5-2016

Multithreaded Collision Detection with Akka's Actor Library in Scala

Bryan Schrock
Trinity University, bryanrschrock@gmail.com

Multithreaded Collision Detection with Akka's Actor Library in Scala

Bryan Schrock

Abstract

This project strives to determine the efficiency of pure actor computing to more conventional methods of parallelizing code. The research involved three algorithms, all of which perform two dimensional collision detection: a one thread, control method that is well documented in previous work, a multithreaded method using a novel technique to avoid race conditions, and an actor model, written using the Akka Actor library. The three algorithms were run at varying input size to get an accurate idea of efficiency.

Acknowledgments

Firstly, I would like to thank Dr. Mark Lewis for his help as my thesis advisor, as well as his constant support throughout my time at Trinity. I would also like to thank Dr. Matthew Hibbs and Dr. Paul Myers for donating their time to my project. Finally I would like to express gratitude to my family and friends that helped with the revision process.

Multithreaded Collision Detection with Akka's Actor Library in Scala

Bryan Schrock

A departmental senior thesis submitted to the Department of Computer Science at Trinity University in partial fulfillment of the requirements for graduation with departmental honors.

April 15th, 2016

Mark Lewis
Thesis Advisor

Paul Myers
Department Chair

Sheryl R. Tynes, AVPAA

Student Agreement

I grant Trinity University (“Institution”), my academic department (“Department”), and the Texas Digital Library ("TDL") the non-exclusive rights to copy, display, perform, distribute and publish the content I submit to this repository (hereafter called "Work") and to make the Work available in any format in perpetuity as part of a TDL, Institution or Department repository communication or distribution effort.

I understand that once the Work is submitted, a bibliographic citation to the Work can remain visible in perpetuity, even if the Work is updated or removed.

I understand that the Work's copyright owner(s) will continue to own copyright outside these non-exclusive granted rights.

I warrant that:

1) I am the copyright owner of the Work, or
2) I am one of the copyright owners and have permission from the other owners to submit the Work, or
3) My Institution or Department is the copyright owner and I have permission to submit the Work, or
4) Another party is the copyright owner and I have permission to submit the Work.

Based on this, I further warrant to my knowledge:

1) The Work does not infringe any copyright, patent, or trade secrets of any third party,
2) The Work does not contain any libelous matter, nor invade the privacy of any person or third party, and
3) That no right in the Work has been sold, mortgaged, or otherwise disposed of, and is free from all claims.

I agree to hold TDL, Institution, Department, and their agents harmless for any liability arising from any breach of the above warranties or any claim of intellectual property infringement arising from the exercise of these non-exclusive granted rights.”

I choose the following option for sharing my thesis (required):

[x] Open Access (full-text discoverable via search engines)
[ ] Restricted to campus viewing only (allow access only on the Trinity University campus via digitalcommons.trinity.edu)

I choose to append the following Creative Commons license (optional):

Multithreaded Collision Detection with Akka's Actor Library in Scala

Bryan Schrock

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reactive programming has grown quickly from a concept into an idealized form of writing multi-threaded code. It has huge implications in web design, where much of the computation is already reactive. In fact, any graphical user interface should be a reactive program.

A popular tool for achieving reactive programming is actors. Actors and actor systems can vary by implementation but the idea remains as a small reactive unit of code. The popular full-stack Scala based framework Play uses Akka actors to implement their framework. Actors have been rising in popularity, and the Akka Actor library is praised as an efficient way to build a reactive multithreaded system, and effective for solving inherently reactive problems.

This research aimed to determine the efficiency of pure actor computing compared to more conventional methods of parallelization in a problem that is not reactive in nature. I strive to build a scalable two dimensional collision detection system using actors, while adhering to Akka programming guidelines as much as possible. In this thesis, I will discuss the prior studies on which this is built, the design steps taken for the actor model as well as control models, and the results I have found.

1.1 Two Dimensional Collision Detection

The biggest reason for the selection of this algorithm for testing is simply the precedent set by Dr. Mark Lewis in previous research. His familiarity with the algorithm was hugely beneficial and his ideas for methods of multi-threading the algorithm spurred the models that were produced in the process of this research.

1.2 Concerns With the Java Virtual Machine

I chose to implement a reactive actor system using Scala. It is a familiar language to me, and has higher-order functionality. The Akka Actor library is a well-documented, easy to implement extension. All of that being said, there are some immediate concerns with this choice that I will address here before continuing.

The obvious concern is that the Java virtual machine (JVM) is not the most efficient choice. Normally a simulation such as this would be written in C++ or a similar highly efficient language. Scala runs on the JVM which can and will have effects on both efficiency and memory usage. I pose that these issues are solved simply because *all* of my code runs on the JVM, thusly comparing between iterations of algorithms will be meaningful. Whether the JVM directly affect the efficiency of actors but not that of the other algorithms will be left for a future project.

Chapter 2

Related Work

In this chapter, we will give a brief introduction to the work of Dr. Mark Lewis, among others, on which this research is based. We will introduce the algorithm that spurred all three algorithms used here, as well as the ideas and coding practices used in the actor model.

2.1 Single threaded two dimensional collision detection

To introduce the slightly modified algorithms eventually implemented, it is useful to begin by looking at a single threaded approach. This approach can be summarized with the following pseudocode [2]:

```
1) Build spatial data structure.
2) Find potential collisions based on initial conditions and
add them to a priority queue.
3) For(predefined number of timesteps)
    a) While there are potential collisions on the priority
        queue.
```

```
i) Remove the next collision from the priority queue.
ii) Process that collision.
iii) Remove all future potential collisions involving
    either of the colliding particles from the priority
    queue.
iv) Find new potential collisions involving those
    particles based on their new trajectories. Add
    those happening in the current time step to the
    priority queue.
```

The term "potential collision" is used to describe a class containing two particles that will collide given their current attributes, and a time signifying when that collision might occur. The uncertainty of the collision is due to the fact that a potential collision may be altered by previously occurring collisions and the changing conditions that accompany them. This single thread approach is used for the control tests.

2.2 Hard-Body Simulations vs. Soft-Body Simulations

There is an important difference in simulating collisions between rigid spheres, as was done in this thesis, and simulating collisions between more deformable things, referred to as soft-body objects. When rigid bodies are used, many of the techniques used for collision detection are heavily based on data structures that can be more or less pre-computed before the simulation starts [5]. Soft-body physics, on the other hand, is frequently used in games, and requires the ability to simulate in real time. Often, the requirement for accuracy is far less important in soft-body simulation, as long as it appears accurate to an observer. This thesis used hard-body collisions. These require accuracy, because their intended use

is scientific simulations.

2.3 Parallelization of the single threaded approach

Parallelizing the algorithm adds a little complexity to the problem. According the the Akka guidelines, shared memory like the priority queue described in step 2 of the pseudocode is not allowed. The priority queue being shared memory could be solved by either using a thread safe queue or smartly applying blocking. The processing loop itself poses a more difficult problem. It is important to realize that while the order in which collisions are processed is very important, collisions happening far enough away from each other can be processed simultaneously. What constitutes "far enough away" depends on a few things. The grid based spatial data structure is created based on the particle with the most velocity. Taking into account the distance that this particle can travel, we can create a grid where we are guaranteed no particle will cross more than one grid boundary in one time step. After constructing this grid, we know that we can process any two grid cells that are not neighbors simultaneously. What we do with this information varies between the two multithreaded algorithms.

2.4 The Actor Model

In Derek Wyatt's *Akka Concurrency*[6], the book we used as a guideline in following what we will call "The Actor Model," two methods of achieving concurrency are introduced. The first is shared-state concurrency, the "default" or "normal" way to program multithreaded code in Java, for example. Other authors call the shared-state model the async-finish model[1]- either way, this model describes when a parent thread spawns child threads and uses blocking to solve race conditions. The other method of achieving concurrency is message passing,

and that is the model on which the actor model is based.

Akka was not the first actor design. The idea of reactive concurrent programming has been around for a long time. Scala had an actor library before Akka was released, and Martin Odersky describes it in a detailed paper [4].

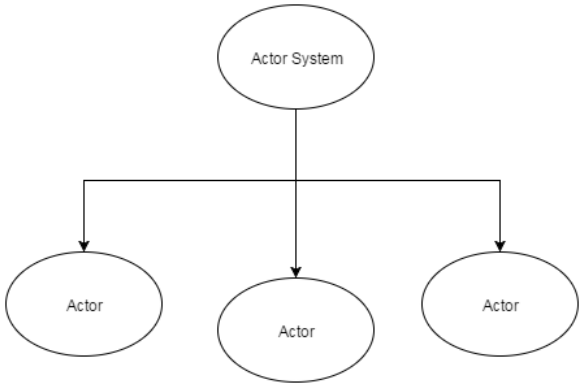

Figure 2.1: A simple diagram of an actor system

With actors, many of the issues that come up in parallel programming are handled simply by following the guidelines setup in the actor system. An actor is an object that has the capability to process incoming messages, with a well-defined lifecycle and rules regarding its actions in different states. Message passing with actors is accomplished by sending messages between actors and listing reactions to those messages. In figure 2.1, we see the main actor (The actor system) which knows about all of its child actors. The system will send messages during runtime to the children and they will react according to a predefined set of rules. The actors can also send messages to each other as long as they know about each other. The other big consideration to think about is that Akka recommends a minimum amount of mutable data, in order to minimize race conditions in actor processing. In fact, the messages passed should contain *no* mutable data whatsoever. The actor's local

state is entirely encapsulated within the actor itself, and cannot be accessed by outside objects. Due to this, and to the ability to communicate between actors asynchronously, actors are inherently concurrent and are a great format to accomplish multithreaded code [1]. We will discuss our specific implementation of the actor model in a later section.

2.5 Limitations of the two models

The async-finish model is well suited to deterministic problems where the "finish" of the children threads generally occurs in a similar time frame. However, many problems are not this way. The quicksort algorithm, for example, is widely regarded as a simple and effective algorithm to parallelize; With the async-finish model, there is not a way to compute partial results of the sort on the occasion that one side of a split set is available early. We must wait for both sides to become available and process them then [1]. In an actor system, we can merge two sides of a quicksort as soon as they are available to merge, adding some efficiency.

The actor model's main limitations stem from its inability to simulate non-blocking synchronous replies. It cannot guarantee the arrival time or order of messages and thus cannot guarantee the order of responses. Similarly, achieving "global consensus" in a group of actors can lead to issues [1]. The typical approach to solving this problem is to use a coordinator actor to process the order of messages in the coordinated actors. This coordinator actor can sometimes become a bottleneck. This coordinator/coordinated approach is how I solve this problem in this research.

2.6 Messages

Messages can be any object. We decided to use Scala's Case Class functionality. The list of possible messages is simply a list of case classes with specified names and specified contained information. Each actor has a reaction method which is filled with a case switch on the message name, calling functions with the parameters contained inside the case classes. The main actor and the child actors react differently to different messages.

2.7 Implementing the actor model

Implementing the briefly discussed parallelized algorithm for two dimensional collision detection using the actor model takes a little extra thought. As shown, the algorithm even in its single thread form, requires some spatial data structure. For the actor model simulation, we use a grid to fill this need along with the need for an ActorSystem. The grid is told to begin its process and main thread now is waiting for an answer from grid.

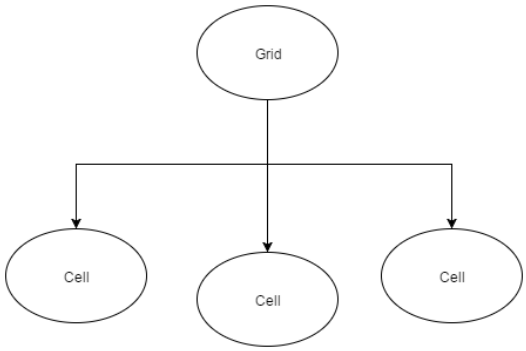

Figure 2.2: A simple diagram of our implementation of the actor model

Upon starting, the grid creates an array of actors, in our implementation called cells, the size of which corresponds to the number of rows and columns in the grid (something

which is generated previously and will be discussed in a later chapter). After generating the array of cells, the grid provides each one with a collection of its neighbors. The cells are also passed a message with a collection of particles that they store for later processing use. This list cannot be used by other cells directly.

2.8 Non-actor parallel model

In order to get a reference for the efficiency of Actors, we implemented a multithreaded algorithm that does not use actors but instead uses the previously mentioned shared state method. There are many similarities between the two. Both models have a main grid based data structure containing cells which do most of the actual work. However, in order to handle race conditions, we made some adjustments to the order of processing. The algorithm is again heavily based on Dr. Lewis' previous work with both Berna Massingill [3] and with Zachary Langbert [2]. The specific changes we made will be discussed in more detail in a later chapter.

Chapter 3

The Algorithm Design

3.1 The Single Thread Model

The single thread model was discussed primarily in section 2.1, with a set of pseudocode largely taken from Zachary Langbert. The algorithm I used is almost exactly the same as the one discussed there. It will be used as a control in the experiments to be compared to the two multithreaded models.

3.2 Detailed Description of the Pseudocode

For reference, here is the pseudocode again.

```
1) Build spatial data structure.
2) Find potential collisions based on initial conditions and
add them to a priority queue.
3) For(predefined number of timesteps)
    a) While there are potential collisions on the priority
```

```
queue.
i) Remove the next collision from the priority queue.
ii) Process that collision.
iii) Remove all future potential collisions involving
    either of the colliding particles from the priority
    queue.
iv) Find new potential collisions involving those
    particles based on their new trajectories. Add
    those happening in the current time step to the
    priority queue.
```

1. In the single thread model, as well as the other models, I use a grid for the spatial data structure. It works well for two dimensional collisions and accomplishes everything it needs to for this project. In each model, the grid acts as a wrapper for a smaller unit of computation. In the single thread, it is simply a two-dimensional array of lists of particles, but in the other two models there is a cell class that handles much of the internal computations needed during processing. The number of rows and columns in the grid is determined based on the maximum velocity of all particles, combined with the user-defined number of time steps in the simulation. Given these two values, we can know the size the cells have to be to guarantee a particle will not pass more than one grid boundary in one time-step. This is a vital guarantee to have, because the simulations only look at neighboring cells when searching for collisions. If a particle travels across two cell boundaries in one time step, it will be in uncharted territory and any collisions it should have would not be found.

2. The conditions for the creation of a potential collision are as follows. First, a time of collision is calculated based on the position and velocity of each of the particles. The grid

is important here- only cells of the grid that are neighbors of each other are considered for potentials. The entire simulation is run in steps, so if a collision is found and will occur during this time step, it is added to a list of potential collisions.

3. In a single, shared-memory model, this step is not too complicated. The potentials are stored in a collection and sorted as they are found, so popping the first element of the collection finds the collision that will happen the soonest. The particles are moved in space to the exact point of collision, and new velocities are calculated. The cell in which the collision occurs and its neighbors are told about the new cells so that new potentials can be found if necessary. Then, the next collision is popped and processed in an identical way. This is repeated until there are no more potentials, and then a new time step is started. The exact implementations of this step in the pseudocode will be described in their relevant sections.

In a well-implemented and tuned version of this code, collision detection hopefully becomes an O(n) algorithm, a huge improvement on the quadratic problem that this would be otherwise.

3.3 Walkthrough of a simulation

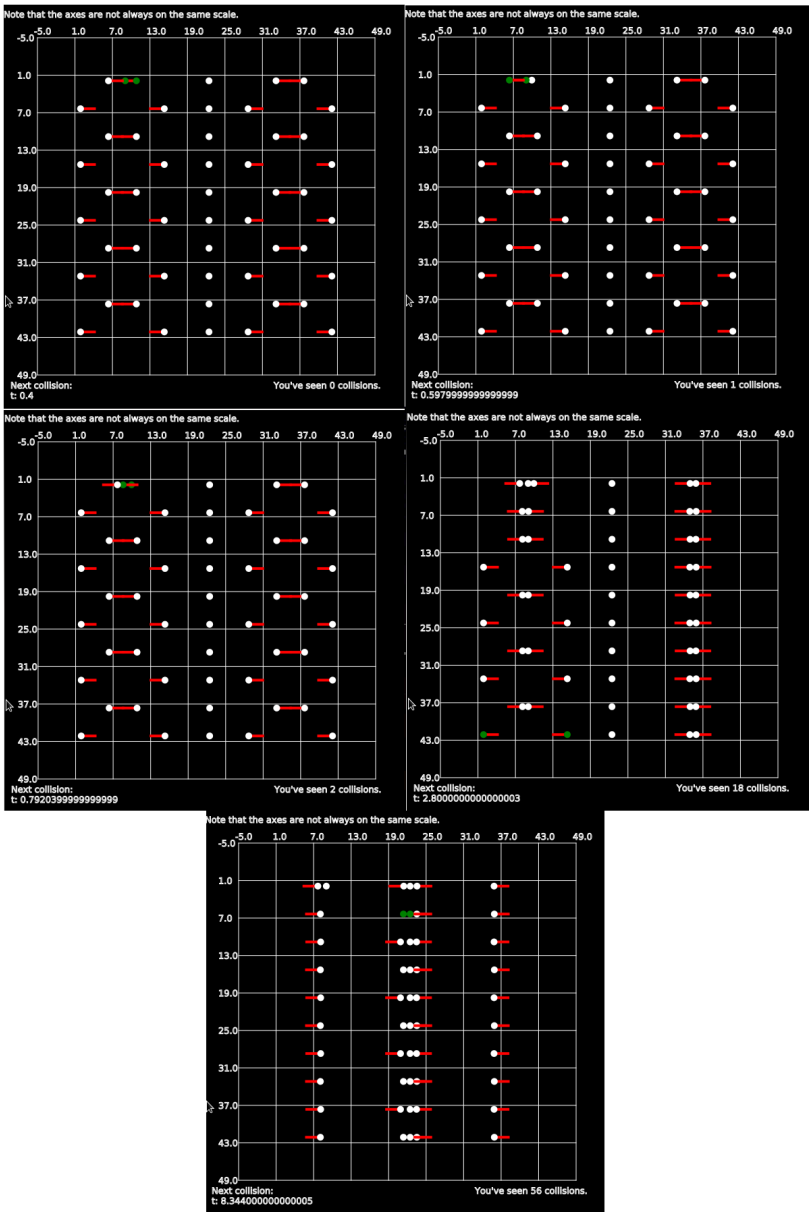

Figure 3.1: Five snapshots during the processing phase of a simulation.

Examine figure 3.1. The first diagram (the top left) is the input set of particles before any collisions occur. The white circles are particles, with red lines showing their to-scale velocities. The two particles that are colored green are the particles involved in the next collision. This input repeats the same pattern many times, but notice that the top left set of particles is slightly different, with three particles in a line. We can follow the collisions as they occur, in the next two diagrams, to the right and then below the first, we see the three particles in the top left of the input bouncing off each other. For purposes of brevity, the next diagram is a snapshot many collisions later. Notice that, due to the nature of the algorithm, particles are only moved when a collision happens. This reflects the algorithm, which only moves particles if they are involved in a collision or when going to a new time step.

The final diagram is the output of the program after all its time steps are run. While there may still be collisions in the set of particles, the number of time steps constrains the run time of the simulation to here.

3.4 The Six-Linked Threaded Priority Bucket Queue

In step 2, the pseudocode mentions a priority queue. For the single thread model in this research I wrote a priority queue specifically for collisions which did a lot of work for me. This priority queue is visualized in figure 3.2. The class contains two bucketed arrays: one based on particle number and one based on time of collision. Each of these arrays simply contains pointers to the first node and it's doubly linked list.

The nodes (visualized in the Figure as the three squares labeled a b and c) each contain 6 pointers:

1. Time Pointers (Previous and Next): Previous and next pointers to nodes based on

Linked Threaded bucket queue.png

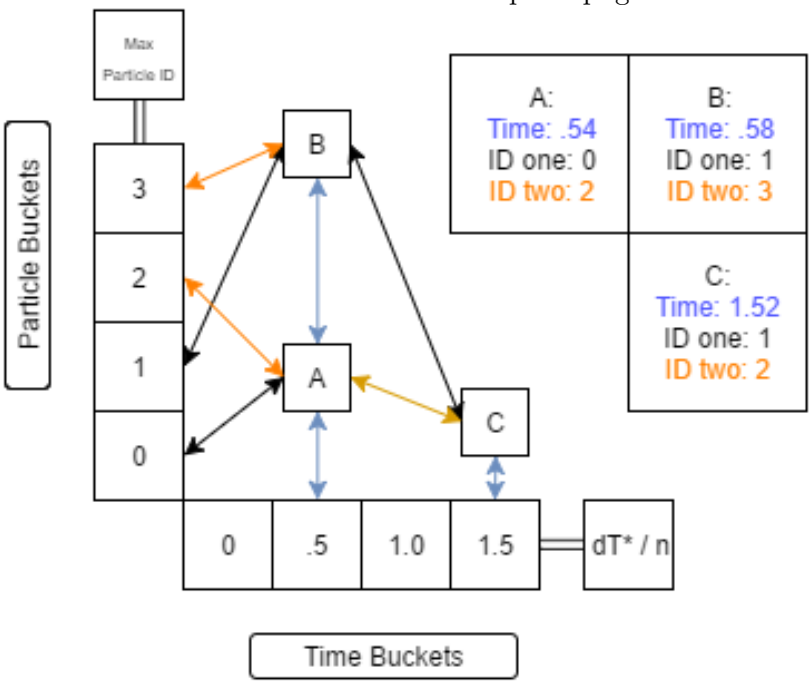

Figure 3.2: The six linked threaded bucket queue.

time (visualized in the figure as the blue arrows coming out from the bottom and top of the nodes). Each time bucket has a range of times that a collision must be within to be contained in the bucket, ranging from 0 to (dT/n) where n is the number of particles, and dt is delta time, the amount of time in one time step for the current simulation. This allows for priority part of the priority queue, having a sorted collection of collisions by time.

2. Particle Pointers (One, Two, Previous, Next): The orange and black arrows in the figure show the *particle pointers*, references to the nodes that involve the particles that match this node. Examine node *A* in the figure. Its particles are id numbers 0 and 2. We can see the black arrow on the left pointing to bucket 0, and the orange

arrow (representing particle 2) pointing on the left to bucket 2, and on the right to node C. Looking at C, we see it also has particle ID 2 in its collision.

The particle pointers are actually circularly linked. This means that if a path followed a nodes one next pointer repeatedly, it would not end but instead wrap back to the beginning. In the figure, this means that As two next is C, but not pictured is that Cs two next is in fact A. If a node is the only node involving a particle as A is with particle 0, its particle pointers involving that particle will point at itself.

The reason the particle pointers exist is to enable the fast deletion of all collisions involving a certain particle. When a particle is processed, its trajectory is changed, and potential collisions that are held in the priority queue need to be validated before proceeding. This is accomplished simply by deleting all potential collisions involving the particles, and calculating new potentials based on their new trajectories.

3.5 Global Class

In order to properly test the efficiency of the algorithms, code is shared where possible. Shared code is managed by the Globals class. The class contains the following important methods:

- double findCollision(Particle, Particle)

  Returns the time that two particles will collide using vector math.

- goToNewTimeStep

  Changes the internal values necessary for keeping track of time steps.

- findNewVelocities

Uses vector calculations to give back the new velocities of two particles bounced off each other.

- translateCoordToCell

  Gives back the indexes of a cell based on pre-calculated information surrounding the list of particles, and given coordinates.

It also has the following attributes:

- allParticles

  The original array of particles that everything else references. Passed in to the constructor of Globals. Upon creation, Globals figures out many of the constraints for the simulation based on this list. For example, the side lengths of the cells is based on the smallest and largest points in the list, combined with density information.

- dT

  Delta time. the length of each time step. Important for ensuring the property where no particle crosses two cell boundaries in one step

- T

  Time. Equal to the number of time steps so far times dT.

- currentTimeStep

  How many time steps the simulation has run.

- numTimeSteps

  User defined, the number of time steps that the simulation will go through.

3.6 The Bounced Particle Problem

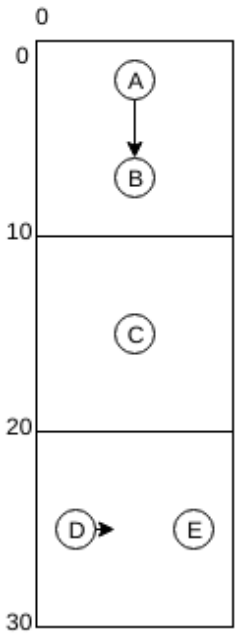

Figure 3.3: An example of the bounced particle problem.

When considering options in how to multithread the collision detection algorithm, there is a key problem to consider. If the algorithm was simply run in parallel with each cell having its own queue (solving shared memory problems at first), another, more hidden race condition will happen:

If two cells share a neighbor, and both cells currenly contain local minima, it is possible for race conditions to appear. Consider figure 3.3. The collision between A, a particle with very high velocity, and B will cause a chain reaction in B hitting C. Because D is moving so slowly, the collision with B and C should be processed before the collision between D and E. This specific issue could be handled relatively simply, but it shows the real source of the

problem: by running two cells with a shared neighbor simultaneously, the two might try to add collisions to the shared neighbor or otherwise access the memory, leading to a shared memory scenario. A good way to handle the issue at hand is to avoid simultaneously processing cells that share neighbors. If cells do not share a neighbor, each can process secure in the knowledge that they are the local minimum and that nothing will modify their neighbors without them knowing.

3.7 Multithreading the Single Thread Algorithm

The single thread collision detection algorithm is relatively simple to parallelize. It is possible to parallelize the discovery of potential collisions, as long as the priority queue is thread safe [2]. I used Java's PriorityBlockingQueue, which is thread safe by default. When this queue is asked to enqueue or dequeue an element, it makes sure nothing else is accessing it and if it is not currently safe, it queues the action and performs actions in order. The grid in this model was largely unchanged. Beginning the simulation, finding potential collisions is trivial. The calculations themselves do not depend on each other at all and can be run entirely simultaneously. Scala's parallel for loop handles this nicely. The part that does contain inter dependencies is adding the found potentials to the cells. As discussed above, this potential problem is thoroughly solved with the PriorityBlockingQueue. After calculating the potentials, we move to the processing phase. It is here that my algorithm strays the most from those discussed in the previous chapter.

3.8 The Bounced Particle Problem in the Non-Actor Model

As discussed in section 3.6, the best way to handle the bounced particle problem is to simply avoid simultaneously processing two cells with a shared neighbor. I achieve this in the non

actor multithreaded model with the following steps. After finding potentials, the processing stage uses a parallel loop to begin the processing of every third row. Each row is processed

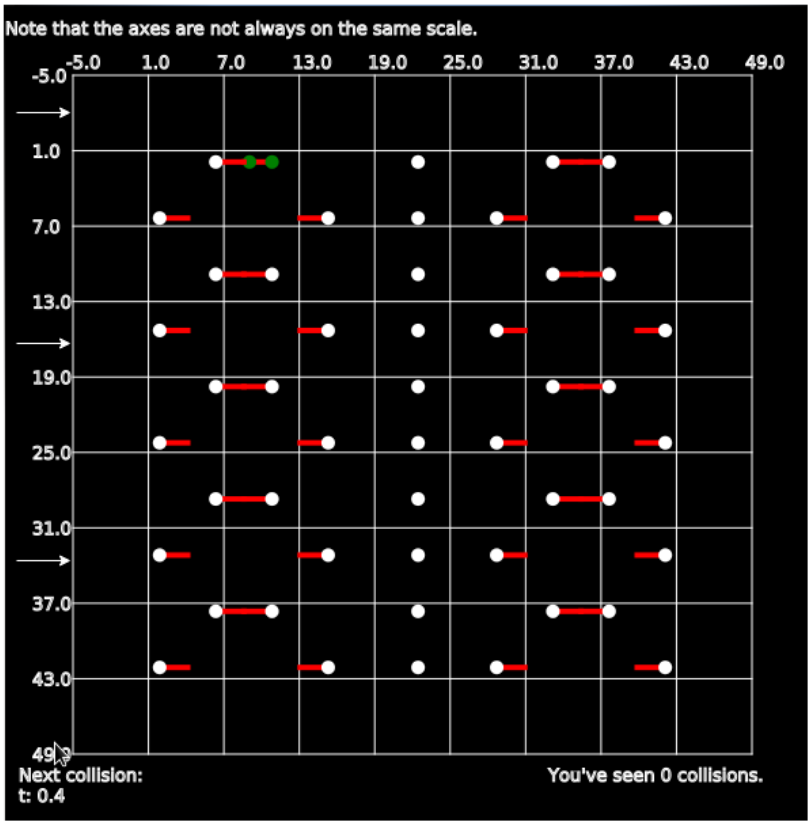

Figure 3.4: Initial run in the non actor multithreaded model.

left to right in one thread. Because a row is processed in a single thread, there is no chance of race conditions in that local computation. The lack of race conditions in the higher scope is guaranteed by skipping to every third row. Consider a four by four grid containing some particles and their various potentials. The grid will begin the processing stage by asking rows 0 and 3 to begin. If the cell located at (0,0) is the local minimum, we know that no collisions happen in (0,1) before the first collision in (0,0). Simultaneously, row 3 has begun

and if that is reported as the local minimum, then we know that no collisions happen in (0,2) before the collision in (0,3). Figure 3.4 shows the initial run through of this algorithm. The white arrows on the left side of the diagram show which rows are run simultaneously first. This approach takes advantage of more threads as the simulation is scaled up, but it is more clear in the figure to show only a few particles.

3.9 The Actor Model

Actors support a highly reactive system. In fact, that is the way they work. When writing for an actor system, the programmer defines a set of reactions to specific messages. The important messages to understand in my work are the following:

- Particle message:

  This is a message for cells and contains a collection of particles. The cell will compare the particles to its own particles and add any collisions it finds (within the time-step) to its list of potentials.

- Progress report:

  This is a message that is sent to the grid. It contains information about a cell and can signify a number of things.

- Begin processing:

  The grid sends this to cells. When receiving this, the cell asks its neighbors to send back their first collision.

- Minimum message:

Cells send this to each other. It simply informs a neighbor at what time cell's minimum collision will occur.

- Finished message

  Cells send this to grid when they are done processing. The conditions for determining when this is true are discussed below.

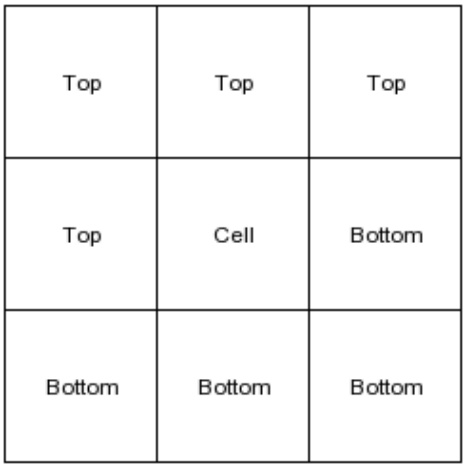

Figure 3.5: Demonstrates which neighbors belong to which group.

In this implementation of the pseudocode, I start with the grid, an actor in itself that contains a two dimensional array of cells. Cells are also actors (See figure 2.2). The grid begins the process by splitting the Global's list of particles by location and distributing the particles to the proper cells. It also tells each actor about its neighbors in two sets: Those neighbors above and to the left (I will call these "top neighbors"), and those neighbors below and to the right (I will call these "bottom neighbors"). See figure 3.5 explaining neighbors. Finally, it passes a message to every cell saying to find the initial potential collisions.

When the cells receive said message, they immediately run through their own list of

particles, looking locally for collisions contained entirely within themselves. Afterwards they send a particle message to each of their "top neighbors" with a copy of their particles, so that the neighbor can find potentials between the two cells. Note that the particle message only needs to be sent to top neighbors. This is because only one of the two involved cells in an inter-cell collision needs to know about the collision. I arbitrarily decided the owner would be the one above or to the left of the collision. The cells send the grid a progress report when they have processed a number of particle messages equal to the number they expect (four, in the majority of cases, but fewer if they are on a bottom or right edge).

This step acts as a conventional block accomplished with the grid. In order to avoid telling a cell to process while it is expecting particle messages and mixing up important steps, the grid keeps track of how many cells have sent the progress report. Only when it receives one from every cell does it begin the next step of the simulation.

The processing steps begin. Every cell asks its neighbors to reply with a minimum message. These responses are compared to its own minimum to decide whether it is the local minimum or not. If it is, it processes the collision, finding the particle's new velocities and sending a particle message to all neighbors. if it is not, it does nothing and waits. On any change in a cells potentials list, it sends another minimum message to all neighbors. This process continues until a cell has no potentials, and neither do its neighbors. Then and only then the cell will send a finished message to the grid. When the grid gets finished messages from all its children, it begins the next time step.

The bounced particle problem is handled by default in the actor model. Actors queue up actions, so any race conditions are handled automatically.

3.10 Challenges in the Actor Model

The biggest challenge in designing the actor model was determining a finished state. It is possible for a cell to have no potential collisions at one point in a time step and to gain some later on in the same step, as a particle might be bounced into the cell from a previous collision. To handle this, I use the same assumption that a particle cannot pass two cell boundaries in a step. A cell is only done if it contains no collisions and none of its neighbors do either. Then, within one time step, there can be guaranteed no change in potentials within the center cell.

Another challenge is one typical of hard body sphere collision simulation. Due to well known problems with floating point rounding in computation, particles sometimes find themselves intersecting during the processing of a collision. This caused my simulation to continually add and process collisions between the two intersecting particles, sinking into an infinite loop. The solution was simply to detect when two particles were overlapped, and to push them apart enough to rid myself of the issue but maintain most of the integrity of the simulation.

Chapter 4

Results

The following tests were all run on the Trinity Computer Science department owned "pandora02." It has 32 2.1 GHz cores. The single thread algorithm only takes advantage of one of the 32, so the speed of the machine will not be apparent until the parallelized tests come up.

The tests consisted of inputs crafted with a randomized algorithm. Particles were created within a given range in space, spaced evenly in a grid pattern. They were each given an evenly distributed random velocity, and pushed around a small random amount to deviate from the rigid grid shape. The random number generator was seeded per trial, so each algorithm was run on the same set of five tests. A trial consisted of ten runs per algorithm, starting with 16 particles (an arbitrary small starting place) and increasing by one hundred thousand particles until finishing with 1 million. In order to see the scalability of the parallel algorithms, larger tests were run on them. These tests were created with the same random algorithm, but ranged up to 3 million particles.

4.1 Single Thread Model

Table 4.1: The table of results for the single thread algorithm.

Single							
NumParticles	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Single Average**	**Std Dev**
16	141	143	141	141	142	1.416	0.894427191
100000	7850	7865	7848	7751	7871	7837	49.05609035
200000	14633	14493	14481	14436	14516	14511.8	73.74754233
300000	21194	21047	20996	21839	20866	21188.4	382.1862112
400000	29690	29761	29327	29734	28269	29356.2	632.6284059
500000	36468	36928	38328	35431	36857	36802.4	1041.486102
600000	45395	44071	45230	43214	43473	44276.6	997.1149884
700000	52793	54073	54807	53313	50945	53186.2	1466.890316
800000	64189	60667	58876	59014	64599	61469	2765.374206
900000	71741	92576	68030	73725	69694	75153.2	9972.207213
1000000	79973	83412	75258	82729	76925	79659.4	3551.480719

The single thread algorithm performed admirably. The purpose of the grid based algorithm is to turn collision detection into a linear algorithm, and this one was linear. It out-performed the actor model considerably.

4.2 Multithread, Non-Actor Model

Figure 4.2 shows the results for the non actor parallel model between 0 and 1 million particles. The multithread, multiqueue algorithm had linear performance between 0 and 1 million particles, and appears to be growing at a slower rate than both actor model and the single thread. While this model was the fastest, I had hoped it would be faster by a larger margin, maybe something around ten fold faster rather than three fold. This model was also the most reliably efficient, with the lowest standard deviation of all three.

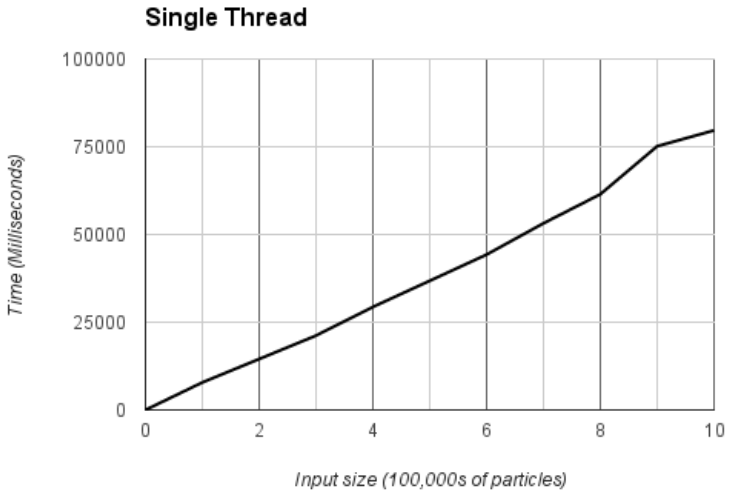

Figure 4.1: The results of the single threaded control algorithm.

Table 4.2: The table of results for the multi thread, non actor algorithm.

Multi							
NumParticles	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Non Actor AVG	Std Dev
16	4240	92	102	132	135	940.2	1844.738247
100000	2671	2714	2645	2555	2713	2659.6	65.37430688
200000	5071	5206	5600	4869	5923	5333.8	424.1034072
300000	6412	5828	5588	5950	6205	5996.6	321.5350681
400000	9071	7870	7798	7712	7640	8018.2	594.9018406
500000	13548	10200	11860	9771	9535	10982.8	1697.578481
600000	15190	11766	12004	11905	12016	12576.2	1464.592162
700000	18699	13990	14145	14631	15613	15415.6	1941.78006
800000	21867	17114	16613	16669	16912	17835	2262.831965
900000	27573	27515	20079	25135	25244	25109.2	3048.767489
1000000	30133	23856	29773	23409	22944	26023	3604.293481

4.3 Actor Model

The actor model's results for the smaller range are shown in figure 4.3. It grows linearly, but is far slower than both of the other two models. I think the majority of issues with the

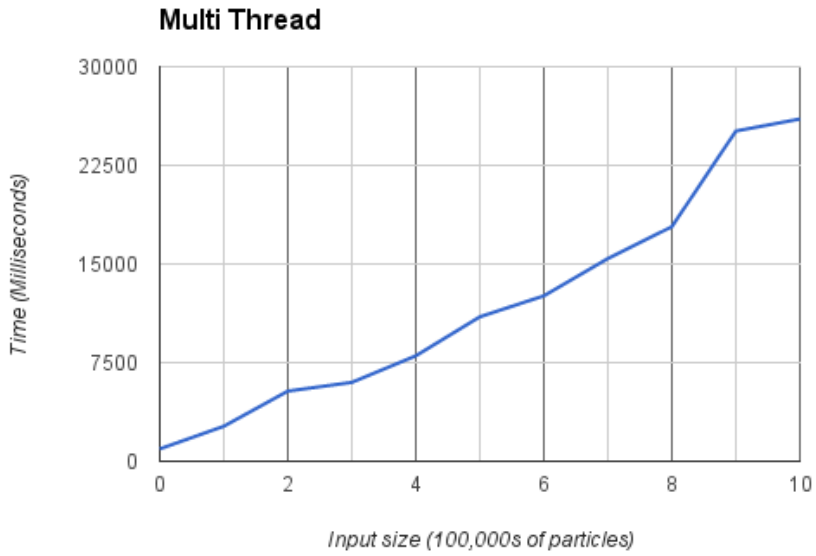

Figure 4.2: The results of the multi threaded, non actor algorithm.

actor model stem from the fact that I took no shortcuts in programming this algorithm, making it as pure as I could. There is no shared memory when there are opportunities to use shared memory, and there is a possibility that this algorithm could run successfully without the blocking that it currently has in place. The overhead of an actor system also probably contributes to the speed of the actor model. Scalability of the parallel models will be discussed in a later section.

4.4 Comparisons

The actor model is by far the slowest, as we can see in figure 4.4. In order to understand why it's so slow, I counted the number of messages being passed between the actors. In a simulation with 200,000 particles, close to 13 million messages were passed over the

Table 4.3: The table of results for the actor model algorithm.

Actor							
NumParticles	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Actor AVG**	**Std Dev**
16	232	252	245	178	265	234.4	33.70904923
100000	10101	9917	9787	10415	10188	10081.6	243.2772081
200000	19561	21550	20462	20760	21043	20675.2	740.7156674
300000	28573	29331	29365	27855	29584	28941.6	717.6599473
400000	46005	43765	40593	43841	42645	43369.8	1972.209725
500000	56492	56517	60347	52441	55501	56259.6	2827.732448
600000	78241	77622	73488	79494	77525	77274	2257.287642
700000	90986	89651	85749	85166	87269	87764.2	2499.845335
800000	100236	109246	106318	107066	103333	105239.8	3507.886144
900000	124692	120616	119240	123152	122507	122041.4	2142.166147
1000000	138521	140688	136829	143252	132643	138386.6	4012.532903

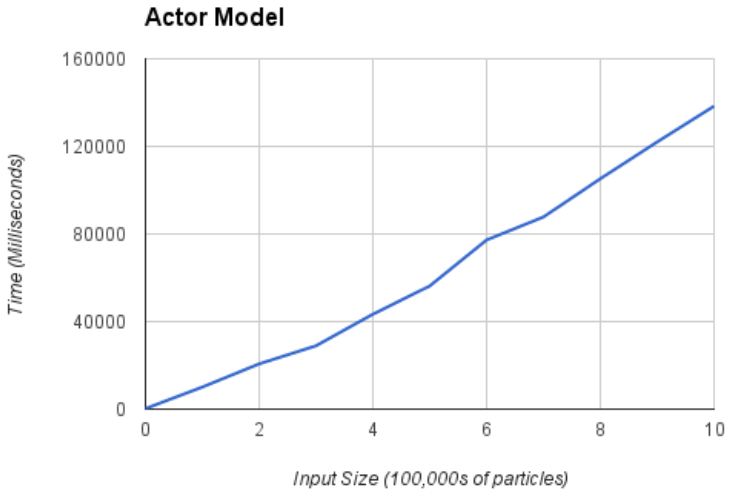

Figure 4.3: The results of the actor model algorithm.

runtime. Only 55,000 collisions were found. Specifically, there were 237.8 messages passed per collision and 65.7 passed per particle. I believe this overhead is the reason the actor model is slow. Ideas for solutions will be discussed in the next chapter.

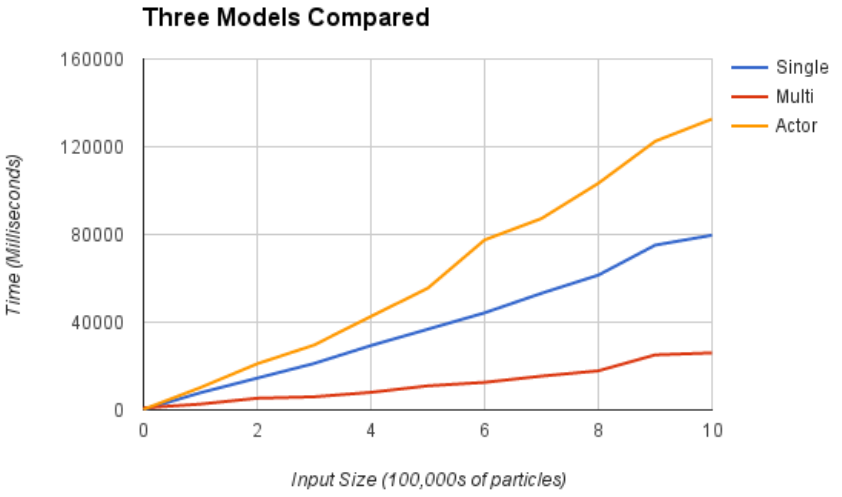

Figure 4.4: The results of all three algorithms overlayed.

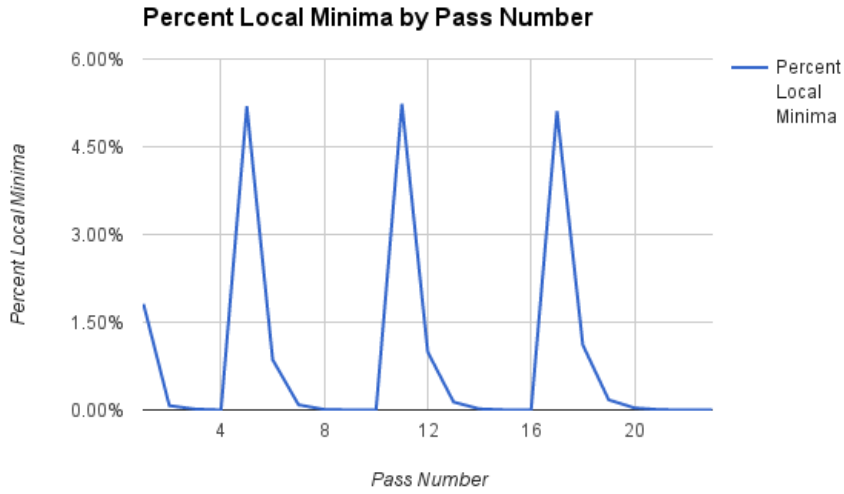

Figure 4.5: Cells that are local minima divided by the number of cells per pass number.

The multithreaded, non actor model is the fastest, but not by an amount that we might expect running the processes on a 32-core machine. I ran a test to examine the percentage of cells per pass that were local minima. One would expect this percentage to decrease during a step- the first pass processes the highest number of potentials, and as the number of potentials decreases, so does the percentage of local minima. The results I found reflected that exactly. Examine figure 4.5. It shows the percentage of cells that are local minima by pass. Changing the density of particles per cells could raise the percentages above their current maximum of five percent, a low value showing the algorithm is doing extra work.

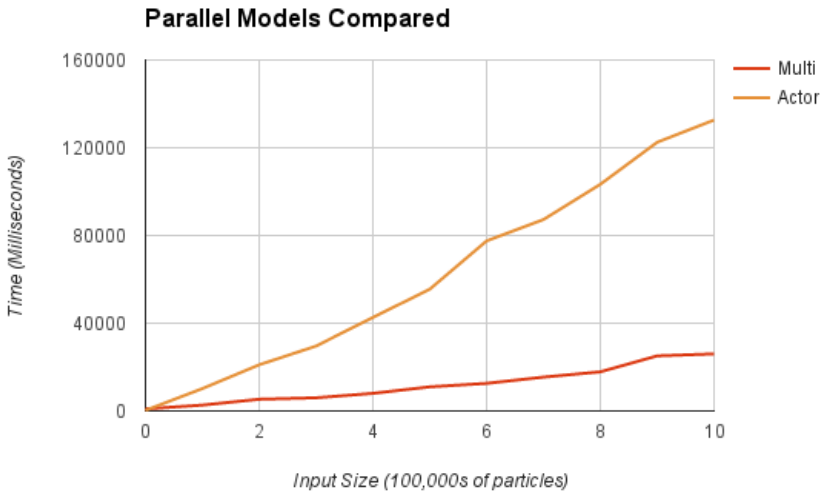

Figure 4.6: The results of the two multi threaded control algorithms.

4.5 Pushing the Parallel

In order to see the limits of the actor model, I ran the two parallel models through some considerably bigger tests, going up to three millions particles. They both scaled admirably.

I believe, however, that there would be a point at which the actor model would have to begin doing a lot of garbage collecting, leading to a decrease in efficiency. However, that point is past 3 million particles.

Table 4.4: The table of results for the large tests on the parallel models.

Timing #	100000s of particles	Trial 1	Trial 2	Actor AVG	Trial 1	Trial 2	Non Actor AVG
1	0	23	26	24.5	669	4416	2542.5
2	2	38027	18527	28277	3794	8158	5976
3	4	42565	76530	59547.5	6878	11662	9270
4	6	66898	72121	69509.5	15309	15845	15577
5	8	96540	99068	97804	17577	31445	24511
6	10	129888	137843	133865.5	22440	38640	30540
7	12	172992	166116	169554	29002	52032	40517
8	14	218004	214518	216261	61050	36339	48694.5
9	16	327169	315567	321368	49437	50330	49883.5
10	18	361596	344613	353104.5	73840	41784	57812
11	20	406120	386807	396463.5	56560	58972	57766
12	22	424843	429599	427221	68417	67100	67758.5
13	24	466301	478751	472526	74645	78079	76362
14	26	508940	504169	506554.5	83169	82158	82663.5
15	28	535457	559071	547264	85585	98517	92051
16	30	604768	596611	600689.5	149786	138092	143939

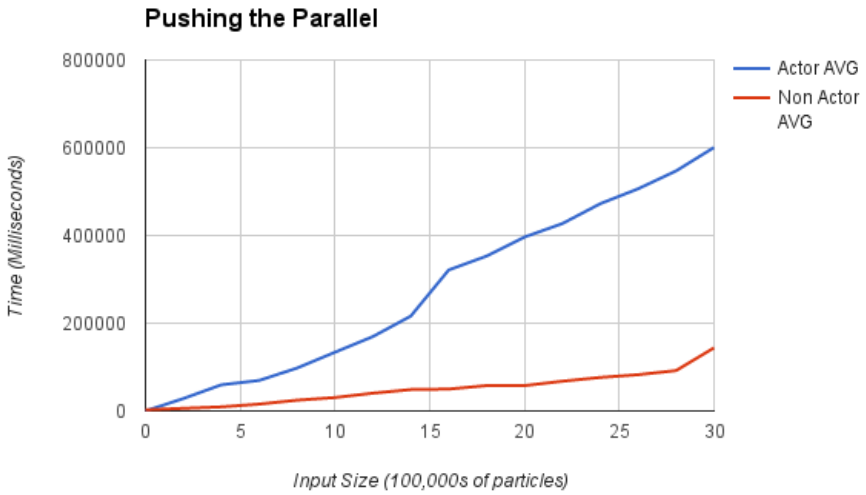

Figure 4.7: The results of the two multi threaded control algorithms.

Chapter 5

Conclusion

This research shows that such a strict implementation of the actor model is not a very efficient way to parallelize the collision detection algorithm. There are some possible effects and how many many actors are in a system as well; The reduction in speed could be because of the sheer quantity of actors (At 3 million particles there are close to 1800 actors running simultaneously) all running on 32 cores. An amendment to the algorithm to allow more cells per actor could be an interesting speed boost.

This research also created a new method of multi-threading the original pseudo code, at least in the scope of Dr. Mark Lewis's work. The model that skips to every 3rd row and computes those rows simultaneously, the non-actor method, proved much faster than the actor method. Comparing this method to a more conventional blocking method could also yield interesting results.

5.1 Further Projects

While this work shows the actor model under performing compared to conventional methods, there are projects that could be done to find further information. Here are some examples of possible further research:

1. Comparison of the non actor, multiqueue model to an actor model that follows the guidelines less strictly. I believe careful application of shared, mutable memory to an actor system would boost the efficiency of the actor model considerably.

2. This research was all tested on the same machine with the same amount of available cores. A comparison of the parallelization techniques with varying number of cores would be interesting. How does the ratio of actors to cores effect performance?

3. As discussed in the third chapter, I solved the bounced particle problem in this work by processing every third row simultaneously rather than using blocking. A comparison of the these two ideas would show the possible performance benefits of this algorithm over the blocking technique.

4. Before some small efficiency fixes were added, the actor model at high input size had a very sharp increase in speed in the same place through two trials. This outdated result is shown in the appendix in figure A.1. An interesting future project would be to see if this increase was due to machine constraints or due to caching problems, or somehow due to actors themselves.

Bibliography

[1] Shams M. Imam and Vivek Sarkar. Integrating task parallelism with actors. *ACM / SIGPLAN Notices*, 47(10):753, 2012.

[2] Zachary Langbert and Mark C. Lewis. Processing hard sphere collisions on a gpu using opencl. In *PROCEEDINGS OF THE 2014 INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS*, volume 1, pages 35–41. CSREA Press, 2014.

[3] Mark Lewis and Berna L Massingill. Multithreaded collision detection in java. In *PDPTA*, pages 583–592, 2006.

[4] Martin Odersky Phillip Haller. Scala actors: Unifying thread-based and event-based programming. 2008.

[5] David E Stewart and Jeffrey C Trinkle. An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction. *International Journal for Numerical Methods in Engineering*, 39(15):2673–2691, 1996.

[6] Derek Wyatt. *Akka Concurrency*. Artima Press, 2013.

Appendix A

Outdated Figure

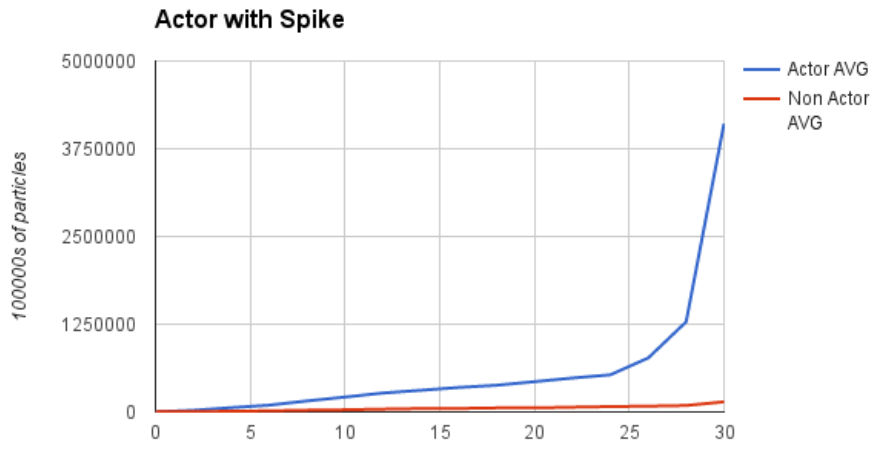

Figure A.1: The old results of the two multi threaded control algorithms.