UNIVERSITY OF ILLINOIS IN URBANA-CHAMPAIGN

CS 598 WG Project Report

Antonio Maria Franques Garcia (franque2)
Thomas Shull (shull1)
May 11, 2018

This is the project report for Antonio Franques's and Thomas Shull's CS598 WG Class Project.

1 INTRODUCTION

It is a well-known problem within the computing community that programming parallel systems is overly complicated. Forcing programmers to consider all of the nuances of a parallel system places a huge burden on the programmer and often results in applications that are buggy, have suboptimal performance, and require countless programmer hours to develop and maintain.

However, recently many new frameworks and languages designed to simplify the process of creating parallel programs have become popular. The goal of these frameworks is to allow programmers to use a simpler, more intuitive programming model, helping to increase programmer productivity and the likelihood of writing correct code. The frameworks are able to accomplish these goals by abstracting away many of the low-level details of modern machines. Instead of being concerned about hardware and OS-specific details such as the number of cores on a machine, the OS threading model, or how data is passed between different threads, these frameworks encourage the programmer to have an abstract model of a machine and leave it up to the framework to efficiently map the model down to the hardware being used. In addition to the framework abstraction, these frameworks are mostly written in Managed Languages such as Java, Scala, and C#, where the runtime manages memory allocation and performs many runtime checks to ensure the correctness and safety of the program execution.

Unfortunately, these levels of abstraction which reduce the programmer's burden are not free, but instead impose an overhead on the execution time. Furthermore, these levels of

abstraction can make it more difficult to reason about program performance, as it is up to the framework and compiler to map the high-level code to the low-level operations supported by processors.

To better understand these tradeoffs, for our class project we have decided to implement a parallel program in one of the most popular of these modern frameworks, Akka [1], which is based on the actor model. in the Actor model multiple isolated computation nodes communicate with each other through asynchronous message passing. We evaluate its performance against a performance expectation model to see the amount of overhead these frameworks impose on a system and how the levels of abstraction affect the accuracy of a performance model. To further explore these overheads, we also leverage hardware performance counters and dynamic binary instrumentation to observe how the program is being mapped to the machine. Finally, based on our analysis, we modify our original program and discuss the improvements we observe.

2 Background

2.1 Actor Model

The actor model was developed in the 1970s by Carl Hewitt [8]. This model realizes concurrency through the creation of many different entities, known as "actors". Each actor has its own private data and executes in isolation from the rest of the system. Communication is modeled by passing asynchronous messages between different actors. An actor's execution is message-driven, or, in other words, a given actor's execution is the reaction to the reception of a message. In parallel, each actor processes messages, which may invoke the actor to modify private state and also potentially send new messages to other actors. Each actor processes messages one at a time via the following process:

1. Pop message off of actor's personal message queue.
2. Match message to behavior to invoke.
3. Perform behavior. This can modify data private to the actor and potentially result in messages being sent to other actors and/or itself.
4. Check if there are more messages in queue. If so, return other step 1. Otherwise, remain idle until a message arrives

A program finishes once all actors are idle and there are no more transient messages within the system.

This model simplifies programming parallel systems in many ways. First, the actor model is meant to be intuitive for programmers to understand. The actor model tries to imitate the way people normally communicate with each other. Like a person, each actor is independent and capable of being autonomous. Furthermore, in the same way that people can request for another person to complete a task, an actor can message another actor to perform an action. This programming model is natural for programmers and makes it easy to reason about parallel code.

Another way the actor model simplifies parallel programming is that one does not have to worry about data races, as there is no shared data. Since each actor has its own private data and information can be exchanged only via messages, it is easy to reason about who has "ownership" or is accessing a specific piece of information at a given program state.

Finally, the actor model creates abundant parallelism that can easily be scaled as the requirements of an application grows. As each actor can run in parallel, many normal bottlenecks or critical paths within a program are eliminated by this design. In addition, it is possible to replicate or create more actors dynamically throughout the running on the program to match the needs of the application at any given time. This is possible due to the transparency implicit in the model. When an actor sends a message to another actor to complete a task, it does not care how or which actor performs the task, only that the task is completed. This allows for the number of actors to be scaled based on the load of the system or the number of processing nodes available in the underlying system.

2.2 Scala & Akka

Scala is a language created by Martin Odersky in 2004 [10]. Like Java, Scala is complied down to Java bytecode which is then executed on top of a Java Virtual Machine (JVM). However, Scala is meant to improve on several of Java's shortcomings, including better support for functional programming, providing a limited type inference while still being strongly typed, and the ability to write more concise code patterns. Scala can be considered a superset of Java – within Scala one can write traditional object-oriented code that looks very similar to Java code. However, in Scala one can also write in a functional manner that was originally not possible within Java. Over the last several years, Java in fact has incorporated features originating from Scala. Scala's influence in the programming community has continued to grow over the last several years and it is now the language of choice for many programmers within the Big Data community.

Over the years, many frameworks have been developed to bring the actor programming model to a variety of languages. Currently one of the most popular frameworks implementing the actor model is Akka [1].

According to Wikipedia [6]:

> Akka is an open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM. Akka supports multiple programming models for concurrency, but it emphasizes actor-based concurrency, with inspiration drawn from Erlang.

Akka is currently one of, if not the most popular framework based on the actor model. Many companies, including Walmart, Huffington Post, Intel, and Samsung [2] are using Akka in projects.

Akka itself is written in Scala, but it allows for one to write code adhering to the actor model in all JVM-based languages. To lower the overheads of actor execution on small systems, Akka relaxes several components of the actor model. Most notably, in Akka data is allowed to be shared between multiple actors and an actor can query any other actor in the system, which

lowers the amount of transparency and isolation of individual actors. In the code we write, we do take advantage of the sharing of data to limit the copies of read-only data (which reduces memory pressure and better utilizes the cache), but we do not take advantage of the ability to send messages to any other actor in the system, and instead we pass messages to other transparent actors, so that our program is more scalable.

2.3 Java Virtual Machine

A Java Virtual Machine (JVM) is a program used to execute Java bytecode. The JVM has the responsibility of running the Java bytecode on the native machine and also providing all of the runtime support specified in the Java bytecode standard (garbage collection, reflection support, exception handling).

Traditionally JVMs Just-in-Time (JIT) compile Java bytecode down to machine code for better performance. In addition, in order to better profile the bytecode's behavior and optimize for the common case, code is not immediately JITted, but instead is run within an interpreter. The time it takes to profile the code and then generate machine code can be very time consuming. Hence, for JVM-based languages the application must endure a long warmup period before peak-performance is attained.

The most popular JVM implementation is HotSpot [9], an open source project that is the basis for Oracle's JVM implementation. For this project we use Hotspot as the execution engine for our code.

2.4 Direct Gravitational N-Body Simulation

An N-Body Simulation is a categorization for programs where each of the N elements in the system requires information about system's other $N-1$ elements. This model can be used to model many features within the physics domain. In the Direct Gravitational N-Body Simulation problem, the force of gravity felt by each of the N particles of the system is evaluated over time. At each timestep, initially the force felt on each particle is calculated. Next, each particle's new velocity is calculated. Finally, the particle is moved to its new position based on the granularity of the timestep. As the particles move throughout the simulation, the force inflicted by other particles in the system changes, causing the calculations to be repeated at each timestep.

3 Benchmark Implementation

3.1 Coding Practices and Framework

We focused on making our code very easy to understand. Our code is written in a very functional manner and is very configurable via an akka configuration file. Via this file one can adjust the number and distribution of planets, which type of instrumentation to perform, and various other runtime parameters.

We split our code into three files: one to initialize the execution, one which is in charge of the hierarchy, and the final file contains the force calculations necessary for the planets.

3.2 N-BODY

Our N-body implementation can be broken down into three components: the universe hierarchy, the message passing patterns, and the planets force and velocity calculations. Below we describe the implementation of each of these components

3.2.1 HIERARCHY

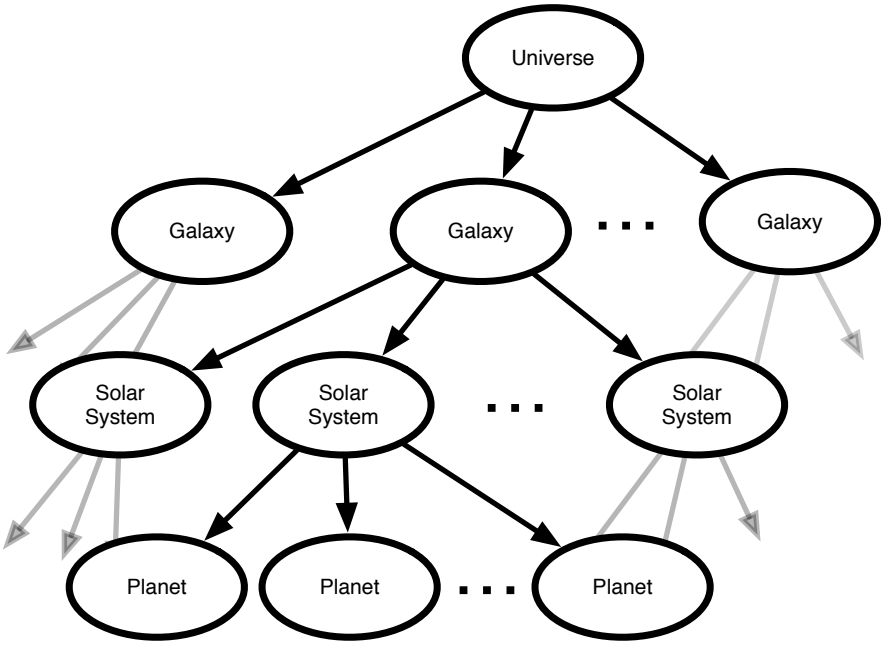

Figure 3.1: N-Body Simulation Hierarchy

We structure our N-body simulation to mirror the setup of the actual universe. Our simulation is broken down into an universe, galaxies, solar systems, and planets. Each galaxy can have multiple solar systems, and each solar system can have multiple planets; however, there is only one universe. Figure 3.1 shows a high-level diagram of how we arrange these elements in our simulation. Each of these elements is its own actor and has references only to its parent and children. In other words, actors that are not either a given element's parent or children are unknown to the element. This helps to make our simulation very adjustable and scalable. We are easily able to adjust the number of galaxies, solar systems, planets, etc.

As mentioned above, our simulation is very configurable. Via our akka configuration we are able to adjust the number of galaxies, the number of solar systems per a galaxy, and the number of planets per solar system. Furthermore, instead of having a perfectly balanced hierarchy (the same number of solar systems for every galaxy, etc.), we also allow for the number of children for each component to be either a bounded random number or to be based on a normal distribution with a configurable mean and variance.

3.2.2 Message Passing Patterns

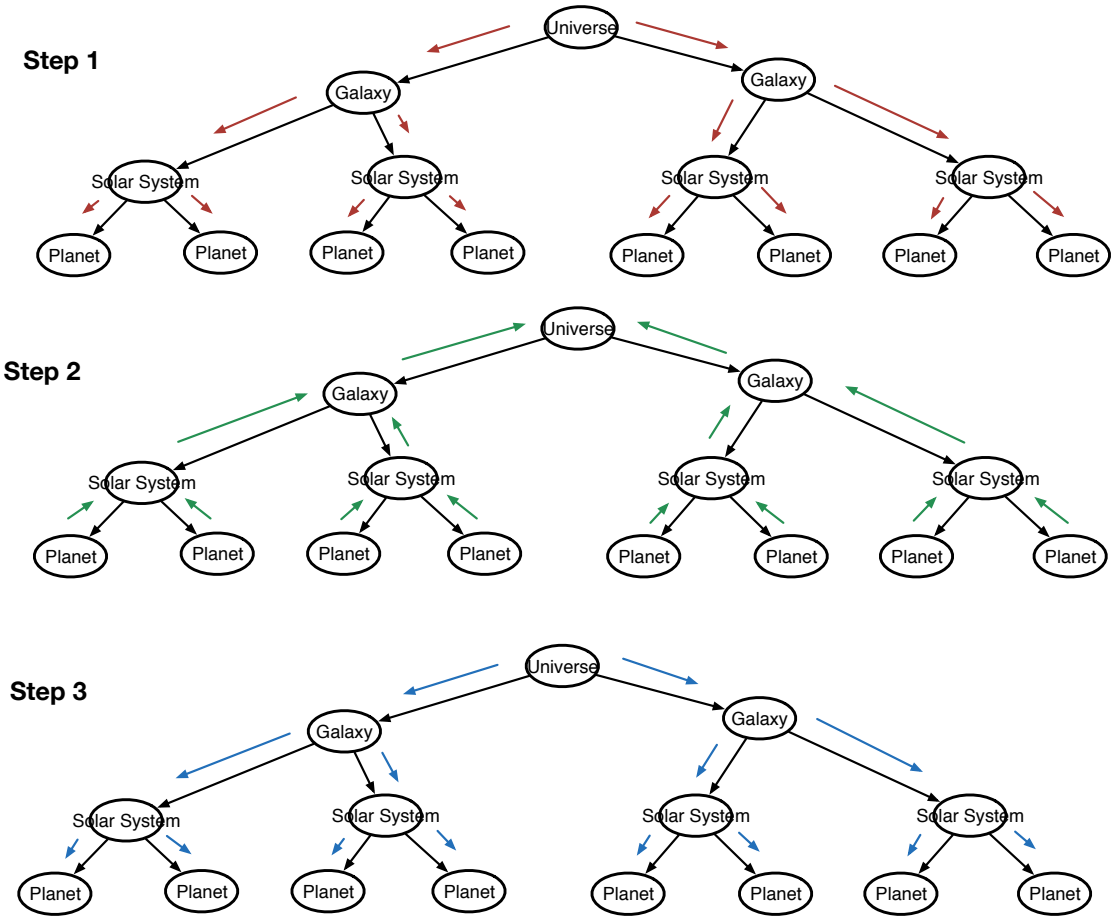

Figure 3.2: N-Body Simulation Hierarchy

In the N-body simulation, during each step, every planet must be informed of the position of every other planet so that their new position can be factored into the force calculation. While there are many ways to accomplish this goal using akka, for our simulation we choose to take advantage of our hierarchy to reduce the number of messages sent.

At the start of each step, the universe will send a message to each of the galaxies to request the position of every planet within each galaxy. Upon receiving this message, then each galaxy sends a message to all of their child solar systems, and all solar systems send a message to each of its child planets. Once this is completed, the chain of messages is reversed, with each planet sending a message to its parent solar system, etc. As the messages containing the planet information propagate up the hierarchy, the information at each level is collected and aggregated into a single message to be sent to the tier above.

Once the universe receives a response to its request for the position of all planets, the universe then propagates a message with this information down the hierarchy until every planet is given access to the list of all planets.

Note that this list of all planets will include the planet itself, not just a list of all other planets as needed for the force calculation. However, in the force calculation, the result of a planet calculating the force on itself will be 0, so it is does not alter the calculation.

Figure 3.2 shows how messages propagate throughout each step of the simulation. In the first part, the initial message requesting the position of planets is being propagated down the hierarchy In the next part, the responses are climbing up the hierarchy while being aggregated at each step. Finally, in the last part, a message containing the list of all planet locations is sent down the hierarchy again.

3.2.3 Planet Calculations

To calculate the position of a planet for the next timestep, several steps are needed. First, the force of all other planets in the system on the planet must be calculated. This equation for the force in one direction is the following:

$$F_{i_x} = \sum_{j=1}^{N} F_{ij_x} \tag{3.1}$$

$$F_{ij_x} = \frac{Gm_i m_j (d_{i_x} - d_{j_x})}{|p_i - p_j|^3} \tag{3.2}$$

where G is the gravitational constant, m_i and m_j are the masses of the planets of interest, $(d_{i_x} - d_{j_x})$ is the distance between the planets in the x direction, and $|p_i - p_j|$ is the magnitude of the distance between the planets. This magnitude is calculated by the following equation.

$$|p_i - p_j| = \sqrt{(p_{i_x} - p_{j_x})^2 + (p_{i_y} - p_{j_y})^2 + (p_{i_z} - p_{j_z})^2} \tag{3.3}$$

Once the force has been calculated, the next step is to calculate the acceleration of the planet. This equation is below:

$$F_x = m * a_x \tag{3.4}$$

$$\text{acceleration} = a_x = \frac{F_x}{m} \tag{3.5}$$

As shown by the above equation, it is easy to derive the acceleration from the planet's mass and force on the planet. Next, from the acceleration, the new velocity of the planet can be calculated. The equation is below:

$$v_{x_{new}} = v_{x_{orig}} + a_x * t \tag{3.6}$$

Finally, from the velocity the new position of the planet can be calculated. The equation is below:

$$p_{x_{new}} = p_{x_{orig}} + v_{x_{new}} * t \tag{3.7}$$

3.3 Code Availability

In all, our code is 729 lines of high-level functional Scala and Java code. The code we wrote is available to download from `https://drive.google.com/file/d/1wIlvImFtbuIK8cQ8dBx_ArwUoozd8pmn/view?usp=sharing`. Note that by using functional constructions and the akka framework we were able to create a parallel application with relatively few lines of code.

4 Performance Expectation

We divide the calculation for the performance expectation into two parts: the memory footprint size and the arithmetic cost of performing the calculation for the next position of a planet. Note that we assume the cost of message passing is only the cost of storing the payload of the message; in other words, we assume the cost of pushing or popping messages from an actor's mailbox is free. In addition, whereas our code is multi-threaded, are performance model is based on the single-threaded model. As we only use up to eight core in our evaluation, this approximation should not significantly affect the accuracy of our model.

For the following equations, let G be the number of galaxies, S be the number of solar systems per a galaxy, and P be the number of planets per a solar system. For this calculation let us assume that S and P are the same for each galaxy/solar system. Finally, let N_P be the total number of planets. N_P can be calculated via G, S, P as:

$$N_P = G * S * P \tag{4.1}$$

Likewise, the equation for the number of solar systems, N_S, is:

$$N_S = G * S \tag{4.2}$$

We make the following assumptions about the cost of various operations:
Cost of Double Computation (C) = 1/3.40 GHz = $2.94e^{-10}$ sec/op
Cost of Double Subtract/Add (A) = 1/3.40 GHz = $2.94e^{-10}$ sec/op
Cost of Double Multiply (M) = 1/3.40 GHz = $2.94e^{-10}$ sec/op
Cost of Double Divide (D) = 1/3.40 GHz = $2.94e^{-10}$ sec/op
Cost of Double Square-Root ($Sqrt$) = 1/3.40 GHz = $2.94e^{-10}$ sec/op
Cost of Load (Ld) = 9835.8 MB/s (from STREAM)
Cost of Store (St) = 9835.8 MB/s (from STREAM)

4.1 Memory Footprint

We divide the memory footprint into two potentially overlapping categories, the load footprint and the store footprint. We assume that even if an address is read/written multiple times, that only the first load and store to that address will incur an overhead, as afterwards the value will be stored in cache. In addition, we assume that this overhead will occur for each timestep.

For each of the footprint categories, there are two main sources: the footprint of the (Universe, Galaxy, Solar System, Planet) fields accessed, and the footprint of the message payload sent to gather the initial position of each planet at the start of the timestep.

4.1.1 Object Field Size

As explained previously, each actor must have links to each actor it wants to send a message to. In the object model, this means that each object must have a field for each of its connections. Table 4.1 breaks down the count for each object type.

Object Type	# Children	# Parents
Universe	G	0
Galaxy	S	1
Solar System	P	1
Planet	0	1

Table 4.1: Number of references to other actors by each tier of hierarchy

Note that these fields are read-only, as aside from initialization, each actor only needs to read the location of each connected actor during the timestep. In addition, we assume that these fields contain a pointer, and hence are machine word size, or 8 bytes in our environment. The total load footprint can be calculated by multiplying each object type sum by the number of objects of that type. The calculation is shown below:

$$\text{Load Footprint} = 8 * (1 * (G + 0) + G * (S + 1) + N_S * (P + 1) + N_P * (0 + 1)) \tag{4.3}$$
$$= 16(G + N_S + N_P) \tag{4.4}$$

While the Universe, Galaxies, and Solar Systems only contain fields linking the components, Planets also have fields containing information about their mass, position, and velocity. While a planet's mass is solely read each timestep, a planet's position and velocity must be both read and updated each timestep. The mass is stored as a double, while the position and velocities are double tuples of size 3. Hence, the footprint of the planet characteristics is:

$$\text{Load Footprint} = 8 * N_P * (3 + 3 + 1) \tag{4.5}$$
$$= 56N_P \tag{4.6}$$

$$\text{Store Footprint} = 8 * N_P * (3 + 3) \tag{4.7}$$
$$= 42N_P \tag{4.8}$$

Adding up the two categories, the footprints are the following:

$$\text{Load Footprint} = 16G + 16N_S + 72N_P \tag{4.9}$$

$$\text{Store Footprint} = 42N_P \tag{4.10}$$

4.1.2 Message Payload Size

As explained in Section 3.2, the process of collecting the position of the planets involves three phases: first a message propagating from the universe to request the planets position, next the planets' position being sent back to universe, and finally the list of planet positions being sent down again to the planets.

Note that the message payload of the first and third phases are minimal. The first step only needs to send the behavior to invoke, while the third phase only needs to send a pointer to the data structure created in the second phase. Since the footprint of these phases is substantially smaller than that of the second phase, we focus on the footprint of the second phase.

For the second phase, each level of the hierarchy combines the planet positions from the tier below and sends the resulting list to the tier above. The final result is a list of all of the planet positions. Below we discuss the memory footprint at each level of the hierarchy.

Planet Level. Each Planet sends its position and mass to its parent solar system. The mass is sent as a double value and the position as a pointer to the reference. Hence a total of two fields are sent.

Solar System Level. At this level of the hierarchy, the planet info is aggregated into a list. We assume that each element contains two fields: a pointer to the planet info, and a pointer to the next element in the list.

Galaxy Level. At this level the planet lists from the solar system level are aggregated into larger lists. In a list, this involves simply changing the next field of an list item. This means that at the Galaxy Level the memory footprint is not expanded. Note that, as we are using immutable data structures, when combining lists, some replication of list items may be necessary. However, we are ignoring this in our modeling.

Universe Level. Finally, the universe aggregates the galaxy's list into larger lists. Like the galaxy level, it also does not increase the footprint.

Therefore, only the planet and solar system levels contribute to the footprint. Note that all of this footprint is both read and written. The footprint of the planet level and solar system levels are shown below:

$$\text{Planet Load/Store Footprint} = N_P * 8(2) \tag{4.11}$$

$$\text{Solar System Load/Store Footprint} = N_S * 8(2) \tag{4.12}$$

4.1.3 Total Footprint Size

Adding of the footprint sizes from the previous two sections together, we arrive at the following totals for the load and store footprints:

$$\text{Load Footprint} = 16G + 32N_S + 88N_P \tag{4.13}$$

$$\text{Store Footprint} = 16N_S + 58N_P \tag{4.14}$$

4.2 Computation Cost

4.2.1 Force Calculation

The first part of the n-body computation cost is calculating the force. Note that the force must be calculated in 3 all directions. The force equation was shown previously in equation 3.1. For N_P planets, then this means this equation has the following number of calculations:

$$\text{Cost(Total Force in one direction)} = (N_P - 1) * A \tag{4.15}$$

Note in the equation we use the symbol for Subtract/Add (A) defined previously. This overhead represents adding all of the individual forces together. Next we must determine the computational cost of equation 3.2. F_{ij_x} has the following performance cost:

$$\text{Cost}(Fij_x) = 3 * M + A + \text{Cost}(|p_i - p_j|) \tag{4.16}$$

The magnitude, $|p_i - p_j|$, has the following performance cost (according to equation 3.3):

$$\text{Cost}(|p_i - p_j|) = 3 * A + 2 * A + 3 * M + Sqrt \tag{4.17}$$

When combining all of these equations, the total cost of computation in one dimension is:

$$\text{Cost(Force in one direction)} = (N_P - 1) * A + (N_P - 1) * (3 * M + A + (3 * A + 2 * A + 3 * M + Sqrt)) \tag{4.18}$$

$$= (N_P - 1) * (6M + 5A + Sqrt) \tag{4.19}$$

This equation must be repeated for each direction. However, magnitude only needs to be calculated once. Hence, the total equation is:

$$\text{Cost(Force in all directions)} = 3 * (N_P - 1) * (6M + 5A + Sqrt) - (2 * (N_P - 1) * (5A + 3M + Sqrt)) \tag{4.20}$$

$$= (N_P - 1) * (12M + 5A + Sqrt)) \tag{4.21}$$

Each planet must perform this calculation, so the total cost is as follows:

$$\text{Total Cost of Force} = N_P * (N_P - 1) * (12M + 5A + Sqrt)) \tag{4.22}$$

4.2.2 Position Calculation

Once the force has been calculated, it must be used to determine the new position for a given planet. This involves calculating the acceleration (equation 3.5), the new velocity (equation 3.6), and finally, the new position (equation 3.7) in each direction.

Calculating the new acceleration for a given planet in one direction requires one division. Both the velocity and position calculation need an add and multiple. Hence, the total calculation of calculation the position of the planets, given that the forces have already been calculation, is shown below:

$$\text{Cost(Position)} = 3 * N_P * (D + M + A + M + A) \tag{4.23}$$
$$= N_P * (3D + 6M + 6A) \tag{4.24}$$

4.2.3 Total Computation Cost

Adding the equations derived in the previous two sections leads us to the total computation cost. This combined cost is shown below:

$$\text{Cost(Total)} = N_P * (N_P - 1) * (12M + 5A + Sqrt)) + N_P(3D + 6M + 6A) \tag{4.25}$$
$$= N_P^2 * (12M + 5A + Sqrt) + N_P * (3D - 6M + A - Sqrt) \tag{4.26}$$

4.3 Estimating Performance Times

Given our footprint and computational costs, we can now estimate the time taken to complete a timestep for a given number of planets. For the sake of this calculation, let us assume that $n = N = S = G$, so that $N_S = n^2$ and $N_P = n^3$. The calculation is shown below:

$$\text{Expected Step Time (sec)} = N_P^2 * (12M + 5A + Sqrt) + N_P * (3D - 6M + A - Sqrt) + Ld^{-1} * (16G + 32N_S + 88N_P) + St^{-1} * (16N_S + 58N_P) \tag{4.27}$$
$$= N_P^2 * (12M + 5A + Sqrt) + N_P * (3D - 6M + A - Sqrt) + Ld^{-1} * (16n + 32n^2 + 88n^3) + St^{-1} * (16n^2 + 58n^3) \tag{4.28}$$
$$= (18n^6 - 3n^3) * 2.94e^{-10} + (16n + 48n^2 + 146n^3)/(9835.8e^6) \tag{4.29}$$

5 Testing Setup

In this section we describe the infrastructure used to test our application.

5.1 Machine and Test Information

For our testing, we use the OpenJDK Runtime Environment as our VM. Specifically, we use OpenJDK version 1.7.0_111, which uses the HotSpot compiler build (24.111-b01). The machine we use for all test has one Intel i7-6700 Skylake processor running at 3.4GHz with 32GB of DDR4 memory and is running Ubuntu 14.04.3, kernel version 4.2.0-41. This processor has 4 2-threaded cores. We leave prefetching, hyperthreading, turbo-boosting, and dvfs enabled so that maximum throughput can be attained.

To measure the performance of our n-body simulation, we test our program with many configuration sizes. Specifically, we adjust the number of planets by adjusting the branching degree for the universe, solar system, and galaxy actors. For each test, we set each of these components to have the same branching degree so that the number of planets for branching degree N is N^3. We test sizes of the branching degree from 2 (8 planets) to 40 (64,000 planets)

Event name	Abbrev.	Description
cycles	CYC	number of cycles execution
instructions	INS	number of instructions execution
cache-misses	CacheM	number of cache misses
cache-references	CacheR	number of cache accesses
LLC-loads	LLC-L	Number of loads from the Last-Level Cache
LLC-load-misses	LLC-LM	Number of misses of the Last-Level Cache
LLC-stores	LLC-S	Number of stores to the Last-Level Cache
LLC-store-misses	LLC-SM	Number of store misses to the Last-Level Cache
context-switches	CS	Number of context switches performed
cpu-migrations	CPUMig	Number of Process Migrations

Table 5.1: Description of Hardware Event Counters

in steps of two. We run each test for approximately 20 minutes and observe the number of steps completed. For larger test sizes where each step take on the order of minutes we specify to the application the number of steps to complete before terminating to help the fidelity of our results.

5.2 Dynamic Binary Instrumentation

To analyze the composition of machine instructions generated for our program we decided to use Intel's Software Development Emulator (SDE) [4], a tool which is able to provide many types of analysis through dynamic binary instrumentation.

We leverage SDE's footprint tracking and instruction recording features in our analysis. Specifically, we have SDE record the dynamic footprint touched during one timestep for various branching degrees. In addition, we also use SDE to count the dynamic instructions by category for a single timestep.

For this evaluation we monitor the execution of the code for one timestep. We only test the branching degrees from 2 to 20, in increments of 2. We found that due to the overhead of instrumentation larger sizes where unable to complete due to insufficient memory.

5.3 Performance Counters

To further analyze and justify the performance of our code we also decided to observe various machine performance counters during the execution of our code. Modern processors include many machine counters which can be use to evaluate the performance of a system. Processors include counters for events such as the number of instructions executed, the number of branch prediction misses, the hit/miss frequency of various levels of cache, and the number of instructions committed per a cycle.

To read these counters we use the tool perf-stat [3]. This tool is included within the Linux kernel and reads a processor's performance counters via accessing machine-specific registers (MSRs).

Category	Event Counters Enabled
CPI	CYC, INS
Memory	CacheM, CacheR, LLC-L, LLC-LM, LLC-S, LLC-SM
System	CS, CPUMig

Table 5.2: Hardware Event Counters Groupings

Table 5.1 shows the counters we collect using perf-stat as well as a brief description of what each counter collects.

Processors allow only a limited number of performance counters to be activated simultaneously. On most Intel machines the maximum number of performance counters that can be accessed at the same time is four [7]. To combat this, perf-stat multiplexes the counters and cycles through the counters during the sampling period. However, to minimize the degradation due to cycling through the counters, we perform three different runnings, with relevant metrics grouped together. Table 5.2 shows how we split up the collection of counter information.

Each of the three categories has a specific emphasis. The first category, CPI, is focused on determining the cycles-per instruction (CPI). The next category, Memory, focuses on the cache performance of each run. The last category, System, looks at system-level events during the execution of the application.

To measure the counters, we arrange perf to take five one minutes samples of the execution after the application has been executing for five minutes at optimal performance levels. We take five samples so that they can be averaged, helping to eliminate noise within the samples. Having long samples (1 minute) related to the number of instructions executed per a second also helps to remove noise from the counter information retrieved.

5.4 Application Warmup

As discussed in Section 2.3, JVM-based languages go through a warmup period before running at optimal speeds. Hence, when measuring the performance of the program, we first must allow the program to be warmed-up before taking measurements.

To measure the execution time without including the warmup time is simple; we simply run the code for a couple minutes before starting to record the execution time. However, when trying to measure the performance counters using perf-stat or when using the sde tools this process is more complicated. We chose to include triggers within the application to the start program analysis after sufficient warmup completed. Our program must communicate with libraries written in C to complete this task. To accomplish this, we leverage the Java Native Interface (JNI) API [5]. In normal JVM-based code, the underlying architecture and operating system is meant to be transparent to the programmer. However, in C, programmers can directly write assembly code and also make system calls to the operating system. The JNI API is designed to allow JVM-based codes to call C-routines which can accomplish machine-specific functionality. Using the JNI API, in our code once warmup is completed we are able to initiate the start of measurements for both perf-stat and SDE.

6 Initial Results

In this section we discuss our performance results for the initial version of our code.

6.1 Time Per Step

Figure 6.1 shows the average time required to complete one step of the nbody simulation vs. the number of planets in the universe for the given configuration. Note that execution time is in seconds. In addition, this figure also shows the performance expectation for each number of planets. Our observed results are in red while the performance expectation is blue.

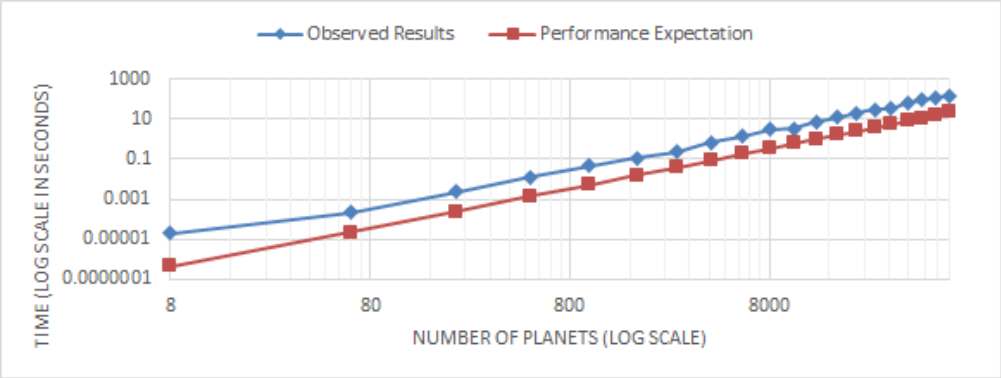

Figure 6.1: Execution Time Per Nbody Step versus the number of planets

# Planets	Time Per Step (sec)	Expected Time (sec)

Table 6.1: Initial Performance Results

As expected, as the number of planets increases the time per a step increases too. This occurs for both performance expectation and observed results. Note that the both the performance expectation and the observed results is the same. In other words, the performance expectation remains off by the same amount for each value. This indicates that the constants should be adjusted for our performance expectation better match the observed performance. However, our performance expectation accurately describes the performance trend and can be used to predict the performance expectation of other planet sizes once this constant is corrected for.

There are many reasons why our constants are off. First, we cost of a floating point operations is potentially higher than one operation per a cycle. Second, in our simplifications we disregard many calculations for each step in handling the data structures. Finally, out of the three phases that it takes to collect the position of the planets for each timestep, our model only considered the second one (where the planets' positions are sent back to the universe). In reality, both first and third phase might be significant enough to making a difference.

6.2 Planets Processed Per Second

Figure 6.2 shows the number of planets which process their new location within a second. Note that as the number of planets increase, more work must be done by each planet to calculate their new location, and hence it is unsurprising that as the number of planets increase the time to process a planet increases.

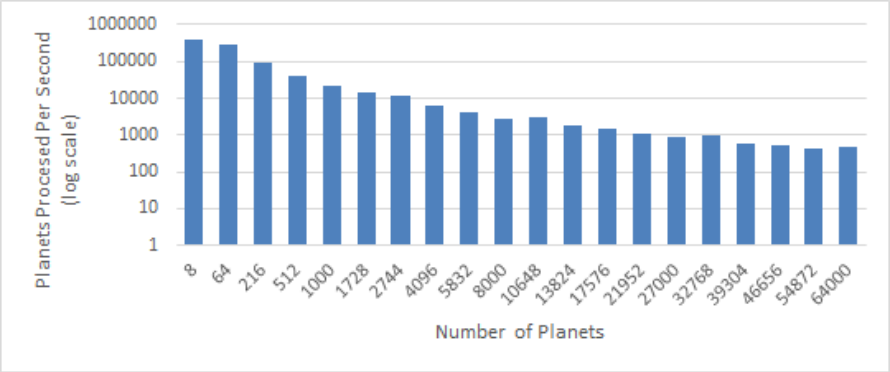

Figure 6.2: Number of planets processed per second

6.3 Effects of Multithreaded Execution

In order to observe the parallel efficiency of our application we disabled a certain amount of cores and then obtained the duration of a timestep over a different number of planets. The results (Figure 6.3) show that when we go from using a single core to two cores, the execution time gets approximately halved. Specifically, the speedup ranges between $S = 1.5$ for the lowest number of planets and $S = 2.3$ for the highest ones. Using the formula of the parallel efficiency, $E = \frac{T_1}{T_2 * N} = \frac{S}{N}$, we get values between $E = 0.75$ (lowest number of planets) and $E = 1.17$ (highest number of planets). This super-linear behavior for the highest number of planets is due to efficient cache usage, and it shows how well parallelized our application is.

If instead we enable all four cores, the speedups versus a single core ranges between $S = 2.67$ and $S = 4.09$, with an average speedup of $S = 3.3$. This translates into a range of efficiencies between $E = 0.7$ and $E = 1.02$, with an average of $E = 0.83$. As we see, although there is still a significant improvement when we go from two to four cores, the efficiency is reducing. Finally, note that even though we also enabled hyperthreading on the four cores, which increases the total count of logical cores up to eight, there were negligible speedups versus the evaluation on four cores. This could be because SMT is not useful since a single-thread per a core and already fully utilize all of a core's resources. If this is the case, then adding addition threads per a core will not result in performance improvements.

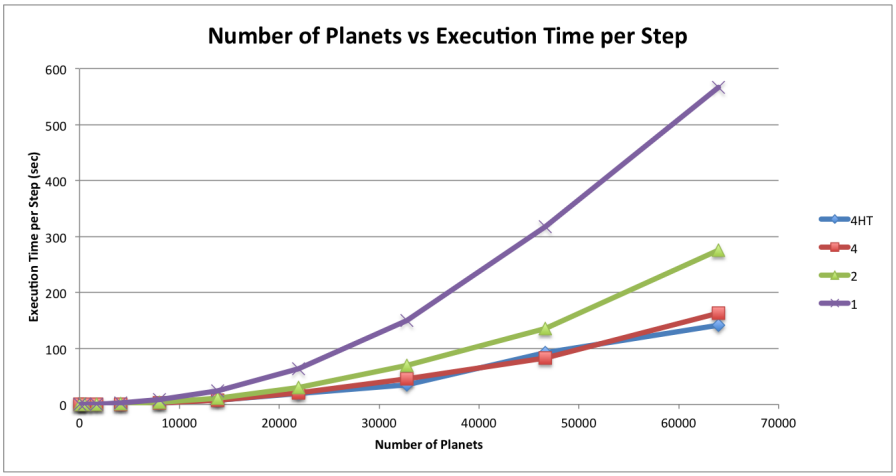

Figure 6.3: Execution time with varying number of compute cores

7 Analysis

7.1 SDE

7.1.1 Instruction Mix

If we observe the total number of instructions executed (Figure 7.1) we see an obvious increase with the number of planets, as the total cost of the application increases. More interestingly, when we normalize each column (Figure 7.2) we can see the breakdown of instruction count per each type, and its evolution with the number of planets. The most prominent category, DATAXFER, relates to moving data between the registers and memory. This is done with loading and stores to data structures as well and when sending messages throughout the system. We also see BINARY having a high impact, which corresponds to the calculations made to obtain the new positions. Continuing in decreasing order we have COND_BR, LOGICAL, PREFETCH (which increases for a big number of planets, where the high memory footprint leads to higher benefits out of prefetching), PUSH and AVX. This last one corresponds to vector operations, and unsurprisingly has higher impact for higher numbers of planets (as there is more data available to be operated on at once).

On the whole the ratios of instructions for different number of planets remain similar; however, there are some slight variations. Most notably, the share of data transfers (DATAXFER) increases for larger number of planets. This is likely because binary instructions have been replacement by vector AVX instructions, so that there are less explicitly arithmetic instructions to perform the same amount of work.

7.1.2 Footprint

Figure 7.3 shows the number of cachelines touched by a program of a given planet size. Note that there is a large jump between a system with 1728 planets vs 2744 planets and after 4096 planets, the footprint evens out. The likely culprit of this is Java's automatic memory manage-

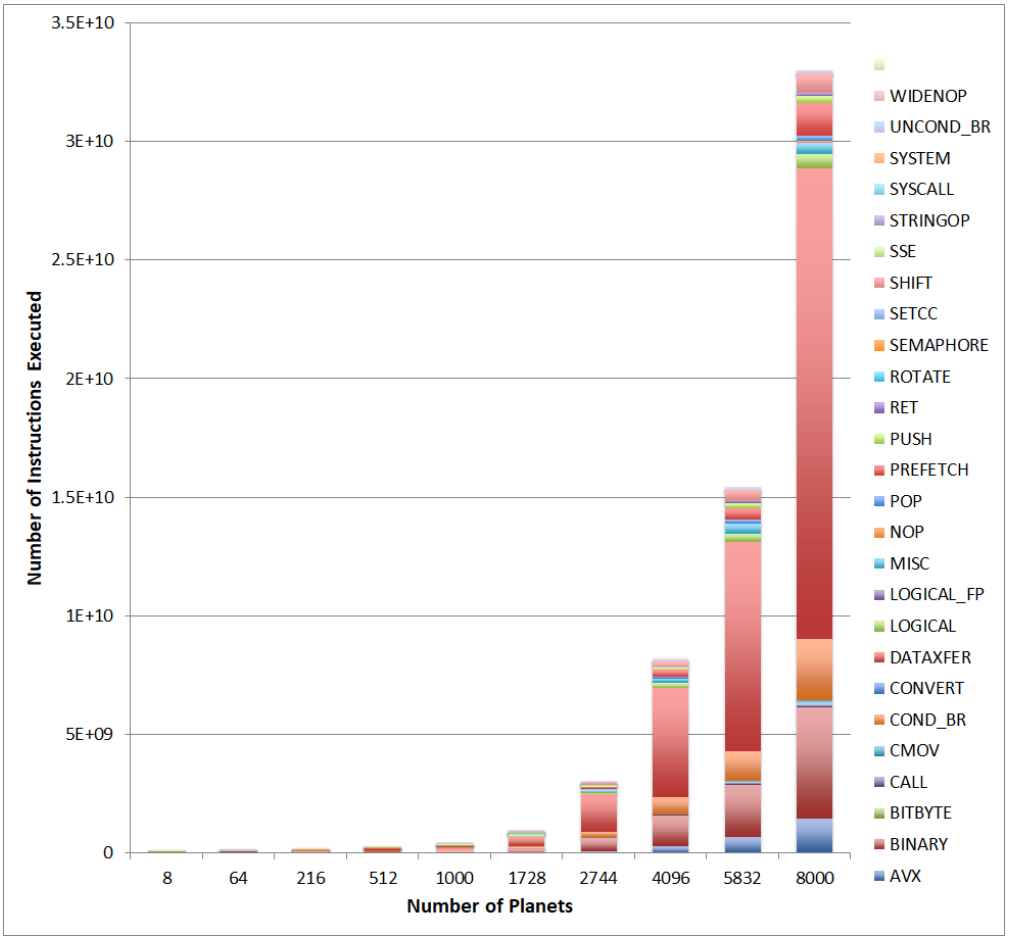

Figure 7.1: Instruction count breakdown over the number of planets

ment coupled with using immutable data structures. Because immutable data structures are used, as the structures are appended then parts of the data structure must be copied and modified while other parts of the structure become garbage, or are no longer reachable from any application pointer. It is up to the runtime to decide how often to perform garbage collection to recover freed objects. As the number of planets increases, immutable data structures are being modified more often creating more trash. The jump between 1728 and 2744 is likely due to the garbage collector deciding to collect trash less often, which will make the application's footprint appear larger. Likewise, the footprint levels off for the larger sizes at around 45M cachelines, or around 3GB. This means that the JVM is performing garbage collection once the heap is becoming a certain size to limit the amount of memory consumed by the application.

Note that by far the largest portion of the footprint is both read and stored to. This is expected, as the main data structures are being both created and read during each iteration.

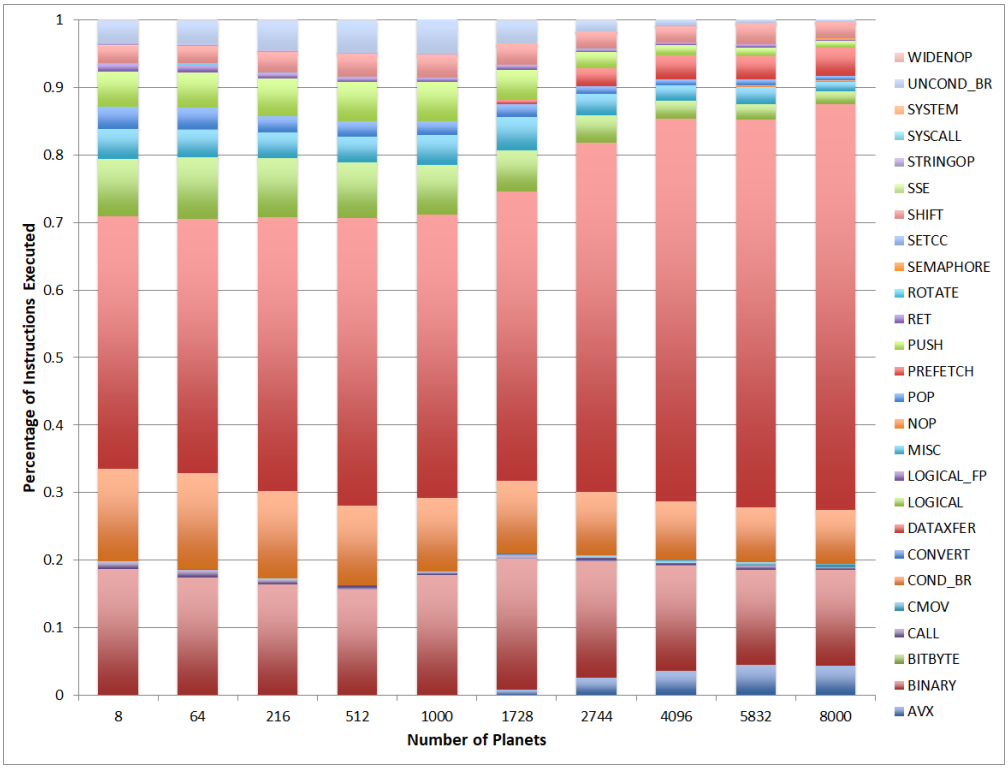

Figure 7.2: Normalized instruction breakdown over the number of planets

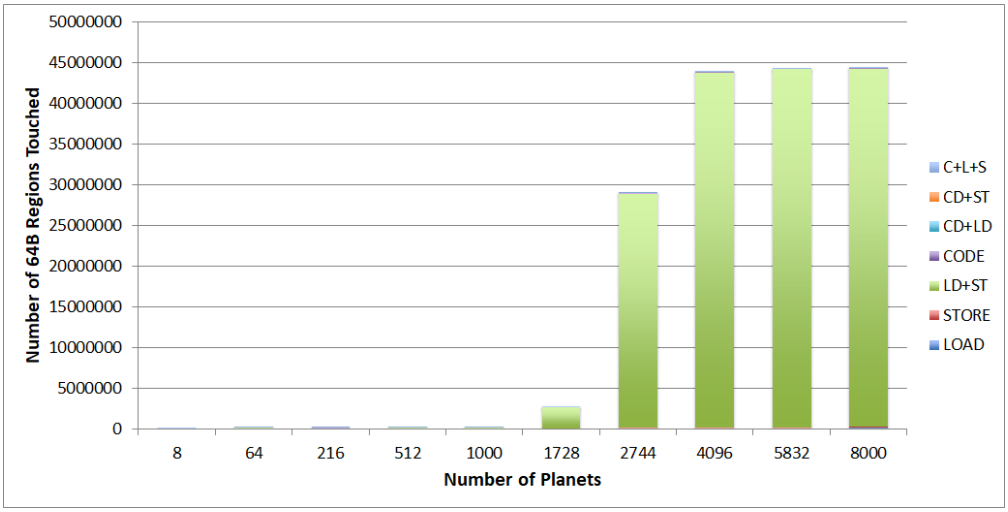

Figure 7.3: Timestep memory footprint over the number of planets

7.2 PERF

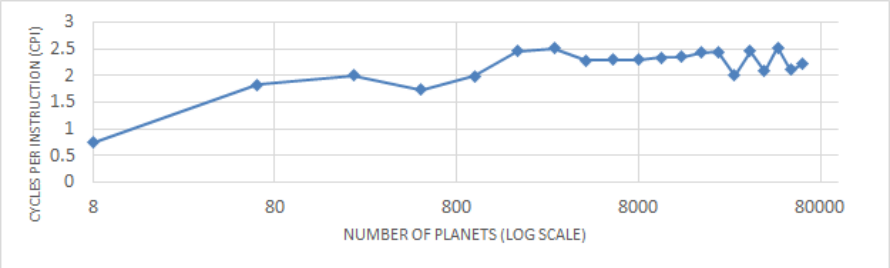

Figure 7.4: Cycles Per Instruction (CPI) over the number of planets

7.2.1 Cycles Per Instruction

Figure 7.4 shows the cycles per instruction (CPI) observed for different planet sizes. Note that as the number of planets in the system increase, it takes more cycles to complete an instruction. However, the CPI levels off around 2.5 regardless of the system size. This indicates that the performance is bounded by the amount of computation, as the CPI does not continue to increase as the problem size become larger as would happen if the memory was the bottleneck. This matches our performance expectation.

7.2.2 System Overheads

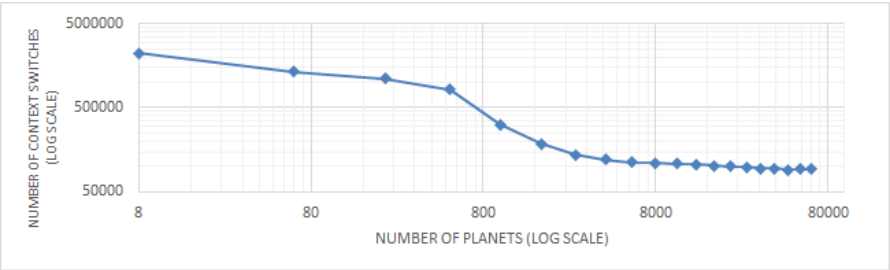

Figure 7.5: Number of context switches over the number of planets

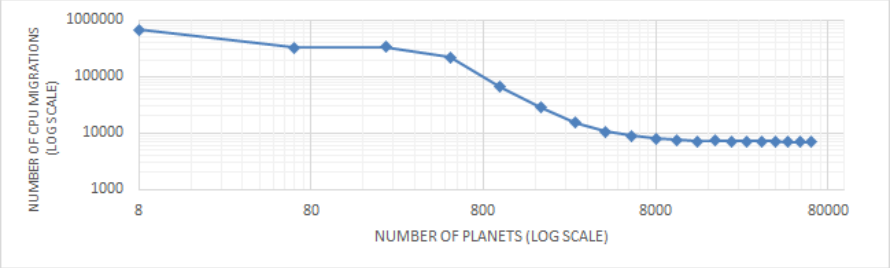

Figure 7.6: Number of CPU migrations over the number of planets

Figure 7.5 and Figure 7.6 show the number of context switches and cpu migrations observed

during the sampling period for each problem size. At first glance, observing that smaller program set sizes have a higher frequency of context switches and cpu migrations is surprising, as these are time-consuming events. However, after further contemplation, the reason for this trend becomes clear: smaller problem sizes have smaller pieces of work for each actor do to. Since the actors can complete their work faster, they are switched out more often than if an individual actor takes a long time to complete a given task. In this way, the higher number of context switches and cpu migrations is representative of more tasks being completed by the smaller problem set sizes.

7.2.3 Memory Pressure

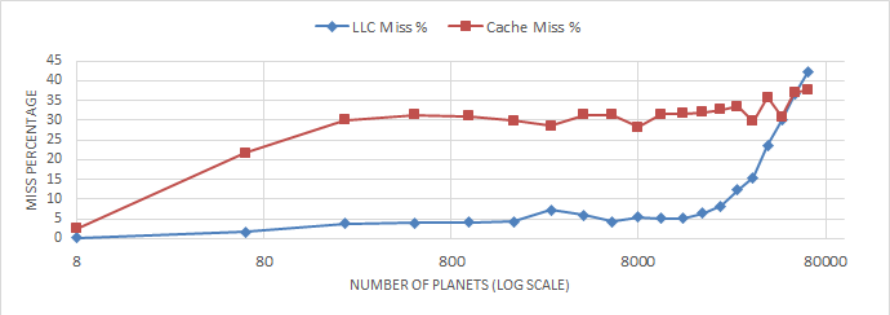

Figure 7.7: Miss Rate Percentage

Figure 7.7 shows the overall miss rate percentage as well as the last-level cache (LLC) miss rate percentage. The overall miss cache rate stays level through the execution. Note though that this includes prefetch misses as well as misses to each tier, so this number is less meaningful than expected. The more telling metric is the LLC miss rate, which steadily climbs as the problem size increases. However, as the CPI remains steady, it appears that computation, not memory is the primary bottleneck. This is confirmed by our performance expectation, as the amount of computation increases exponentially as the number of planets increase while the memory increase is linear.

8 Program Optimizations

In this section we discuss multiple strategies that can potentially improve nbody's performance.

8.1 Optimization Strategies

Two straightforward ways to improve the performance of nbody are the following:

1. Reduce the total number of messages sent
2. Reduce the amount of computation required for each planet

Reducing the total number of messages sent has many performance implications. First, this reduces the total amount of loads and stores the system performs. Each message must be stored in memory and loaded to be read afterwards, which can result in significant memory pressure.

Furthermore, in the actor framework, each message sent to a different actor requires the receiving actor to be activated. On a system with many more actors than processors, this results in either switching threads or loading a new context into the currently running thread, both of which have nontrivial overheads. Reducing the number of messages reduces the number of times actors have to be activated.

In the initial setup, all planets receive the coordinates of every other planet. However, in reality, only nearby planets significantly impose gravitational forces on the planet. Planets in other solar systems or especially other galaxies should apply negligible forces on a planet. Hence, an approximation one can make is to only query for the location of planets in one's solar system or galaxy and ignore other planets, or to approximate the value of planets far away.

8.2 Optimizations Performed

To improve n-body's peformance, we create five different optimization configurations. All of the configurations focus on reducing the amount of computation each planet must perform to calculate the resultant force and also potentially reduce the amount of messages which must be sent between the actors.

8.2.1 Pruning Distant Neighbors

The first optimization is to limit the amount of neighbors needed for the calculation. For this optimization we create two configurations, *galaxy share* and *solar system share*. In solar system share, each planet only receives the coordinates of the other planets in the same solar system. This can improve performance in several ways. First, the list of planets does not need to be aggregated at the galaxy and universe level, reducing the amount of messages sent. Second, each planet's list is substantially smaller which will improve the performance of the force calculation.

The galaxy share configuration does a similar thing as the solar system share configuration, but instead now each planet receives a list of all planet coordinates which shares the same galaxy.

8.2.2 Approximating Distant Neighbors

While limiting the planet list size helps to improve the performance of the force calculation, it also can affect its accuracy as many planets are not accounted for in the calculation. Another option is to approximate the impact of non-neighboring galaxies or solar systems force on a planet's by creating one representative object for a given galaxy or solar system. To accomplish this at the galaxy and/or solar system level, the given actor can, instead of sending a list of all values, calculate and send one value whose mass is the sum of all elements of the list and position is the weighted average location of the list. This optimization also greatly reduces the

size of the list used by each planet to calculate forces, but does not sacrifice as must accuracy as the impact of non neighboring planets is roughly accounted for.

To test the performance impact of this optimization, we create three configurations which vary at what level(s) the neighbors are approximated.

Optimized version *solar system approx* performs this optimization at the solar system level only. Optimized version *galaxy approx* performs this optimization at the galaxy level only. Optimized *solar system and galaxy approx* performs this optimization at both the solar system and galaxy level.

9 Optimized Results

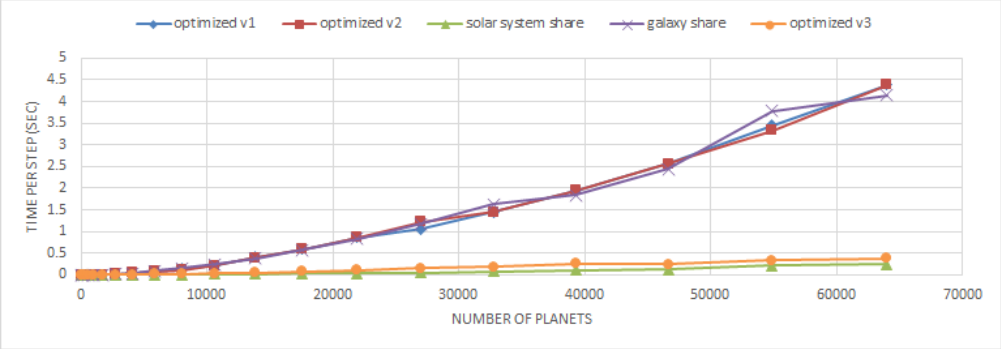

Figure 9.1: Optimized Time Per Step

Figure 9.1 shows the time per step for the five optimized configurations. The original version is left out due to the vastly superior performance of the optimized versions. Overall, these optimized versions have more than an order of magnitude performance improvement compared to the original configuration. The optimized configurations fall into two categories based on the level in the hierarchy the sharing is performed at. *Solar system approx*, *galaxy approx*, and *galaxy share* have similar performance. A main reason for that is that these configurations limit the size of the data structures and number of messages by a similar degree (essentially removing a level of the hierarchy). *Solar system share* and *solar system and galaxy approx*, on the other hand effectively remove two levels of the hierarchy and hence achieve even higher performance results, at the expensive of losing precision in the calculation.

10 Discussion

10.1 Code Repository

All of code is available for download at `https://drive.google.com/file/d/1wIlvImFtbuIK8cQ8dBx_ArwUoozd8pmn/view?usp=sharing`. In total, our application contained 729 lines of high-level functional Scala and Java code. As we are using the actor framework, our code is highly parallel

and can with minimal effort be extended to work on a distributed system. The scripts for recreating our results are also available from the above link.

10.2 Distribution of Work

We made many efforts to ensure that work was evenly split between the two members of this project. Thomas wrote the application and JNI code while Antonio created and performed the analysis using the perf tool and Intel's Software Development Emulator. Both members contributed equally to the writing of this report.

11 Conclusion

For this class project we decided to implement the n-body simulation using an actor framework and with high-level programming abstractions. We successfully created a multithreaded implementation which achieved good scalability on a single core and matched our performance expectation. In addition, we analyzed our application using Intel's Software Development Emulator and by measuring performance counters to better understand our application's performance. Finally, we proposed and implemented five optimized configurations of our application that significantly improved performance by reducing the amount of computation, the number of messages passed, and the size of data structures.

References

[1] Akka: a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala. `https://akka.io/`.

[2] Case studies of Akka in the market. `https://www.lightbend.com/case-studies#tag=akka`.

[3] Intel Performance Counter Monitor. `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`.

[4] Intel Software Development Emulator. `https://software.intel.com/en-us/articles/intel-software-development-emulator`.

[5] Java Native Interface. `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/`.

[6] Wikipedia article on Akka. `https://en.wikipedia.org/wiki/Akka_(toolkit)`.

[7] David Levinthal. Performance Analysis Guide for Intel Core i7 processor and Intel Xeon 5500 processors. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`.

[8] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[9] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

[10] M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, and et al. An overview of the scala programming language. Technical report, 2004.