Modern Concurrency Techniques: an Exploration

by

Daniel Atkins

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2013

Abstract

In this thesis, we investigate some of the options programmers have when writing a concurrent program. We explore the use of manually created threads, thread-pools, actors, and Software Transactional Memory. We use these techniques to implement case studies of various kinds: a video game, a physical simulation, an image-processing application, and a concurrent data structure. Through-out these case studies, we notice a common thread: concurrency, applied correctly, can improve the performance of a program—but the correct application may not be readily apparent. Concurrency is an important tool in the toolbox of the modern programmer, especially with the rise of multi-core architectures and the increasing prevalence of distributed systems. And like any tool, it is important to understand how and when to use it.

Acknowledgments

I would like to acknowledge my supervisors, Alex and Lindsay, for their support and patience through this last year.

I would also like to thank Michael Blockley for letting me bounce ideas off him, and keeping me on task when I needed it.

Many thanks to Elizabeth Waugh, Mel Duncan, and Russ Kale for proof reading and sanity checking.

Special mention to the denizens of #cave, without whom my nights at uni would have been much less entertaining.

Special thanks to Dr. Janice Polito, for her assistance with various AI-related subroutines.

Contents

Chapter 1

Introduction

Concurrency is an important part of computing today—Moore's Law, as far as processing *power* is concerned, is a thing of the past [25]. Thus, in order to continue increasing processing power, the focus has shifted away from faster clock speeds, and towards an increase in the physical number of CPU cores within each processor. Additional cores mean that more programs can be executed in parallel, and that programs written with concurrency in mind can take advantage of having multiple cores to improve performance [21, 20].

Concurrency techniques were initially aimed at sharing limited system resources. This was especially important during the era of computers that could only process one program at a time—if a program required user input, or had to wait on file operations (which both have notoriously high latency [23]), the entire system was basically doing nothing. Early concurrency, in the form of *multiprogramming*, allowed the computer to switch to a different program while the current one was blocked, due to waiting on an event.

As computers became more able to *multi-task* new forms of concurrency were introduced. Instead of being a means of executing multiple programs at once, concurrent concepts began to be applied *within* programs, not just *between* programs. Even though the program might be ex-

ecuting on a single CPU, the introduction of multiple threads of execution inside a program allowed programs to be more efficient and responsive; while one part of the program was occupied with file input/output (I/O), another could be controlling the user interface. This idea of a program being made up of multiple threads of execution became particularly important as the number of CPU cores inside a computer grew above one, as this allowed a program to *actually perform simultaineous actions*. This had a serious impact on programs. Consider a simple program that sorts an array, with no blocking I/O calls. With one CPU, it doesn't matter how many threads you use in your program: it is not going to run any faster, as those threads are all interleaving on the same processor. In fact, due to the overheads associated with threading, the program may in fact run *slower*. On a machine with multiple cores, however, the threads can **actually** execute concurrently, which means that the array is going to be sorted in less time.

How much less time? The answer is not a simple one. Two threads running on two cores, tasked with sorting an array, should hypothetically be able to sort the array in half the time of a single thread on a single core. The fact that this is generally **not** the case begins to hint at some of the interesting problems faced by concurrent programming. This thesis aims to explore some of the more common problems faced by programmers when writing concurrent apllications, as well as introducing and exploring a few different means of implementing concurrent systems. We approach concurrency from both a design and implementation viewpoint, with a focus on the options available for actually introducing concurrency into a program—and the pitfalls and benefits of each. We also look at how different kinds of applications might require different forms of concurrency. To this end, we implement four different case studies which each focus on a unique area of concurrency, and explore the issues that a programmer might face during the design and implementation of such a program.

1.1 Concurrency Issues

- Not all code can be made concurrent—some programs are, by necessity, sequential. Having a sequential section of code in a program limits the performance gains that can be obtained by making the rest concurrent [6].
- Not all segments of code can be safely executed concurrently [7]. Critical sections of code, such as the updating of global variables, or writing to databases/files, can only be executed by one thread at a time.
- Concurrency is conceptually difficult, and requires a different approach than sequential programming [26].

The first is the simplest case; a program (or part of a program) that just cannot be written in a concurrent fashion. Sometimes, code just **must be sequential**. A good example of this is a sequential stream-cipher; every encryption or decryption operation requires the result of the previous operation in order to proceed. In situations where this lack of concurrency only applies to some of the program, the performance gained by making the rest of it concurrent is limited by how much of the code needs to be sequential, with the total speed-up given by Amdahl's Law [6]:

$$S(N) = \frac{1}{(1-P)\frac{P}{N}}$$

Where N is the number of processors, and P is the proportion of parallel code within the program.

However, even if all of the code for a specific task, such as decrypting a stream filter, cannot take advantage of concurrency to improve performance, we may be able to gain some measure of performance benefit by using concurrency to perform multiple instances of the task at the same time. A program like this is explored in Chapter 5, where we explore the

option of using concurrency to apply a set of image processing functions to multiple images at once.

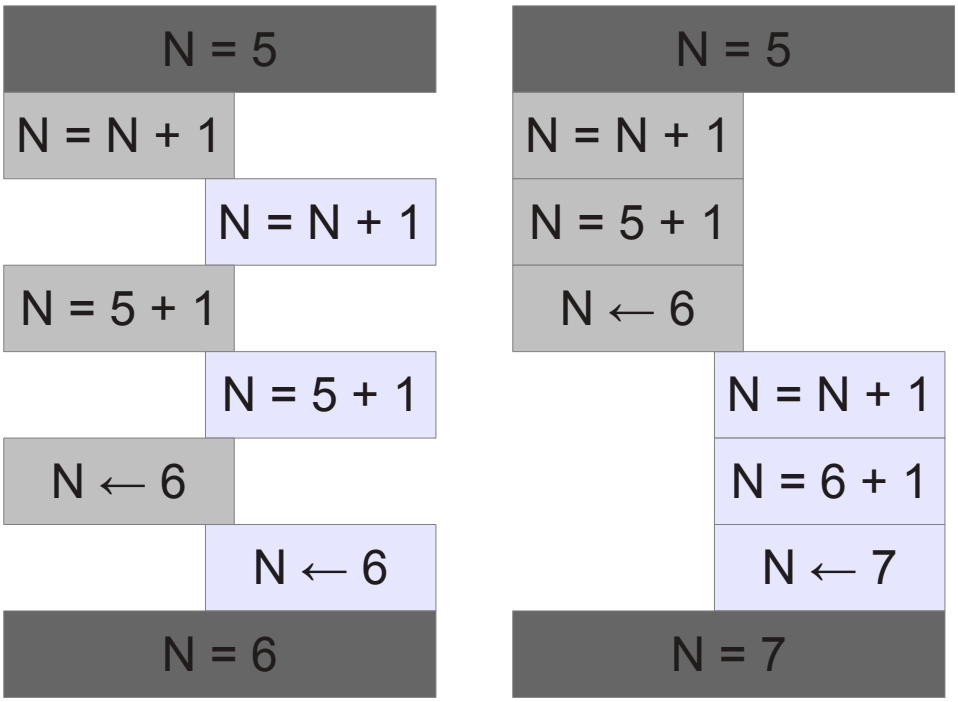


Figure 1.1: Different interleavings of instructions

The second point is an inescapable fact of concurrency: sometimes, you simply *can't* do two things at once (safely, at any rate). The simplest case of this is updating a global variable in a shared-memory environment. Say there are two process, p1 and p2, and they share an integer variable n between them. As shown in Figure 1.1, both processes are attempting to increment the value of n by 1. Logically, this should end up incrementing the value of n by 2, but as we can see, this is not always the case. The incorrect result occurs because one process reads the value of N before the other updates it, resulting in both processes seeing N as 5. There are many ways to solve this problem, but the most common solution is to use a lock to ensure that only one process has access to the variable at a

time, enforcing behaviour similar to the sequence on the right in Figure 1.1. Chapters 3 and 4 go into more detail about the use of locks.

The final point is perhaps an entirely subjective matter of opinion, but it bears mentioning as it is a *very widely held* opinion. Sequential programs are conceptually easy in terms of program flow: they proceed in order, following instructions line-after-line; and given the same set of initial conditions, a sequential program will always follow the same path through the code. The same cannot be said for concurrent applications.

> Probably the greatest cost of concurrency is that concurrency really is hard: The programming model, meaning the model in the programmers head that he needs to reason reliably about his program, is much harder than it is for sequential control flow [25].

> Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism [17].

If a programmer misses a potential race condition in their code (ie: a situation where the output depends entirely on which randomly determined interleaving of operations is used), it may not be immediately apparent—the program might run perfectly fine the first few thousand times it's executed, but then due to some strange cosmic alignment (and the randomess of the scheduler), it crashes spectacularly the next time it's run. Naturally, when you then try to debug the program to figure out what happened, it runs perfectly fine. Bugs related to concurrency can be extraordinarily hard to reproduce.

1.2 Concurrency Techniques

Next we will introduce the concurrency techniques being explored in this thesis, as well as providing a little historical context for them—technical details are left to the next chapter. Each of the case studies implemented in this thesis are implemented using each of these methods. In order to provide a base-line for analysis, all of the case studies also have a totally sequential implementation of the code. This allows us to readily determine if there is any performance benefit from the concurrent versions of the code by comparing it to a baseline performance obtained from the sequential version of the code.

1.2.1 Threads and Thread Pools

A thread is a portion of executing code that is contained inside a process. Threads, unlike processes, share a common address-space in memory, allowing inter-thread communication to be performed in a shared-memory space, instead of relying on message-passing, like inter-process communication.

A thread pool is a set of threads that process a sequence of incoming jobs (usually submitted to a queue).

1.2.2 The Actor Model

In the Actor Model, actors form the base unit of concurrency within the system. An actor can be essentially any part of the system: a CPU, a memory device, a program, part of a program, etc. For the purposes of this thesis, we will be only considering the role of actors *within* a concurrent program, where they behave much like threads but with some very key differences:

- Actors do not share state or memory.

- Actors communicate via message passing.
- Actors only activate when processing a message.

1.2.3 Software Transactional Memory

Software Transaction Memory (STM), unlike the techniques outlined above, isn't a method of creating concurrency in-and-of itself. STM is rather an alternate means of ensuring atomicity when executing critical sections. It is used to replace traditional locks to improve performance when many threads are attempting to acquire and release locks.

1.3 Case Studies

To inspect the differences between the models present above, we have written four case studies. Each case study has been implemented five different ways, corresponding to the models above. Evaluation of each case study is based on performance with respect to the non-concurrent version of the code, and several subjective measures: ease-of-use, library support, ease-of-understanding.

The first case study we investigate is an implementation of the well-known game Asteroids. Through this case study, we intend to investigate the effects of concurrency on updating the game state, and using it to perform collision detection.

The second case study is a physical simulation of gas diffusion. Through this case study, we intend to investigate the effects of concurrency on large computations, as well as examine how the division of tasks amongst units of execution affects performance.

The third case study is an image processing program that applies various filters to input images and then outputs them to the file system. Through this case study, we intend to examine the use of concurrency to perform

multiple, totally independant, operations. We will also investigate the performance effects of having concurrent computations be I/O bound, CPU bound, and memory bound.

The fourth, and final, case study is a HashMap that has been extended to allow concurrent reads and writes. Through this case study, we intend to examine the effectiveness of various mechanisms for controlling access to shared data, as well as how these mechanisms perform for different ratios of read operations and write operations.

1.4 Thesis Outline

Chapter 2 presents more detail about the concurrency techniques that will be investigated in this thesis.

Chapter 3 through to Chapter 6 look at the four individual case studies outlined above. Each chapter is self-contained, presenting results and a discussion about each case study alongside the general description and technical details.

Chapter 7 concludes the thesis with a discussion that encompasses all the case studies.

Chapter 2

Background

This chapter looks at the concurrency techniques used in this thesis in a more technical manner, and provides examples of the techniques in action. It addresses the key similarities and differences between the techniques, and hypothesises as to how well they will perform at different sorts of tasks.

2.1 Threads

Threads are the building-blocks of concurrency *within* a program. They are similar to processes, but much more lightweight. Threads, unlike processes, share much of the state of the program between themselves: with the exception of CPU register values and execution stack, all of the state is available to any thread at any time. This has several key advantages over using separate processes to make a program concurrent, namely:

- No need for message passing to notify the other threads about state changes—all changes are immediately visible
- Faster context switching due to shared state not needing to be altered
- Threads are lighter than processes, so more of them may be spawned

2.1.1 Multi-threading in Java

```
1   int[] array = ...; //Some array of integers
2   Worker worker1 = new Worker(0,10,array);
3   Worker worker2 = new Worker(10,20,array);
4   Thread t1 = new Thread(w1);
5   Thread t2 = new Thread(w2);
6   t1.start();
7   t2.start();
8   //Wait for the threads to finish before continuing
9   try{
10      t1.join();
11      t2.join();
12  } catch (InterruptedException e){
13      //Exception handling code
14  }
```

Figure 2.1: An example of creating two worker threads in Java

The most common and simple way of obtaining a new thread in a program is to **fork** a new thread. The method of doing this varies from language to language, and from library to library. Figure 2.1 shows a snippet of Java code that creates two threads and beings executing them. Creating a thread in Java is a straight-forward affair that involves creating a new instance of the `Thread` class, which takes an instance of `Runnable` as an argument (Note: the `Worker` class implements the `Runnable` interface, as shown in Figure 2.2). Alternatively, the Worker class could *extend* Thread, and provide an implementation of the required `run()` method, as shown in Figure 2.3. Such an approach would change the Thread creation in Figure 2.1 to look more like Figure 2.4.

Once the thread has been instantiated, it is started by calling the `start()` method upon it, whereupon the JVM spins off a new thread to execute

```
1   public class Worker implements Runnable {
2       private int start;
3       private int end;
4       private int[] array;
5
6       public Worker(int s, int e, int[] a){
7           start = s;
8           end = e;
9           array = a;
10      }
11
12      //To satisfy the Runnable interface
13      public void run(){
14          for(int i=start; i<end; i++)
15              doSomeOperation(array[i]);
16      }
17  }
```

Figure 2.2: An example a Worker that implements runnable

```
1   public class Worker extends Thread {
2       private int start;
3       private int end;
4       private int[] array;
5
6       public Worker(int s, int e, int[] a){
7           super();
8           start = s;
9           end = e;
10          array = a;
11      }
12
13      //We need to provide this method
14      @Override
15      public void run(){
16          for(int i=start; i<end; i++)
17              doSomeOperation(array[i]);
18      }
19  }
```

Figure 2.3: An example of a Worker that extends Thread

```
1   Thread t1 = new Worker(0,10,array);
2   Thread t2 = new Worker(10,20,array);
```

Figure 2.4: Instantiation of a Worker Thread

the code within the object's `run()` method. Note that calling the `run()` method instead of the start method does **not** execute the method in a separate thread.

To force the current thread to wait for another thread to finish before continuing, the `join()` method should be called on the other thread, as demonstrated on lines 10-11 of Figure 2.1. Notice that in Java joining a thread requires exception handling in the event that the joined thread is interrupted (via the `interrupt()` method call).

2.1.2 Thread Pools

A thread pool is a set of existing threads that are already resident in memory, and ready to start executing code whenever they are required to. Unlike the threads described above, the threads in a thread pool don't have a specific function coded into their `run()` methods; instead, they take **jobs** from a queue of available jobs and execute the code contained within the job. Thus, multiple jobs can be submitted to the pool, and when a thread is free, it will take the first available job. In Java, a job is typically required to implement some for of interface so that each job has a common method that can be used to execute it. Java has built-in support for thread pools using the `ExecutorService` class. Two implementations of a Java thread pool are described below, but there are several others available, including:

FixedThreadPool Maintains a pool with a fixed number of threads. Used for the Work-Sharing pool described below.

CachedThreadPool Maintains a variable-sized pool that creates new threads when required, but re-uses old threads if they are available. Removes un-used threads from the pool after a certain time limit.

SingleThreadExecutor Not really a "pool" at all; uses a single worker thread to complete jobs in the queue.

Work-Sharing

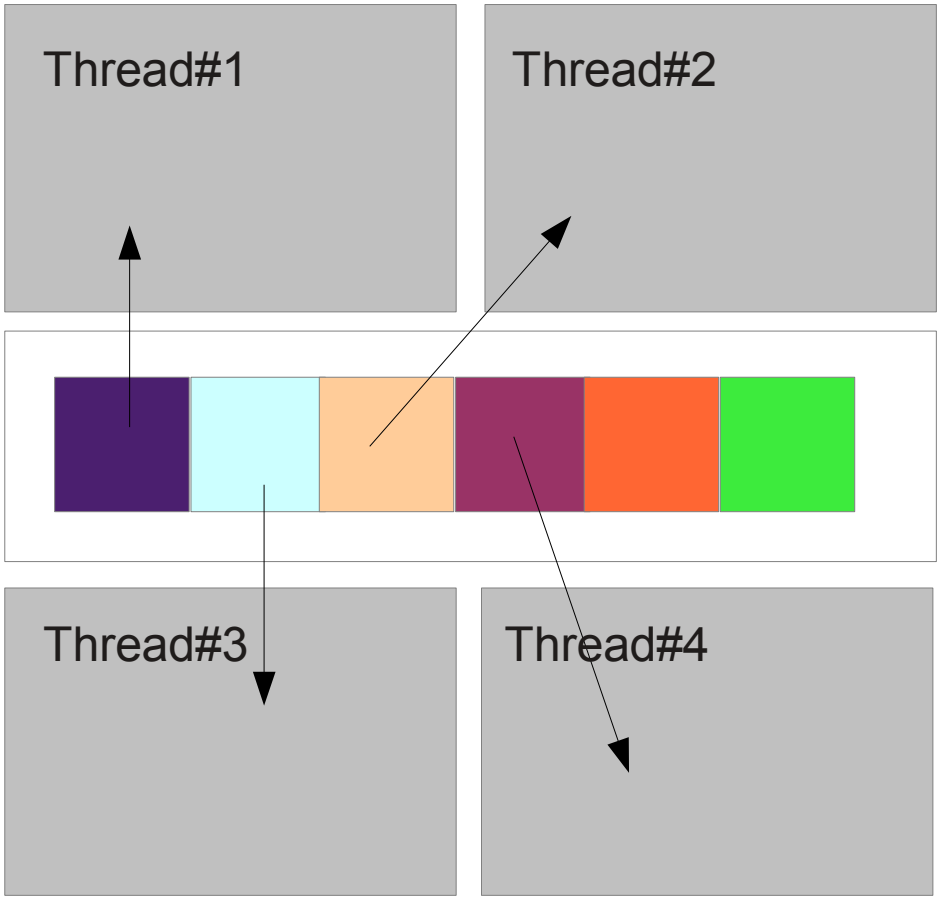


Figure 2.5: Four threads in a Work-Sharing Pool

A work-sharing pool, like the one shown in Figure 2.5, is a thread-pool that maintains a global queue of jobs. When a thread in the pool is ready to accept a job, it removes the first available job in the queue and begins to process it. In order to ensure that all threads see a consistant state of the queue, any operations on the queue must be *thread-safe*: that is, the queue must guarantee that race-conditions will not occur, and that all changes

to the queue are immediately visible to all threads in the pool upon an operation completing. This synchronization of threads in the pool in done by the pool itself.

```
1     int[] array = ...; //Some array of integers
2     ExecutorService pool = Executors.newFixedThreadPool(4);
3     Worker w1 = new Worker(0,10,array);
4     Worker w2 = new Worker(10,20,array);
5     Future f1 = pool.submit(w1);
6     Future f2 = pool.submit(w2);
7
8     try{
9         f1.get();
10        f2.get();
11    } catch(CancellationException | ExecutionException
12            | InterruptedException e){
13        //Exception handling code
14    }
```

Figure 2.6: An example of using a Work-Sharing thread pool in Java

In Java, a Work-Sharing queue may be contructed and used as shown in Figure 2.6—notice the similarities to Figure 2.1. However, instead of creating new threads, we instead `submit()` the Worker (the *same* worker, in fact—the jobs must implement the `Runnable` interface) to the pool. Submitting the job returns a `Future`, an object that is used to keep track of the job's status, i.e. if it has complete, and if so, what its result is. Like manually creating and running a thread, we can force the current thread to wait for the job to finish executing before continuing; this is accomplished via the `get()` method of the `Future` as shown on lines 9-10 of Figure 2.6. Notice that `get()`, like `join()` requires exception handling in the event that the job is cancelled, throws an exception itself, or the thread executing it is interrupted.

An important note: a job that implements the `Runnable` interface **cannot return a value**, as `run()` is declared as returning `void`. If a return value is required, jobs should instead implement the `Callable<T>` interface, with the appropriate return type as the type parameter. For the case studies used in this thesis, return values were not required.

Work-Stealing

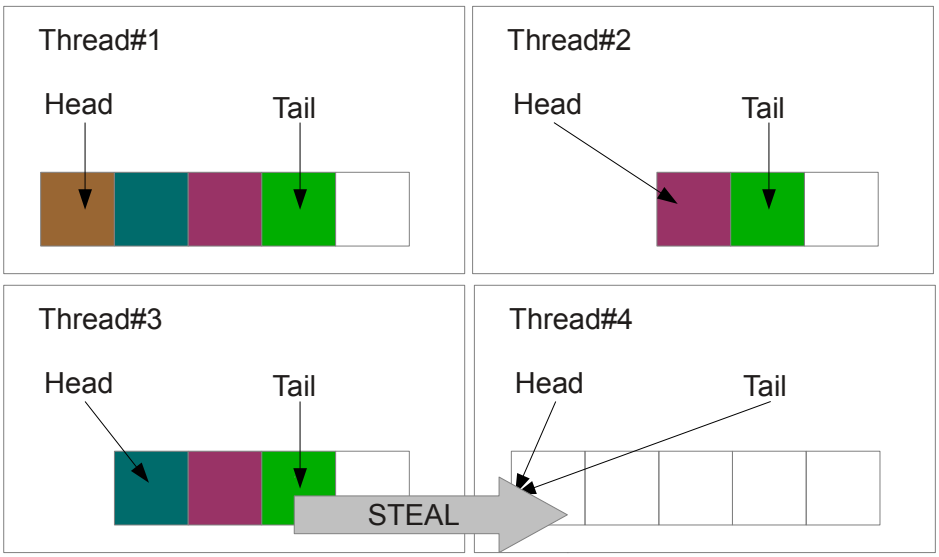


Figure 2.7: Four threads in a Work-Stealing Pool

In a work-stealing pool like the one shown in Figure 2.7, there is no global job queue like a work-sharing pool. Instead, each thread maintains its own queue of jobs. Such a queue is typically double-ended, with a reference to the **head** and **tail** of the queue being maintained by the thread [10]. A thread processes jobs from the head of the queue until its personal job queue is empty, at which point it proceeds to *steal* jobs from the tail of another thread's queue. The implementation of the work-stealing pool used in this thesis is contained in the JSR166Y package, authored by

Doug Lea [4], and is based on prior research [8, 18]. The lack of a global queue means that the worker threads don't need to synchronize as heavily as they would in a work-sharing queue. Figure 2.8 demonstrates an example of using the `ForkJoinThreadPool` contain in JSR166Y, in Java. Notice that the **only** difference to the work- sharing code (Figure 2.6 is the constructor used on line 2—one of the advantages provided by the use of the `ExecutorService` interface means that both pools have exactly the same methods, even if the underlying implementation is different.

Important Note: the version of Java used in this thesis is Java 6. The contents of JSR-166 (specifically `ForkJoinPool`) are available as part of the standard Java library as of Java 7.

```
1   int[] array = ...; //Some array of integers
2   ExecutorService pool = new ForkJoinPool(4);
3   Worker w1 = new Worker(0,10,array);
4   Worker w2 = new Worker(10,20,array);
5   Future f1 = pool.submit(w1);
6   Future f2 = pool.submit(w2);
7
8   try{
9       f1.get();
10      f2.get();
11  } catch(CancellationException | ExecutionException
12          | InterruptedException e){
13      //Exception handling code
14  }
```

Figure 2.8: An example of using a Work-Stealing thread pool in Java

2.2 Actors

The Actor Model proposes that every part of a system can be modelled as an Actor [14, 9, 15], an entity that performs operations in response to messages being sent to it. Actors are *internally sequential*, though many such actors may be executing in parallel. As such, an actor in a system can only process one message at a time; the rest get stored in the actor's *mailbox*, which typically takes the form of a queue. In terms of concurrent software, Actors operate on a paradigm that is more similar to processes than threads (though they are much lighter-weight than both), and as such **do not share state**.

As an example of this difference, imagine a program that uses actors to sort an array using merge-sort. If you were using threads, the array might be stored in a location that is accessable to all threads, and they would read and write to that same location. With actors, however, there are no such locations available to multiple actors—each actor can **only** read/write to its own internal state. Any communication with any other actor in the system **must** be done via message passing. So to split an array into chunks to be processed by an actor, we must send each actor a message containing a copy of their section of the array. Once they have sorted it, they will send a copy of the sorted chunk to a different type of actor who will recombine them into a fully sorted array, and then send that onwards to its destination.

Because of this lack of shared state, messages need to be complete *copies* of the original data. Take this snippet of Java code for example:

```
1    PrintingActor pa = new PrintingActor();
2    String text = ''Hello World'';
3    pa.tell(text);
```

Figure 2.9: Psuedocode Actor Example

The code in Figure 2.9 is **semantically incorrect**, as we are sending a *reference* to `text` to the PrintingActor. If the PrintingActor stores this incoming text somehow, we now have a reference that points inside an actor's state. While this is certainly *permitted* by the Java programming language, it goes against the intended semantics of the Actor System. A more correct version of the code would send a *clone* of the string. This requires a substantial paradigm shift for those programmers who have learned little but Object-Oriented Programming—the ability to just reach in and alter an objects state is something a OOP-programmer might take for granted. Likewise, a global state which can be altered by any thread at any time is a luxury the usual OOP paradigm allows programmers as a given. To suddenly find that global state is something that needs to be manually passed around as a message in your system is perhaps a little disconcerting to those unfamiliar with functional programming.

As an example of this, imagine a simple 2D game in which you pilot a ship through an asteroid field. Figure 2.10 gives an example of what actors might be involved in the system, and the messages they might send between each other. An update message from the Game Controller would contain the elapsed time since the last update, which the objects in the game (the ship and the asteroids) would then use to update their positions according to how they were moving. Once they have done that, they need to send their new location (and a copy of their collision box/sprite) to the Collision Dectector and the Renderer, which process that information. If a collision is detected, the collision detector needs to be able to inform the game object that it has collided with something. In a paradigm that allows for shared memory, this scenario looks and behaves very differently: the game controller maintains a global list of game objects, which the renderer and collision detector have direct access to. One of the benefits of the actor model over the shared-memory model is that *because* the data isn't shared, there are no concurrent access issues that need to be addressed. However this results in duplication of information; at the very least, there are three

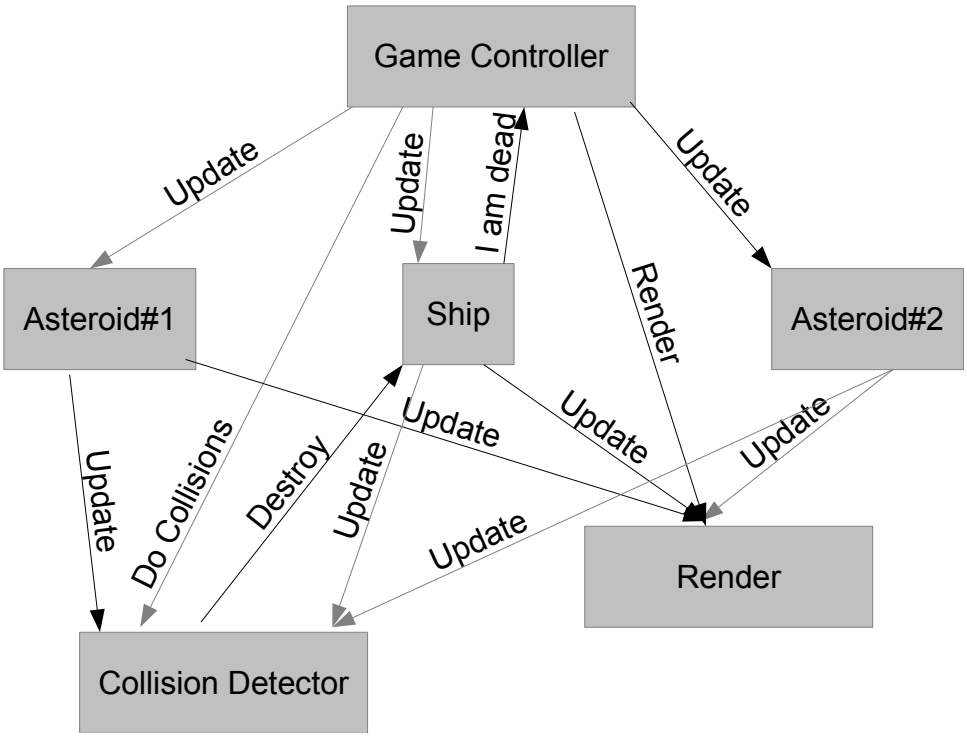


Figure 2.10: A game using Actors

different copies of a game objects location—one inside the game object actor, and another for the renderer and collision detector. Because of this memory overhead, and the overhead of passing and storing messages in the system, the Actor Model seems like it may be better suited to tasks that require less information to be passed between each actor in the system.

We will be using Akka [1] to provide the implementation of the actor system for our case studies.

2.3 Software Transactional Memory

A discussion of concurrency is not complete without an examination of the means of controlling access to shared memory. The actor model does

away with this issue by eliminating the concept of shared memory entirely; however, threads (and by extension thread pools) are firmly entrenched the shared-memory model. In Java, the traditional approach to ensuring that shared data is only modified by one thread at a time is to use the `synchronized` keyword (as shown in Figure 2.11); this behaves as a **lock**—only the thread that has acquired the lock may execute the code encapsulated by `synchronized`. However, locks can be a little cumbersome to use, and can often impede the performance of a program; for example, if every method on a data-structure is declared `synchronized`, only one thread may use it at a time, no matter what operation is being performed. If the program makes extensive use of this data-structure, then this is really no different than only having one thread in the first place.

Software Transactional Memory (STM) provides an alternative mechanism for controlling concurrent access to memory. STM encapsulates accesses in *transactions* and performs (or *commits*) these transations in a non-interfering atomic way. It typically provides a lock-free solution to the problem of concurrent modifcations to memory, and ensuring atomicity [22]. Importantly, *unlike* using `synchronized` it makes a distinction between transactions that are just **reading** values, and transactions that actually modify the state.

We are using DEUCE to provide an implementation of STM on the Java Virtual Machine [3]. DEUCE uses "an original locking design that detects conficts at the level of individual *fields* without a signifcant increase in the memory footprint or GC overhead" [11]. It does this by intercepting classes before they are executed, and instruments them to provide STM support where indicated by the programmer. DEUCE was remarkably simple to use, requiring only the addition of the `@Atomic` annotation to any method that must be executed as a transaction, as shown in Figure 2.12.

```
1       private int x;
2
3          public synchronized void increment(){
4                   x++;
5          }
```

Figure 2.11: Synchronized Example

```
1       private int x;
2
3          @Atomic
4          public void increment(){
5                   x++;
6          }
```

Figure 2.12: STM Example

2.4 Experimental Technique

All of the case-studies in this thesis were executed on identical hardware (the Dell Optiplex 990), as per the specifications in Table 2.1. To allow the Java Virtual Machine (JVM) to warm up, the first 5 results from every experimental run were discarded, and the remaining 95 used to calcuate the mean execution time for the program.

CPU	Intel(R) i5-2400 CPU @ 3.10GHz (Four Cores)
RAM	4GB
OS	Arch-Linux (3.7.5-1-ARCH x86_64)
Java Version	1.6.0_37, 64-bit
Actors Library	Akka-2.0.1
STM Library	Deuce-1.3.0
Work-Stealing Library	JSR-166Y

Table 2.1: Hardware and Software Specifications

Chapter 3

Case Study: Asteroids

The first case study is the well-known game of Asteroids, one of the most popular arcade games of all time [27]. The program used in this case-study was implemented from scratch in Java, and does not use any third-party libraries above those mentioned in Section 2.4. The focus of this case study is determining how well each of the concurrency techniques scale as more objects are added in to the system.

3.1 Description

In this version of Asteroids, the player controls a spaceship that flies around the screen, shooting down incoming asteroids. A screen-shot of a game in progress, showing the different size-classes of asteroid, the player's ship, and some bullets, can be seen in Figure 3.1. Large asteroids, when destroyed by bullets, split into two medium-sized asteroids. Medium asteroids behave similarly when destroyed, producing two small asteroids. Small asteroids do not split when destroyed. Bullets have a limited life-span, and cannot collide with each other or the player's ship. Asteroids cannot collide with other asteroids, but can collide with, and destroy, the player's ship. There are only three different types of game entity in our implementation of Asteroids: the player's ship, asteroids, and bullets.

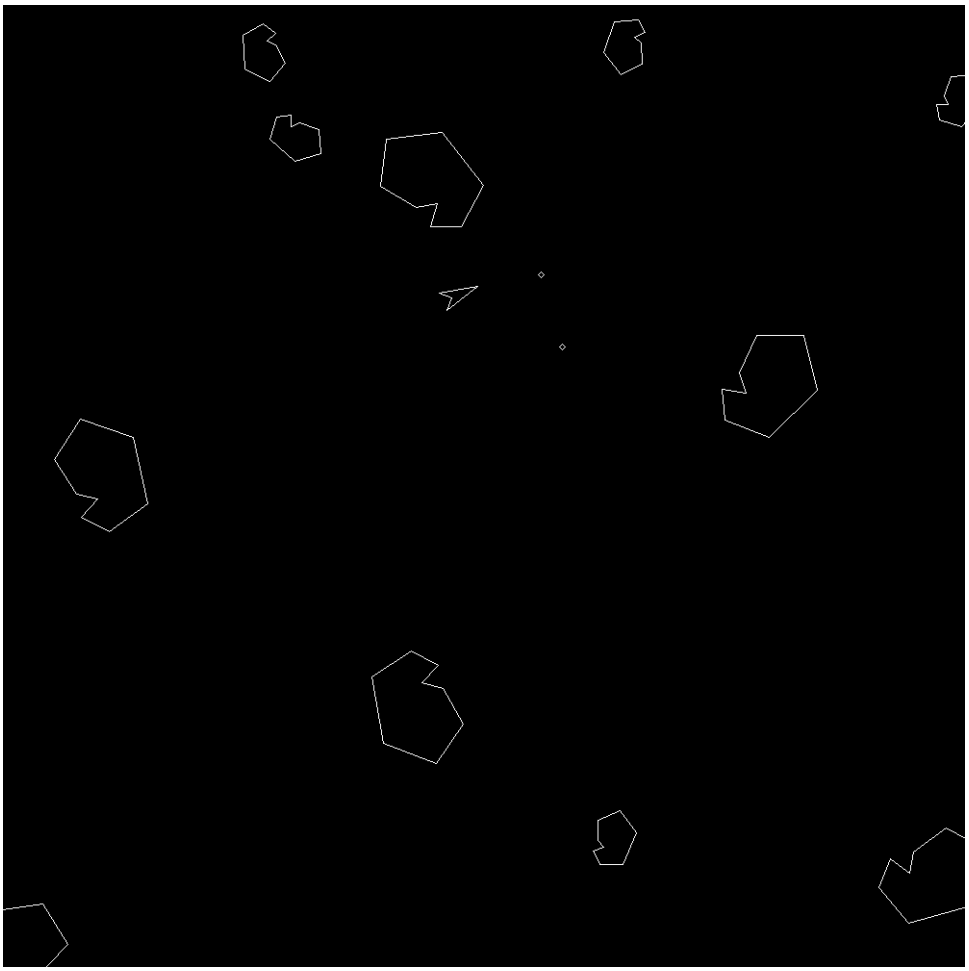


Figure 3.1: A game of Asteroids

The player's ship is controlled via keyboard input, and moves about on the screen. It can **rotate**, **accelerate**, **decelerate**, and **shoot**. The game maybe be launched without a user-interface, in which case an automated player takes the helm—this is to emulate user input during automated testing. If the players ship comes into contact with an asteroid, it is destroyed—this interaction is not present when running the game without a UI.

Asteroids have a constant velocity, and begin play moving in a random

direction, rotating with a random angular velocity as they move. All asteroids begin the game as the largest size. If an asteroid splits, the two new asteroids fly apart in random—but opposite—directions. All asteroids of the same size class have the same shape. Asteroids cannot collide with each other.

Bullets are emitted by the player, and can interact only with asteroids. A bullet colliding with an asteroid causes both entities to be destroyed—this may cause the asteroid to split, as outlined above.

The screen geometry wraps around at the edges, so an entity leaving the top of the screen will re-enter from the bottom, and vice-versa. The same is true for the left and right edges.

Like many games, our implementation of Asteroids is structured in a game loop which has multiple phases. At its simplest, a game loop consists of two phases—the **Update Phase**, which updates the state of game entities (movement, etc), and the **Rendering Phase**, which renders the current game state to the display. The update phase can be further broken down into sub-phases: for this implementation of Asteroids, it can be considered to be comprised of a *movement* sub-phase, and a *collision-detection* sub-phase. Figure 3.2 demonstrates this splitting of tasks.

3.2 Design

There are several ways to make this game loop concurrent. One approach is to have the update phase and rendering phase run in parallel with each other. This would hypothetically allow a single iteration of the game loop to take as long as the duration of the longest phase; instead of the duration of both phases together. However, for a game as simple as asteroids, this approach is unnecessary as the program can calculate updates to the game state far more quickly than the the 30 updates per second that are required to maintain a rendering speed of 30 frames per second. The rendering itself is capped at 30 frames per second *regardless* of how quickly the update

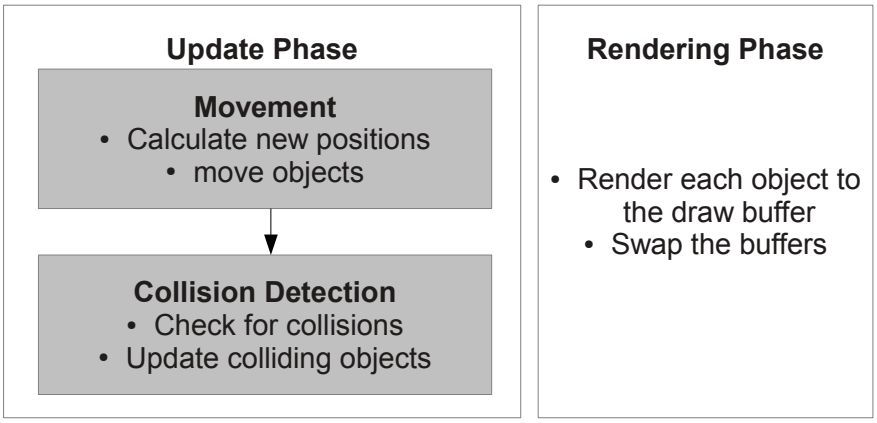


Figure 3.2: Phases in the game loop

phases take to calculate, as the actual rendering of a frame takes significantly longer than the calculations required to produce it. Another reason not to render in parallel with the update phase is the simple fact that the update phase might only be half-complete when the renderer executes, leaving the display in an inconsitant state. Another approach is to parallelize *within* phases. Within the rendering phase, trying to render objects in parallel can cause some issues; ordering, as an example, is a rather large one. If certain objects are required to be on top of, or behind, certain other objects, then there is a strict order in which they must be rendered. Support for concurrent rendering is also a problem: some libraries just don't allow concurrent access to the draw buffer. There are yet more issues if you're using a library like OpenGL, which is a state machine and thus requires parallel operations to essentially be atomic—and thus essentially sequential.

Within the update phase, however, this limitation does not apply. A close examination of the update phase shows that it does two things:

- Updates the position of game entities based on the time since the last update.

- Performs collision detection between game entities, and updates their state if required.

There are two main ways to check for collisions between objects. The first involves applying small movement updates to each object, and then checking to see if any objects overlap. The second is to compute the trajectory of each object (or each point in the object) and check if any trajectories intersect. For this implementation of Asteroids, the former is used. While the second approach is possible, the rotational velocity of each asteroid makes trajectory calculation much more complex, as it involves the rotation of a polygon as it translates through space. For the sake of simplicity, the incremental approach was chosen. The function that checks for overlap first checks to make sure the two game objects can actually collide (bullets with asteroids, asteroids with the player), and then does a basic bounding-box check to see if a collision could be possible. If the two bounding boxes overlap, it checks each line in each polygon involved in the collision to see if they cross any of the lines in the other polygon. If any line crosses, it detects a collision. Due to the choice of collision detection algorithm, collision detection is dependant on the movement update—it **must** occur *after* movement has been calculated. This means that the update sub-phases *cannot* be executed in parallel. We need to go deeper, and look inside the sub-phases.

The movement update calculation is performed in the exact same way on every game entity, and only depends on the amount of time that has passed since the last update. Since there is no dependency *between* game entities, and there is very clear way to break the problem down into multiple units (i.e. individual game objects) this is an example of an "embarrassingly parallel problem" [13]. This has the benefit that updating of the positions of each game object can be done with no need for any communication between entities.

The same cannot be said of the collision detection sub-phase, as the algorithm inherently requires knowledge of the locations of multiple game

entities. The same approach can be taken, but care needs to be taken to avoid different processes attempting to update the same objects simultaineously, as this could lead to an inconsistant game state. A solution to this is to require any thread that wishes to apply updates to a game entity—for example, if a bullet hits an asteroid—to obtain a lock on that game entity, which is released when the update is finished. A thread that wishes to read that entity's state must also acquire the same lock, thus ensuring that state is consistent between all threads. In the case of asteroids, *not* doing this might result in an asteroid being struck by multiple bullets (thus spawning more smaller asteroids than it should), or one bullet hitting multiple asteroids.

3.3 Implementation

This section discusses the implementation of each technique, and any difficulties faced by trying to apply the technique to the problem at hand.

3.3.1 Manual Threads

Implementing threading within a program manually, as discussed in Chapter 2, consists of constructing worker threads to perform a specific task. As we have already identified the sections of code that are to be parallelized (as above), implementation of these worker threads was easily inferred from the design process—at least to begin with.

As outlined above, the movement sub-phase of the update phase needs to finish in its entirety *before* collision detection can be done. This would seem to imply that these two sub-phases need to be made concurrent independently, using `join()` to ensure that all the previous threads have finished before the next set are created, as outlined in Figure 3.3. A MovementThread runs through its allocation of gameobjects and performs the movement update. Likewise, a ColliderThread performs collision detec-

```
1  while(true):
2          deltaTime = start - now;
3          start = now;
4
5          for i in NumThreads:
6                  create new MovementThread
7                  start thread
8          join on all MovementThreads
9          for i in NumThreads:
10                 create new ColliderThread
11                 start thread
12         join on all ColliderThreads
13         render
14
15 MovementThread:
16         for each object in allocation:
17                 object.move(deltaTime)
18
19 ColliderThread:
20         for each object in allocation:
21                 for each otherObject in global list:
22                         object.collide(otherObject)
```

Figure 3.3: Running the Game Loop (Pseudocode)

tion between its allocated objects and the other game objects in the game. In all cases, each thread is given an allocation of the game objects in the global object list, equal to $\frac{NumObjects}{NumThreads}$.

```
1  while(true):
2          deltaTime = start - now;
3          start = now;
4
5          for i in NumThreads:
6                  create new UpdaterThread
7                  start thread
8          join on all UpdaterThreads
9          render
10
11 UpdaterThread:
12         for each object in allocation:
13                 object.move(deltaTime)
14         wait on barrier
15         for each object in allocation:
16                 for each otherObject in global list:
17                         object.collide(otherObject)
```

Figure 3.4: Running the Game Loop with a Barrier(Pseudocode)

However, Java presents another option: Barriers. Barriers are objects that are used to control access to sections of code, like locks (discussed in Chapter 2). Unlike locks, they are used to ensure that all threads enter the section of code they protect at the **same** time. They do this by blocking any thread that attempts to access the barrier until a specified number of threads have tried to access it—at which point it unblocks all the threads, and they all enter the contained section at the same time. The resulting pseudo-code for the game loop is given by Figure 3.4. The MovementThread and ColliderThread have now been combined into one Up-

daterThread.

So which approach is better for this situation?

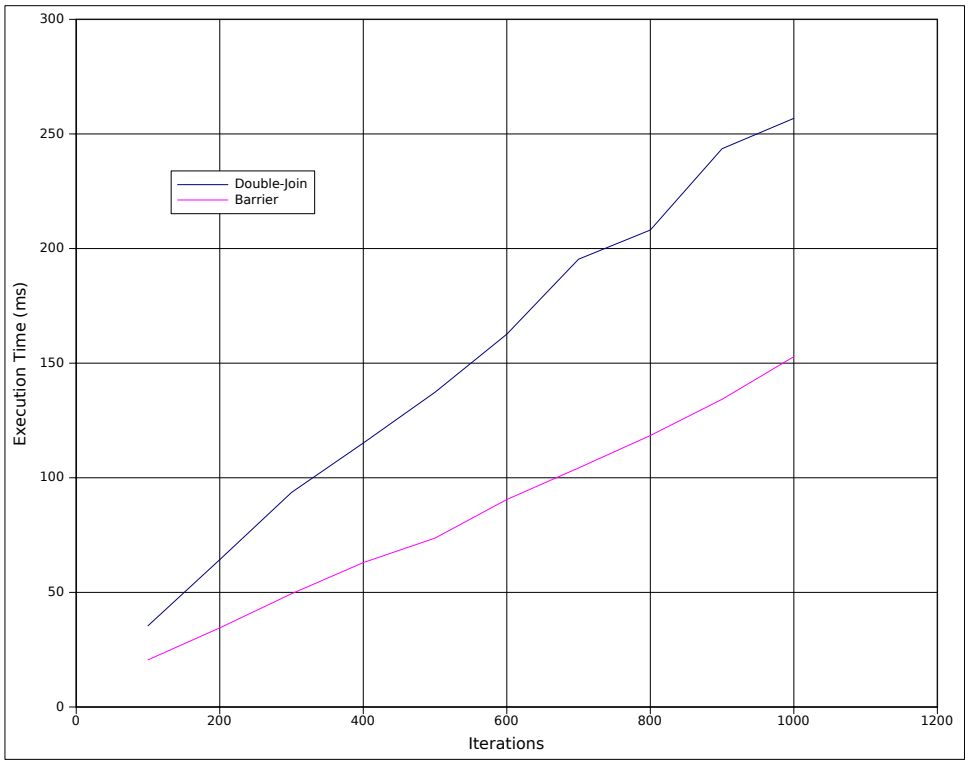


Figure 3.5: Execution times for the Double-Join and Barrier approaches

Figure 3.5 demonstrates the execution times for both approaches when updating 100 game-objects over varying numbers of iterations. As shown, the overheads caused by creating threads twice in one iteration of the loop are not insubstantial. To avoid this extra overhead, the barrier option was chosen, and the created threads are forced to wait on it at the end of the update phase. This ensures that the sub-phases are synchronized correctly across multiple threads, while avoiding the overhead of creating entirely new threads.

As the movement update phase operates on each object individually, and doesn't need to read or write to any sort of shared state outside of

that object, changes to the game objects' state do not need to be protected by any form of synchronisation. For collision detection however, some kind of protection is required; otherwise it becomes entirely possible to collide with an object that has—in another thread—just been marked as destroyed. In game terms, this means a single bullet can penetrate multiple asteroids instead of just one, or when multiple bullets hit a single asteroid, it gets "destroyed" once for each bullet (thus spawning far more than two smaller asteroids), instead of just once. This was accomplished by using Java's built-in `synchronized` keyword, which acts as a lock on the entire game object. Deadlock was avoided by ensuring that all threads acquire and release objects in the same order.

3.3.2 Work-Sharing and Work-Stealing Queues

As seen above, creating a new thread (or two) every iteration can be expensive. Fortunately, Java offers a solution in the form of the `ExecutorService` class. Jobs (objects that implement `Runnable`) are submitted to the ExecutorService (in this case, a `ThreadPoolExecutor`). The thread-pool contains a fixed number of threads, and these threads work through the jobs in the queue. Figure 3.6 shows how the game loop would be structured under this scenario.

For this program, there were two kinds of jobs that were submitted to the queue: MovementUpdate jobs, and a CollisionDetection jobs. These jobs are analogous to the MovementThreads and CollisionThreads from the previous solution. After all the MovementUpdate jobs are submitted to the queue, their Futures are used to ensure they have completed before the CollisionDetection jobs are submitted to the queue. A key point of difference with the previous implementation is that the jobs submitted to the queues deal with *one object at a time*, not a range of objects. As per the previous technique, concurrent modification of game objects was an issue, and was dealt with in the same way.

```
1  while(true):
2          deltaTime = start - now;
3          start = now;
4
5          for each object in gameObjects:
6                  submit new MovementJob(object, deltaTime)
7          wait on all futures
8          for each object in gameObjects:
9                  submit new CollisionJob(object, deltaTime)
10         wait on all futures
11         render
```

Figure 3.6: Running the Game Loop with a Thread Pool (Pseudocode)

3.3.3 Software Transactional Memory

This implementation of Asteroids is essentially the same as the Work-Sharing Queue, except that STM is used to manage concurrent data access, instead of Java's `synchronized` keyword. This means that any update to the state of a game object, or read access to that state, is wrapped in a transaction and handled by the STM engine.

3.3.4 Actors

There are 6 different types of Actor in this implementation of Asteroids; one for each of the three entity types, one to handle rendering, one to handle collision detection, and one to keep the game running. They are descibed below:

AsteroidActor This Actor type maintains a single Asteroid game object. All of the game object actors recieve `update` messages from the Game Controller, at which point they execute movement updates and send their new positions and copies of their polygons to the Ren-

derer and the Collision Detector.

BulletActor This Actor type contains a single bullet game object.

ShipActor This Actor type contains a single Ship game object. It receives messages from the input controller to direct the ship's actions.

RendererActor This Actor maintains an internal list of Polygon objects that are sent to it by the game object actors. Whenever it is sent a `Render` message by the Game Controller, it renders the list to the screen.

CollisionActor This Actor maintains a mapping of the addresses of the current game object actors and their in-game polygons and positions. When it receives a `collide` message from the Game Controller, it uses its local copies of the game object's polygons to perform collision detection. If it detects a collision, it informs the involved game object actors that they have collided with another object, and that they should take the appropriate actions.

GameControllerActor This Actor is responsible for maintaining the main game loop. It is sent a single `Run` message at the start of execution, and then takes control of the program. It dispatches `Update` messages to the game object actors to inform them to perform movement updates. When all of the game object actors have responded with `UpdateComplete` messages, it dispatches `Render` and `Collide` messages to the RendererActor and the CollisionActor telling them to perform their respective duties. When collision detection is complete, it issues a `Run` message to itself to start the game loop again.

3.4 Results

Three sets of experiments were performed using the implementations outlined above. The first run was with 100 asteroids over numbers of itera-

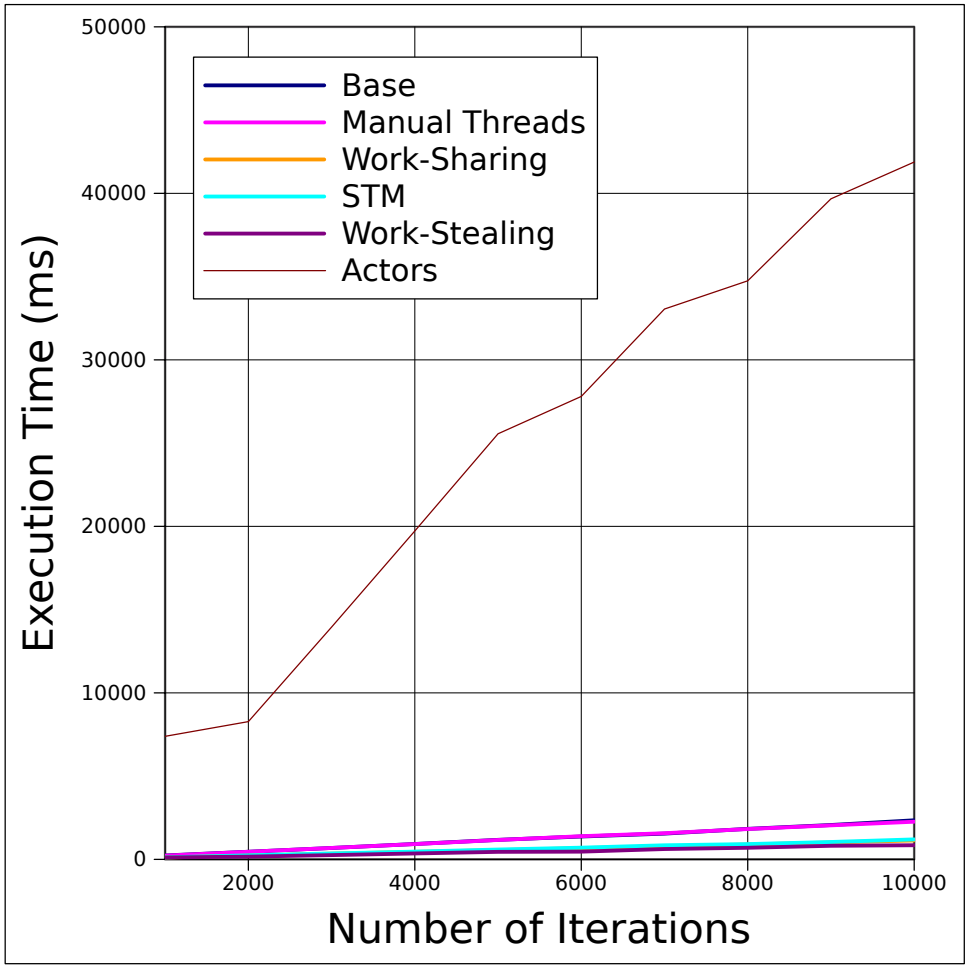


Figure 3.7: Execution times for 100 Asteroids

tions that varied from 1000-10000 in increments of 1000. The second was over the same variation of iterations, but with 1000 asteroids. The final experiment was with a varying number of asteroids (100-1000 in increments of 100) over a fixed number of iterations (10,000). In all cases, the number of threads manually created, in thread-pools, and available to the Actor subsytem was 4.

From the results presented in Figures 3.7 and 3.8, we can see that the

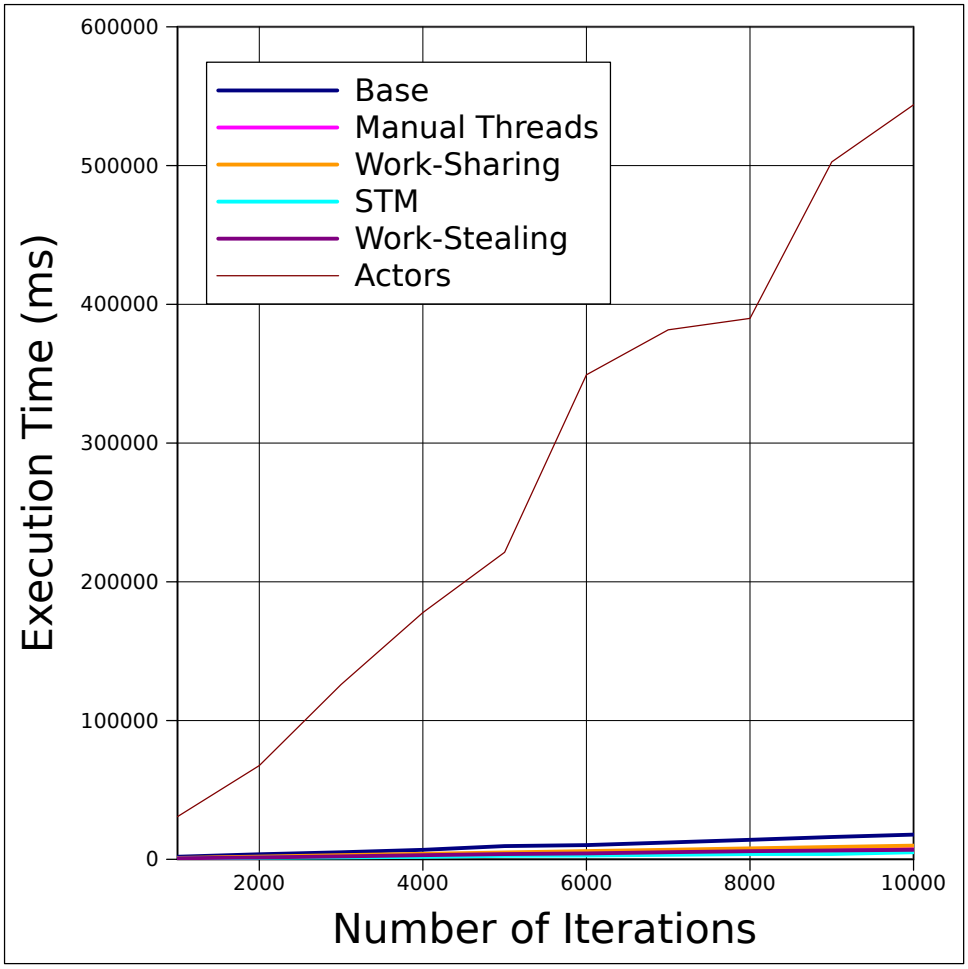


Figure 3.8: Execution times for 1000 Asteroids

Actors approach is by far the worst performing approach. Possible reasons for this will be explored in the next section. We re-present the results for the other approaches in Figures 3.9 and 3.10.

Interestingly, Figure 3.9 indicates that for a small number of asteroids (100), there is very little difference between manual threading and having no concurrency at all. However, for more asteroids (1000), it is clear that we gain some performance benefit by using manual threading over the

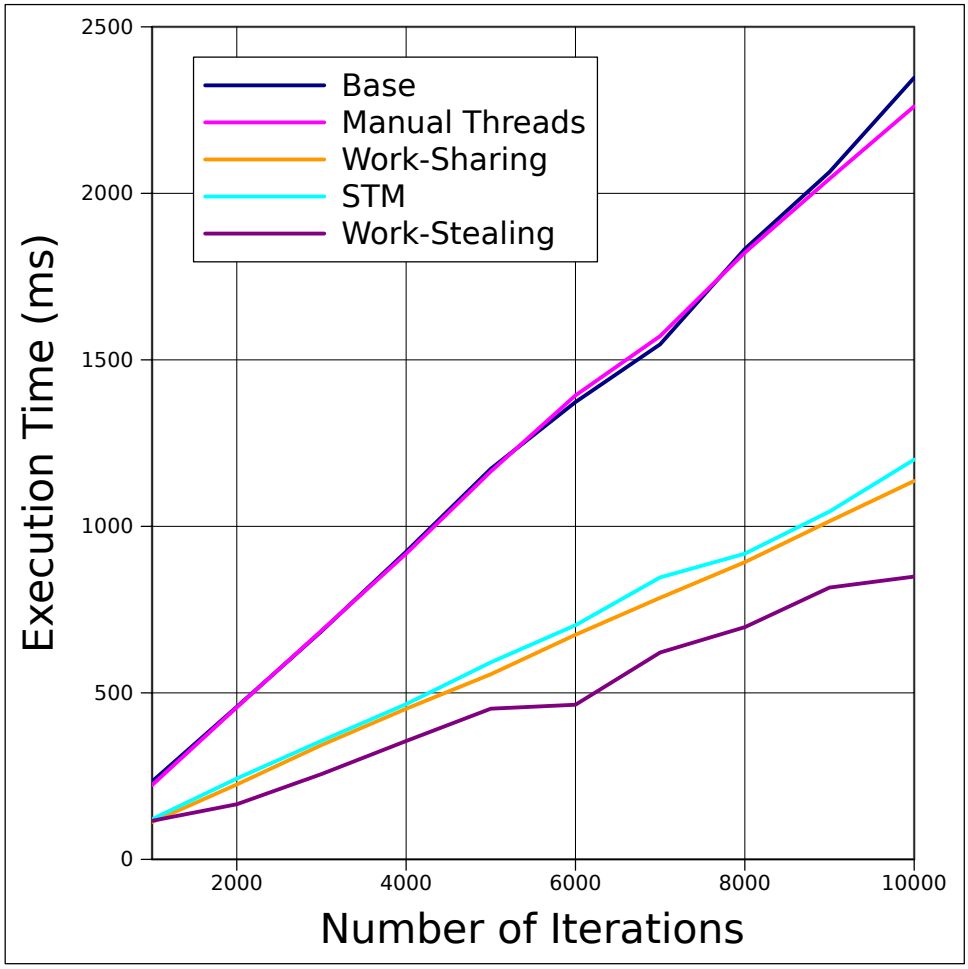


Figure 3.9: Execution times for 100 Asteroids, without Actors

non-concurrent case. To investigate this further, the experiment was re-run with a fixed number of iterations (10,000) and a varying number of asteroids. Figure 3.11 demonstrates these results.

Of the two thread-pool approaches, work-stealing outperforms work-sharing in all cases, and for small numbers of asteroids is the fastest overall performer. For larger numbers of asteroids, we see that STM takes its place as the fastest option.

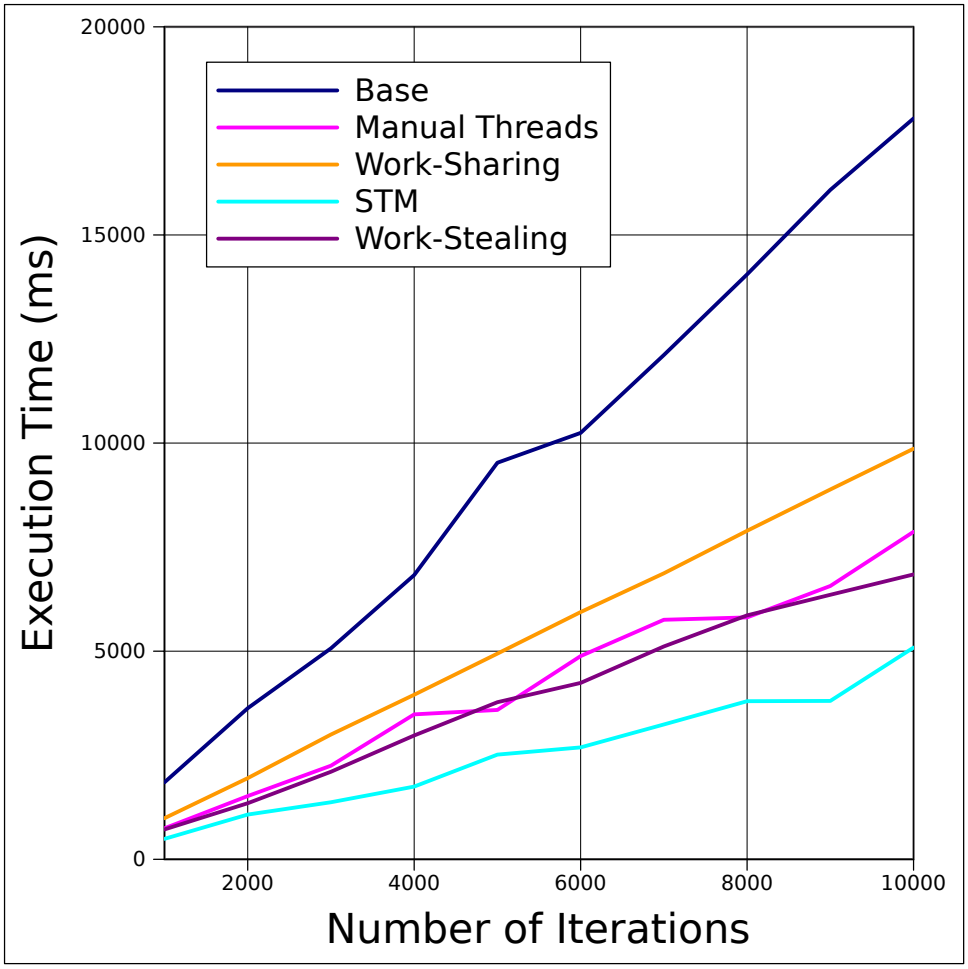


Figure 3.10: Execution times for 1000 Asteroids, without Actors

3.5 Discussion

From the results shown in the previous section, it is clear that Actors are not a suitable solution for providing concurrency in this problem. On the surface, actors appear to lend themselves well to making a game such as Asteroids concurrent: each game object is an individual entity that does not share state with any other game object—much like actors themselves. However, a game is not made of just its entities; the renderer and the col-

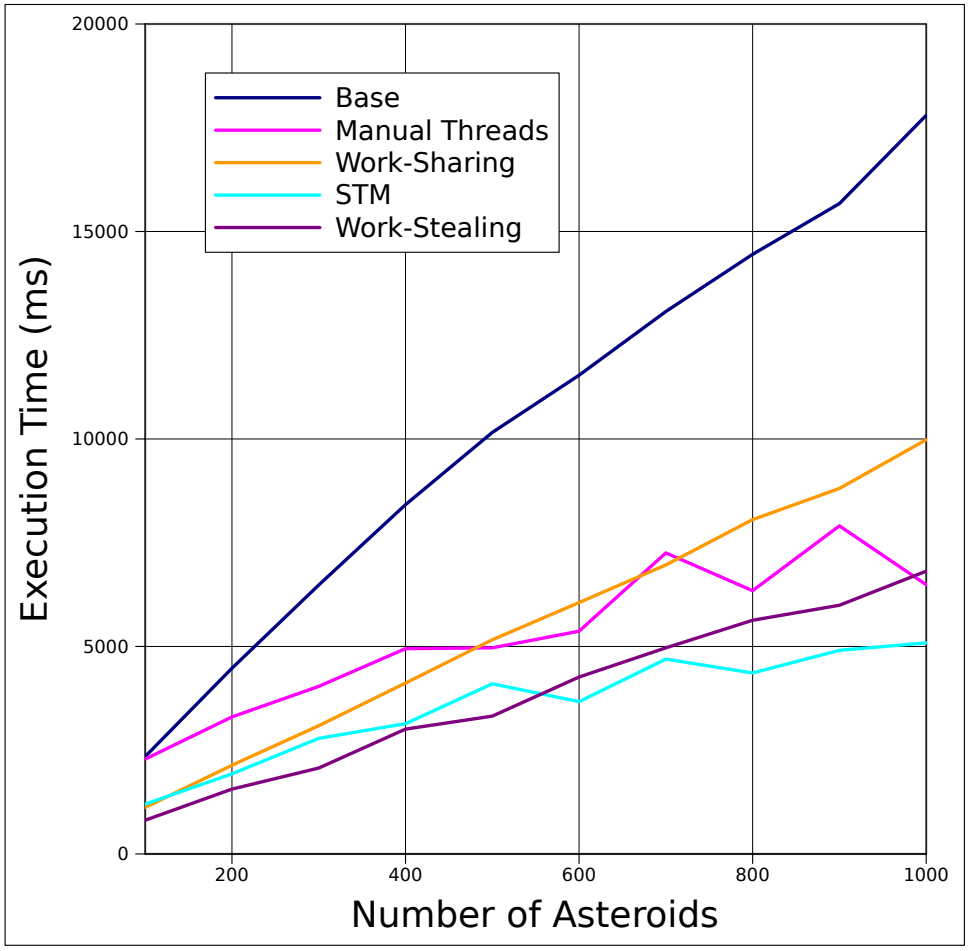
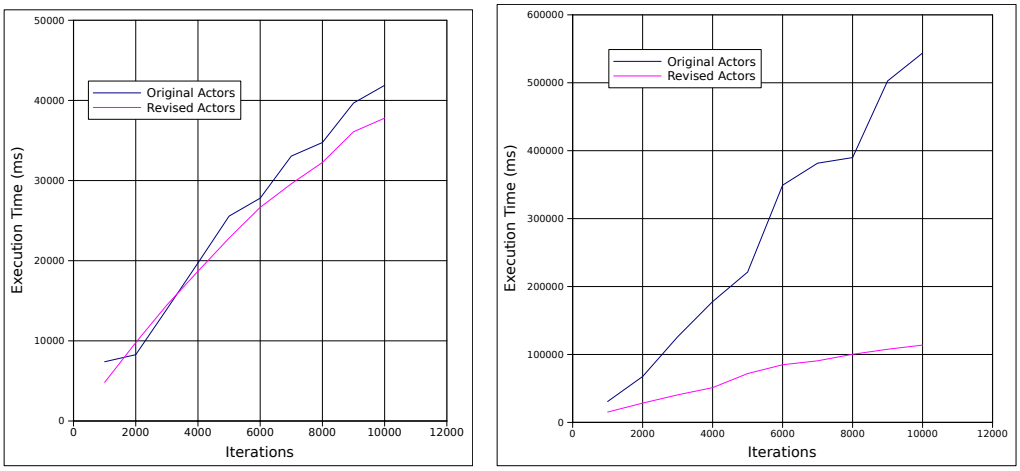


Figure 3.11: Varying numbers of asteroids over 10,000 iterations

lision detector are vital parts of the program. Here, we begin to see how the actor model might break down when applied to a game. The renderer needs to know about the position and shape of each game object in order to draw it. Similarly, the collision detector needs to know the location and outline of each object in order to collide them properly. In a shared-memory model, this is straight-forward: the renderer and the collision detector both have access to the list of game objects, and can directly obtain the information they require. In the Actor model, this must be ac-

complished by the collision dectector and the renderer sending a message to the game objects requesting the information, and the game objects must then reply and send the information back. This must happen every tick for the collision detector, and every displayed frame (30 per second) for the renderer. Just how much overhead does this incur?



(a) Execution times for 100 Asteroids (b) Execution times for 1000 Asteroids

Figure 3.12: Revised Actors charted against original actors, for 100 and 1000 asteroids

I re-wrote the Actor version of the case study to use a hybrid Actor/Shared-Memory approach. Individual game entities were still actors, communicated with the game controller and each other via the message-passing system. The renderer and collision dectector, however, operated in a sequential shared-memory environment where they could address each game entity directly. It should be noted that this is a violation of the Actor Model, but due to the fact this is written in Java such restrictions cannot actually be enforced without fundamental alterations to the JVM. The previous experiment was run with the new code, and produced Figures 3.12 and 3.13.

For 100 asteroids (Figure 3.12(a)), we see very little initial difference

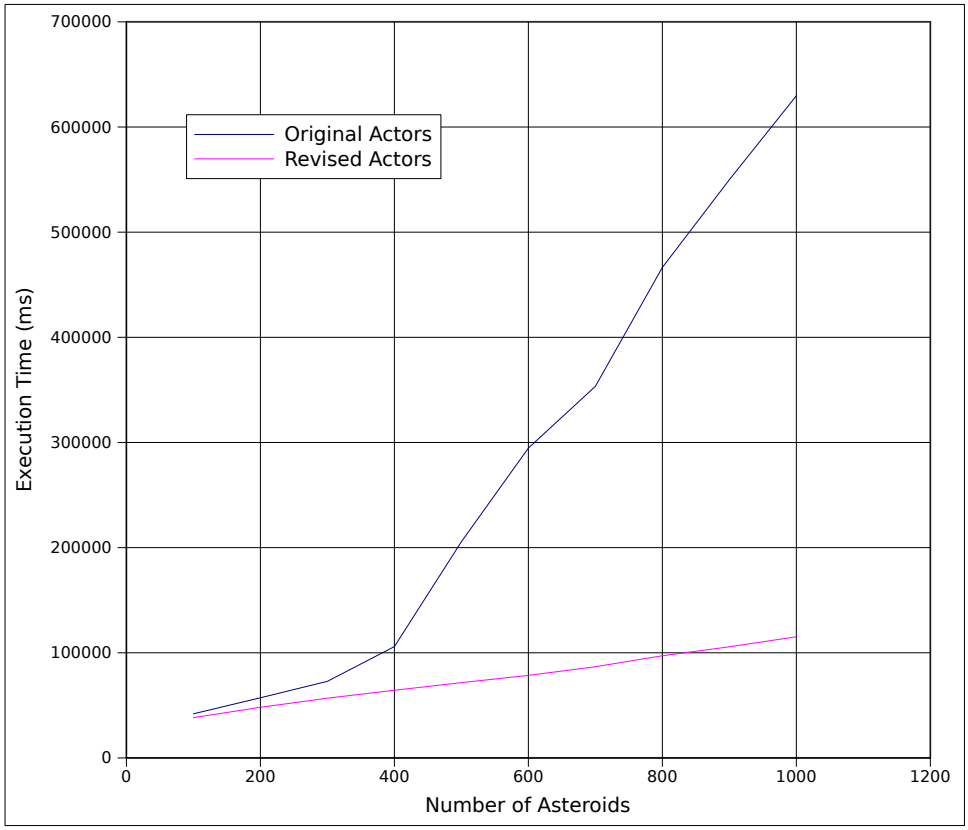


Figure 3.13: Revised Actors charted against original Actors for varying numbers of asteroids

between the two approaches as the number of iterations increases. When the number of asteroids is increased to 1000 (Figure 3.12(b)), however, we begin to see a very stark difference between the two versions of the code. Both increase roughly linearly over the number of iterations, but the revised code has a **much** shallower gradient. As before, we then fixed the number of iterations to 10,000 and varied the number of asteroids; this resulted in Figure 3.13. As the figure shows, the revised version of the actor code is significantly more scalable than the original version. Clearly the overheads incurred by game objects sending update messages every tick to the collision detector is not insubstantial. However, we also note that

despite the improved performance, the revised actor code is *still* outpeformed by the entirely sequential code. Actors, it seems, are not a good solution to this problem.

With actors no longer a viable choice for this program, what can we say about the other approaches?

3.5.1 Manual Threading vs Thread Pools

In all cases, the Work-Stealing thread-pool out-performs both Manual Threading and the Work-Sharing thread-pools. It doesn't have the same thread-creation overhead as Manual Threading, and because of its ability to re-distribute jobs amongst the threads in the pool, it doesn't run into the a problem that Work-Sharing can have of one thread taking too long to finish its jobs while the other threads sit idle with nothing to do. Interestingly, we can see in Figure 3.11 that the performance of Manual Threading actually begins to exceed that of the Work-Sharing approach as the number of asteroids increases. To see why this is, we need to look at the mechanism that underpins how work is allocated in both approaches.

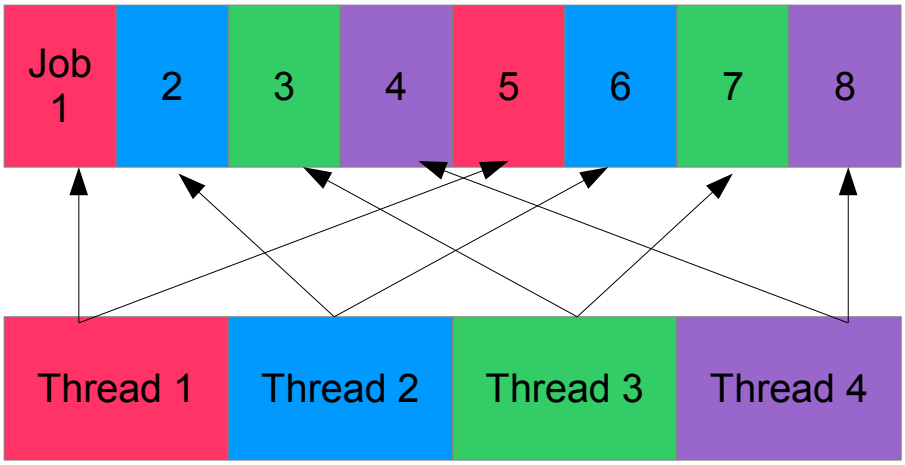


Figure 3.14: Job allocation in a work-sharing queue

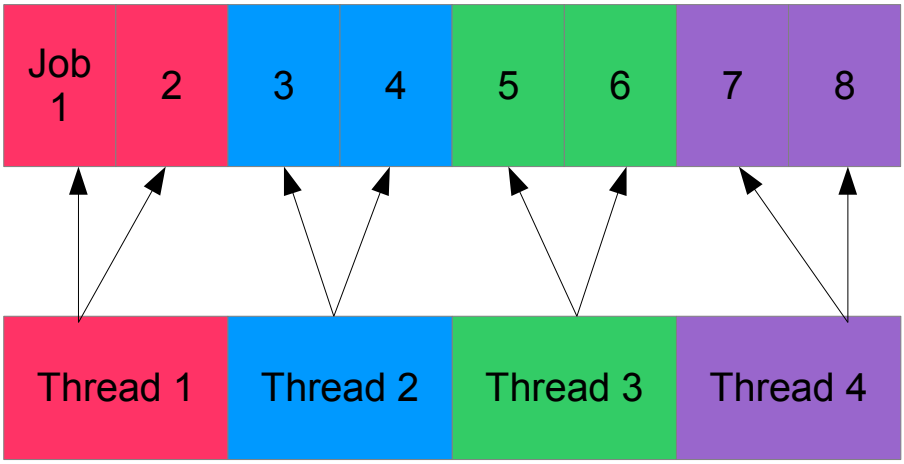


Figure 3.15: Job allocation under manual threading

For the Work-Sharing approach, jobs are taken off the queue by the threads in the pool as needed. If these jobs were to all take the same amount of time no matter which thread takes them, we would end up with a job allocation much like that shown in Figure 3.14, meaning that each thread acquires every fourth job in the job queue. For manual threading however, this allocation is entirely different. While there are still the same number of threads, the allocation is done by assigning continuous blocks of the jobs to each thread, as shown in Figure 3.15. So how does the difference in allocation then affect the performance?

First, we need to look at how objects are added into the game, be it from an asteroid splitting, or a bullet being fired. In this implementation, the game objects are stored in a list, and new objects are added to the end of that list. Bullets, both being created often and having a short life-span, are typically located towards the end of that list. In the allocation method used by Manual Threading, this means that *for large numbers of objects in the list*, the majority of the bullets are being handled by a **single thread**. Collision detection, which only occurs between Bullets and Asteroids in these experiments, is thus being processed by one thread—avoiding any

contention issues over locks when updating the state of an object.

3.5.2 STM vs Synchronized

This case study shows that for large numbers of game objects, STM offers a performance increase over the use of the `synchronized` keyword. This is hardly surprising: for large numbers of objects, the likelihood of locking conflicts is higher when performing collision detection, and STM is designed to be more efficient than using locks when contention is high. However, we can also see from Figure 3.9, for small numbers of objects—where the chance of contention is lower—the overhead of performing every collision as a transaction is higher than the overhead of using locks to protect the collisions. While STM is clearly not the best choice for small numbers of game objects, as the graph in figure 3.11 demonstrates, it is the *most scalable* approach, having the smallest performance decreases by adding more objects. This scalability is one of the hallmarks of STM.

3.5.3 Evaluation

At first glance, the actor model seemed like it would suit this case study in terms of modelling the interaction between game objects. Certainly from a design stand-point, it makes sense to consider each game entity as an actor in the system. And it does, in a way—but there are problems when it comes to collision detection and rendering. The lack of shared state means that these operations quickly become cumbersome, and require far more overhead than in a shared-memory system. Sending a game object to the renderer, for example, requires making a clone of the polygon that makes up the object on screen, and sending that entire object—not just a reference to it. Coming from a mostly OOP background, this idea that you're not allowed to share state between actors was a totally different approach to what I was used to.

In terms of ease of implementation, the thread-pool approaches (Work-Sharing and Work-Stealing) were definitely in the lead. The pools themselves are easy to use, and using the simplest case of "one game object per job" made writing the Jobs fairly straightforward too. This extended well for the Manual Threading implementation, where the Jobs were simply adjusted to take a range of objects instead of just one. There were, as mentioned above, some concerns about whether to create threads twice per iteration, or to use a CyclicBarrier to synchronize the update phases that stopped Manual Threading being as easy to implement as the thread-pool versions.

Chapter 4

Case Study: Gas Simulation

The second case study is a simulation of molecular gas diffusion in two dimensions, using physical laws of diffusion. The focus of this case study is investigating how job and task distribution affects the performance of a concurrent program.

4.1 Description

AtmosSim is my implementation of a cell-based physical simulator for gas systems. Each cell is either floor, walls, or a vacuum. Floor cells contain some number of moles of various gases—which determines their partial pressures–and diffuse their contents to surrounding cells as indicated by pressure differences. All gasses flow from high partial pressure to low partial pressure. Wall cells are impermeable and contain no gases. Vacuum cells are considered to always be at 0 pressure, and remove any gases entering them from the simulation (consider a hull-breach in a space-ship). Floor cells may also contain injectors and filters. Injectors inject a small amount of a given gas into the cell every update tick, while filters perform the reverse. Humidity is also tracked; any cell that contains gaseous water has a non-zero humidity. If the cell reaches saturation, water begins to condense in that cell.

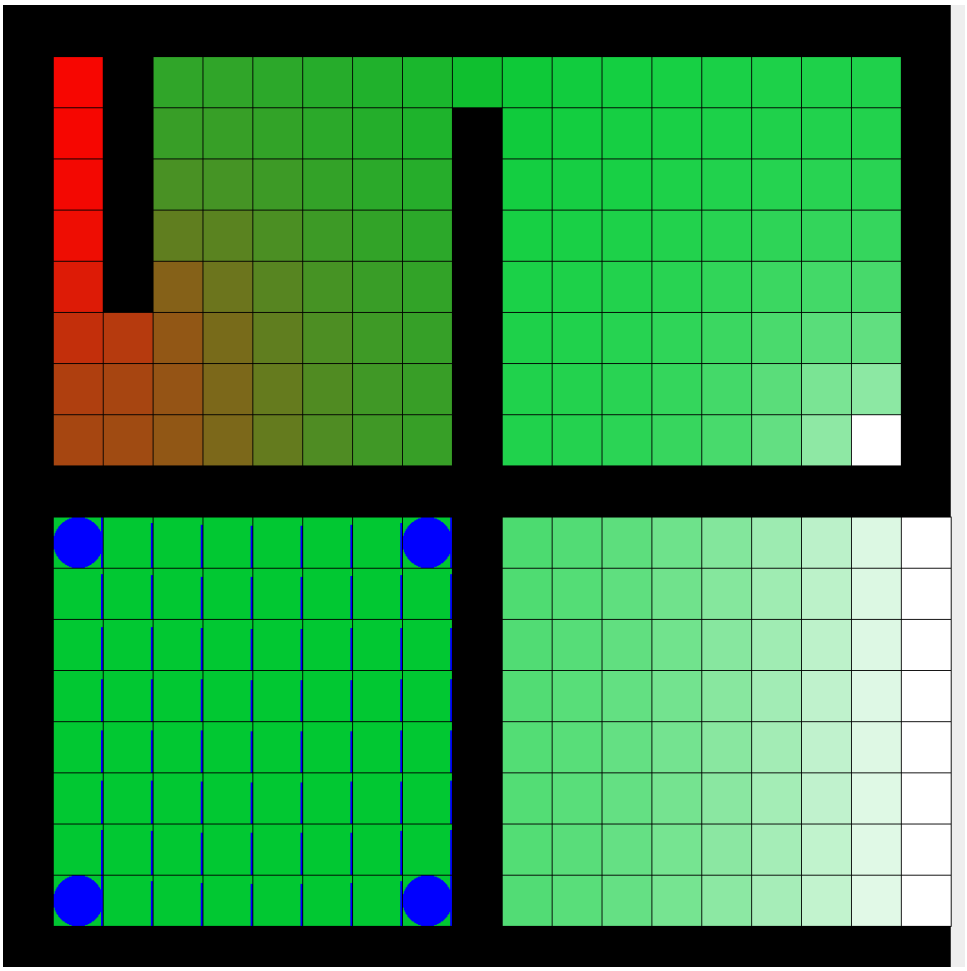


Figure 4.1: A gas simulation, showing carbon dioxide (red) diffusing through a Nitrogen-Oxygen mix (green).

Gas molecules diffuse between cells according to Fick's first law of diffusion (one-dimension) [12, 16]:

$$J = -D\frac{\partial \phi}{\partial x}$$

where:

J is the diffusion flux, in $\frac{mol}{m^2 \cdot s}$

D is the diffusivity of the gas, in $\frac{m^2}{s}$

ϕ is the concentration, in $\frac{mol}{m^3}$
x is the distance, in m

For higher dimensions (AtmosSim uses 2), the equation becomes:

$$J = -D\nabla\phi$$

Where ∇ is the gradient operator.

This can be simplified by assuming that all interactions use the instantaneous difference between adjacent cells, the results of interactions are constant for a fixed time interval, and at the end of this time interval, the calculations are performed again. Thus, the equation becomes the following psuedo-code:

```
for each cell:
  for each neighbouring cell:
    for each gas in that cell:
      diff = amount_in_neighbour - amount_in_this
      if diff < 0:
        J = -(gas.diffusivity) * diff
        amount_transferred = J * delta_time
        amount_transferred = flowOut(amount_transferred)
        neighbour.flowIn(amount_transferred)
```

Note that gas is only transferred if the central cell contains **more** of that gas than the neighbouring cell that is being inspected. This stops the algorithm from transferring the gas *again* when it reaches that neighbouring cell in the outer-loop; as the contents of each cell are not actually updated until *after* the changes to every cell have been calculated, the neighbouring cell will still have less gas than the central cell.

Why delay the updates to the contents of each cell? In each iteration of the simulation, the equation above must be applied to each cell *simultaneously*. This means that each cell must see its neighbours **exactly as they**

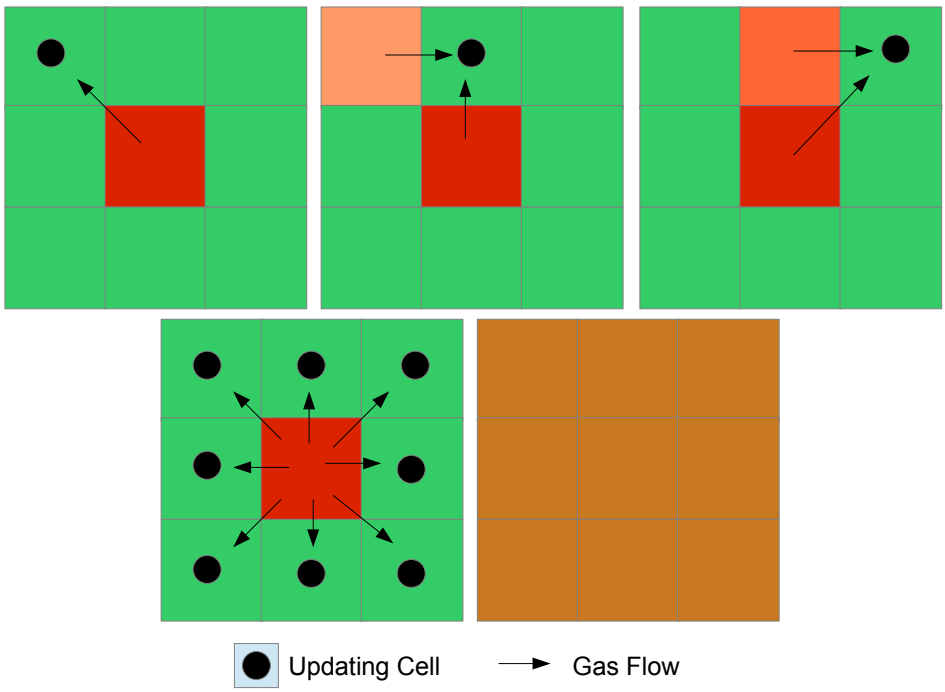


Figure 4.2: Updating cells. Top: immediate updating. Bottom: delayed updating.

were at the start of the iteration. If this condition isn't met, some rather non-physical results can occur. Figure 4.2 displays one possible scenario resulting from the violation of this condition—note how the gas flow is calculated incorrectly in the top row, which does **not** delay cell updates until every cell has been calculated. Note that this isn't an issue created by using concurrency! Even a sequential program will behave in a physically inconsistent way if the condition is violated—however, unlike a concurrent version of the algorithm, it will misbehave in a predictable and repeatable way, as calculations will always proceed in the same order.

4.2 Making it Concurrent

This program is a prime candidate for concurrent computation. There is a clear base unit of computation, and provided the *current* state is kept constant during the computation, there seems to be little in the way of issues related to having shared memory—we only need to control access to the `flowIn` and `flowOut` methods; all other updates to the state of a cell are performed by the cell itself (strictly speaking, only access to `flowIn` needs to be controlled, as `flowOut` is only ever called by the cell that owns it). In the non-STM versions of this code, access is controlled by use the built-in Java `synchronized` keyword; for the STM-based implementation, the annotation `@Atomic` is used to indicate to the STM engine that these methods are to be executed atomically.

But, naturally, it's not quite that simple. While the individual cell seems to be the perfect unit of concurrency in this program, the overheads required by spawning a thread to deal with each cell individually simply cannot be overlooked. So this case study begins to examine a more subtle problem—granularity of job division. The answer is not so simple, as the division of tasks in a concurrent program are highly dependent on the nature of the tasks. Therefore this case study is handled a little bit differently to the others. While the usual structure of testing each implementation against each other will be maintained, within each implementation, the division of jobs is varied as well. These results are presented below.

4.2.1 Manual Threading

For this concurrent approach, a number of threads (ranging from $NUM\_PROCESSORS$ to NUM_CELLS) are created at the beginning of each update step and tasked with performing the update computation on some subset of the cells, determined by dividing the entities amongst the threads so that each thread has an equal share of the total cells. Cells are allocated in blocks, starting from the first cell in the grid and counting along rows until the

quota for each thread has been met.

For the experiments in this chapter, the number of processors was 4.

4.2.2 Work-Sharing and Work-Stealing Queues

For these concurrent approaches, the queues accept one type of job; a Cell Update Job. This job updates one or more cells as detailed above.

The division of job sizes ranges from one cell per job, to $\frac{1}{NUM\_PROCESSORS} \times 100\%$ of the cells per job, following the same allocation method of cells as above. Also as above, the experiments used 4 processors.

4.2.3 Actors

Unlike the above approaches, the Actor-based implementation of this case-study requires a little more thought, as *Actors cannot share state*. This presents an issue as the grid of cells cannot be shared between all active Actors, making the computation mechanism a little bit more complex. Two different ways of solving this problem were explored.

One Actor per Cell

The most obvious solution is to have each cell represented by one actor, which then contains all of the state required for that cell. This also eliminates any access-control issues, as actors are, within themselves, entirely sequential. Care still needs to be taken that cells aren't updated before the current computation cycle is complete, but again, actors provide a very good mechanism for controlling this in the form of messages.

Because of the lack of a shared state, an additional step must be performed in the calculation: actors must query their neighbours to obtain the values of the surrounding cells. Not only does this require cells to maintain additional state about who their neighbours are, it adds additional processing time to the simulation. Unlike the shared-state models

above, a cell cannot simply just access its neighbours directly—messages must be sent, recieved, acted on, and the results sent back to the original sender. Having one actor per cell means that each cell is both sending eight messages and responding to eight requests (obviously edge and corner cells have less to send and respond to). Once the calcuations have been performed, there is another round of message-sending to inform cells of updates to their contents based on the computation. Again, this requires a message to be sent, acted on, and then replied to—flowing gas out of a cell returns a result, which is used to flow gas in to the receiving cell. Clearly this approach will generate far more overhead than necessary: A single (non-edge, non-corner) cell, in a single iteration, will need to send 16 messages, and will receive and act on 16 further messages. For the grid used in Figure 4.1, this amounts to 80,000 messages **per iteration**, and the number of iterations is typically very high (several hundred, if not thousand, per second). Even if the system can process these messages quickly, that seems like quite a high number of messages to be passing around every iteration.

Each Actor Allocated a Block of Cells

One solution is to have each actor be allocated more than one cell—which immediately raises the question: *how do you split the grid up?* The approach used for the shared-state models (dividing the grid by index ranges) splits the grid up along rows—this is not an optimal split for this case. Consider Figure 4.3; in the worst case, an actor is responsible for a row somewhere in the middle of the grid. To update any of its cells, it requires the information from the surrounding cells—seven of which belong to actors other than itself. While it can (and does) receive all of the cells from an adjacent actor in one message, this looks a lot like the previous setup.

Instead, what if the grid is subdivided into blocks, as per Figure 4.4? While each actor is still required to query the other actors regarding cells along the edge of a block, the actor requires much *less* information in order

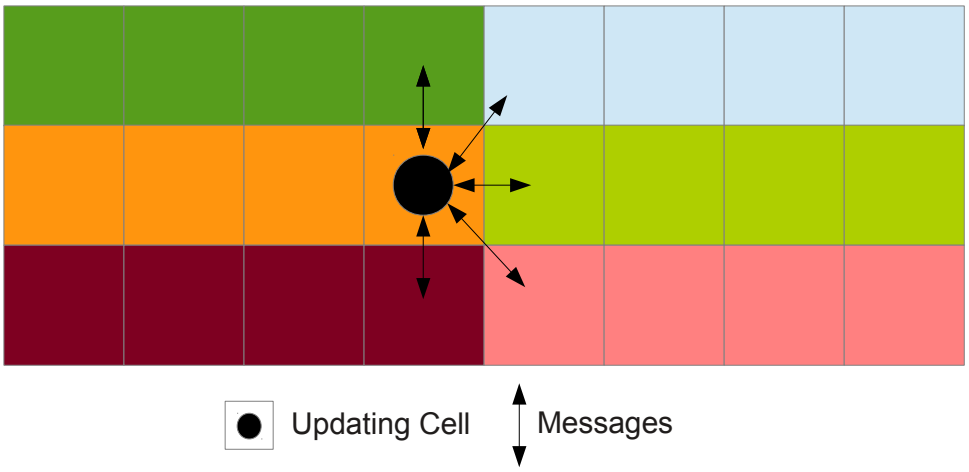


Figure 4.3: Update messages required to update a cell in a badly allocated grid

to update the cells it is responsible for, and could even start performing the calcuations for the non-edge cells while it waits for a response from its neighbours, though this requires constant monitoring of the message-queue and was **not** implemented.

4.3 Results

Figure 4.5 shows the running times of the simulation over an increasing number of iterations, averaged over 100 runs at each iteration value. These results were obtained using four threads (on a four-core machine) in all cases except the unthreaded version. Actors#1 refers to the "One Cell per Actor" implementation, while Actors#2 refers to the block-based approach.

As shown in the graph, the Work-Sharing Queue and Work-Sharing with STM are the fastest performing implementations at any number of iterations; they are also the most scalable in terms of iterations performed,

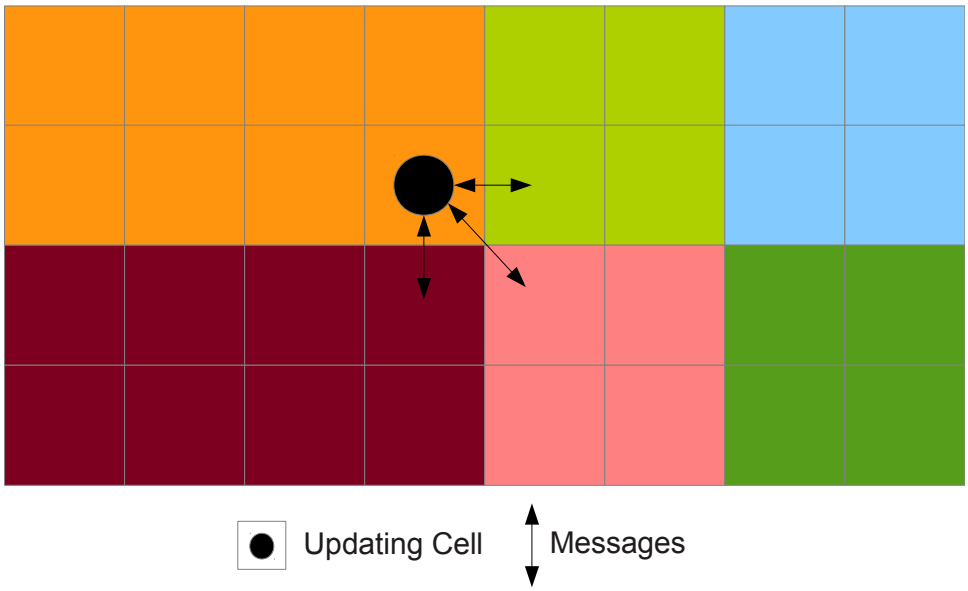


Figure 4.4: Update messages required to update cells, with an improved allocation of cells to actors

as shown by their slower increase in time taken as the number of iterations increases. The first of the actors implementation, as somewhat expected, is the slowest and least scalable in terms of number of iterations. Notably, the second actor implementation is *outperformed by the unthreaded version*. Clearly the overheads of using actors to solve this problem greatly outweigh the performance benefits of using concurrency.

But these results don't take into account the size of the jobs being processed, simply the number of iterations over which the simulation is run. How does the performance of the program change with respect to the size of the jobs performed by each thread/actor?

The first thing to note is that adding more jobs to the Manual Threading implementation also increases the number of threads being used. As shown in Figure 4.6, splitting the jobs by creating more threads is eminently unscalable, as it begins to dominate the graph. The Actor imple-

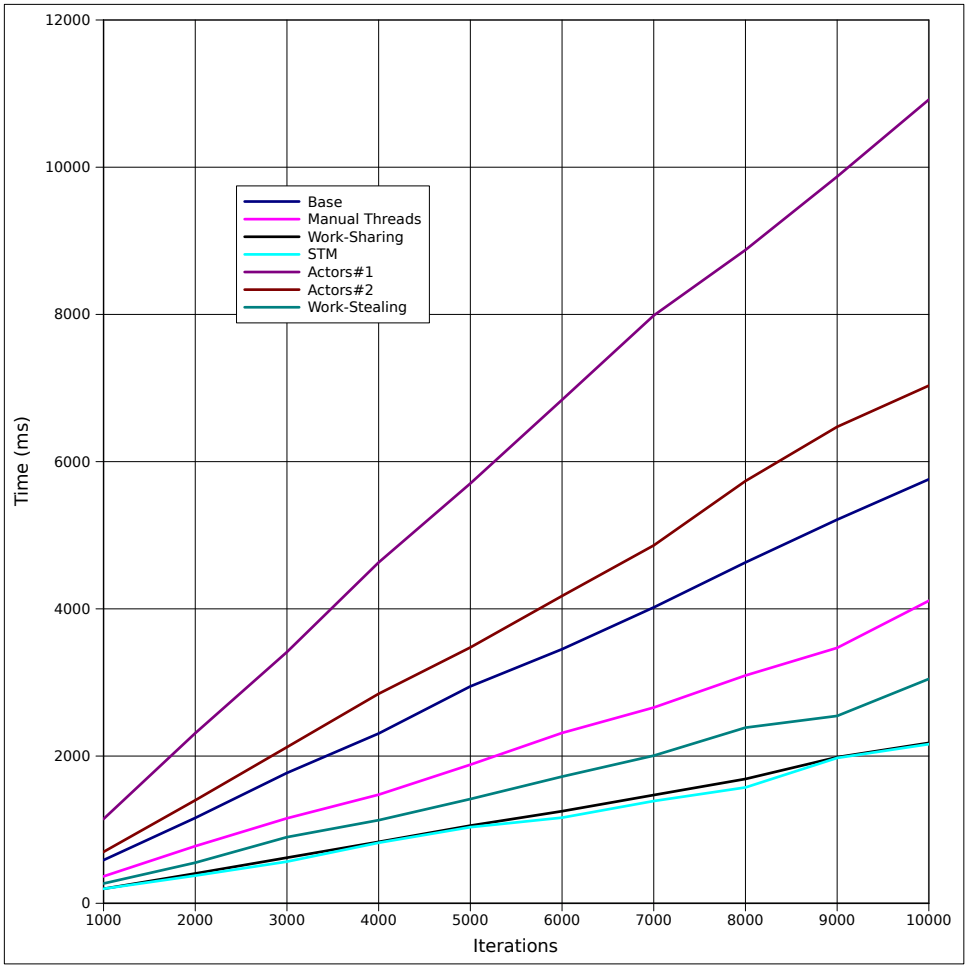


Figure 4.5: Results from trials with increasing numbers of iterations

mentation suffers a similar fate, as additional jobs mean exponentially more messages being sent between actors, to the point where it actually became impossible to run the simulation with more than 1024 actors. The reason behind this is unclear, but the program would not run to completion with higher numbers of actors in the system.

Figure 4.7 show the other two approaches in zoomed-in detail. As one might expect, the Work-Stealing queue is faster when there are more jobs

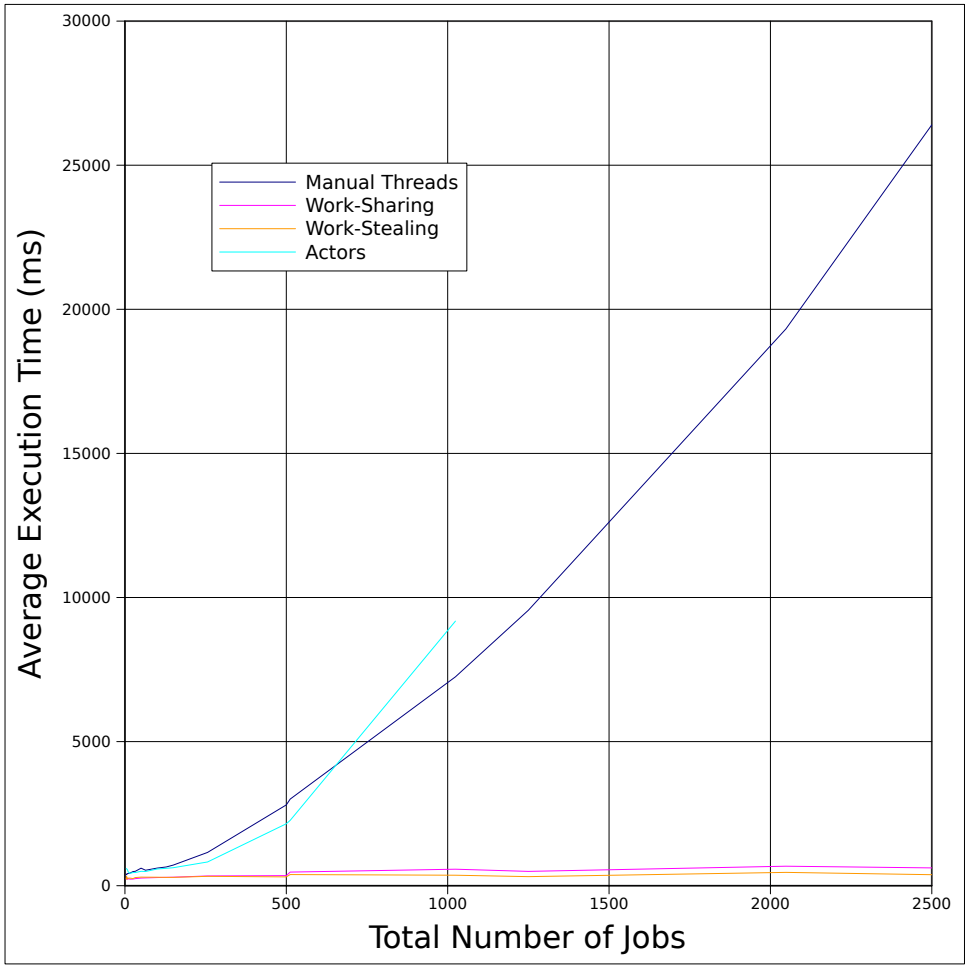


Figure 4.6: Results from trials with differing job sizes, for all approaches

available to actually be stolen. Note the sharp drops in both approaches at certain points—these correspond to the number of jobs being a factor of the total number of cells, meaning that the splits of cells between jobs is perfectly equal.

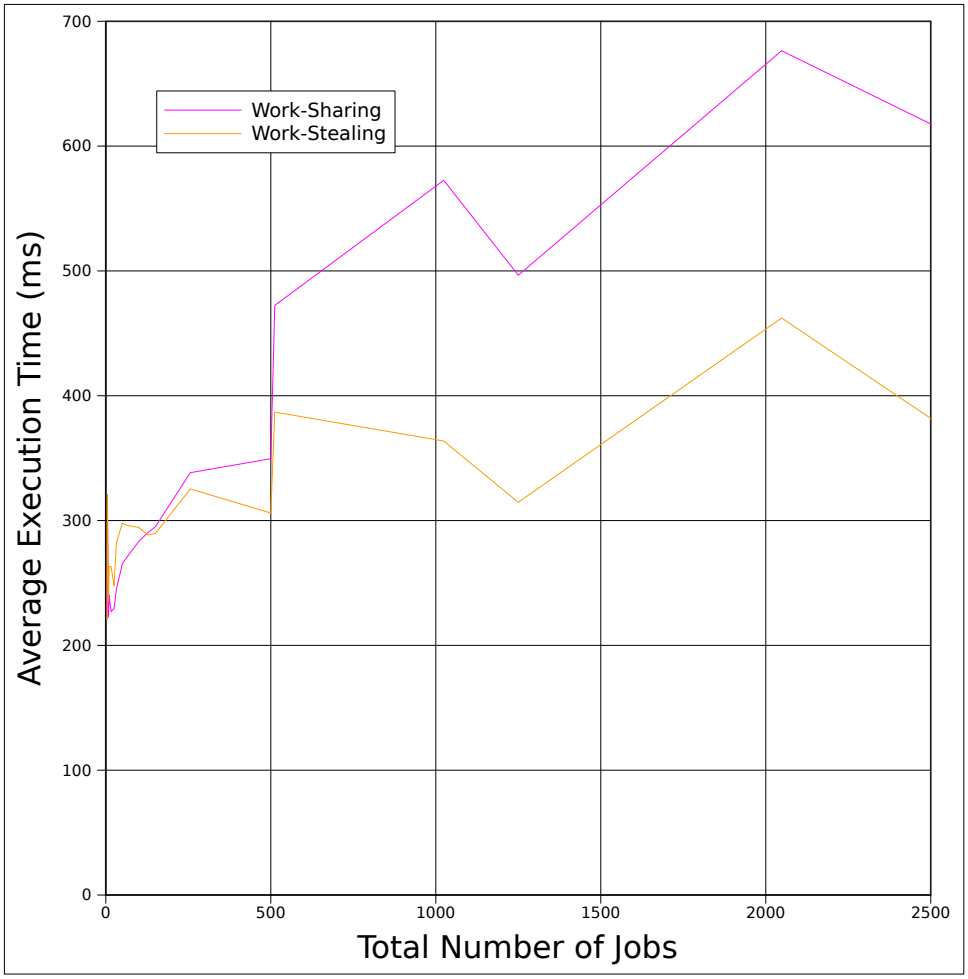


Figure 4.7: Results from trials with differing job sizes for the thread-pool approaches

4.4 Discussion

Much like the previous case study presented in Chapter 3, the actor-based implementation(s) of this case study peform rather badly compared to the other implementations. While this might be expected of the first actor implemention (with one actor per cell) due to the high volume of messages that need to be sent between actors, it is not so clear that this should be

the case for the second implementation. However, we note that execution time scales quite poorly as the number of jobs increases—once again, the overhead of running the entire actor system seems to cause the program to perform worse than the unthreaded implementation of the case study.

When limited to one job per thread, as per the first set of results, we see that Work-Sharing and Work-Sharing+STM are the best performing implementations. Manual Threading, with the overheads caused by spawning threads every iteration, falls between the unthreaded implemention and the thread-pool implementations. As there are only four threads working at once, contention for locks is relatively low, especially considering there are very few cells that need to be accessed by more than one thread: thus it stands to reason that STM offers no improvement over the work-sharing queue.

The Work-Stealing queue falls between Manual Threading and Work-Sharing. As there is only one job per thread in this first set of experiments, there isn't any opportunity for worker threads to *steal* jobs from another thread. This gives us a baseline for the overhead that the work-stealing queue generates when it is running in a very similar way to the work-sharing queue.

The results for execution over different jobs sizes (Figures 4.6 and 4.7) are somewhat more interesting. As expected, Manual Threading scales poorly as the number of jobs increases, due to the increased overhead of spawning more and more threads. Likewise, Actors scale poorly due to the increased number of messages that are required to process an update. We can see that the overhead of using the Work-Stealing queue means that it is out-performed by the Work-Sharing queue at low numbers of jobs. As the number of jobs increases we see that it soon overtakes the Work-Sharing queue as the best performing implementation, as jobs are now actually available to be "stolen" by the worker threads. Of note, we can see very obvious and sharp increases in performance at several different quantities of jobs submitted to the queue—these values are perfectly

divisible by four, the number of threads running on the system. This is an important result, as it tells us that an exactly equal division of work is better than an inequal division of work when it comes to making a program concurrent.

4.4.1 Evaluation

Again, the design of the system appears to suit the Actor model quite well; isolated cells that only care about maintaining their own internal state, happy to simply ask their neighbouring cells for data when required. Unfortunately, that is required quite often, as we discovered with the Asteroids case study, having to pass a large number of messages between actors all the time is not the most beneficial for performance.

In terms of implementation, the Actor model was much easier to apply for the one-cell case. Extending it to allow for the multiple-cell case was a bit trickier, requiring very careful thought about the best way to send cell data between actors. However, due to this, the problem of job division was **most** apparent when designing the Actor implementation of the case study. In order to maximise the content of a message (thus reducing the number of messages), it made sense to try and minimise the number of neighbours a block of cells had to query. Unlike the shared-state version, it was clear that just dividing the grid up by counting along rows was not a good solution for this.

That said, the Manual Threading and thread-pool variants of the code were, yet again, the least complicated to implement. Care still had to be taken, of course, that shared data structures were updated safely, but the fact that shared data structures were permissible made the entire process a lot easier to reason about. Splitting up the grid for jobs wasn't nearly as complex an issue, as accessing the neighbouring cells is as simple as accessing the cells a thread is responsible for updating.

Chapter 5

Case Study: Image Processing Server

This case study is a simple image processing program that takes a folder of images and performs a series of processing operations upon them. The focus of this case study is how the different concurrency approaches deal with blocking on I/O.

5.1 Description

Image processing is a large part of many computer science fields, from Computer Vision, to Artificial Intelligence [24, 19]. The program developed in this case study is an example of the sort of program one might write while working in such a field. It takes, as input, an entire folder of images. It then applies a series of operations to those images—this case study has two different sets of operations:

Mostly File I/O This sequence of operations is relatively simple and very fast to calculate. It begins by scaling the image down by 50% using a quick operation that simply takes the average value of each 2x2 block of pixels, and uses that to create one pixel in the target image. This

Figure 5.1: A test image (upper left), and three output images of different convolution filters (clockwise: sharpen, emboss, edge-detection)

is then followed by two convolution operations that perform image sharpening, and edge detection.

Mostly Computation This seqeuence of operations is almost the same as the first one, but instead of scaling the image down, it scales it up to double the size, using a non-trival cubic B-Spline interpolation algorithm [2], given below.

$$F'(u,w) = \sum_{m=-1}^{2} \sum_{n=-1}^{2} F(i+m, j+n) R(m-dx) R(dy-n)$$

$$R(x) = \frac{1}{6}\left[P(x+2)^3 - 4P(x+1)^3 + 6P(x)^3 - 4P(x-1)^3\right]$$

$$P(x) = \begin{cases} x & \text{when } x > 0 \\ 0 & \text{when } x \leq 0 \end{cases}$$

Where:

$F(i,j)$ is the value of the pixel at (i,j) in the original image
$F'(u,w)$ is the transformed pixel at (u,w) in the transformed image
u and i are related via $u = \lfloor i \cdot \frac{width'}{width} \rfloor$
w and j are related via $w = \lfloor j \cdot \frac{width'}{width} \rfloor$
dx is the non-integer remainder of $u - i$
dy is the non-integer remainder of $w - j$
$R(x)$ is the cubic scaling function

Since the image is being doubled in size, every pixel in the original image becomes four pixels in the target image. So the pixel $(0,0)$ in the source image would be interpolated to $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$ in the destination image. Notice that this means that there is only a 1-to-1 mapping from each pixel in the source image to *one in every four pixel* in the destination image. This can be readily shown by the relationship between source and destination pixels given above. Specifically, we can only directly transfer the value of $(0,0)$ from source to destination. The pixels located at $(0,1)$ and $(1,0)$ in the source can be similarly directly transferred to $(0,2)$ and $(2,0)$ in the destination ($2 = \lfloor 1 \cdot 2 \rfloor$). So what about the pixels at $(0,1)$, $(1,0)$, and $(1,1)$ in the destination? The relationship for pixel locations would seem imply that $(0,1)$ in the *destination* needs to be taken from $(0,0.5)$ in the source—halfway into a pixel. This is where the dx and dy terms of the above equation come into play: for a scaling factor of 2, the

value of dx is 0 when the value of u maps to an integer value of i, and 0.5 when it does not. dy is calculated similar fashion, using w and j.

Thus, the equation to calculate the pixel at $(0, 1)$ in the destination image would be:

$$F'(0,1) = \sum_{m=-1}^{2} \sum_{n=-1}^{2} F(0+m, 1+n)R(m-0)R(0.5-n)$$

5.2 Making it Concurrent

The purpose of this case study is to investigate the advatanges to executing multiple tasks in parallel. To this end, the concurrency in this case study was introduced at a higher level of the program than in previous case studies. Instead of using concurrency to calculate the image filters over multiple pixels within an image simultaineously, concurrency was used to apply the sequence of image filters to multiple images at once.

5.2.1 Implementation

For the non-concurrent implementation, the program iterates over every image file in a given input folder and applies a set of image-processing filters to the images. The resulting image is then saved into a specified output folder.

The Manual Threading implementation creates a new thread to handle every new image. This was a deliberate design choice, as the case-study is supposed to represent a server of sorts, and creating a new thread to handle every incoming request is a popular method of writing low-traffic servers.

The thread-pool implementations (Work-Sharing and Work-Stealing) treat every individual image as a seperate job, and submit them to the pool.

The Actor-based implementation uses an ImageProcessingActor to process images, with one actor being created for each image. The images are then given to the actors via message-passing, and the system waits for each actor to signal that they have completed their task.

STM was not used in this case study as the lack of any communication or shared information between jobs means that any form of concurrent access control (ie: STM or `synchronized`) was not required.

5.3 Results

Each implementation was executed 100 times on an input folder consisting of 37 512x512 pixel images. The total execution time and the total I/O times for each run were recorded. For the first experiment, the image processing operations used on the images were:

- Scale the image down by half
- Apply a 3x3 Sharpen convolution filter
- Apply a 3x3 Edge Detection

The execution time of these filters is very short, allowing the majority of the total execution time of the program to be caused by waiting for File I/O.

For the second experiment, the image processing operations were:

- Scale the image up to twice its original size
- Apply a 3x3 Sharpen convolution filter
- Apply a 3x3 Edge Detection

The first operation is a lot more computationally intensive than scaling an image down, allowing I/O to form a very small part of the execution times for this experiment.

Finally, the second experiment was repeated with a more memory-efficient version of the code after it became clear that excessive memory usage was causing one of the concurrency techniques to perform quite poorly.

Model	**I/O Time (ms)**	**Total Time (ms)**
Non-Concurrent	17039.10	20966.46
Manual Threads	125960.69	5242.06
Work-Sharing	17807.96	5570.01
Work-Stealing	16836.54	5460.01
Actors	49164.73	5501.97

Table 5.1: Execution times for the first experiment

Model	**I/O Time (ms)**	**Total Time (ms)**
Non-Concurrent	111299.16	1924368.09
Manual Threads	716871.38	3796964.24
Work-Sharing	112142.64	487058.66
Work-Stealing	113333.18	549864.47
Actors	334499.67	527122.49

Table 5.2: Execution times for the second experiment

The first set of experiments dealt with a set of CPU-light operations—Table 5.1 gives ≈82% of the execution time being spent on I/O for the non-concurrent version of the program ($\frac{\text{I/O time}}{\text{Total Time}}$). Note that I/O times for the other versions of the program are totals across all threads, and not an indication of real-time (as evidenced by the totals being much larger than the actual total execution times). From the table, it is evident that parallelising I/O operations has a marked performance increase for a program. The largest gain was using the manual threading approach to create a new thread to process each image; but how does this hold up when the bulk of the operation is CPU-bound?

Model	I/O Time (ms)	Total Time (ms)
Non-Concurrent	115693.15	2029835.56
Manual Threads	726131.80	501719.88
Work-Sharing	97068.08	530300.40
Work-Stealing	97032.52	531253.46
Actors	281239.26	567913.54

Table 5.3: Execution times for the repeated second experiment, with memory optimisations

The second set of experiments investigated this by using the up-scaling algorithm described earlier. Table 5.2 demonstrates that I/O forms ≈6% of the execution time for this set of operations (again, calculated using $\frac{\text{I/O time}}{\text{Total Time}}$ for the non-concurrent case). Note how badly the manual threading approach works in this case, while the other concurrency approaches maintain a similar performance boost over the unthreaded version. These results raise an interesting question: why does the manual threading approach work *slower* than the non-concurrent version?

The problem was the implementation of the upscaling algorithm: it was far too memory-intensive. More than 50% of the heap space was being used by `java.awt.Color` objects and multi-dimensional `double` arrays—to ensure as much accuracy as possible when interpolating the pixels of the target image. Refactoring the code (as discussed below) to eliminate the need for using any of these objects gives the results present in Table 5.3. Note that the other concurrent approaches remain at similar execution times, but the manual threading approach is **greatly** improved.

5.3.1 Refactoring

The original implementation of the scaling algorithm uses a double array to store the RGB values for a pixel that is returned from the function $F(u, w)$. Refactoring the code to be more memory efficient started by re-

placing these arrays with a single integer. Accuracy *is* lost in this process, but the end result of the original process is a 32-bit representation of the pixel, so the accuracy gained by using doubles to represent a pixel's individual RGB values is not required. In this way, the representation of a pixel in memory went from an array of 3 64-bit values to a single 32-bit value. Pixel values are packed into the integer like so: 0xRRGGBBAA, where each RGBA value is a byte long, allowing for a range from 0-255. This mirrors the internal representation of the pixels in the class that stored the images, which further allowed the use of `Color` objects to be excised from the scaling code. Previously, these objects had been used to create the 32-bit integer representation of the pixel from the array of doubles. With the 32-bit representation already in hand, these objects became superfluous.

When refactoring was complete, the scaling algorithm did not use any non-primitive data types except for the images themselves.

5.4 Discussion

This case study demonstrates one of the approaches to concurrency that can offer the greatest increases in performance, and the greatest ease of implemention: parallel tasks with no absolutely no interaction. A lack of interaction between tasks means that code doesn't need to be "thread-safe"—there is no shared data to control access to. Each thread or actor is simply executing the exact same task with different input. A lack of communcation between tasks means that each task can simply be started and left to run; they require no synchronization, control, or any of the additional overhead that has been required in the previous case studies. However, that is not to say that this approach to concurrency is not without its own pitfalls. We begin the discussion by looking at the three different forms of task presented in this case study.

5.4.1 I/O-Heavy Operations

It is clear from the results presented above that the addition of concurrency to a program that involves a great deal of parallel file operations provided benefits. We have shown that adding concurrency over four-cores results in an almost four-fold reduction in execution time for this case study, with the best results coming from the manual threading implementation. Actors, for perhaps the first time in this thesis, also appear to be providing a significant performance boost. File I/O remains one of the slowest parts of many programs (other than networking, perhaps), as the majority of computers still use machanical hard-drives to store data. While non-blocking I/O (ie: asynchronous) was not investigated in this case-study, we can see that even blocking (synchronous) I/O calls can be made faster by executing them in parallel. For each implementation, the total amount of time spent on I/O was also recorded—note that this is not a measure of "real" time, so much as "effective CPU time", as the total I/ O times are distributed across multiple threads of execution operating in parallel. For manual threading, this number seems rather high: recall that in the manual threading implementation, one thread was spawned for *each image in the input folder*. This means that the manual threading implementation had far more threads executing at once than the others; 4 for each of the thread-pool implementations, and 16 (each core thread of the actor subsystem has 4 worker threads) for the actor implementation (which *also* has a significantly higher I/O time than the other implementations). This high total I/O time is caused by all of the threads effectively initiating their file I/O calls simultaineously, and then being swapped out for threads that have become unblocked (ie: finished their I/O). In the thread-pool implementations, **every** thread becomes blocked as soon as four jobs are being processed, leading to shorter over-all I/O times, but a slightly slower overall performance.

5.4.2 CPU-Heavy and Memory-Heavy Operations

Table 5.2 gives a startling set of results given the results of the previous experiment. We immediately notice that manual threading is performing very badly, giving a total execution time that is approaching *double* that of the unthreaded version. However, the other multi-threaded operations give a very similar performance to last time, again showing an almost four-fold reduction in execution time. The only real difference between the manual threading implementation and the other multi-threaded implementations is that the manual threading version spawns more threads—could the cause of this terrible performance be simply due to having more threads running at once?

Using the JVisualVM tool (which is included as part of the Java Development Kit) to visualise the program in execution, we noticed that the manual threading implementation was using **far** more memory than the other implementations, mostly bound up in `java.awt.Color` objects and `double[][][]` objects. The culprit was the scaling algorithm used to scale the images to a larger resolution. In order to scale the image with as much accuraccy as possible, the algorithm was storing pixel data as a multi-dimensional array of doubles, and using the Color objects to construct new pixels of the appropriate value.

5.4.3 CPU-Heavy and Memory-Light Operations

Having isolated the cause of the slow-down in the manual threading implementation, the scaling algorithm was re-written to be calculated entirely using primitive integers and bit-shift operators. The resulting execution times, shown in Table 5.3, are mostly unchanged by any significant amount—except for the manual threading implementation, which we now note as being the fastest once more. A huge change in performance for very little in terms of code changes.

5.4.4 Evaluation

This case study highlights a key issue that crops up when writing concurrent code: memory usage. Care must be taken when executing large numbers of task in parallel that the machine (VM or physical) can handle the extra memory requirements. If a single method is written poorly and uses far more memory than it should, then running that method 40 times concurrently is going to cause a lot of stress on memory—especially when those objects are located on the thread's local stack, or if the tasks are so large that switching requires virtual memory to be paged to and from your hard-drive. Under these situations, any performance advantage you gain from writing concurrent code is likely to be totally lost in the extra time required for memory management.

In terms of implementation, parallel tasks that require no communication are among the easiest to understand and implement forms of concurrency. We're not looking to do anything more complicated than take advatange of having multiple cores by executing multiple copies of the same task, albeit with different input. In these terms, all of the various approaches were fairly easy to implement. Actors make a lot of sense for this kind of implementation as, unlike the previous case studies, this one doesn't require any nasty shared state that really seems to get in the way of making Actors work well. The messages are fairly minimal too, only requring two messages total, compared to the larger numbers required for the other case studies.

Chapter 6

Case Study: Concurrent HashMaps

This case study takes a deeper look at something that underpins the first two case studies: concurrent access to data structures. The structure we have chosen to look at in this case study is the Hash Map. To investigate the effects of different approaches to controlling concurrent access to a data structure, we have implemented different wrapper classes for Java's built-in HashMap.

6.1 Description

Concurrent use of a data structure is a vital element of concurrent programming. This case study aims to investigate some different means of controlling concurrent access to a shared data structure in a safe manner. We will be investigating the effects of different ratios of operations (read/write) on a shared data structure, and how the different implementations perform in response to this. We will also be investigating how this changes under different forms of read operation, from just using the fast `get` and `contains` operations, to the more time-intensive operations such as `values` and `keyset`.

To this end, we have implemented different approaches to making a HashMap concurrent.

6.2 Making it Concurrent

Unlike the previous case studies, this section will not focus on individual concurrency techniques, but rather on the different ways in which access to the HashMap was implemented.

Many of the implementations below are based on a well-known solution to controlling concurrent access to a shared data-structure: the Readers-Writers Problem [7]. In the Readers-Writers problem, threads/actors are divided into two groups:

Readers Any thread/actor that needs to perform an operation that cannot be performed while a Writer is operating on the data-structure, but can be done concurrently with other Readers

Writers Any thread/actor that needs to perform an operation that cannot be performed concurrently with *any* other operation.

One of the easiest ways to solve this problem is to only allow one thread/actor to perform their operation at a time; this is presented below as the Synchronized HashMap, and the Atomic HashMap. Another approach is to have a structure that keeps track of any currently waiting Readers and Writer, and lets them perform their operations in accordance with the requirements above. These solutions are presented below as Actor HashMap, ReaderWriter HashMap, and Java's built-in ConcurrentHashMap.

6.2.1 Synchronized HashMap

The Sychronized HashMap is the most basic and naïve approach to making a data structure concurrent. It uses the `synchronized` keyword to

control access to each of its methods, thus ensuring that multiple threads cannot change the HashMap at the same time. Unfortunately, it also ensures that multiple threads cannot **read** from the HashMap at the same time either. By requiring mutual exclusion on *every method*, this implementation of a concurrent HashMap suddenly doesn't seem very *concurrent*—so why bother even including it? Recall that one of the purposes of this thesis is to highlight the pitfalls and traps of writing concurrent programs, and there is none so great as the keyword `synchronized`.

The Synchronized HashMap is used with the manually threaded implementations, as well as the Thread-Pool based techniques: Work-Sharing, and Work-Stealing.

6.2.2 Atomic HashMap

Much like the Synchronized HashMap, this implementation of HashMap protects every method call—this time with Deuce's `@Atomic` annotation. This is similar to the way Synchronized HashMap was implemented, but uses STM to provided mutal exclusion, not `synchronized`. The STM engine, based on the results of previous case studies, seems more efficient than just putting locks on everything, so we expect the Atomic HashMap to have better performance than the Sychronized HashMap.

6.2.3 Actor HashMap

We have violated Actor Semantics slightly in this implementation of a HashMap, as we allow multiple actors to access the same data-structure. Due to the internally sequential nature of actors, implementing this the sementically correct way—i.e. with one actor containing the HashMap, which it updates in response to messages—would result in an essentially non-concurrent Map. Instead, each actor may request access to a shared HashMap. To control access to the HashMap, we have implemeted a HashMapControllerActor, to which all Actors wishing to gain access must

send a message requesting access. In addition to this, the message must also include the *type* of access the Actor is requesting: read or write. The controller automatically allows the first actor to request access into the HashMap by returning a PermissionGranted message to the requesting Actor, along with a reference to the shared map. The controller allows any number of actors that have requested read-permission to use the HashMap simultaineously—unless there is an actor waiting to write to the HashMap. If there is, the controller will place any futher read requests in a pending pool and allow the writer into the hashmap when the current readers pool is empty. When the writer is done, the controller checks for any more writers and allows them in one at a time before approving the entire pool of pending readers.

The Actor HashMap is used only with the Actor implementation of the code.

6.2.4 ReaderWriter HashMap

Much like the Actor HashMap, the ReaderWriter HashMap implements a solution to the Readers-Writers problem, using Java's built-in `ReaderWriterLock` in place of the `synchronized` keyword. This allows multiple readers to access the HashMap concurrently, and ensures that only one writer is updating the HashMap at a time.

The ReaderWriter HashMap is used with the manually threaded implementations, as well as the Thread-Pool based techniques: Work-Sharing, and Work-Stealing.

6.2.5 Java's ConcurrentHashMap

The final concurrent HashMap used in this case-study is Java's own implementation, the aptly named ConcurrentHashMap.

The ConcurrentHashMap is used with the manually threaded implementations, as well as the Thread-Pool based techniques: Work-Sharing,

and Work-Stealing.

6.3 Results

Two experiments were run using the implementations of HashMap outlined above. Each consisted of 500,000 randomly generated method-calls on the HashMap, with the ratio of read operations to write operations varying from 100:0 to 0:100. The first experiment consisted of only `get` and `put` operations, the cost of which for a HashMap is O(1). The second experiment allowed method calls to be drawn from the entire pool of available method calls (except `clear`), which are as follows [5]:

containsKey Returns true if this map contains a mapping for the specified key.

containsValue Returns true if this map maps one or more keys to the specified value.

entrySet Returns a Set view of the mappings contained in this map.

get Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

isEmpty Returns true if this map contains no key-value mappings.

keySet Returns a Set view of the keys contained in this map.

put Associates the specified value with the specified key in this map.

putAll Copies all of the mappings from the specified map to this map.

remove Removes the mapping for a key from this map if it is present.

size Returns the number of key-value mappings in this map.

values Returns a Collection view of the values contained in this map.

Only a subset of these operations are O(1), namely `get`, `put`, `size`, `containsKey`, `containsValue`, and `isEmpty`, and even these operations are only O(1) if the HashMap is sufficiently sparse. The other operations are at least O(n) over the number of items within the Map.

For each run of the experiment, each map is initially randomly populated with 50,000 Integer keys and values.

Figure 6.1 shows the execution times for varying ratios of read operations to write operations, with very short read operations (500,000 read operations processed in 560.51 ms for the Non-Concurrent implementation $\approx 1.1 \times 10^{-3}$ ms per operation). We can immediately see that we do not gain any performance benefit from applying concurrency in this situation—in fact, it approximately **doubles** the execution time of the program. The operations being performed on the HashMap, being $O(1)$, don't seem to be taking up enough time to actually make concurrency a viable option for this case study. The overhead introduced by requiring access control clearly outweighs any benefit the concurrency might be adding, especially in the actor-based implementation.

So what's the point of making a data structure like a HashMap concurrent in the first place? From the results in Figure 6.1, it looks to be a terrible idea! But does this still hold if the operations we perform are a little more complex?

Figure 6.2 shows the execution times for varying ratios of read operations to write operations for long read operations (500,000 read operations in 18245 ms for the non-concurrent case gives an average length of 0.036 ms per operation; approximately 33 times slower than the short reads shown above). Notice again how the Synchronized HashMap shows very similar results to the unthreaded case. These implementations demonstrate a linear increase in execution time as the number of read operations increases, which is not unsurprising. In these implementations, all operations are being performed on the HashMap sequentially (as `synchronized` is enforcing mutual exclusion on **all** methods), so as the number of read

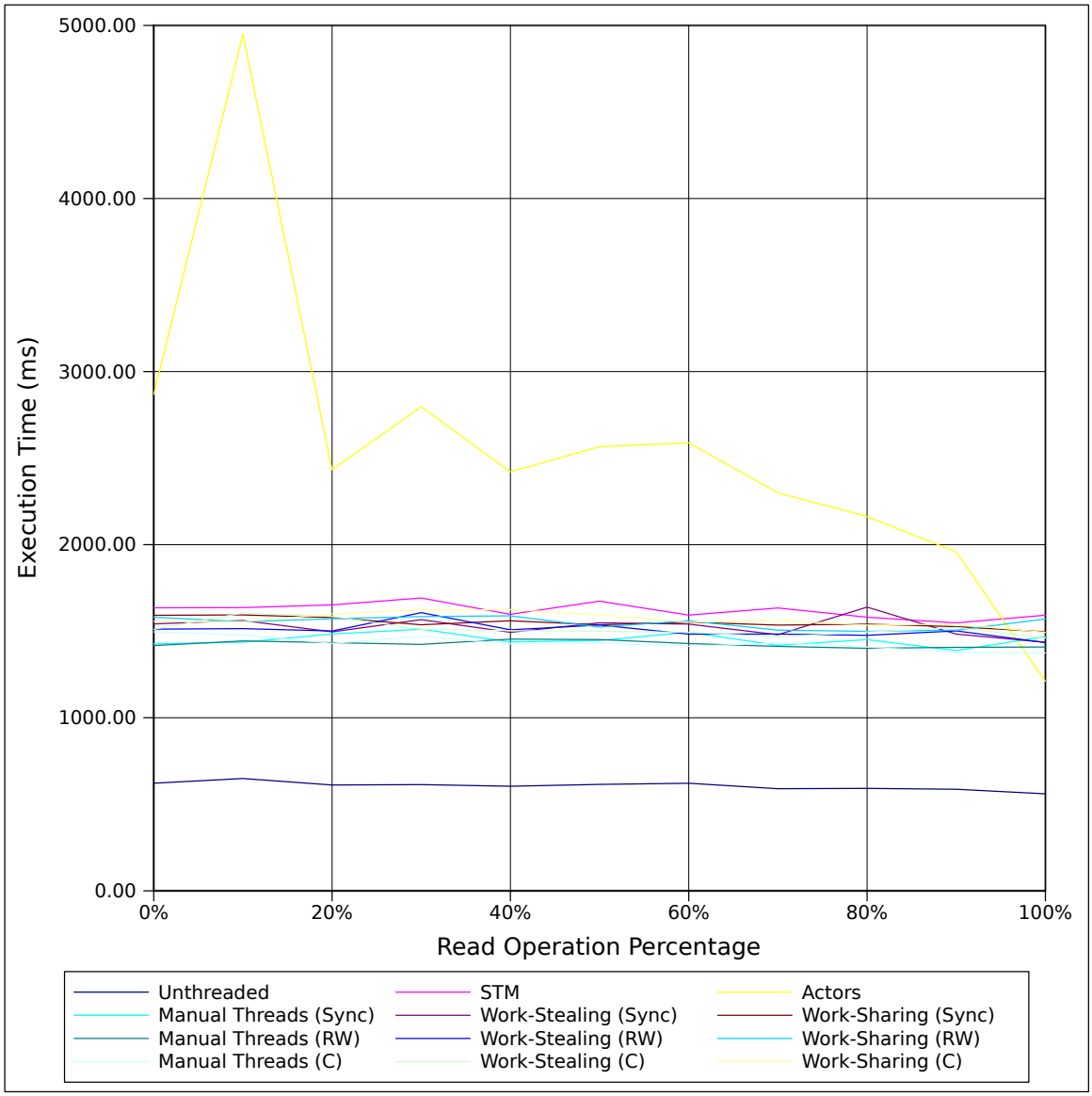


Figure 6.1: Execution times for short read operations

operations increases, the total execution time, as evidenced by the graph, will increase accordingly.

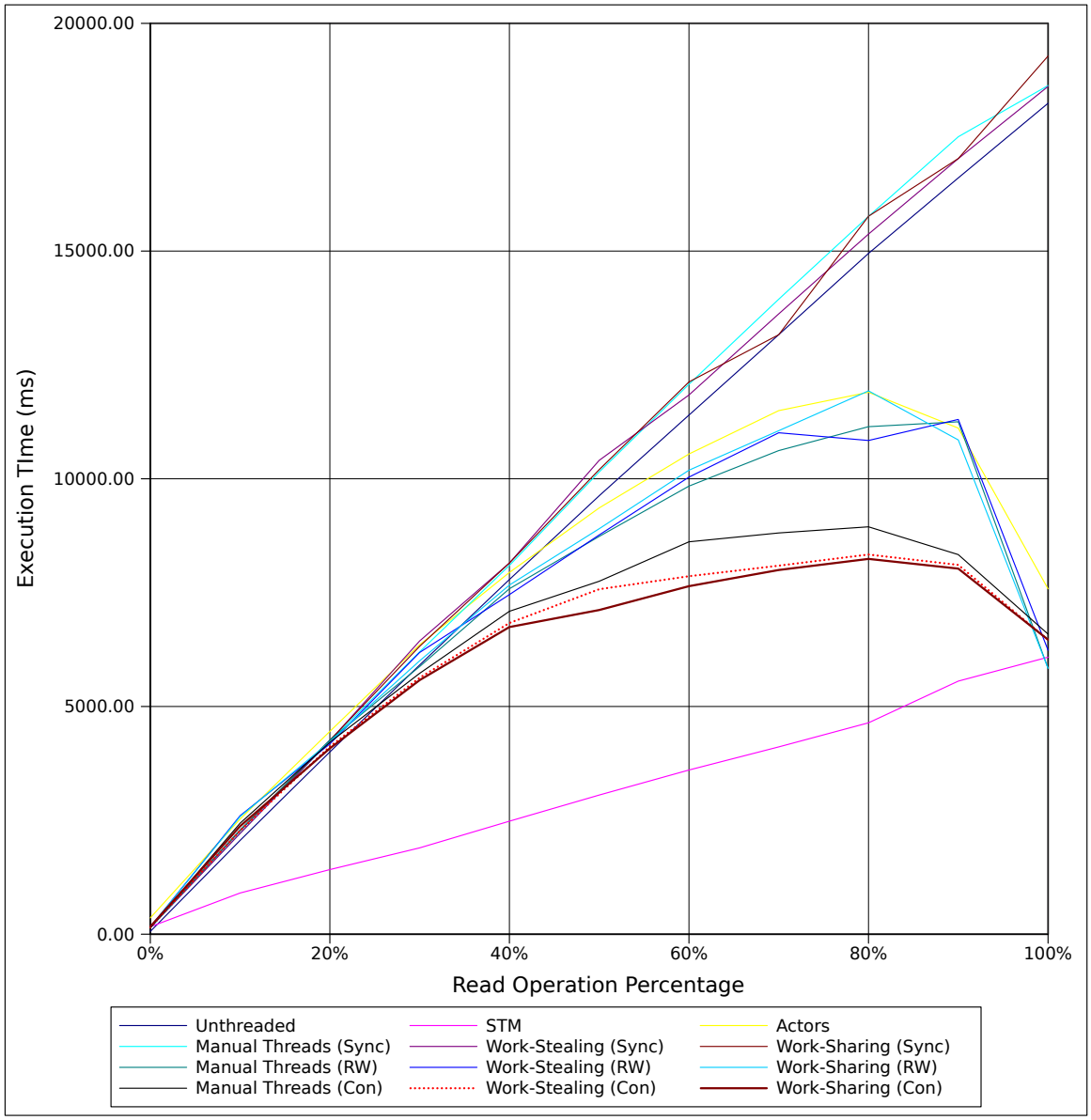


Figure 6.2: Execution times for long read operations

For the Readers-Writers-based Maps, we see a more interesting result. For longer read operations, the penalty for waiting for write operations to complete before allowing readers access is clearly evident from the shape

of the curve, with the worst case occuring when the ratio is 80:20. At higher read percentages, the chances of multiple reads overlapping *without* a write interrupting them is much higher, and leads to improved performance. When every operation is a read, the performance is *significantly* improved over the non-concurrent case, as expected.

STM is the clear winner in this scenario; while it still exhibits a linear increase with respect to the number of read operations, the gradient is far less than any of the other linear results. It maintains a faster execution speed than any of the other approaches, except at the 100% read ratio, where it ties with the non-Actor Readers-Writers Maps.

6.4 Discussion

In this case study, we looked at the effect on performance of making a HashMap thread-safe via a number of different means. What we found was somewhat surprising: for very fast operations like `get` and `put`, the overhead of managing concurrent access to the HashMap significantly increases the amount of time it takes to execute those operations. In retrospect, this should not have been surprising at all: in a decently implemented HashMap, the get and put operations are O(1)—they execute in constant-time. What remains surprising is that this still occurs even when every operation is a read operation, as for the STM, ReadersWritersLock, and ConcurrentHashMap implementations multiple readers are permitted at once. Lock acqusition in Java therefore seems to be quite expensive, even if the thread is **not** forced to wait.

But `get` and `put` are far from the only operations available on a HashMap in Java. Once we add the remaining operations to the pool of available operations, we begin to see a marked difference in performance for each of the implementations. STM begins to really shine under these conditions, becoming the fastest performing implementation. The fact that all of the approaches *increase* in execution times with increasing reads (at least, ini-

tially) might seem odd, but is due to the fact that the majority of the read operations being performed take far longer to complete than the write operations; 300 times longer for the non-concurrent case, which can perform 500,000 write operations in 60.78 ms, but takes 18245 ms to perform that many read operations. Of note, the approaches that maintain mutual exclusion on **all** operations demonstrate a very linear increase in execution time as read operations become more prevelant. This is especially noticable in the Synchronized implementations of the HashMap, as they describe the same curve as the non-concurrent implementation—this should not be surprising, as the use of `synchronized` in *all* of the operations essentially means that it executes sequentially. STM is also affected by this, as it too maintains mutual exclusion across all operations on the map, but is clearly far more light-weight than using the `synchronized` keyword.

The remaining three implementations—Actors, ReadersWriters, and Java's built-in ConcurrentHashMap—create far more interesting curves. Like the other implementations, the initial performance increase is linear wrt to the increasing proportion of read operations, but soon begins to diverge. We notice that when these implementations are executing **only** read operations, their performance is vastly superior to the unthreaded case, performing the same number of operations in approximately a third of the time. Each possess a maxima, where the ratio of reads to writes seems to be such that it causes the worst performance. This appears to happen at a ratio of 80:20 for most cases, implying that at this ratio the write operations maximally interfere with the read operations.

6.4.1 Evaluation

Java's built-in ConcurrentHashMap was the easiest version of the code to implement, as using a ConcurrentHashMap requires no additional code to using a regular HashMap beyond changing the constructor. Implementing a solution to the Readers-Writers problems using Actors was clear and

understandable, despite the gross violation of the intended actor semantics by allowing a shared data-structure. It is aided by the fact that communication between actors is performed via message-passing, and that actors can only process one message at a time—in this way, we can guarantee that the "lock" Actor is only processing one request for entrance at a time. Because Java is a relatively permissive language, we were able to violate the intended semantics of the Actor Model and construct a concurrent HashMap that has better performance that a purely lock-based implementation, though it is still out-performed by the ReadersWritersHashMap on which it is based.

Implementing different versions of a HashMap in Java is made quite simple by the OO nature of the language. SynchronizedHashMap and ReadersWritersHashMap were simply classes that implemented Java's Map interface, and internally mapped method-calls onto a regular HashMap. The only alterations were how the methods were accessed. For SynchronizedHashMap, all of the methods were given the `synchronized` keyword, requiring all threads to obtain a single global lock on the objects, ensuring mutual exclusion on every method in the class. For the ReadersWritersHashMap, the `synchronized` keyword was replaced with method-specific calls to a ReaderWriterLock that either attempted to obtain the Read Lock or the Write Lock, depending on the nature of the method call. This made the code easy to understand, and straight-forward to implement: two things that go a very long way when writing a concurrent program.

Chapter 7

Conclusion

In this thesis, we have investigated some of the options programmers have when writing a concurrent program. We used these options to implement case studies of various kinds: a video game (Chapter 3), a physical simulation (Chapter 4), an image-processing application (Chapter 5), and a concurrent data structure (Chapter 6). Through-out these case studies, we notice a common thread: concurrency, applied correctly, can improve the performance of a program—but the correct application may not be readily apparent. Concurrency is an important tool in the toolbox of the modern programmer, especially with the rise of multi-core architectures and the increasing prevalence of distributed systems. And like any tool, it is important to understand how and when to use it.

To this end, we also explored some of the pitfalls that might befall programmers when writing concurrent code, and performed a subjective analysis of the approaches we used to write the case studies. We judged implemenation efficiency and understandability, as well as the more conceptual problems that might arise from trying to code in a concurrent fashion. We explored the utility of certain concurrency models in different situations, and how easily they were adapted to fit a different scenario.

7.1 The Actor Model

To a programmer trained in OOP, the Actor Model initially presents a conceptual difficulty. The lack of shared state between actors leads to quite a different design-space than the traditional OOP model. Instead of method calls, one has to think about literal messages being passed around—messages that need to contain all of the state required to perform the function being invoked. This is similar enough to the functional paradigm that any programmer familar with functional programming will be able to adjust more readily to the Actor Model. Having used actors to solve a variety of concurrency issues, I have to admit that the model has presented itself as a reasonable solution in many situations. Asteroids, at least on the surface, appears to lend itself well to being implemented with the Actor Model. The truth, however, is that the Actor Model is not a model of concurrency that meshes well with Java as a language. Java is too rooted in OOP paradigms to allow a "proper" implemenation of actors without significantly overhauling the JVM to use actors instead of threads as its internal concurrency model. This is echoed by the results presented in this thesis: the only time actors performed as well as any of the other approaches was when there was almost no message-passing or shared state of any kind. This leaves us with a not particularly unexpected conclusion: actors are best when performing isolated tasks that require very little in term of shared information. They are a perfect model to use for server-side client handling, for example.

7.2 Thread Pools

It should really come as no surpise that code that used thread-pools generally out-performed code that manually created and started threads whenever it required them. Threads, while still far more light-weight than an entire process, are not inexpensive. Populating a pool with threads at the

outset saves a lot of overhead. The only case-study in which this was **not** readily apparent was that of the image-processing server. The overheads incurred by creating large numbers of threads at once were clearly mitigated by the gains in having those large numbers of threads to perform file I/O—unless, of course, the code is badly written or uses a lot of memory, in which case having large numbers of threads is not a good idea.

Work-Stealing, in general, proved to be the more efficient implementation of a thread-pool. The only time this was not the case was in the AtmosSim case study—when there was only one job per thread in the pool. This is clearly not an optimal case for the Work-Stealing thread-pool, which performs much better when there are actually jobs to steal in the first place.

7.3 Software Transactional Memory

STM proved that in many cases—but not all—it is a much more efficient and scalable approach to managing concurrent access to shared state than the other approaches used in this thesis. Asteroids, AtmosSim, and the Concurrent HashMap show that STM is generally more efficient than using `synchronized`. The only time that this was not the case was when contention was quite low in the first place, thus the overheads of STM cause it to be slightly less efficient than `synchronized`. The Concurrent HashMap case study also showed that it out-performs more finely-crafted methods of controlling concurrent access; specifically Java's built-in ConcurrentHashMap and the use of a ReaderWritersLock to allow concurrent reads but not writes—though at the point where every operation was a read, the performance of all three was roughly equal.

7.4 Future Work

The landscape of concurrency is a very large and constantly changing one. This thesis has presented but a small sliver of the depths of concurrent programming. Java is but one language amongst many, and every language handles concurrency in different ways. For example, Erlang is built with the Actor Model as its default model of concurrency. C, and by extenstion C++, have numerous threading and concurrency libraries available for them. The work done in this thesis could easily be extended to compare concurrency in different languages. Even sticking to just Java, there are other Actor and STM libraries available than the ones used in this thesis—though comparing similar concurrency models is something that is already a topic of extant research [15].

One thing that was not explored in this thesis is the effects of varying the number of available processors. We present results for the non-concurrent (ie: single core) case, as well as multiple concurrent implementations that execute on four-cores. Extending the case studies to utilize larger numbers of cores would be an interesting avenue of investigation, though would require additional hardware.

Another avenue of research may be to explore the conceptual difficulty of concurrency. *Why* is it hard? What sorts of internal mental models of concurrency do successful programmers use in order to avoid the difficulty of concurrent programming—and how can we use them to make statements about concurrency? In the end, we're left with the question: what can we do to make concurrency easy? The answer, I fear, will be as complex as the history of programming languages themselves.

Bibliography

[1] Akka Website.
`http://akka.io` accessed: 20 Feb 2013.

[2] Bicubic Interpolation for Image Scaling.
`http://http://paulbourke.net/texture_colour/imageprocess/` accessed: 20 Feb 2013.

[3] Deuce STM Website.
`http://www.deucestm.org/` accessed: 20 Feb 2013.

[4] JSR-166y Website.
`http://gee.cs.oswego.edu/dl/concurrency-interest/index.html` accessed: 20 Feb 2013.

[5] Map Interface JavaDoc.
`http://docs.oracle.com/javase/6/docs/api/java/util/Map.html` accessed: 20 Feb 2013.

[6] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (New York, NY, USA, 1967), AFIPS '67 (Spring), ACM, pp. 483–485.

[7] BEN-ARI, M. *Principles of concurrent and distributed programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[8] CHASE, D., AND LEV, Y. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2005), SPAA '05, ACM, pp. 21–28.

[9] CLINGER, W. D. Foundations of actor semantics. Tech. rep., Cambridge, MA, USA, 1981.

[10] DINAN, J., LARKINS, D. B., SADAYAPPAN, P., KRISHNAMOORTHY, S., AND NIEPLOCHA, J. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 53:1–53:11.

[11] FELBER, P., KORLAND, G., AND SHAVIT, N. Deuce: Noninvasive concurrency with a Java STM. In *Electronic Proceedings of the workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)* (2010), p. 10 pages.

[12] FICK, A. Ueber diffusion. *Annalen der Physik 170*, 1 (1855), 59–86.

[13] FOSTER, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] HEWITT, C., BISHOP, P., AND STEIGER, R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence* (San Francisco, CA, USA, 1973), IJCAI'73, Morgan Kaufmann Publishers Inc., pp. 235–245.

[15] KARMANI, R. K., SHALI, A., AND AGHA, G. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (New York, NY, USA, 2009), PPPJ '09, ACM, pp. 11–20.

[16] KITTEL, C., AND KROEMER, H. *Thermal Physics (2nd Edition)*, second edition ed. W. H. Freeman, Jan. 1980.

[17] LEE, E. A. The problem with threads. Tech. Rep. UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

[18] MICHAEL, M. M., VECHEV, M. T., AND SARASWAT, V. A. Idempotent work stealing. *SIGPLAN Not. 44*, 4 (Feb. 2009), 45–54.

[19] MORRIS, T. *Computer vision and image processing*. Palgrave Macmillan, Sept. 2003.

[20] MURTHY, P. Parallel computing with x10. In *Proceedings of the 1st international workshop on Multicore software engineering* (New York, NY, USA, 2008), IWMSE '08, ACM, pp. 5–6.

[21] SARASWAT, V. A., SARKAR, V., AND VON PRAUN, C. X10: concurrent programming for modern architectures. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2007), PPoPP '07, ACM, pp. 271–271.

[22] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1995), PODC '95, ACM, pp. 204–213.

[23] SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. *Operating system concepts (7. ed.)*. Wiley, 2005.

[24] SONKA, M., HLAVAC, V., AND BOYLE, R. *Image Processing, Analysis, and Machine Vision*. Thomson-Engineering, 2007.

[25] SUTTER, H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal 30*, 3 (2005).

[26] SUTTER, H., AND LARUS, J. Software and the concurrency revolution. *Queue 3*, 7 (Sept. 2005), 54–62.

[27] WOLF, M. J. *The video game explosion : a history from PONG to Playstation and beyond*. Greenwood Press, Westport, Conn, 2008.