Investigating methods for real time simulation of an n-body system.

David Ireland

BSc (Hons) Computer Games Technology,
2021

School of Design and Informatics
Abertay University

Table of Contents

Table of Contents i
Table of Figures iii
Table of Tables v
Acknowledgements vi
Abstract vii
Abbreviations, Symbols and Notation viii
1. Introduction 1
1.1. Overview 1
1.2. Aim, Objectives and Research Question 3
Aim 3
Objectives 3
Research Question 4
1.3. Remaining Chapters 4
2. Literature Review 5
3. Methodology 11
3.1. Initial Developments 11
3.2. Euler Method 12
3.3. Verlet Method 13
3.4. 4^{th} order Runge-Kutta 14
3.5. Tree Method 17
3.6. GPU Parallelisation 19
3.7. Recording Results 21
Test Case 1 21
Test Case 2 21
Test Case 3 22
4. Results 23
4.1. Overview 23

4.2. Test Case 1 23
4.3. Test Case 2 24
4.4. Test Case 3 26
5. Discussion 31
5.1. Performance 31
5.2. Energy 32
5.3. Efficiency and Efficacy 34
5.4. Simulation Considerations 35
6. Conclusion and Future Work 38
List of References 41
Appendix A – Raw timing results 44
Appendix B – Barnes-Hut Tree diagram 63
Appendix C – Artefact Input and Output 64

Table of Figures

Figure 1: A screenshot from Universe Sandbox showing the simulation of an N-Body System (Universe Sandbox, 2020) 2
Figure 2: A screenshot from Kerbal Space Program showing orbits of bodies. (Kerbal Space Program, no date) 2
Figure 3: shows a system of bodies split in 2D space 8
Figure 4: shows the system from Figure 3 represented in a quad tree structure 9
Figure 5: UML diagram of inheritance from Solver Class. All classes override virtual Solve() function, classes have access to inherited CalculateAcceleration() function. 12
Figure 6: Verlet pseudo code 14
Figure 7: RK4 pseudo code 16
Figure 8: Node Structure 17
Figure 9: Tree Update process pseudo code 17
Figure 10: Pseudo code for SplitNode function 18
Figure 11: CalculateForces pseudo code 19
Figure 12: TraverseNode function pseudo code 19
Figure 13: Run times of each method 23
Figure 14: Change in Energy for 10 bodies shows the RK4's maximum change in energy to be around 1/6th of the maximum change in energy of the Verlet 24
Figure 15: Change in Energy for 100 bodies shows the RK4's maximum change in energy to be around 1/8th of the maximum change in energy of the Verlet 25
Figure 16: Change in Energy for 1000 bodies shows the RK4's maximum change in energy to be around 1/7th of the maximum change in energy of the Verlet 25
Figure 17: Change in Energy for 10000 bodies shows the RK4's maximum change in energy to be around 2/9^{ths} of the maximum change in energy of the Verlet 26
Figure 18: Change in energy of Euler method 27
Figure 19: Change in energy of Runge-Kutta method 27
Figure 20: Change in energy of Verlet method 28
Figure 21: Change in energy of Barnes-Hut with θ=0.5 28
Figure 22: Change in energy of Barnes-Hut with θ=1.5 29
Figure 23: Change in energy of Euler GPU method 29
Figure 24: Oscillation of Energy 30

Figure 25: Barnes-Hut Tree Construction and Force Calculation Diagram 63

Table of Tables

Table 1: Mean of absolute changes in energy 30
Table 2: Recorded times for Euler Method 46
Table 3: Recorded Times for RK4 Method 48
Table 4: Recorded Times for Verlet Method 50
Table 5: Recorded Times for GPU Euler method 53
Table 6: Recorded Times for Barnes-Hut with theta =0.5 55
Table 7: Recorded Times for Barnes-Hut with theta =0.75 57
Table 8: Recorded Times for Barnes-Hut with theta =1.0 60
Table 9: Recorded Times for Barnes-Hut with theta = 1.5 62

Acknowledgements

Thanks to Dr Craig Stark for his help and guidance throughout this project.

Thanks to my friends, flatmates, and family for their support.

Abstract

N-body simulation is the simulation of systems of particles, with each particle exerting a force on every other particle. Methods that simulate every force are called direct methods and are slow. Therefore, methods of approximation such as the Barnes-Hut method, which utilises a tree structure can be used to reduce the number of calculations. Every method has some level of error, reducing the timestep between each simulation step will lower the error but results in the simulation taking longer. The potential error in simulation and the slow performance makes implementing N-body simulations into real time applications challenging.

This Project aims to develop a real time N-body simulation that can be used to compare methods of real time simulation and further investigate direct methods, the Barnes-Hut and GPU parallelisation.

This project implements 3 direct methods: the Euler, the Runge-Kutta and the Verlet. A parallel CUDA Euler method is implemented to improve performance. The Barnes-Hut implemented constructs a tree to represent particle positions in 3D space. When calculating forces, the tree is traversed and groups of particles that are deemed suitably far away are approximated using weighted average positions and combined mass.

Performance of each method is recorded, timing the simulation step. Each method performs 100 steps and median times are plotted. Energy readings are taken to find the change in energy for each method with different timesteps.

As expected, the Runge-Kutta is the most accurate but all direct methods are slow and unlikely to be viable for real time applications with larger N values. We find that the Barnes-Hut provides good performance for larger N, but the GPU Euler method performs best for large N.

The artefact achieves the goal of implementing methods of N-body simulation and this project has successfully compared these methods and evaluated their suitability to real time applications. Potential future work would involve implementing collisions of bodies and fragmentation or other methods such as the FMM and a CUDA implementation of the Barnes-Hut.

Abbreviations, Symbols and Notation

RK4 – 4^{th} order Runge-Kutta

KSP – Kerbal Space Program

GPU – Graphics Programming Unit

N – Number of Bodies

FMM – Fast Multipole Method

1. Introduction

1.1. Overview

N-body simulation refers to the simulation of a dynamic system of particles under the influence of physical forces such as gravity or electromagnetic forces. Gravitational simulations are often used to model the dynamics of small systems of bodies such as the moon, earth and sun or larger systems such as a universe which can be used to study and understand the physics of these systems. Plasma N-body simulations simulate the interaction of electromagnetic forces on a group of charged particles. In a system, each body or particle, effects every other particle. Therefore, in a system of 100 bodies, each particle will have 99 forces acting on it from each other particle. For gravitational simulations the amount of force applied by a particle is dependent on the distance between the particles and the mass of the particle applying the force. With a greater mass the force will increase, and with a shorter distance the force will also increase. Given that the simulation must simulate every particle's effect on every other particle the simulation becomes very computationally expensive with a larger number of bodies.

Two games that are good examples of the difficulty of N-body simulation are Universe Sandbox (Giant Army, 2015) and Kerbal Space Program (Squad, 2011). Kerbal Space Program is a space flight simulation game but does not include n-body simulation, this is due to the unpredictability of N-body simulations which could alter orbits and cause catastrophic events to ruin the game's core gameplay. Instead Kerbal Space Program opts to use a simplified simulation by using a technique called patched conic approximation. Universe Sandbox does implement N-body physics, they provide multiple integrators to be chosen by the user. Universe Sandbox lets the user alter the simulation speed and method, letting the user prioritise speed or accuracy. The simulation speed is limited by the hardware, value of N and method. The game attempts to make sure the timestep, which effects the simulation speed, is not too high that it would cause accuracy to go below a certain threshold which would result in a divergence too far from realistic simulation.

Figure 1: A screenshot from Universe Sandbox showing the simulation of an N-Body System (Universe Sandbox, 2020)

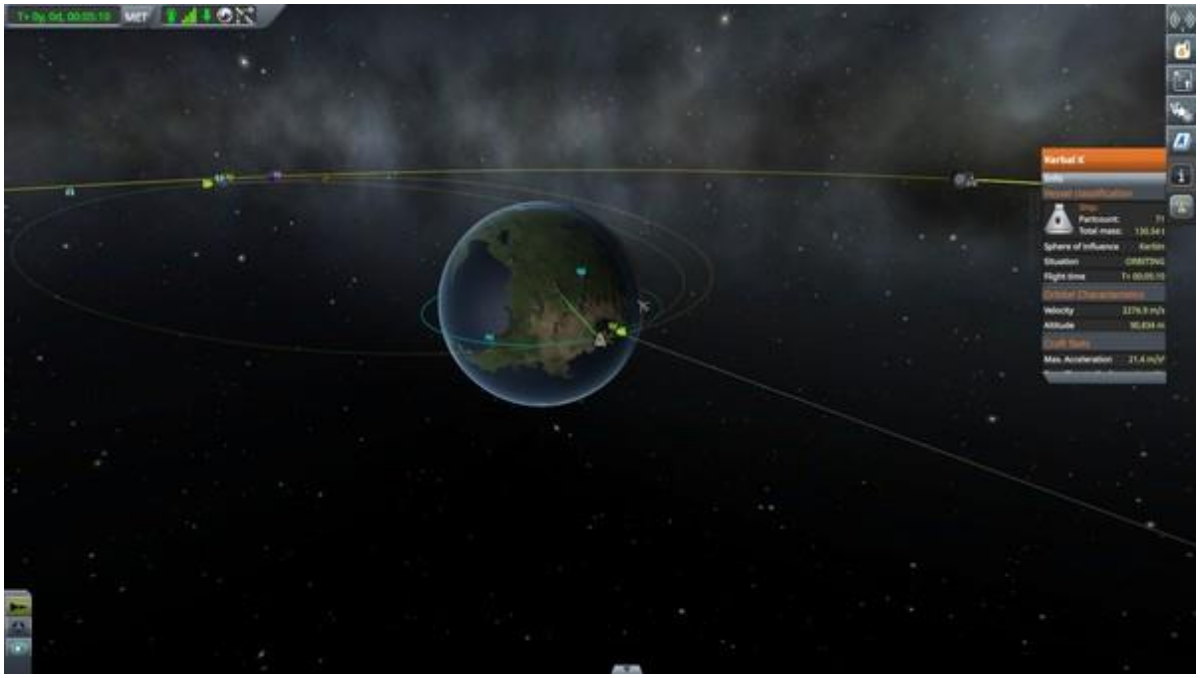

Figure 2: A screenshot from Kerbal Space Program showing orbits of bodies. (Kerbal Space Program, no date)

This project investigates integration methods such as Euler, Verlet, and 4th Order Runge-Kutta, known as brute force methods, since they calculate all forces applied and are generally slow. Also investigated is the Barnes-Hut method which stores particles in a tree structure and then approximates certain calculations based on distance. Finally, the project will investigate the use of parallelism on the GPU to speed up the process of a brute force method such as the Euler method.

Running N-body simulations in real time requires a lot of processing power, to understand which methods are viable for real time applications, we must consider the run time taken to achieve a step of simulation. However, it is also important to understand that all numerical methods result in error. Large errors will result in inaccurate simulation. Keeping error low is important to avoid potentially chaotic consequences. Each method will provide different levels of accuracy. One way to keep error low is to keep the timestep low. A timestep of 0.1s will provide more accurate results than a timestep of 1s, however will take 10 times as long to simulate the same period of time. To consider the viability of each method we must consider both the accuracy and speed of the method. A third factor to consider is the complexity of implementation. The Euler is by far the simplest method to understand and implement and may be suitable for some simulations, therefore it could be the most effective solution.

The simulation developed in this project is specifically looking to investigate the methods for calculating forces for gravitational systems. The simulation therefore will not feature collisions. The artefact developed will allow the user to choose the method they wish to use, the timestep and the number of bodies being simulated. The artefact also provides functionality to output the energy and run times of the simulation. By varying the input variables of N and the timestep for each method, the output can be used to compare each method's accuracy and performance as the system increases in size.

1.2. Aim, Objectives and Research Question

Aim: To create a real time simulation of an N-body gravitational system and compare the efficiency and efficacy of direct methods to the Barnes-Hut method and to a GPU parallelised method for real time simulation.

Objectives:

- Investigate uses of N-Body simulation in current technologies and games.
- Create and visualise an N-Body system simulation in real time
- Compare to find the most efficient and effective way to run the simulation in real time.

Research Question: Considering direct methods, parallelism, and methods of approximation for N-Body gravitational simulation, what methods are efficient and suitable for real-time simulation?

1.3. Remaining Chapters

The Literature Review will discuss further the challenges of simulating N-body systems and the applications where they are used or omitted due to these challenges. Further chapters discuss the process of implementation of each method and how the accuracy and speed of the methods can be considered. The discussion and conclusion chapters evaluate each method's advantages and disadvantages in the context of performance and accuracy to answer the Research Question.

2. Literature Review

In preparation for the undertaking of this project, the following section investigates literature relating to the field of N-body dynamics simulations. The focus of this project is to discuss the methods for the purpose of real time simulation on a single GPU for a relatively simple system of gravitational particles. Keeping this focus is important when discussing literature because many of the studies into N-body Dynamics are completed with large budgets using clusters of GPUs and attempt to simulate extensive systems of particles, often with interests of more specific subjects of Astrophysics compared with the aim of this project which is to investigate the computational side of the simulation.

Generally, games developers stay away from N-body simulation. This is because accurately simulating a gravitational system is very expensive computationally. Gemmeke (2005), explains that unstable orbits occur due to numerical errors. Given that small errors can cause orbits to become unstable, in a game like Kerbal Space Program (Squad, 2011) where realistic physics is key, the developers chose not to implement N-body simulation (PCGamesN, 2019) due to the potential for error to cause catastrophic circumstances. Instead KSP makes use of an approximation method called patched conic approximation; this allows KSP to divide space into subsections and assign each particle a sphere of influence, using these spheres of influence to reduce the calculations to individual two body problems which are much easier to solve, less computationally intensive and more predictable. This method provides Kerbal Space Program with a good approximation but does result in some methods of space travel not being possible (Koon et al., 2011).

One game which does implement N-body simulation is Universe Sandbox (Giant Army, 2015), which focuses on simulating real astrophysics conditions, allowing players to create and simulate solar systems. This is one example where creating a realistic physics system is a fundamental part of the game. Universe Sandbox however is unable to avoid these errors. In the game's development blog (Universe Sandbox, 2016), the developers explain that when the game jumps forward larger time steps, the error is greater. The game lets the user choose the method of integration, letting the user choose whether to prioritise speed or accuracy (Universe

Sandbox Wiki, 2020). As well as a number of brute force integrators, Universe Sandbox allows the user to enable a gravity tree method. The gravity tree is used to speed up simulations of large numbers of bodies, however, can cause a decrease in performance for small systems, probably due to the time associated with building the tree structure. This method allows the user to change a tree setting the game calls cell distance ratio, which is referred to as theta in most other simulations. The game notes that a higher value of theta will result in faster computation but will result in potentially greater errors. The simulations in Universe Sandbox attempt to avoid reaching an error that will cause orbits to fall apart. When the timestep is increased significantly the potential for orbits to become unstable also increases, resulting in moons crashing into planets, Mercury being shot out of the solar system and other chaotic events (Universe Sandbox, 2016). A Simulation Speed parameter can be changed by the user. This speed is limited by the hardware and performance that will determine how fast steps can be simulated, the timesteps between integrations that will determine the accuracy, and the method used. Universe Sandbox automatically finds a timestep that will result in a balance of accuracy and performance. Universe Sandbox also makes some simplifications to reduce the number of particles being simulated. Firstly, massive objects like suns and moons which are made up of lots of smaller particles are treated as point masses, where the whole mass is represented at a single coordinate in space. Secondly, certain low mass objects are treated as non-attracting objects, these objects are still affected by other objects gravitational force but do not exert their own gravitational force. Universe Sandbox is a game that can afford to implement expensive N-body simulations and utilises the unpredictability of the simulation as part of its gameplay, despite this it must still make some approximations in order to achieve better performance.

The simplest solution to the N-body problem is to integrate over a timestep using the Euler method. Euler method is a first order integration method and assumes constant acceleration over the time step, at smaller timesteps the Euler method keeps error low and therefore is effective. An alternative is the Runge–Kutta method or RK4. RK4 is a method of numerical integration using timesteps, like the Euler method, small time steps result in lower error but becomes more computationally expensive. Eagan explains that RK4 improves on Euler by finding an average of 4 acceleration values within the timestep (Eagan, 2017). These values are represented by k1 to k4. RK4 finds how the acceleration will change throughout the timestep.

Eagan's simulation however is not a good example of RK4 as it only integrates the acceleration using the RK4 method and then integrates the velocity with Euler's method after. Since the values of v(velocity) and x(position) are dependent on each other, the values of v and x should be integrated using RK4 in parallel calculating each kx1 value using the appropriate v value and calculating the kv1 value using the appropriate x value. Before moving on to calculate kx2 and kv2 all values of kv1 and kx1 must be calculated.

An integrator that's time step is reduced by 10 resulting in a factor of ten reduction of error is known as first order. Runge-Kutta is considered fourth order because accumulated error scales with O(dt-4) (Prappleizer, 2020). Pasha plots velocities and positions of a particle as calculated using the RK4 method against the analytical solution to show the accuracy of the method. Pasha then shows an example of the method being used to simulate the orbit of the earth around the sun, this clearly shows that the RK4 method is a valid method for simulating N-Body dynamics. However, the paper does not compare the RK4 method to any other methods in detail to show the advantages of using a 4th order integrator such as RK4.

The Barnes-Hut method is used to reduce the time taken to process each force. Aarseth (2003) explains that the basic idea of a tree method such as the Barnes-Hut is that interactions between particles can be described using a small number of parameters based on the mass distribution. The total force is obtained by summing the contributions of far particles based on these parameters and adding the forces of nearby particles as normal. Forces are calculated recursively, the algorithm checking if $l/D < \theta$, where θ is the opening angle or tolerance, l the size of the cell and D the distance from the cell's centre of mass to the particle. Aarseth mentions that some simulations use a value of around $\theta = 1$ but this will result in relatively large errors. Aarseth also explains that with $\theta = 0$ the algorithm reverts to a direct summation method. Aarseth's discussion around the Barnes-Hut algorithm revolves primarily around creating efficient and accurate simulations rather than for what is necessary for real time applications. This is further shown in the next chapter of the book where Aarseth discusses data structures, opting to go for an ordered array of pairs, updating the array regularly to be in sequential order. Aarseth explains that this is to allow sequential force calculation. The time to re-order these arrays becomes

negligible for very large N simulations and speeds up the simulation. For the size of simulations this project will be investigating this step that would increase the complexity of implementation significantly will not be necessary. The complication this would cause would also make further features involving different particles harder to implement. As the purpose of this project is to prove the viability of N-body simulations beyond simulations alone, this would not be a favourable outcome. Figure 3 shows a system of particles split up into a grid in 2D space. Figure 4 shows this system stored in a quad tree, since this system is only in 2D, it is in a quad tree. In the artefact, a 3D system, the simulation would be split into 3D octants and represented in an octree.

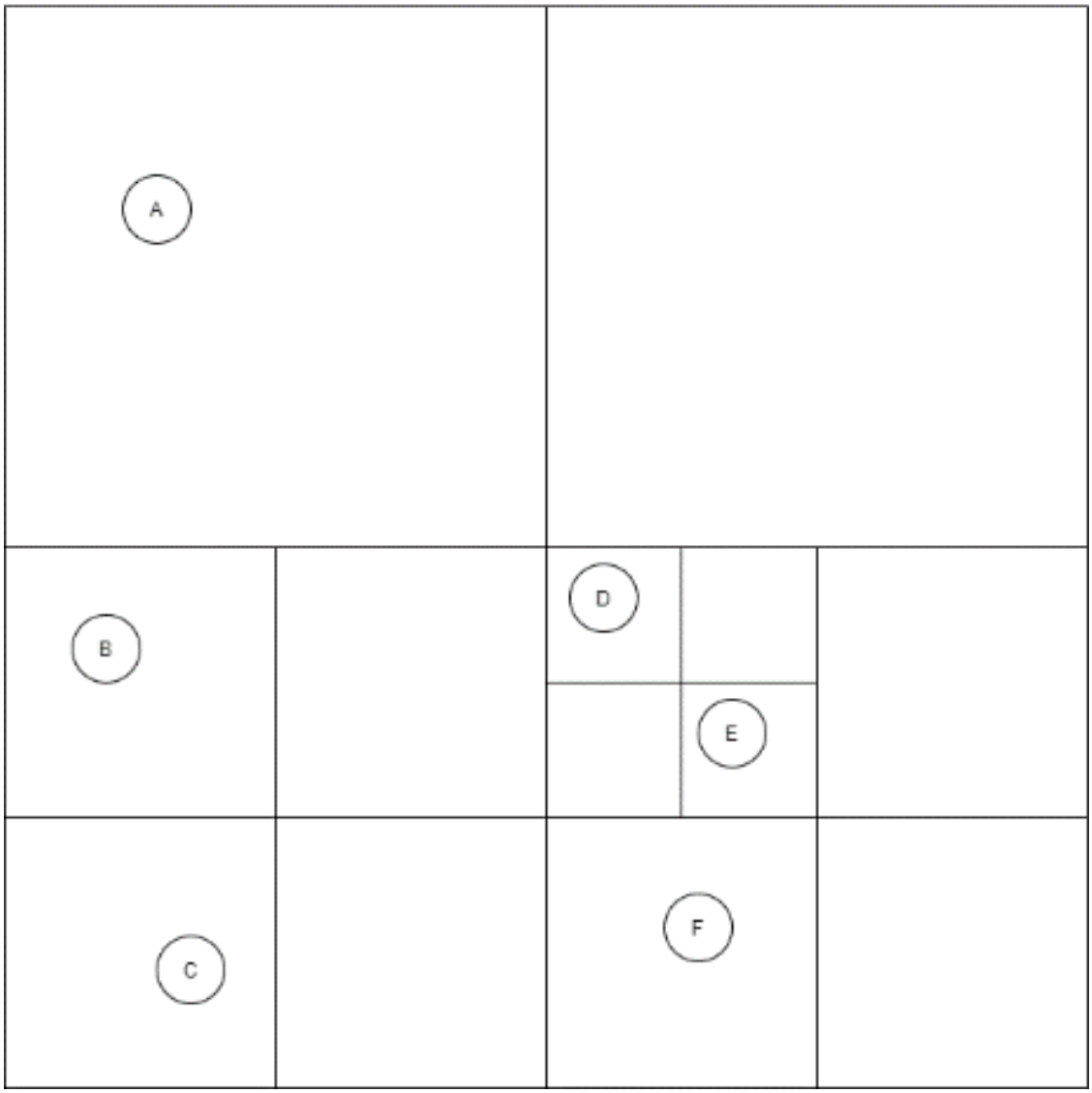

Figure 3: shows a system of bodies split in 2D space

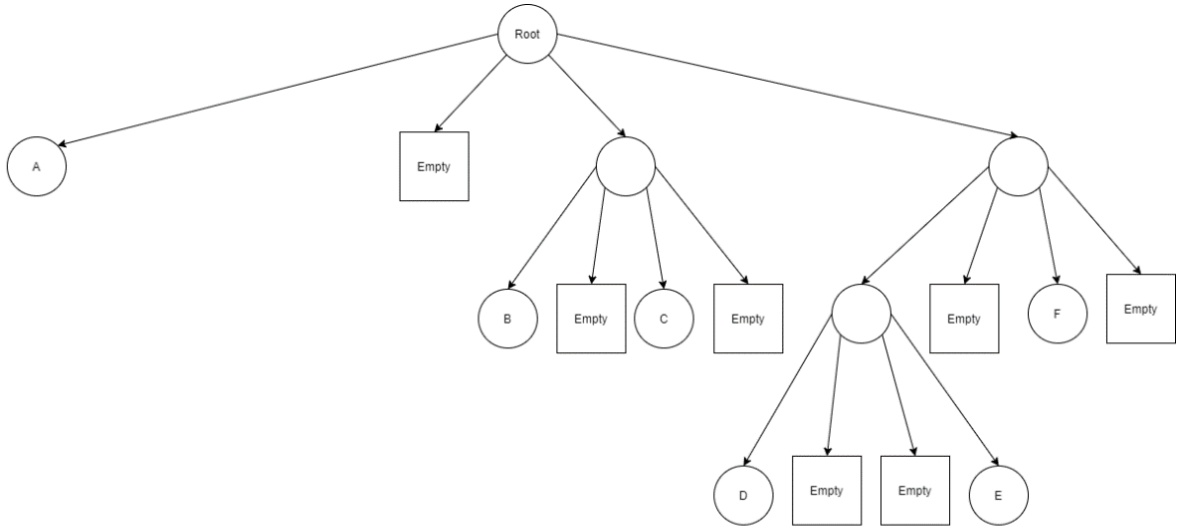

Figure 4: shows the system from Figure 3 represented in a quad tree structure

Nyland, Harris and Prins (2007) discuss how CUDA (Nvidia, 2007) can be used to implement the direct Euler method on the GPU. The method used is known as the All-Pairs method and is a brute force method meaning that it will evaluate all interactions between all bodies. The algorithm is suited well to GPU programming due to its relatively straightforward parallelism. While in theory each force from each body could be calculated on a different thread and then summed this would be inefficient due to the amount of memory needed. Therefore, some of the work is serialized meaning each kernel will calculate all forces acting on one body in serial with multiple kernels doing work in parallel. They find that their implementation makes use of straightforward parallelism and that their method runs more than 50 times as fast as the serial version on a CPU. Besides the obvious benefit of doing this in parallel, the GPU implementation benefits from the fact that complex operations such as the inverse square root executes much faster on the GPU than the equivalent execution times on the CPU. This article was first published in 2007 and the results are tested on a GeForce 8800 GTX GPU, a graphics card that was released in 2006, therefore both GPUs and CPUs will have advanced considerably since then, despite this the methods are still very relevant.

Hamada et al. (2009) compare the performance of direct methods and tree algorithms when parallelised using GPU and CPU. The comparison is of 4 different

methods of GPU parallelisation and 1 CPU parallelisation method. This report would be improved had it compared more than 1 CPU method to demonstrate the differences between GPU and CPU better. However, the report does find that all GPU direct methods outperformed the CPU. When comparing the tree algorithms, the difference in performance was negligible or worse for 3 of the GPU tree algorithms and only the modified code by Hamada et al. achieved significantly better performance. This is achieved by sending multiple groups of particles that will interact with each other, known as walks, to the GPU simultaneously allowing multiple walks to be processed at once by each thread. The study simulates over 500 million particles and uses 128 GPUs valued at around $168,000, therefore the techniques used may not be relevant to the smaller scale simulations on a single GPU this project will be simulating. However, the report does clearly show the performance gains that can be achieved using GPU parallelisation and the advantages of using the GPU over the CPU.

3. Methodology

3.1. Initial Developments

The program is implemented using C++ due to experience with the language but also the performance that is provided by C++ and the option to allow for GPU programming through CUDA. Initial Development focused on the requirements of the application to facilitate implementation of N-body simulations. To display the simulated particles, OpenGL (Khronos Group, 2017) was implemented. OpenGL allows for geometric shapes to be easily and simply rendered. Early developments also included basic systems such as the Vector3 class that allows 3-dimensional float values to be stored and the camera class which are used throughout the application. Also implemented are the `particle` and `particleManager` classes that store the required information about the particles, with the `particleManager` controlling the technique being used to update the particles. `particle` stores `mass`, `velocity`, `acceleration` and `position` variables relating to the particle. Particles in the `particleManager` are stored as a pointer to an array of particles. Using an array to store the particles means that the particle class cannot be as easily resized as it would be using a vector however CUDA does not support Vectors or other standard library structures so using an array is required to implement CUDA, due to this the decision was made to use an array.

The program makes use of polymorphism, `Solver` is used as the base class for each method. `Solver`, stores `time`, `g` (the gravitational constant) and contains 3 functions. The `CalculateAcceleration` function takes two positions and a mass and returns a `Vector3` with the x,y,z acceleration values. Passing the positions and mass individually rather than the whole particle allows for the function to be reused in subclasses where the position or masses being used in the calculation may be modified. The `Solve` function is a pure virtual function that has no implementation in the base class but instead is overridden in sub classes. Using this system allows the program to easily swich between methods, with the `Solve` function being called from each subclass.

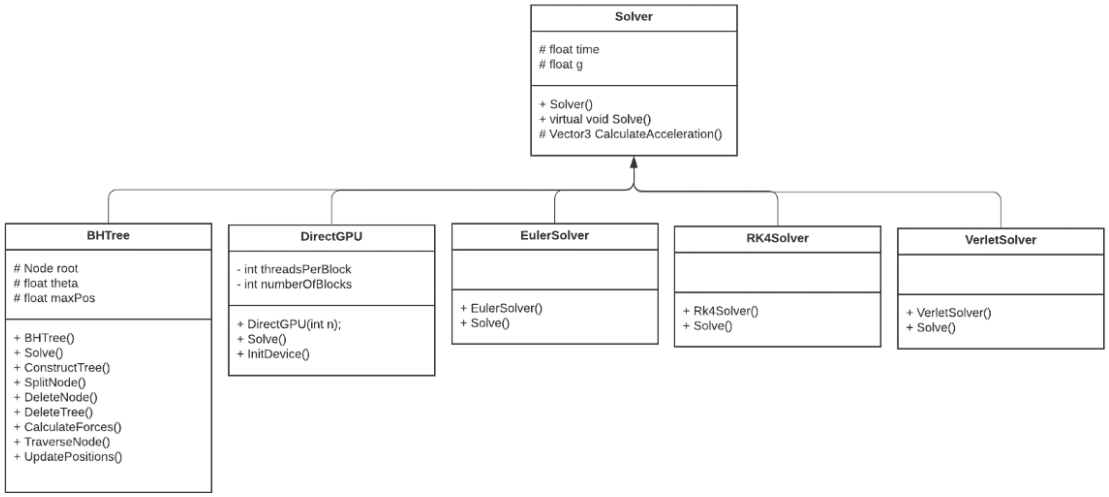

Figure 5: UML diagram of inheritance from Solver Class. All classes override virtual Solve() function, classes have access to inherited CalculateAcceleration() function.

UI is implemented to communicate to the user the method, number of bodies, timestep, the runs the program will do before outputting results and value of theta. The default values can be altered by changing the values in the file setup.txt, allowing for easy testing. Values can also be changed by using keyboard controls. As the time step is changed the simulation will speed up or slowdown in real time. Changing other values will take effect when the 'r' key is pressed, restarting the simulation.

3.2. Euler Method

The Euler method is the simplest method of solving the N-body simulation and therefore it was the first to be implemented. Like all direct methods, all particle's gravitational effect on every other particle must be summed. To achieve this a for loop iterates through all particles, using this iteration the program will find the force acting on this particle, the i particle. Inside this loop is a second for loop that loops through every particle again, this is the j loop. Since particle i cannot apply a force to itself, the program checks that $i \neq j$ to make sure the two loops do not refer to the same particle. The force acting on particle j is defined in Equation 1.

$$F_i = \sum_{i<n, j<n}^{i \neq j} \frac{G m_i m_j (r_j - r_i)}{\|r_{j-r_i}\|^3}$$

Equation 1: Force Calculation

Where m is the mass of the particle, r is the position of the particle's centre of mass and G is the gravitational constant which is 6.6743x10^{-11}. Since $F_i = m_i a_i$, rather than calculate the force and then divide by m_i, the equation for acceleration can be derived by dividing the force formula by m_i as is shown in Equation 2.

$$a_i = \sum_{i<n, j<n}^{i \neq j} \frac{G m_j (r_j - r_i)}{\|r_{j-r_i}\|^3}$$

Equation 2: Acceleration Calculation

After the j loop has completed for one particle, all accelerations have been summed for this particle using the acceleration calculation function inherited from the solver class and therefore the velocity and new position can be calculated using Equation 3.

$$v(n+1) = v(n) + a(n)h$$

Equation 3: Euler for velocity

Where $v(n+1)$ is the new velocity, $a(n)$ is the acceleration that has just been summed and h is the timestep. Next, Equation 4 is used to calculate the position.

$$x(n+1) = x(n) + v(n)h$$

Equation 4: Euler for position

Where $x(n+1)$ is the new 3-dimensional position and $x(t)$ is the current position. It is important not to update the positions of the bodies until after all accelerations have been calculated, if the positions were to updated during the same stage some accelerations would be calculated based on position at time n and some at position from time n+1. This could cause inaccurate and unstable orbits.

3.3. Verlet Method

Verlet integration can be also used to integrate the velocities and positions of particles. Equation 5 is used to calculate positions.

$$x(n+1) = x(n) + v(n)h + \frac{a(n)h^2}{2}$$

Equation 5: Verlet for Position

Velocity is then calculated using Equation 6:

$$v(n+1) = v(n) + \frac{a(n) + a(n+1)}{2} h$$

Equation 6: Verlet for velocity

Since the velocity calculation requires the position of all particles to be calculated to find $a(n+1)$, an initial loop calculates all new positions at time $t+h$, shown at line 1 and 2 of figure 6. Once the initial loop has calculated the positions, line 3 and 4 start a new i loop and a nested j loop, line 5 uses the new positions in the acceleration calculation to find the new accelerations. Lines 6 to 8 adds the values of $\frac{a(n)}{2} * h$ and $\frac{a(n+1)}{2} * h$ to $v(n)$, and updates the acceleration.

```
1   Loop all particles
2         Calculate new position
3   Loop i for all particles
4         Loop j for all particles
5               if i != j
                      Sum acceleration
6         velocity += oldAcc * 0.5 * timestep
7         particle acc = summed acceleration
8         velocity +=  newAcc * 0.5 * timestep
```

Figure 6: Verlet pseudo code

3.4. 4th order Runge-Kutta

The 4th order Runge Kutta method considers the position and acceleration of particles at 4 timesteps. The RK4 method is defined in equation 7.

$$y(n+1) = y(n) + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$

Where k_1, k_2, k_3, k_4 are defined as:

$$k_1 = f(t(n), y(n)),$$
$$k_2 = f\left(t(n) + \frac{h}{2}, y(n) + h\frac{k_1}{2}\right),$$
$$k_3 = f\left(t(n) + \frac{h}{2}, y(n) + h\frac{k_2}{2}\right),$$
$$k_4 = f\left(t(n) + \frac{h}{2}, y(n) + hk_3\right)$$

Equation 7: 4th order Runge-Kutta

The equations of motion are applied to equation 7. Since the calculation for the acceleration requires the position to be calculated and the calculation for position requires the velocity which is calculated using acceleration, the k value for position and velocity must be calculated before the next k values can be calculated. Therefore 4 nested loops are required to sum the k accelerations and find the k position values.

```
1   //k1
2   For i to n
3         kx1[i] = velocity[i] * timeStep
4         For j to n
5               If i != j
6                     acc =CalculateAcceleration(position i,
                      position j, particle j mass)
7         acc.scale(timestep)
8         kv1[i] += acc
9
10  //k2
11  For i to n
12        kx2[i] = (velocity[i] + kv1[i]/2) * timestep
13        For j to n
14              If i != j
15                    acc =CalculateAcceleration(position i      +
                      kx1[i] /2, position j +
                      kx1[j]/2,particle j mass)
16        acc.scale(timestep)
17        kv2[i] += acc
18
19  //k3
20  For i to n
21        kx3[i] = (velocity[i] + kv2[i]/2) * timeStep
22        For j to n
23              If i != j
```

```
24                  acc =CalculateAcceleration(position i
                    + kx2[i]/2 , position j + kx2[j]/2,
                    particle j mass)
25        acc.scale(timestep)
26        kv3[i] += acc
27
28  //kv4
29  For i to n
30        kx4[i] = (velocity[i] + kv3[i]) * timeStep
31        For j to n
32             If i != j
33                  acc =CalculateAcceleration(position i
                    + kx3[i] , position j + kx2[j],
                    particle j mass)
34        acc.scale(timestep)
35        kv4[i] += acc
36
37  //x and v
38  For i to n
39        kx1 = kx1/6, kx2 = kx2/3, kx3 = kx3/3, kx4 = kx4/6
40        kv1 = kv1/6, kv2 = kv2/3, kv3 = kv3/3, kv4 =     kv4/6
41        particles[i].position += kx1[i] + kx2[i] +  kx3[i] +
          kx4[i]
42        particles[i].velocity += kv1[i] + kv2[i] +  kv3[i] +
          kv4[i]
```

Figure 7: RK4 pseudo code

The first K values calculated in lines 1 to 8 of figure 7 are the same as those found by the Euler method, since this is at time t, therefore the position and velocity are calculated using the same values and same structure. For k2 at line 15, the positions passed to the CalculateAcceleration function are modified by adding the k1/2 values, this must be done for both the i and j particles otherwise this would be calculating the forces based on positions from different times. The k3 calculations at line 24 follow the same structure, using the values of k2 instead of k1. Similarly, at line 33

the values of k4 are calculated but the k3 value is not divided by 2 as stipulated in equation 7. With all values of k calculated, a final loop is used to find the position and velocity of each particle in lines 38 to 42.

3.5. Tree Method

The Barnes-Hut tree makes use of the node structure in figure 8.

```
1  //Node
2  int : particleCount
3  Vector of Node pointers : children
4  Vector3 : averagePos
5  float : combinedMass
6  float : sideLegnth
7  Vector3 : position
8  Vector of particle pointers : particles
```

Figure 8: Node Structure

The `particleCount` value stores the number of particles in this node, in the root node this will be the total number of bodies, the vector `particles` points to each of the particles inside the node's domain. The `children` vector stores pointers to the children of the node. The float value `combinedMass` is used to store the combined mass of all particles stored in the node. The Vector3 `averagePos` stores the weighted average position of all particles in the node. `Position` stores the 3-dimensional position in space representing the centre point of each octant. The constructor of the Barnes-Hut Tree class adds all particles to the root node. The next step is to find the `maxPos`. `maxPos` is found by comparing the absolute value of each particle in the x,y and z and finding the maximum value. `maxPos` will be updated when the position is updated if the absolute value of any particle's position becomes greater than the current value of `maxPos`. When updating the tree, the update function follows the procedure outlined in figure 9.

```
1  Delete Tree
2  Construct Tree
3  Calculate Forces
4  Update Particles
```

Figure 9: Tree Update process pseudo code

The delete tree function clears any allocated memory in the tree since every update particles move and need to be stored in a different structure. The function starts at the root and works down the tree recursively, deleting and removing pointers. The construct tree function passes a reference to the root to the `SplitNode` function, this starts the iterative tree construction process. Pseudo code of this process is provided in figure 10. If the function reaches line 14, the function is called again, this time splitting the child that the loop at line 5 has reached.

```
1    halfSide = currentNode.sideLegnth* 0.5
2    Create 8 child nodes
3    Loop all particles in currentNode
4         Find which child this particle belongs to and insert
5    Loop all children
6         if child's particle count is 0
7              Set to Null
8         if child's particle count is 1
9              Set combined mass to mass of particle
10             Set averagePos to position of particle
11        else child has >1 particles
12             Set combined mass
13             Set averagePos
14             Pass this child to SplitNode()
```

Figure 10: Pseudo code for SplitNode function

The combined mass is found by simply summing all masses in the node as shown in equation 8.

$$cm = \sum_{i=0}^{n} m_i$$

Equation 8: Combined Mass

Where cm is the combined mass and n is the number of particles in the node and m is the mass of the particle**.** The weighted average position is used to find the centre of mass distribution, shown in equation 9.

$$p = \frac{\sum_{i=0}^{n} x_i m_i \, m}{cm}$$

Equation 9: Weighted average position

Where p is the average position and n is the number of particles in the node, m is the mass of the particle and x is the position of the particle, cm is the combined mass. With the tree constructed, the next step is to calculate the accelerations of all particles. The `CalculateForces` Function, shown in figure 11, loops through all particles, and traverses the node calculating the acceleration using the average position and combined mass if it is suitably far from a group of particles otherwise using exact positions and forces.

```
1   Loop all particles
2         TraverseNode(currentParticle, root node )
```

Figure 11: CalculateForces pseudo code

The `TraverseNode` function is used to recursively traverse the node to find the acceleration. It is passed a node and the current particle.

```
1   for children of current Node
2         if child has 1 particle
3               Calculate Acceleration using exact position
4         else if child is not null
5               if child's sideLegnth /distance to particle
                from node's avg pos < theta
6                     Calculate Acceleration from average
          position and combined mass
7               else
8                     Recursively call traverseNode
```

Figure 12: TraverseNode function pseudo code

With all accelerations calculated and added onto the velocity, the position can now be updated using the Euler Method.

3.6. GPU Parallelisation

To further increase the performance of the Simulation, CUDA is used to parallelise the Euler Method. The kernel code for Euler is in fact very similar to the code

implemented in 3.2 and uses equations 3 and 4, and the method would be almost the same if ran on a single thread. One major difference is that the GPU does not allow the creation of new instances of classes defined in CPU code or access functions from those classes. Therefore the code does not use the `Vector3` class or its functionality such as the length function that would return the magnitude of a vector, instead we have created arrays with 3 elements that the GPU understands and avoid calls to functions by reimplementing functionality. This is not a major issue but is an example of the difficulty presented by CUDA. Inside the particle manager, the particle array is allocated `n * sizeof(Particle)` memory in unified memory using `cudaMallocManaged`, this allows the particles to be accessed by both the GPU and the rest of the program. The threads per block is set to 256, this was chosen as threads per block should always be a multiple of 32 for efficiency since kernels issue instructions in warps of 32 threads. The number of blocks is set to `(n + threadsPerBlock - 1) / threadsPerBlock`. This finds the appropriate number of blocks for the number of bodies being processed, keeping the number of bodies each thread requires to process to 1 if possible. When `Solve()` is called the `EulerAcceleration` kernel is called, and is ran on `threadsPerBlock` threads and `numberofBlocks` blocks as defined. This is passed `n`, `particles` and `timestep`. The Kernel starts by finding the `id = blockDim.x * blockIdx.x + threadIdx.x` which represents the current thread's id. Similarly, `stride = blockDim.x * gridDim.x` which represents how many bodies to jump in the i loop. The i loop begins at the id and will run while `i < n`, incrementing by `stride` each loop. Then the j loop begins as normal, the kernel checks that `i != j`, calculates and sums the acceleration for each j particle. Once all j accelerations are summed these are multiplied by the time step and added to the velocity.

After the call to the kernel is the `cudaDeviceSynchronize()` call which makes the program wait for all threads to finish before moving on. This is important as some kernels will take longer than others and positions cannot be updated until all accelerations are calculated.
Once all kernels are finished the second kernel is called, this is the `EulerPosition` call which runs on the same number of threads and blocks as defined earlier. Again, `id` and `stride` are set in the same way. The kernel then loops for i, while i < n,

incrementing by the value of `stride`. Each loop will update the position using equation 4. A final call to cudaDeviceSynchronize is made to make sure all positions are successfully updated before the program continues.

3.7. Recording Results

To investigate the efficacy of each method, timings were recorded to show the time taken to solve the simulation step. To make sure that the recordings reflect the method and are not affected by other parts of the program, the recording starts directly before the solve step and stops immediately after, before continuing with other steps such as updating the graphics which could result in times that do not properly reflect the methods themselves. To understand the efficiency and effectiveness it is important to also understand the accuracy of the simulation. Since the law of conservation of energy applies the total energy in the system should stay constant throughout the simulation if there was no error. Equation 10 shows the summation of kinetic and potential energy to find the total energy of the system (Medium, 2020).

$$E_t = \sum_{i<n} \frac{1}{2} m_i v_i^2 + \sum_{i<n, j<n}^{i \neq j} \frac{G m_i m_j}{|r_j - r_i|}$$

Equation 10: Total Energy

The summation of energy does not need to be done at every step as it should remain close to constant, instead it is recorded every 10 steps since it takes significant time to record, performing a nested loop over all other particles.

Test Case 1

Test Case 1 records the run time of each method at each timestep, the simulation is run for 100 timesteps, with n being varied. All methods apart from the Barnes-Hut are ran once with the Barnes Hut being run multiple times with the value of θ being varied.

Test Case 2

Secondly, a test is run to find the energy of the system. The simulation is run with N varied for each method. Once again, the Barnes-Hut is tested with the value of θ varied. This test records energy at the start of the simulation and then records the total energy 9 times at every 10th timestep.

Test Case 3

A final test case considers the energy of the system of 1000 bodies over a period of 100 seconds. Again, the Barnes-Hut is tested multiple times with θ varied. The time step is varied by an order of magnitude for each method, recording the energy at every 10^{th} timestep for the values of timestep = {0.01,0.1, 1, 10}.

4. Results

4.1. Overview

Following the implementation of all methods mentioned in the methodology section and the implementation of timing and energy recording functionality the tests outlined in 3.7 were conducted.

All tests were performed on a system utilising; Intel Core I7-9750H CPU with a clock speed of 2.60GHz, 16GB of RAM and an NVIDIA GeForce GTX 1660 Ti.

4.2. Test Case 1

Test Case 1 produced a set of 100 times for each method (4 separate sets of results for Barnes-Hut with θ at 0.5, 0.75, 1.0 and 1.5). From these the median was found in each test varying the value of N. These medians are plotted in figure 13 which shows the times increasing with the increase in N. Table 1 also shows that the GPU method provides slow times at small N but produce the fastest run times at larger values of N. Similarly, the Barnes-Hut times at small N produce slower times than the direct methods but perform faster at larger values of N.

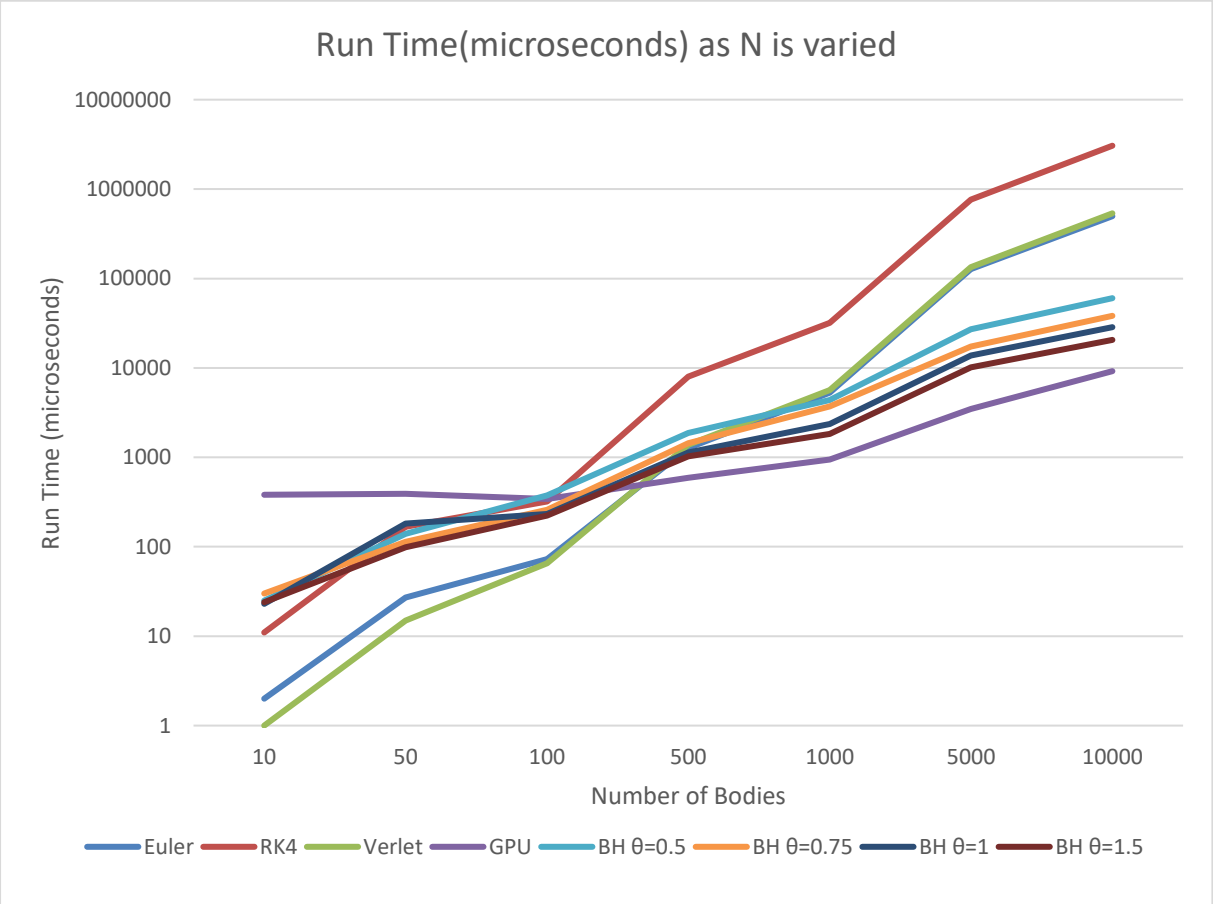

Figure 13: Run times of each method

4.3. Test Case 2

Test Case 2 recorded 10 energies for each method as the value of N is varied. The change is shown in both negative and positive values to show the spread of values for each method. The Barnes-Hut is tested at θ=0.5 and θ=1.5 to find if there is a difference in energy change when using a low value of θ and a higher value. The first energy of the system is recorded before the first step of the simulation and is an accurate reading of the total energy in the system. To show the accuracy of each simulation the other 9 values recorded every 10 timesteps are used to find the change in total energy at each time. Figures 14 to 17 show the change in energy for each method as N is varied. These figures show the Runge-Kutta to be the closest to constant, and the Verlet integrator to have the largest variance.

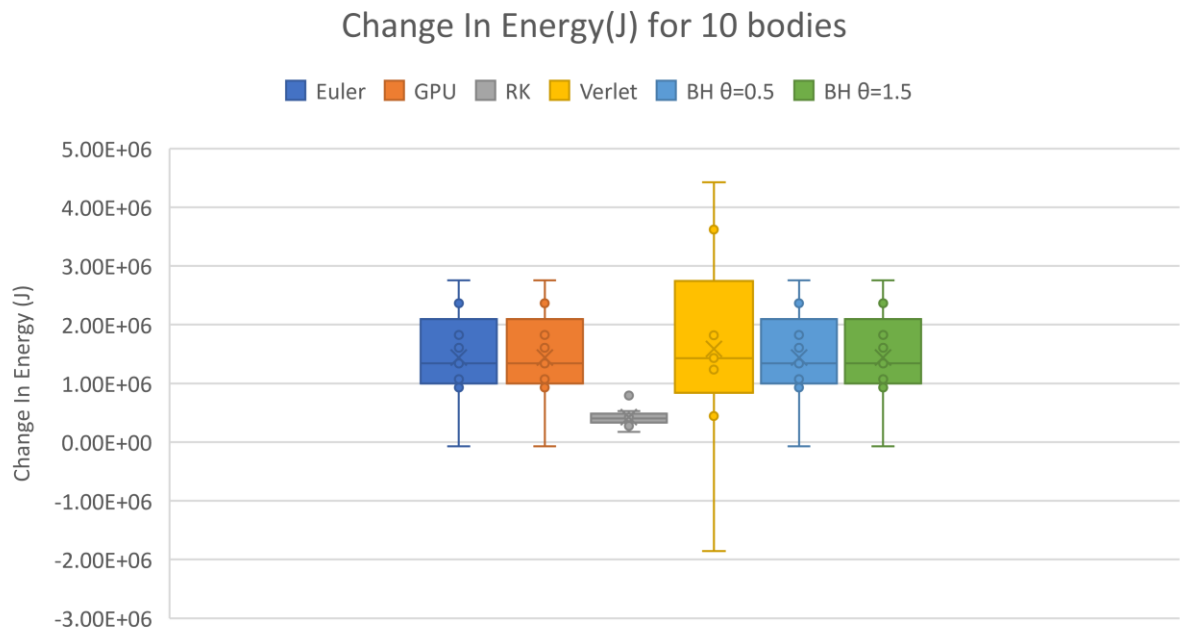

Figure 14: Change in Energy for 10 bodies shows the RK4's maximum change in energy to be around 1/6th of the maximum change in energy of the Verlet

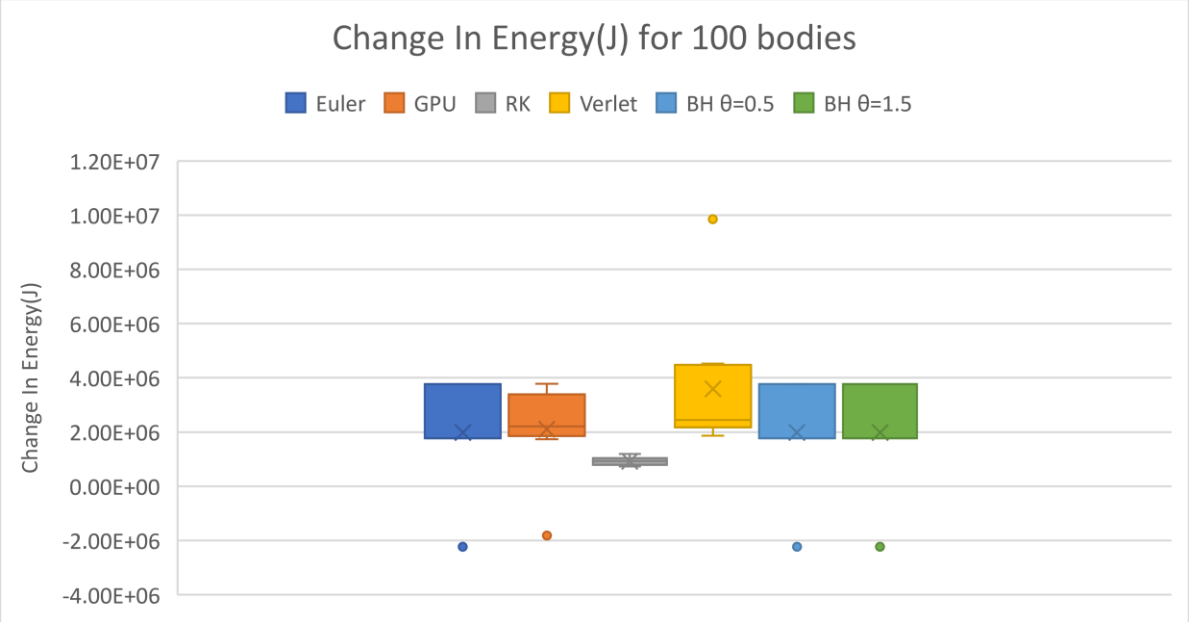

Figure 15: Change in Energy for 100 bodies shows the RK4's maximum change in energy to be around 1/8th of the maximum change in energy of the Verlet

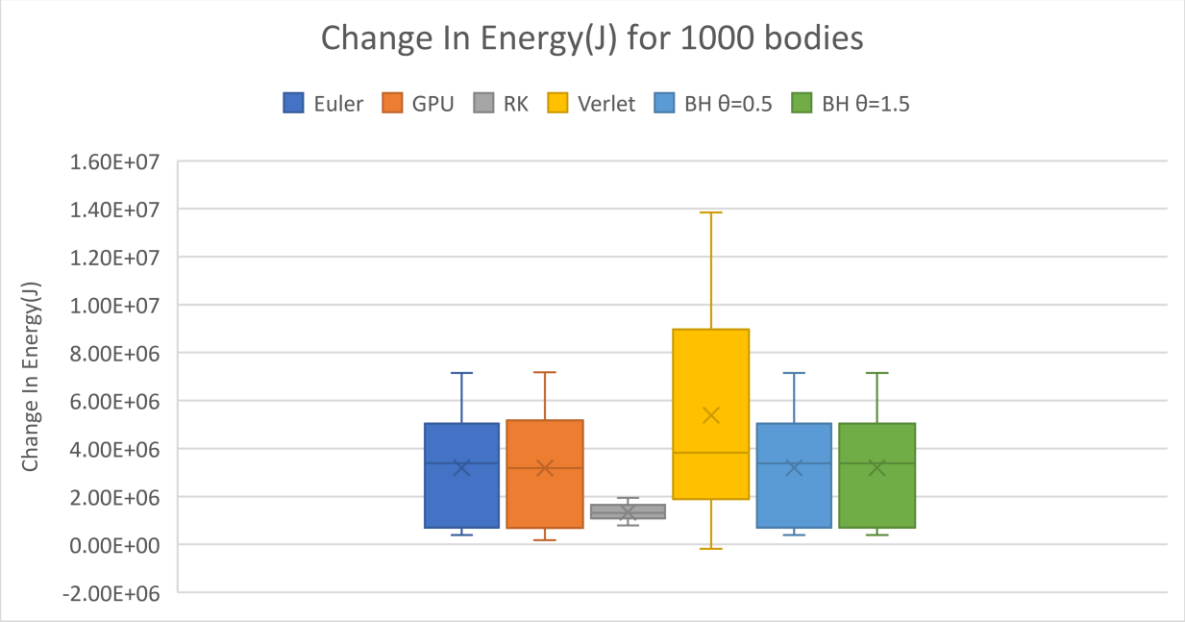

Figure 16: Change in Energy for 1000 bodies shows the RK4's maximum change in energy to be around 1/7th of the maximum change in energy of the Verlet

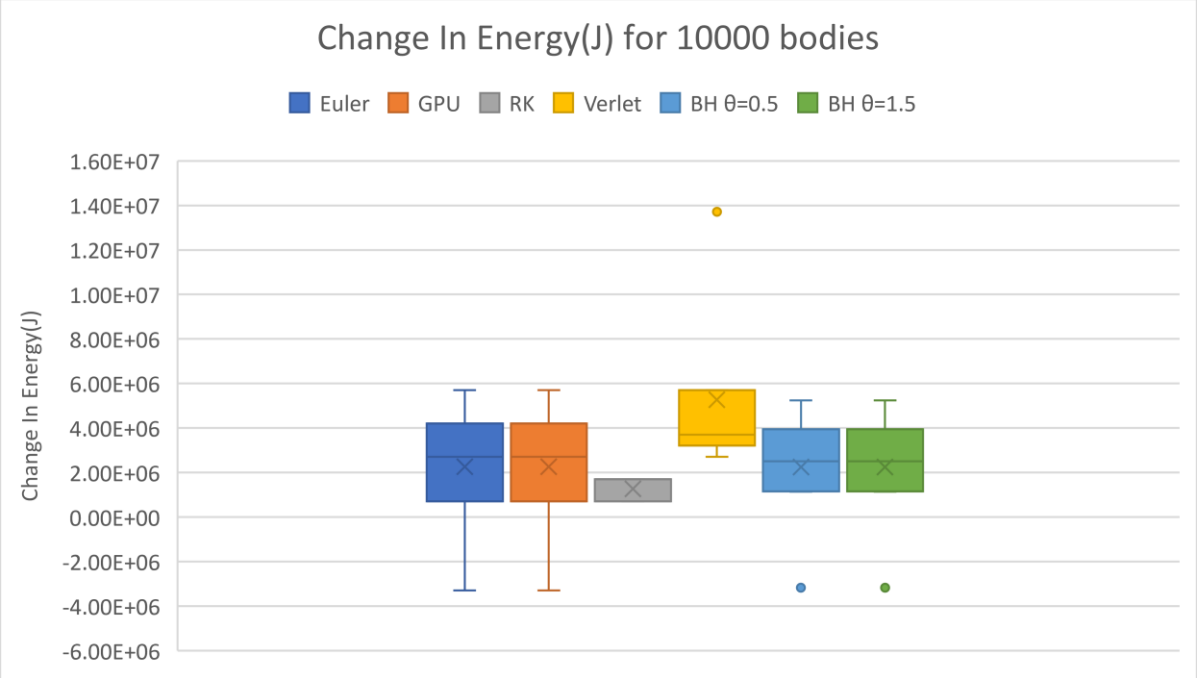

Figure 17: Change in Energy for 10000 bodies shows the RK4's maximum change in energy to be around 2/9^{ths} of the maximum change in energy of the Verlet

4.4. Test Case 3

The third test case found the initial energy for a system of 1000 bodies and simulated this system for a period of 10 seconds, recording the energy in the system every 10 timesteps. This test was repeated varying the timestep by a magnitude of 10, starting with 1000 steps of 0.01 seconds until completing just one step of 10 seconds. Figures 18 through 23 graph the difference in energy between time = 0s and time =10s, showing that larger timesteps result in a bigger change in energy. At timestep 1 the Runge-Kutta Method has the largest change in energy, but at timestep 10 the change in energy is greater for Euler, Euler GPU, and both Barnes-Hut methods. At low timesteps the change for all methods is closer to 0. The results are shown in different figures as the results of some methods are very close and would therefore overlap in one graph. Table 1 shows the same results but rather than the total change in energy of the whole system it shows the mean change in energy. Finally, figure 24 shows the change in energy over the period of 10s. This figure shows the change in energy oscillating around the line of 0J.

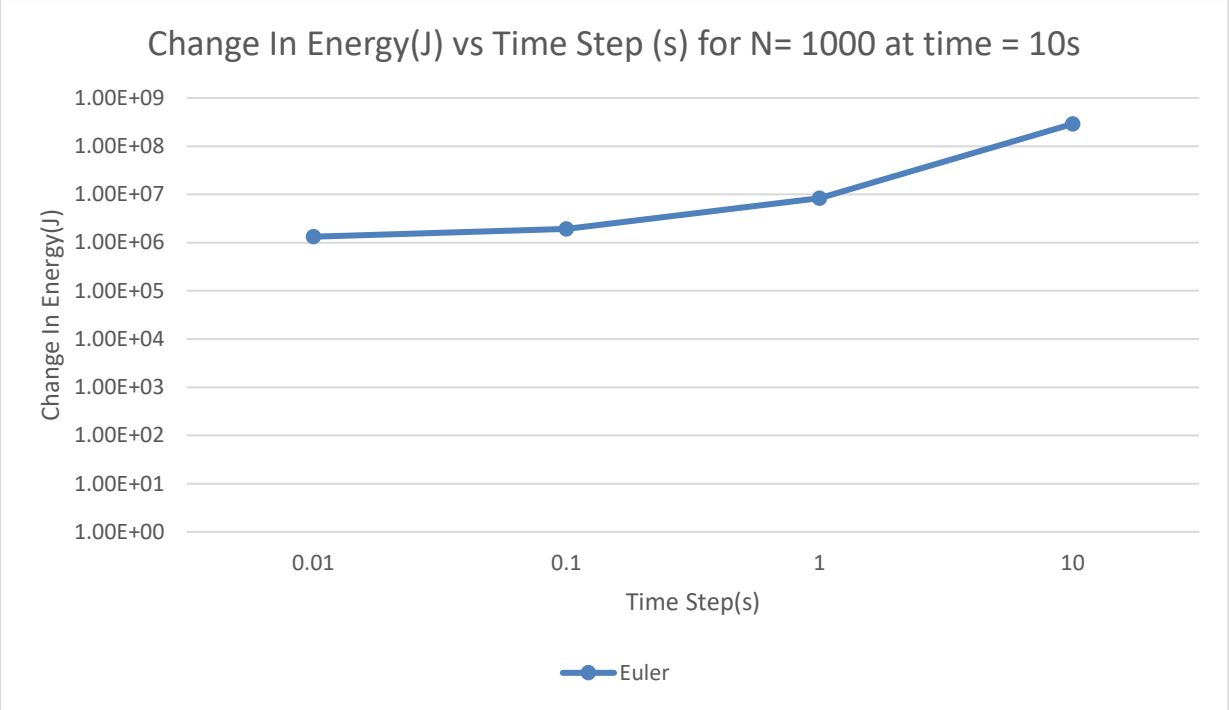

Figure 18: Change in energy of Euler method

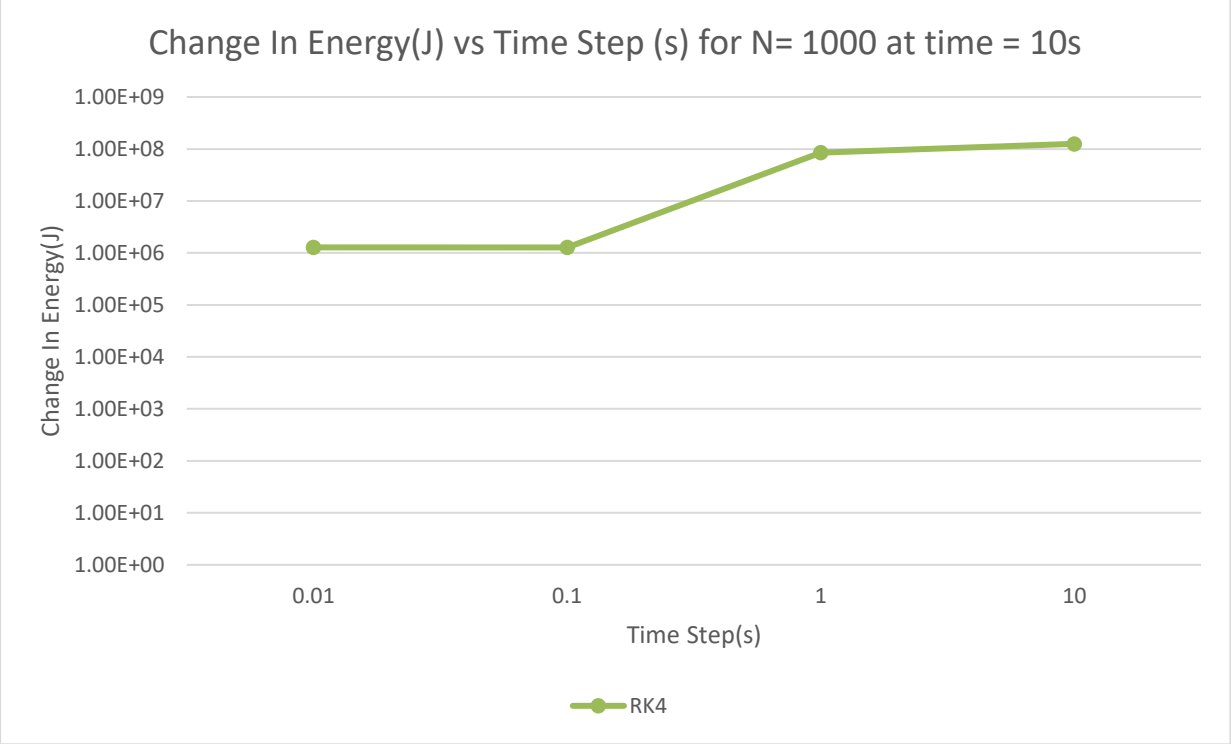

Figure 19: Change in energy of Runge-Kutta method

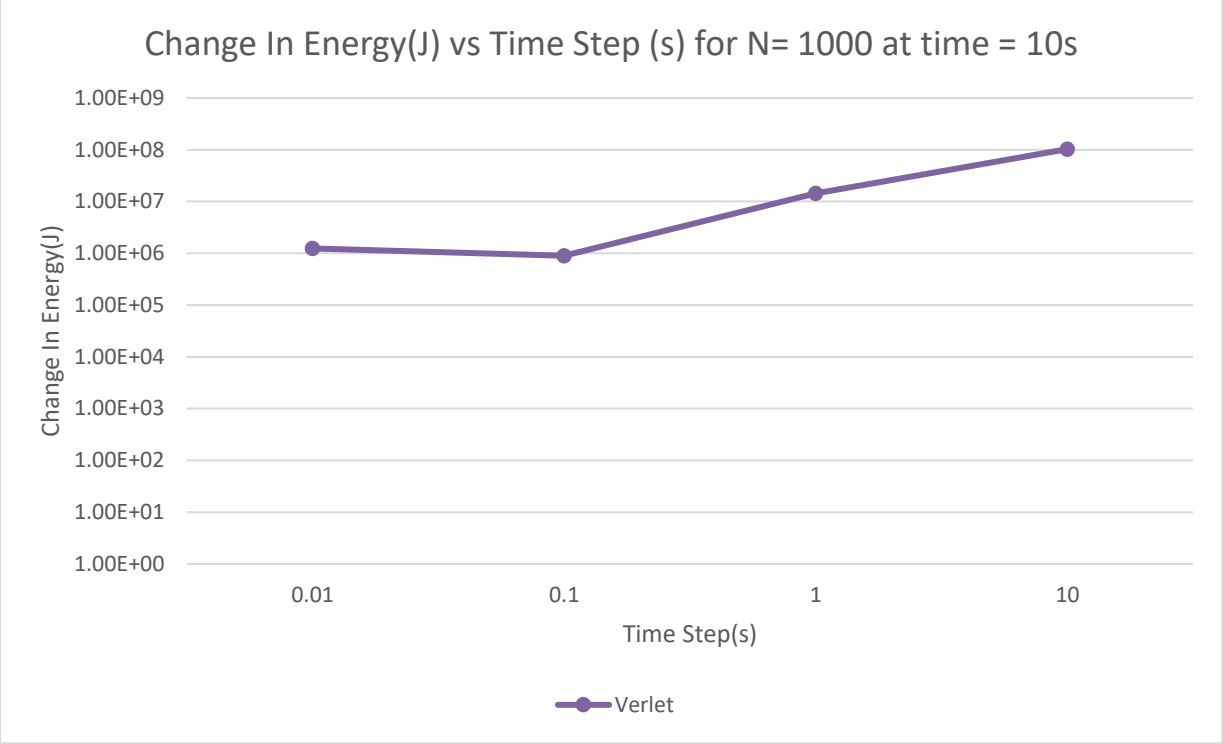

Figure 20: Change in energy of Verlet method

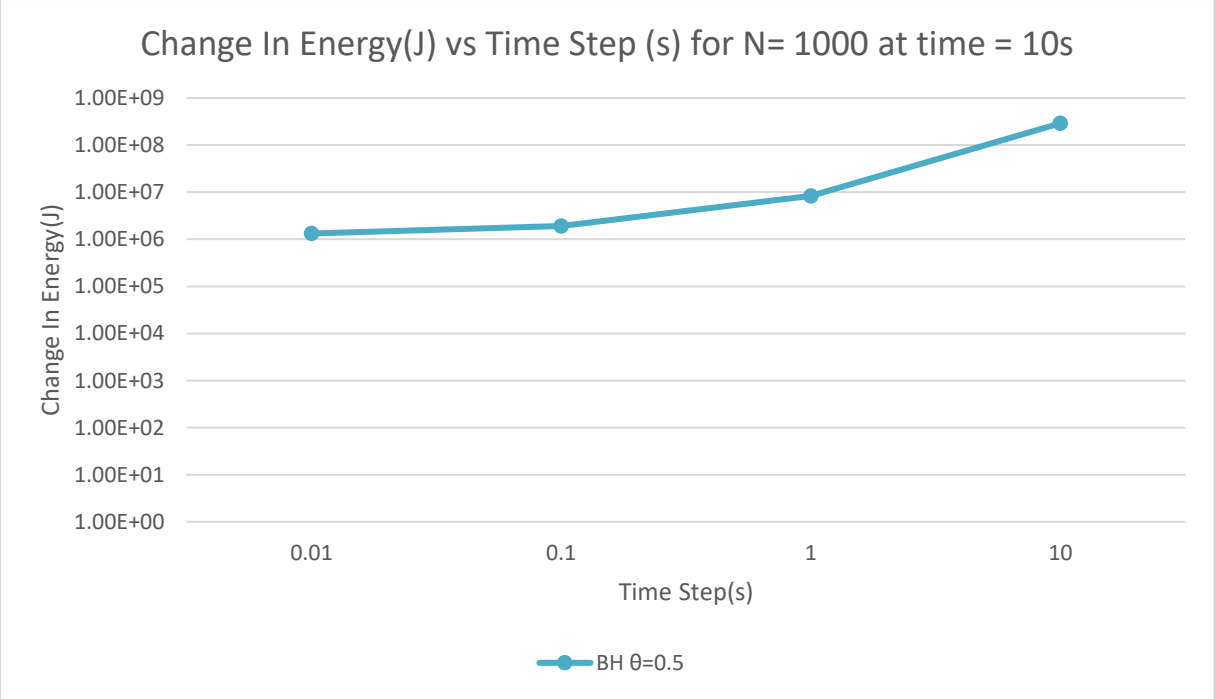

Figure 21: Change in energy of Barnes-Hut with θ=0.5

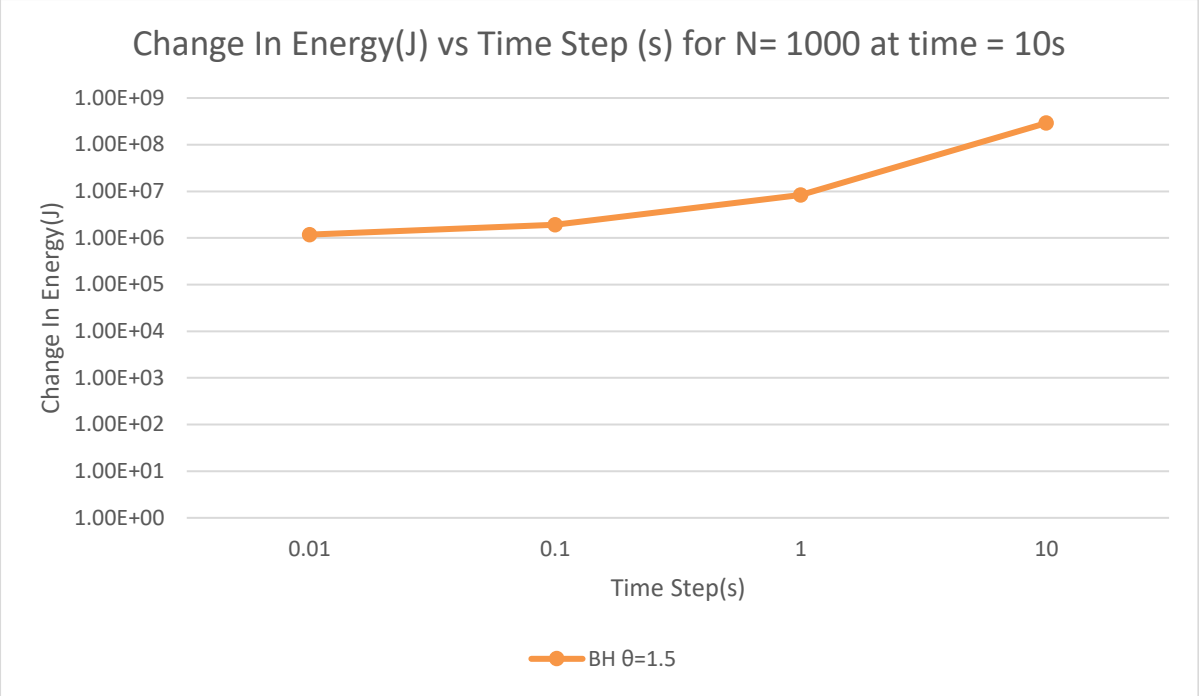

Figure 22: Change in energy of Barnes-Hut with θ=1.5

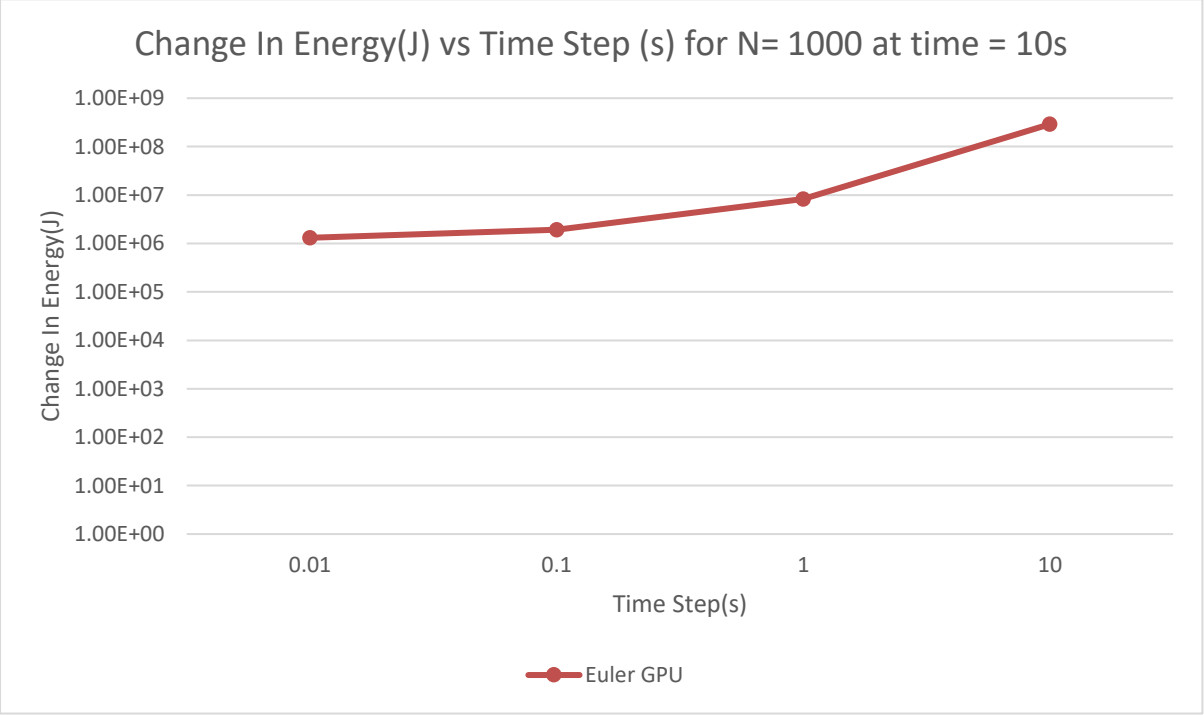

Figure 23: Change in energy of Euler GPU method

Timestep	0.01	0.1	1	10
Euler	11583.42	241500	4214000	2.90E+08
RK4	11551.45	114160	9165700	1.26E+08
Verlet	11955.04	257340	4118600	1.03E+08
Euler GPU	11581.42	241480	4213900	2.90E+08
BH0.5	11583.42	241500	4214000	2.90E+08
BH 1.5	11583.42	241500	4214000	2.90E+08

Table 1: Mean of absolute changes in energy

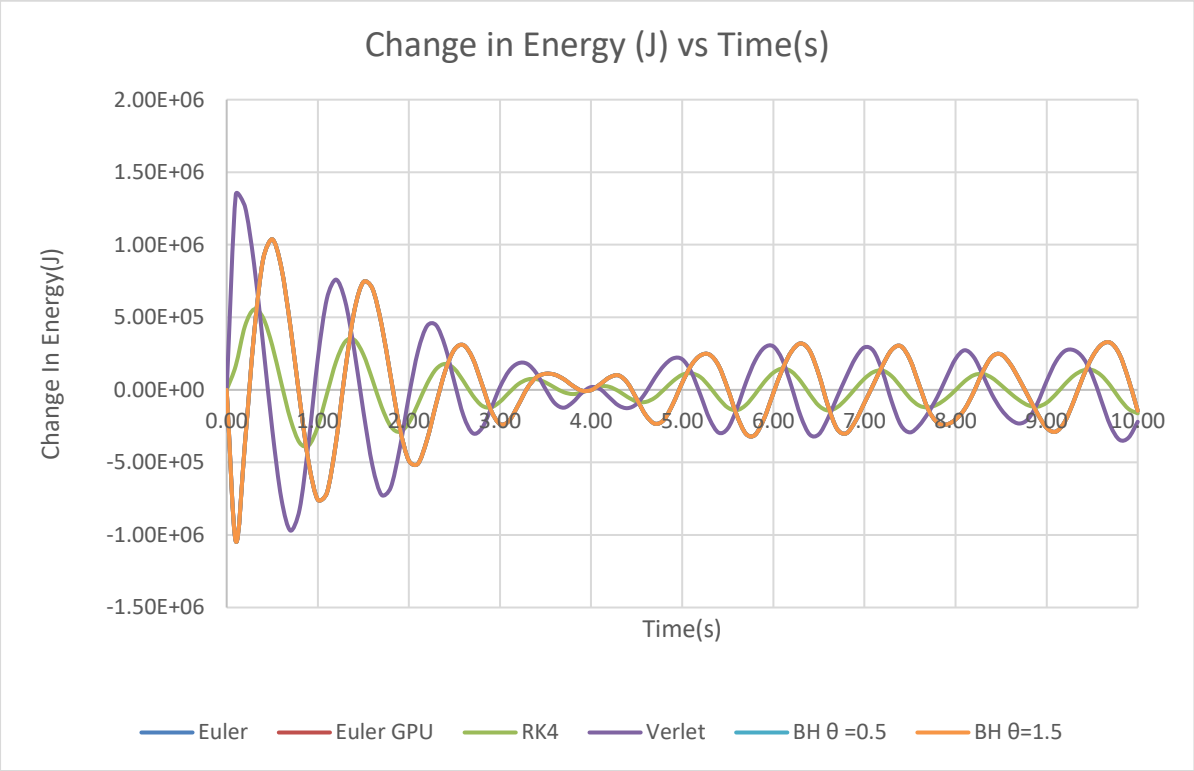

Figure 24: Oscillation of Energy

5. Discussion

Using the run time and energy results obtained, the research question can be addressed, discussing the advantages and disadvantages presented in terms of accuracy and performance of each method for different contexts.

5.1. Performance

Before testing, the hypothesis was that the Runge-Kutta would be overall the slowest method, followed by the Verlet method and then the Euler method for the Direct CPU methods. The Barnes-Hut would be slower for very low N but would be significantly faster than the direct method as N increases. The hypothesis expected the GPU method to perform better than the direct method for every value of N tested.

Looking at figure 13, as expected, the Runge-Kutta is the slowest of the direct methods, this is because it must calculate 4 position and acceleration values in 4 nested loops which will take more time, comparing the pseudo code of the Verlet and RK4 from figure 6 and 7 the complexity of the RK4 method is clear. The Verlet and Euler methods produce similar results, for small N the Euler method performs slower than the Verlet method, but at N=500 the timings become faster than the Verlet. It is expected that both produce similar results since both make use of one nested loop and a second single loop. The Verlet method becomes slower due to the extra calculations when adding to the velocity as can be seen when comparing equation 3 and equation 6. The Euler simply scales the acceleration by the timestep and adds it to the velocity of the particle. The Verlet has extra processing to do, scaling the old acceleration, adding this to the velocity, updating the acceleration and then doing the same with the new acceleration, the same is true for the position calculation. As the bodies increase this step will take more time and will therefore slow down the Verlet method. All direct methods approximately follow $O(n^2)$ scaling.

At low n, the Barnes-Hut is very inefficient and slows down the simulation. Considering the Euler simulation median time is 2μs for 10 bodies, for the Barnes-Hut to achieve a speed up at 10 bodies, the tree building stage would have to take less than this time. Therefore, the Barnes-Hut is unable to achieve a faster speed than the brute force Euler method until there are 500 bodies in the simulation. With

500 bodies the speed up is only achieved for the higher values of θ as the tree building still takes significant time. As N is further increased to 1000, the performance of the Barnes-Hut θ = 0.5 becomes about 1.2 times as fast. This performance gain increases significantly as N increases, at N =10000, the Barnes-Hut θ = 0.5 is more than 8 times faster than the direct Euler and higher values of θ provide even faster results, with θ = 1.5 achieving an increase in over 24 times the standard Euler method. The Barnes-Hut methods scale with O(n log n) complexity.

The Euler GPU method proves to be very inefficient for low N and does not return a performance increase until N =500. The kernels are launched in parallel on 256 threads per block. The number of blocks will not increase from 1 until there are 257 bodies so it is expected that there will be some decrease in performance due to only making use of one block of threads. The simulation still benefits from parallelism as the threads on that block will run in parallel, however it is the memory overhead associated with accessing unified memory that will slow down the system. For larger values of N, this overhead is insignificant as the simulation times grow and the GPU outperforms the Barnes-Hut and all direct methods. From N =500 the performance of the GPU scales with approximately O(n) complexity. The tests were only performed on one CPU and one GPU due to limited access to hardware, running tests on varying levels of hardware would have been interesting to see the effect of differing CPUs, GPUs and memory.

5.2. Energy

Tracking the energy in the system can be used to find the accuracy and stability of the system. Figures 14 to 17 show the change in energy for different systems. A perfect simulation would have no change in energy; however, every method will have some level of error and therefore will result in a change in energy. The Runge-Kutta as expected has the smallest change in energy for every system, this is because the Runge-Kutta takes extra steps to calculate the weighted average of four increments, therefore finding more accurate accelerations and positions than the Euler method. The Verlet method has the largest variance in results, this at first makes it seem less accurate than the other methods but looking at figure 24 shows that the energy does vary a lot at the start but in general oscillates closer to the line of 0J than the Euler

based methods, however it is not by a great margin and is visibly not as accurate as the Runge-Kutta.

The Euler, Barnes-Hut and GPU methods all provide very similar results. It makes sense that the Euler and GPU would produce almost exactly the same results as these mathematically are the same. The Barnes-Hut would produce the same results with theta at 0 as it would compute the same calculations. Despite the approximations made by the Barnes-Hut there is no significant change in energy for either of the Barnes-Hut tests compared to the direct Euler. This is because the calculations that the Barnes-Hut approximates are for particles very far away from each other therefore having a small effect on the accuracy. Since the system consists of N-1 particles with small masses orbiting around one massive particle and there is more mass in the centre particle than all other masses combined, the accelerations are affected more greatly by the single massive particle than any other particle. The masses that are most likely to be approximated are the small particles on the edges of the system that will have little effect on the overall acceleration and therefore maintain accuracy. A system that has multiple particles that are comparatively massive or all similar may prove to have less accuracy with larger values of θ.

Figures 18 to 23 show that in general smaller timesteps result in a smaller change in energy and therefore a more accurate result. However, the Verlet method has a smaller change in energy for the 0.1 timestep than the 0.01. Test case 2 however has shown that the Verlet produces the biggest variance in results. Therefore, we can say that this is caused by the oscillation shown in figure 24. Similarly, the Runge-Kutta is shown to have a larger change in energy but this is most likely due to different rates of oscillation at the end point. Since figures 18 to 23 do not take these oscillations into consideration they are fairly limited in showing the level of scaling but in general show the trend that with larger timesteps the change in energy becomes larger.

5.3. Efficiency and Efficacy

These results prove why games often steer clear of N-body simulations where possible, accurate integrators such as the RK4 method are slow and would be unsuitable for simulating long periods of time at high speeds. Even though the RK4 method would provide more accurate results than a simpler Euler method, over a long period of time the inaccuracies would add up and could cause unstable orbits as suggested by Gemmeke (2005). Despite being significantly faster than the RK4, the Euler and Verlet methods are still slow processes. Games have many other systems and processes they need to use their processing power for, therefore sharing a significant percentage of this processing power is not an option. This is even evident when running the artefact at large values of N, the times output in the results are accurate of how long the simulation step took however since it is a visual simulation the rendering step also takes significant time. A game like Universe Sandbox (Giant Army, 2015) can focus more of its processing power to N-Body simulation since this is one of the core mechanics of the game. Despite this, Universe Sandbox omits the 4th order Runge-Kutta in favour of less accurate integrators such as Euler, Verlet and the 2nd order Runge-Kutta. Most likely this is due to how computationally expensive the 4th order version is. There is a fine line between accuracy and efficiency that must be achieved to avoid chaos of unstable orbits and the performance to achieve playability.

The Barnes-Hut is effective at achieving significant speed up while keeping errors low. The approximations made in the simulation clearly had little impact on the accuracy but had a great impact on the performance. To achieve a fast running simulation using the Runge-Kutta the timestep would need to be increased to make up for the slow run times of each step, increasing the timestep is likely to cause large errors. This contrasts to the Barnes-Hut, where the timestep can be kept small to avoid error. In a real-time application, where the simulation must simulate a period of time efficiently the Barnes-Hut may provide more accurate results because it is able to keep the time step low. While there was found to be no significant difference in accuracy between the Barnes-Hut and direct Euler method in figures 14 to 17 this was likely due to the conditions of the system, the potential lack of consistency could be a concern for applications where accuracy and predictably is required.

The GPU Euler provides the fastest execution times for most systems. It is unsurprising that the GPU can achieve much faster times than the serial CPU versions. The greatest limitation of the GPU is itself; GPUs are expensive and not every PC will have one and the CUDA code written will only execute on NVIDIA GPUs, therefore applications cannot rely on GPU methods solely as they need to be accessible to a wide range of system requirements. Nyland, Harris and Prins (2007) claim that their GPU code runs more than 50 times as fast as the Serial version, figure 13 shows that this is true for the system with N=10000 but is an over statement for smaller N systems. The GPU code is a general solution for all sizes of systems. Given that the GPU code performs poorly for low values of N, splitting the work differently for different values of N and implementing different solutions may result in more optimized performance.

So far, we have considered the run time performance and accuracy of each method, an important consideration to the efficacy of the methods however is the complexity of implementing each method. The Euler method is a very simple method that can be easily implemented. The simplicity of it also made it an obvious candidate for parallelisation. The Verlet and 4th order Runge-Kutta methods are more complex and require a more in depth understanding of maths and integration. The complex nature of these integrators also makes them more difficult to parallelise. The Barnes-Hut method is also complex and requires greater development time. CUDA is a complicated language and has a steep learning curve, hence why only the simpler Euler method was implemented in CUDA. Since CUDA was implemented last it was also difficult to add into the existing project, the setup itself taking time. Beyond this, CUDA is difficult to debug and understand, increasing development times.

5.4. Simulation Considerations

Considerations were taken to make sure the results produced by the test cases were consistent and an accurate representation of the artefact but there are still limitations. All timings were taken as the sole program running on a single PC. The simulation runs 100 timesteps for each method in test case 1 which provides a good sample of data. However, there is some obvious outlying data in the results, using the median of the timings helps to mitigate these outliers. Summing the total energy of the system is a computationally expensive process and therefore is only

performed every 10 timesteps. Summing the energy every step would provide a more accurate representation of the energy in the system but would have slowed the process down considerably. Therefore, it is possible that the largest deviation from the starting energy may be larger than the change shown in the results, despite this the results still provide a good evaluation of the change in energy in the system. Test case 3 only finds the energy for a simulation period of 10s. In applications where N-Body simulation could be applied like games such as Kerbal Space Program (Squad, 2011) or Universe Sandbox (Giant Army, 2015) the simulations will be required for much larger periods of time up to potentially millions of years. In the context of this timescale the miniscule changes in energy add up to become much larger and therefore accuracy will decrease.

The simulation has some limits, firstly the program only simulates a disk system of N-1 bodies orbiting round 1 massive body, simulating other systems of particles would provide more robust results. In reality, planets, moons, suns and every object are made up of lots of individual particles with different masses. It would be impossible to simulate every single particle making up every object, therefore objects are simulated as point masses, a single point in the simulation that represents every particle making up that object. This should not be a major limitation of the simulation however, especially as the simulation is not aiming to directly simulate any specific system. This approach is common amongst games and other simulations and is the approach taken by Universe Sandbox. The system does not simulate collisions and so cannot fully simulate the path of objects that should collide. Issues can occur if particle positions are very close to each other or in the exact same position, which could occur because of the lack of collision. As the distance approaches zero the force will increase to infinity, causing the particle to shoot out of orbit and diverge from the accurate path. An option to prevent this is to add a very small number called the softening parameter to the distance. The softening parameter should be small enough to keep the simulation accurate. For the disk system, this is not an issue so softening is not used as it would cause accuracy to decrease. If two particles have the exact same location, the simulation will try to divide by the distance, 0, causing the acceleration and position to become unusable. Since every particle is affected by the unusable positions and accelerations, the simulation breaks. This also causes the Barnes-Hut to crash, unable to split the two particles into different cells it will run

out of memory. These issues do not occur in the simulation as set up but are limitations of the simulation should the starting conditions be altered.

6. Conclusion and Future Work

The discussion finds that there is no one method that fits all purposes, instead looking at the benefits and drawbacks of each method alongside the requirements of the simulation is necessary to understand which method is best suited. The choice of method is dependent on the conditions of the system, the number of bodies and the speed of simulation.

While the most accurate, the Runge-Kutta method is probably the least useful for real time applications due to its slow run time as shown in figure 13. Its use would be limited to small N simulations with aims to produce accurate results. Most games would likely not be able to spare the processing power to simulate so many calculations. Other direct methods such as the Euler and Verlet method, as well as other direct methods not implemented int this project would likely not be viable either for most applications as they scale poorly. The Barnes-Hut method is probably the most viable, but in most applications, θ would have to be kept low to keep the error low enough that it will not cause catastrophic consequences, so the gain in performance will most likely not be as dramatic as the performance boost by the Barnes-Hut with θ at 1.5 in figure 13. The Euler GPU method is an interesting method as it does not make approximations but still improves upon the Barnes-Hut even at θ =1.5. Most games however will need to utilise the GPU's processing power to render graphics, therefore while the GPU provides good results in this simulation, when adding in more complex rendering, the GPU may prove less effective. When considering all these methods, the most suitable method seems to be what the application can afford to do. If the application can afford to approximate, then the best method is the Barnes-Hut. If slow execution is not an issue and N is low, then the Runge-Kutta would be a good fit. When the GPU is not being used extensively then the GPU method will be the best suited.

Factoring in the difficulty of implementation is also important when deciding on implementation as development times will vary. For many small-scale simulations the Euler method will be sufficient. The Runge-Kutta is a much more difficult to implement method and can be difficult to debug, however if accuracy is the primary goal of the simulation then it is worth the extra development time. Implementation of

CUDA was the most complex, partly due to a lack of experience but also due to the nature of parallelism being difficult to debug. It was also difficult to implement with the system that was already in place, which could be an issue for developers looking to implement CUDA into projects with wide scopes.

Most games do not implement N-body simulations due to their inherent slow run times and unpredictability. Where performance is concerned, most games are unable to provide the processing power required for a full n-body simulation. Inaccuracy and unpredictability are perhaps even bigger roadblocks for developers, who do not want their games to be ruined by decay of orbits due to numerical errors. It is clear that for N-body simulation to be valid for games, they must be central to the game, where unpredictability of systems becomes a feature of the gameplay, otherwise using simplified approximated two body simulations like those used in KSP is much more effective. Beyond games technology, N-body simulations are used often in scientific programming to understand complex physics-based systems, for applications like these, accuracy will be key. Therefore, a slow but accurate scheme such as the Runge-Kutta would be useful.

N-Body simulation is a wide field and there are many paths that could be undertaken for future work. As evidenced from the findings in this study, the Barnes-Hut tree method and the GPU method were the most efficient in terms of time. Therefore, the obvious path for future work would be to develop a CUDA implementation of the Barnes-Hut method. The tree method is not as simply parallelised as the direct method. This can be implemented in 6 optimized kernels (Burtscher and Pingali, 2011). The first kernel computes the bounding box around the whole system. Secondly the tree is built, using a different approach to the method described in figure 10. Particles are inserted in parallel, when attempting to insert into a node already containing a particle, the algorithm will split the node recursively until they are in different nodes. This method is better suited to parallelism, dividing particles between threads. A kernel then finds the weighted average position and combined mass for each internal node. The fourth kernel approximately sorts the particles by distance to optimize the computation. Kernel 5 computes the forces of each body using the tree to determine approximations. The final kernel is a simple position

update and follows the same structure as the Position kernel implemented for the GPU method in this study.

An alternative approximation method is the fast-multiple method, similarly to the Barnes-Hut it divides the space into a grid and performs approximations based on far away particles. The grid is subdivided into smaller cells. In general, the FMM can outperform the Barnes-Hut algorithm (Belloch and Narlikar, 1997) making it potentially more suitable to real time applications.

Other than force calculation methods, collisions would add another level of depth to the simulation. Fragmentation could occur due to collisions resulting in a change in N. A brute force approach would for each particle perform a collision check with all other particles. Like with force calculations, this becomes very slow at higher values of N, to speed this up an initial broad phase detection can be used to find possible collisions. A sort and sweep method projects objects onto x, y and z axes and overlaps are checked. Alternatively, Spatial Subdivision divides particles into cells and checks for possible collisions using these cells. These methods can also gain from GPU parallelisation (Le Grand, 2007).

This project set out to create a visual real time N-body simulation and answer the research question: Considering direct methods, parallelism, and methods of approximation for N-Body gravitational simulation, what methods are efficient and suitable for real-time simulation?
The artefact created visualises the n-body simulation with different parameters and this study has answered the research question by discussing which methods are most suitable for real time simulation in given contexts.

List of References

Aarseth, S.J. (2003) *Gravitational N-Body Simulations*. Cambridge: Cambridge University Press.

Belloch, G. and Narlikar, G. (1997) *Parallel Algorithms - Series in Discrete Mathematics & Theoretical Computer Science*. American Mathematical Society.

Burtscher, Martin and Pingali, Keshav (2011) 'An efficient CUDA implementation of the tree-based barnes hut n-body algorithm', in GPU Computing Gems Emerald Edition, pp. 75–92. doi: 10.1016/B978-0-12-384988-5.00006-1.

Eagan, B. (2017) *N-Body Orbit Simulation with Runge-Kutta*, *Cyber-omelette.com*. Available at: http://www.cyber-omelette.com/2017/02/RK4.html (Accessed: 19 November 2020).

Gemmeke, J. (2005) *Detecting chaotic orbits in N-body simulations*. MSc. Universiteit van Amsterdam.

Giant Army (2015). Universe Sandbox [Video game].

Hamada, T. et al. (2009) 'A novel multiple-walk parallel algorithm for the Barnes–Hut treecode on GPUs – towards cost effective, high performance N-body simulation', *Computer science*, (24) pp. 21-31. doi: 10.1007/s00450-009-0089-1

Kerbal Space Program. (no date) *Steam Page*. Available at: https://store.steampowered.com/app/220200/Kerbal_Space_Program/ (Accessed: 25 April 2021).

Khronos Group, (2017) *OpenGL* (Version 4.6)[Computer Program]. Available at: www.opengl.org (Accessed: 21 November 2020).

Koon, W. et al. (2011) *Dynamical Systems, the Three-Body Problem and Space Mission Design*. 2nd edn. New York: Springer, pp. 4-7.

Le Grand, Scott (2007) ‘Broad-Phase Collision Detection with CUDA’, in GPU Gems 3.

Medium (2020) *Create Your Own N-body Simulation (With Python)*. Available at: https://medium.com/swlh/create-your-own-n-body-simulation-with-python-f417234885e9 (Accessed: 18 April 2021).

Nvidia (2007) *CUDA* (Version 11.3) [Computer Program]. Availlable at: https://developer.nvidia.com/cuda-toolkit

Nyland, L., Harris, M. and Prins, J. (2007) "Fast N-Body Simulation with CUDA", in Nguyen, H. *GPU Gems 3*. Upper Saddle River, NJ: Addison-Wesley.

PCGamesN (2019) *Kerbal devs tried n-body physics, but the simulation “starts to fire moons at planets”.* Available at: https://www.pcgamesn.com/kerbal-space-program-2/n-body-physics (Accessed: 3 April 2021).

Prappleizer (2020). *Writing your own RK4 Orbit Integrator (Part 1: N Body).* Available at: https://prappleizer.github.io/Tutorials/RK4/RK4_Tutorial.html (Accessed 24 November 2019)

Squad (2011). Kerbal Space Program [Video game].

Universe Sandbox (2016) *Working Through the N-Body Problem in Universe Sandbox ².* Available at: http://universesandbox.com/blog/2016/02/n-body-problem/ (Accessed: 10 May 2021).

Universe Sandbox. (2020). *Simulate Gravity*. Available at: http://universesandbox.com/ (Accessed: 25 April 2021).

Universe Sandbox wiki (2020). *N-Body simulation.* Available at: https://universesandbox.fandom.com/wiki/N-Body_Simulation (Accessed: 8 April 2021).

Appendix A – Raw timing results

Recorded times for each method before the median is calculated.

N	10	50	100	500	1000	5000	10000
Time (µs)	1	14	152	1307	5175	126052	501878
	1	14	72	1460	5181	141100	535803
	7	13	144	1308	5119	127469	497089
	1	13	51	1299	5142	129086	495792
	1	14	50	1303	5456	128333	502544
	2	14	108	1302	5317	129260	500226
	1	13	112	1305	5179	127692	499485
	1	13	54	1385	5279	128011	497656
	2	45	52	1272	5268	126202	497284
	1	28	52	1343	5377	127230	496517
	1	22	51	1267	5576	124807	494103
	1	101	172	1272	5363	127335	498753
	1	27	51	1540	5288	126754	499906
	5	24	172	1273	5364	127772	497727
	5	45	103	1271	5510	128140	497318
	5	45	74	1271	5329	128620	496789
	4	28	64	1401	5236	128318	499004
	2	23	172	1270	5294	127166	497903
	2	45	103	1283	5218	130479	498076
	2	27	73	1328	5193	129368	504040
	1	44	64	1513	5288	127513	994214
	2	27	58	1307	5367	128199	498832
	1	24	50	1273	5169	128654	498513
	1	45	132	1271	5262	128084	985753
	1	27	50	1270	5263	128580	496600
	1	20	133	1254	5165	129263	497041
	1	18	88	1270	5051	128649	498537
	1	15	73	1271	5141	128735	497656
	5	45	51	1306	5169	129550	497604
	5	34	89	1270	5092	128249	497391
	3	44	50	1270	5377	126598	494386
	2	27	133	1325	5430	126774	497650
	3	23	88	1331	5346	127336	498643
	2	17	74	1269	5329	126969	509779
	2	15	67	1351	5300	127346	509469
	2	13	51	1272	5223	125666	498436
	4	68	50	1330	5461	125158	496261
	2	35	50	1270	5404	125860	497982
	3	27	172	1269	5141	126888	497038
	2	35	105	1333	5264	126168	498812
	2	26	73	1373	5101	126468	494716

	1	15	64	1272	5402	125874	495835
	1	34	61	1316	5091	127456	497758
	2	14	183	1297	5535	126541	497687
	2	34	103	1272	5258	127680	499280
	3	27	78	1271	5301	126277	498794
	1	23	64	1342	5255	126269	498092
	1	57	332	1395	5302	127405	496328
	6	27	103	1268	5239	129652	499309
	5	21	74	1313	5403	124602	497334
	3	17	50	1268	5289	125699	497945
	3	13	54	1272	5379	127178	499920
	2	14	327	1273	5396	126954	497464
	2	14	86	1338	5414	127029	495904
	2	68	51	1270	5180	127632	498134
	2	34	50	1269	5093	127101	497112
	2	28	172	1278	5318	127963	496744
	2	45	103	1272	5221	126794	500395
	2	17	74	1271	5193	127017	495242
	2	13	52	1325	5220	126451	500560
	2	67	50	1267	5220	126636	493633
	2	35	172	1270	5293	130688	498286
	2	27	103	1341	5246	127139	497328
	3	54	74	1271	5415	126939	497982
	2	27	81	1269	5304	130005	498132
	2	13	52	1349	5234	126601	498777
	1	18	51	1271	5362	128440	497376
	6	34	182	1317	5228	130745	502651
	5	28	86	1374	5319	127790	505301
	6	46	50	1350	5315	127233	502055
	1	27	50	1266	5299	126567	494763
	1	23	50	1436	5324	128047	500834
	1	17	51	1270	5290	126546	500756
	1	34	358	1271	5252	127358	505368
	6	27	103	1394	5335	125566	544127
	5	19	75	1272	5234	127520	507270
	5	17	64	1337	5864	128013	500251
	4	17	52	1329	5861	127295	496871
	2	289	51	1269	5342	127126	496581
	2	47	50	1270	5828	127007	497435
	1	27	50	1272	5342	126777	495584
	6	22	114	1270	5277	126800	499224
	33	24	50	1270	5367	127214	496159
	3	45	53	1346	5320	126434	495972
	2	27	55	1326	5321	125620	498714

	2	22	129	1272	5123	126118	498463
	2	45	107	1307	5327	124660	497873
	2	27	76	1416	5320	123578	496738
	5	23	65	1269	5347	127036	499548
	1	17	52	1350	5267	126245	496796
	1	15	50	1275	5257	127106	493780
	1	15	50	1270	5438	126484	497698
	1	45	53	1270	5095	126599	498047
	2	34	50	1337	5251	127044	497081
	5	27	129	1270	5270	127223	499216
	100	45	106	1271	5438	126394	495818
	5	27	75	1388	5120	127522	494060
	2	39	66	1402	5367	127942	496845
	2	124	52	1270	5263	128139	497432
	1	121	101	1300	5219	126805	496958

Table 2: Recorded times for Euler Method

N	**10**	**50**	**100**	**500**	**1000**	**5000**	**10000**
Time (µs)	7	87	327	8305	32955	789065	3.13E+06
	5	85	550	11530	33493	808820	3.10E+06
	6	83	332	8069	32610	765220	3.09E+06
	5	129	322	8026	31847	768977	3.07E+06
	6	84	336	7966	31884	763457	3.05E+06
	8	206	384	7941	31510	763223	3.06E+06
	20	164	323	8316	32245	764497	3.05E+06
	13	122	322	7945	32447	764644	3.08E+06
	40	274	323	7998	32224	765083	3.06E+06
	21	168	332	8131	32306	769713	3.06E+06
	11	139	391	7957	32345	763893	3.07E+06
	6	275	387	7984	32185	770061	4.20E+06
	7	165	330	8029	31963	784318	3.05E+06
	73	196	378	8276	32035	762041	3.06E+06
	54	356	320	8089	31628	763515	3.07E+06
	21	168	316	7973	31739	767408	3.04E+06
	14	137	339	7984	31176	766648	3.13E+06
	13	349	317	7935	32000	764457	3.06E+06
	14	334	341	8185	32614	762026	3.05E+06
	13	83	316	7985	31622	762788	3.05E+06
	11	204	337	8011	31947	760237	3.05E+06
	12	165	316	8119	32073	764051	3.05E+06
	60	118	391	8057	31830	761436	3.06E+06
	29	105	366	8033	32023	764517	3.05E+06
	32	84	431	8064	32041	767284	3.05E+06
	18	83	549	8232	31471	767263	3.04E+06
	13	454	316	7951	32574	766123	3.04E+06

	11	208	316	8099	47223	767197	3.06E+06
	11	82	316	7986	32309	764053	3.05E+06
	11	289	319	7954	31561	764674	3.05E+06
	9	86	332	8075	31270	758912	3.06E+06
	11	257	313	8098	31999	761075	3.06E+06
	10	275	323	8061	31652	772458	3.05E+06
	11	354	313	8196	31641	762183	3.06E+06
	11	90	361	8000	31988	764061	4.47E+06
	11	213	313	8000	31499	762479	3.05E+06
	10	447	321	7951	31964	763951	3.07E+06
	10	165	319	8033	31768	761780	3.10E+06
	12	137	440	7897	31713	761151	3.05E+06
	7	284	315	7953	31703	760011	3.05E+06
	9	163	320	8511	31727	762876	3.06E+06
	7	85	315	8003	31985	760623	3.04E+06
	13	278	315	7950	32013	762655	3.06E+06
	57	166	316	8065	32419	762738	4.54E+06
	30	143	313	8029	32239	764538	3.07E+06
	31	89	324	8039	31744	762582	3.07E+06
	15	206	313	7957	31862	763841	3.04E+06
	15	266	327	8151	31834	765408	3.86E+06
	11	124	314	7963	32049	770748	3.05E+06
	12	108	317	7951	31642	770751	3.04E+06
	10	288	371	8043	31881	762247	3.05E+06
	10	275	318	7965	31880	764562	3.04E+06
	31	138	315	7977	32266	761881	3.05E+06
	21	348	316	7965	31695	763851	3.04E+06
	15	286	318	8019	31744	762189	3.06E+06
	13	165	316	8007	32273	761290	3.04E+06
	9	428	318	8002	32012	765285	3.04E+06
	13	166	495	8129	32218	763554	3.04E+06
	10	126	318	7996	32106	762460	3.05E+06
	7	244	314	8150	32124	760890	3.05E+06
	5	168	313	7960	31438	759242	3.05E+06
	7	141	314	7999	32090	761546	3.05E+06
	8	264	315	7960	31726	763449	3.06E+06
	9	166	313	7961	32122	772474	3.04E+06
	7	143	316	8130	31677	784187	3.06E+06
	9	104	325	8077	31494	785848	3.04E+06
	8	362	320	7995	31999	768213	3.05E+06
	8	175	319	8003	32202	776382	3.05E+06
	8	143	369	8034	32644	763129	3.06E+06
	8	80	326	7803	31845	763013	3.04E+06
	7	207	313	7938	31987	760349	3.04E+06

	7	304	324	8088	31885	760998	3.07E+06
	8	289	315	7979	32599	762669	3.05E+06
	7	80	318	7842	32506	764537	3.06E+06
	8	218	312	8087	32369	761352	3.05E+06
	7	172	317	7963	31906	761331	3.05E+06
	12	118	378	7997	31793	768644	3.25E+06
	8	312	390	7991	31571	762682	3.20E+06
	11	165	319	8022	33134	762441	3.19E+06
	8	121	315	8033	32198	760717	3.19E+06
	7	447	378	8072	31950	761858	3.21E+06
	7	167	314	8050	32185	760830	3.19E+06
	13	138	380	8081	31686	761423	3.18E+06
	7	206	314	8115	31360	761936	3.17E+06
	11	168	379	7967	31506	760775	3.16E+06
	8	152	316	8069	31668	759722	3.16E+06
	9	288	374	7944	32054	759983	3.18E+06
	9	167	316	7996	31808	761799	3.16E+06
	9	118	313	8061	31984	763860	3.18E+06
	11	282	386	8065	31923	762947	3.16E+06
	12	268	312	8007	31754	761275	3.19E+06
	10	137	374	8122	32021	761482	3.21E+06
	10	355	314	8162	31477	761147	4.56E+06
	12	170	313	8034	31642	763581	3.06E+06
	29	217	371	8005	31915	763724	3.05E+06
	32	104	325	8019	31761	763492	3.05E+06
	30	563	365	7892	31916	761600	3.04E+06
	29	207	320	8000	31628	761281	3.06E+06
	15	262	376	8131	32004	762786	3.11E+06
	18	206	342	8001	31741	764360	3.19E+06

Table 3: Recorded Times for RK4 Method

N	**10**	**50**	**100**	**500**	**1000**	**5000**	**10000**
Time (µs)	1	28	56	1423	5602	251189	540936
	1	14	56	1595	5610	154351	569640
	1	14	55	1345	5367	138741	535477
	1	14	58	1409	5572	136154	538510
	1	14	55	1491	5493	138520	535198
	1	40	55	1348	5896	135037	536442
	1	28	58	1381	5578	135901	534991
	6	21	118	1503	6065	134543	533276
	4	18	184	1460	5661	134943	535414
	4	14	55	3090	5899	132998	538965
	2	17	54	1377	5558	139503	533310
	2	14	57	1405	5657	136736	533719
	2	71	54	1427	5815	136372	532401

	2	14	312	1420	5634	136655	533234
	2	15	138	1375	6045	139008	534212
	2	14	95	1411	5863	134924	533630
	6	48	70	1376	5830	135331	536116
	4	30	61	1374	5735	134756	534194
	3	38	57	1375	5721	135681	533661
	2	30	56	1482	5875	134473	535218
	1	23	53	1373	5783	133959	531352
	1	47	183	1374	6947	135238	534484
	1	28	112	1405	5670	134420	535887
	1	24	54	1445	5710	135273	533404
	1	17	184	1375	5604	134735	535320
	1	14	113	1472	5737	133677	534138
	5	98	54	1421	5692	134036	534644
	5	14	263	1374	5849	133878	534681
	6	15	110	1375	5592	134854	530081
	4	71	79	1448	5618	133736	531717
	3	46	53	1372	5537	133378	530227
	3	36	54	1374	5592	133360	535167
	2	24	55	1472	5624	133591	534528
	2	48	192	1375	5708	134609	545446
	2	29	110	1373	5692	133796	535299
	2	24	78	1374	5662	134266	534276
	2	50	70	1427	5666	133619	535159
	3	36	80	1376	5618	134587	532439
	2	29	56	1761	5572	135113	536135
	2	21	184	1430	5492	134629	708761
	1	16	109	1372	5486	133003	532855
	1	14	55	1373	5662	134768	534617
	1	14	340	1608	5686	135338	535777
	6	14	54	1377	5869	135722	535011
	5	14	54	1373	5771	131557	533698
	5	72	190	1400	5590	133806	538227
	3	36	110	1377	5747	131795	536485
	2	29	78	1375	5849	132968	537477
	2	25	83	1462	5520	134523	536715
	2	19	55	1375	5490	135788	534906
	2	13	53	1373	5979	132069	535508
	1	14	250	1496	5530	133623	531524
	1	14	110	1375	5698	134986	532855
	1	14	79	1379	5513	134159	535551
	1	14	69	1312	5695	134612	532631
	1	14	60	1374	5558	135430	534509
	1	14	54	1428	5572	134706	534891

	1	14	250	1374	5516	133832	536137
	1	16	110	1375	5503	132812	535588
	2	15	78	1379	5585	133484	533590
	1	13	69	1376	5748	131587	543041
	1	15	57	1374	5760	134344	533871
	1	15	54	1497	5525	134507	534832
	1	14	139	1374	5711	135733	531048
	1	14	110	1375	5766	134463	535125
	1	15	78	1500	5734	134293	544138
	1	17	69	1432	5751	134556	537138
	1	14	57	1375	5774	133633	535490
	1	16	63	1476	5858	134273	540931
	1	16	54	1375	5682	144820	545087
	1	14	53	1350	5624	142093	565018
	2	17	54	1379	5799	143408	557642
	1	14	251	1377	5734	135999	552481
	1	14	110	1375	5661	138223	559843
	1	15	79	1457	5798	134691	551600
	1	14	69	1443	5897	137834	551269
	2	15	60	1374	5492	134388	550993
	1	14	55	1461	5575	132785	551051
	2	14	360	1373	5689	134340	547076
	1	15	110	1373	5893	132965	551256
	2	14	78	1459	5635	132536	554185
	2	15	143	1374	5780	134670	555048
	1	14	54	1310	5681	134509	923416
	1	14	55	1472	5656	135655	549966
	1	15	263	1437	5528	134969	550290
	1	15	110	1374	5602	133654	546024
	1	15	79	1496	5916	135177	557987
	2	14	68	1506	5579	134382	550137
	1	18	55	1458	5511	135003	549567
	1	14	55	1574	5817	133298	550908
	1	14	53	1374	5717	131048	550020
	1	15	58	1376	5757	134506	558253
	3	17	58	1460	5621	134361	564956
	1	14	55	1375	5515	134564	579414
	1	15	55	1377	5473	134389	963046
	1	104	55	1469	5615	133515	572493
	1	49	115	1374	5676	148206	544293
	1	36	55	1376	5640	134427	534832
	2	24	55	1381	5824	134024	533867
	1	14	54	1376	5521	133968	535119

Table 4: Recorded Times for Verlet Method

N	10	50	100	500	1000	5000	10000
Time (µs)	244	407	229	683	796	2387	5399
	153	242	344	643	848	3718	7028
	230	449	251	634	1142	3659	10022
	207	388	259	459	787	3746	9438
	273	407	248	540	876	3731	9192
	531	608	284	647	948	3724	9488
	356	421	317	644	695	3707	9495
	384	356	301	487	949	3820	9146
	678	345	346	476	915	3724	9213
	862	297	493	403	950	3582	5498
	496	339	478	643	943	2295	5352
	517	269	455	646	687	3542	9144
	360	305	388	502	962	3571	9123
	354	679	275	489	1144	3541	9124
	447	454	338	727	1122	3515	9455
	360	431	314	679	946	3489	9430
	496	334	364	650	669	2330	9470
	340	601	307	491	941	3669	9445
	426	369	357	460	886	3464	9495
	334	630	337	482	958	3471	9536
	440	532	379	697	944	2315	5319
	362	516	388	662	679	3482	9152
	479	572	340	439	940	3448	9530
	279	328	452	466	1130	3464	9510
	261	500	533	481	968	3478	9707
	264	619	480	646	946	3469	9126
	293	907	377	640	928	3490	9740
	304	516	323	511	983	3447	9768
	257	355	398	500	766	3459	9726
	411	289	382	709	1030	3444	9781
	257	292	325	658	946	2329	5376
	668	534	323	645	1065	3450	9282
	326	670	324	469	945	3429	9224
	277	369	366	456	746	3724	9184
	737	271	734	725	959	3434	9187
	646	556	308	641	941	3635	9150
	696	282	393	645	940	3414	9195
	542	591	314	505	948	3443	9179
	439	457	331	455	795	3622	9166
	377	361	312	705	948	3433	9174
	373	334	343	640	951	2304	5348
	386	528	315	638	918	3660	9418
	382	381	316	487	940	3403	9043

	352	504	310	476	974	3708	8476
	656	458	314	710	887	3700	8824
	409	414	526	640	947	3435	9116
	385	357	559	672	697	3712	9111
	683	781	559	518	955	3706	9132
	680	412	374	459	735	3701	9132
	607	592	599	755	946	3413	9142
	379	321	365	638	946	2297	5278
	389	415	327	614	935	3711	9714
	376	346	456	515	950	3647	9529
	411	572	342	458	765	3710	9167
	349	451	351	712	951	3634	9172
	417	558	313	645	914	3644	9135
	308	331	305	654	1092	3641	9167
	405	294	313	412	943	3634	9143
	278	455	352	463	715	3716	9199
	284	394	550	707	967	3705	9182
	276	345	366	642	911	2321	5381
	474	345	398	643	879	3715	9765
	281	696	464	488	947	3713	9238
	286	456	307	503	686	3708	9196
	295	389	314	678	983	3715	9156
	243	317	305	640	1165	3711	9202
	474	602	314	634	858	3715	9242
	328	304	326	473	950	3719	9162
	792	805	331	460	721	3720	9138
	364	300	316	717	941	3555	9154
	287	421	536	562	949	2302	5367
	342	379	307	559	937	3559	9178
	284	506	399	470	935	3546	9481
	370	285	537	417	684	3545	9321
	286	449	583	731	1024	3539	9354
	305	327	569	642	922	3537	9157
	284	356	350	635	898	3549	9451
	805	336	453	476	968	3488	9433
	680	320	340	457	678	3530	9501
	593	324	311	607	955	3486	9444
	419	373	427	566	941	2315	5363
	390	309	322	586	941	3503	9512
	385	312	310	516	953	3446	9488
	357	297	315	455	711	3465	9442
	646	282	384	645	948	3458	9599
	323	329	325	683	1130	3459	9472
	644	338	382	560	943	3460	9464

	305	626	419	481	912	3457	9491
	415	468	393	460	928	3446	9467
	383	544	454	654	953	3449	9457
	414	354	333	562	945	2302	5358
	430	538	333	598	946	3454	9781
	641	293	284	470	942	3446	9470
	331	516	336	389	674	3432	9233
	352	330	311	641	947	3454	9587
	408	540	358	719	872	3443	9430
	550	375	315	545	939	3459	9210
	395	458	537	486	949	3469	8628
	547	555	534	475	678	3448	9332
	335	357	535	677	945	3452	9127

Table 5: Recorded Times for GPU Euler method

N	**10**	**50**	**100**	**500**	**1000**	**5000**	**10000**
Time (μs)	44	291	245	2404	3971	25656	57248
	13	75	252	1696	4370	26446	59795
	22	94	248	1739	4257	26522	60674
	13	88	219	1665	4430	26427	58453
	27	89	237	1981	4049	25973	59357
	13	303	327	1791	4170	27623	58177
	37	197	977	1787	4432	27788	57729
	31	177	409	1785	4375	30960	57623
	39	213	381	1752	4459	27168	57867
	44	644	392	1788	4533	27118	57454
	86	238	343	2093	4165	26387	55628
	176	197	280	1758	5380	27742	57524
	54	445	299	1799	4102	27028	57537
	33	315	349	2128	4617	27356	57358
	26	234	348	1747	4450	28046	57745
	25	452	891	1815	4519	27414	57672
	38	241	690	1981	4161	27438	57568
	17	209	453	1812	4170	27868	57663
	18	143	405	1726	4266	27372	57307
	24	102	303	1798	4241	27847	57080
	11	189	281	1849	4237	26117	55436
	16	137	967	1798	4331	27036	57389
	31	162	583	1941	4426	27475	57792
	15	109	335	1791	4262	27000	58299
	16	110	377	1796	4565	27855	57826
	16	114	341	1888	4417	27040	58498
	16	155	299	1862	4652	28509	58346
	15	114	1102	1876	4351	31150	58421
	16	131	679	1937	4296	29156	59345

	19	132	436	2067	4201	27386	59312
	25	111	404	1828	4379	26378	57495
	54	112	346	1852	4506	27459	59135
	64	109	334	1920	4309	28791	60533
	87	115	335	1791	4233	29596	60200
	34	562	326	1814	4605	27075	60364
	44	400	317	2292	4112	27177	59913
	17	256	902	1865	4242	27019	60835
	21	113	607	1850	4175	27285	59768
	25	135	427	1788	4393	27185	60107
	22	237	363	1837	4141	27005	60026
	15	129	507	1872	4103	25894	58363
	20	129	306	1910	4640	26926	59819
	19	129	1048	1802	4074	27289	59726
	25	121	538	1913	4341	27048	59710
	16	170	430	1841	4150	26803	59495
	20	113	361	1795	4272	26881	60869
	85	127	294	1816	4220	26533	62759
	74	540	293	1906	4217	26735	61975
	308	329	726	1871	4152	27828	61143
	36	237	608	1759	4181	26796	61540
	33	214	408	1874	4071	26257	62481
	24	164	396	1834	4302	26688	71020
	31	113	277	1802	4239	26661	62507
	38	114	387	1833	4375	27087	61598
	21	119	1135	1823	4386	26729	61100
	15	121	292	1849	4538	26491	60592
	51	109	280	1860	4288	26850	59909
	13	113	930	1987	4749	26976	60418
	30	118	613	1881	4542	26840	62278
	49	545	401	1826	4518	26833	61406
	15	541	376	1892	4400	25986	59597
	27	206	301	1851	4374	26879	59910
	19	147	331	1861	4255	26818	60673
	83	128	1174	1938	4291	26895	59991
	59	111	657	1894	4309	26879	59999
	87	126	412	1884	4364	27057	60357
	30	103	392	1951	4401	27880	61282
	24	104	311	1820	4326	27288	63161
	16	134	680	1802	4504	26997	62327
	25	119	385	2004	4642	26974	79294
	56	121	402	1876	4592	25897	58341
	92	164	360	1883	4613	28450	60626
	30	142	285	1954	4569	27101	62862

	30	125	334	1878	4385	26911	76561
	27	141	274	1884	4551	26722	61694
	23	135	853	1996	4426	27000	60773
	25	163	262	1838	4460	27232	61055
	22	149	363	1900	4437	27250	60831
	25	224	378	1962	4453	27078	61179
	25	137	290	1925	4532	26914	60720
	36	222	365	1938	4416	26078	59771
	19	177	291	1905	4451	26958	62390
	13	142	390	1863	4479	27065	62143
	14	141	306	1890	4702	27187	61469
	23	144	862	1884	4373	27290	64731
	20	125	372	2010	4449	26921	61271
	19	215	365	1908	4366	26984	61288
	17	199	289	1924	4840	27553	61523
	15	132	341	2692	4492	27095	60092
	21	171	500	1876	4589	27355	61491
	17	130	714	1906	4552	25747	59866
	22	132	455	1899	4664	27025	72857
	18	139	342	1930	4412	27013	64411
	85	123	354	1981	4472	27018	60355
	22	183	327	2111	4467	27151	93362
	95	147	308	1995	4521	27010	62007
	20	147	409	1954	4600	27040	61413
	30	104	402	2051	4456	26872	61120
	25	167	437	2286	4495	27045	63260
	31	127	420	1974	4456	27359	82194

Table 6: Recorded Times for Barnes-Hut with theta =0.5

N	**10**	**50**	**100**	**500**	**1000**	**5000**	**10000**
Time (µs)	33	77	317	1401	2751	16389	35943
	18	69	180	1385	2812	16729	37595
	21	84	224	1305	3298	16905	37380
	12	80	203	1205	3603	16386	37295
	15	83	220	1254	3493	16496	37067
	18	102	315	1338	3900	17301	37177
	39	376	265	1374	3977	17252	38122
	101	192	336	1662	3919	16930	39047
	55	208	239	1346	3781	17275	38462
	45	307	236	1374	3493	17308	38312
	53	303	251	1469	3396	16225	36564
	39	164	261	1299	3546	16874	38257
	32	248	254	1551	3547	17983	38276
	36	174	263	1305	4598	17782	38854
	33	285	243	1460	3671	16950	37939

	36	206	305	1452	3852	17084	38023
	28	144	257	1343	4602	16958	38123
	33	123	260	1392	2844	16937	37965
	21	277	399	1360	3873	17940	37790
	20	218	247	1512	3917	16956	38423
	20	566	249	1332	4077	16249	36467
	25	328	246	1554	3260	17350	38321
	31	208	250	1371	3670	17066	38294
	22	146	228	1347	3899	17075	38415
	24	132	234	1413	3676	17065	38427
	25	112	319	1509	3737	17113	38172
	22	102	313	1658	3894	17355	37970
	24	95	287	1364	3624	17107	37651
	24	102	263	1372	4021	17300	37997
	27	121	262	1390	3442	17595	37836
	14	114	258	1381	2942	16331	36245
	20	101	289	1440	3504	17485	38247
	61	134	264	1420	3805	17275	38972
	56	109	265	1355	3734	17020	38081
	39	119	245	1392	3676	17205	38477
	37	98	274	2391	3644	17683	38263
	17	90	254	1946	3810	17275	38115
	18	93	312	1390	2968	17379	38007
	51	111	236	1363	3470	17342	38236
	19	101	253	1595	3489	17486	38733
	16	106	229	1535	3942	16481	36309
	16	120	244	1392	3808	17087	38225
	70	118	262	1526	3762	17853	38654
	299	94	319	1496	3867	17269	37951
	40	323	254	1395	3723	17174	37957
	34	104	276	1431	4536	17258	38084
	38	115	231	1407	3727	17232	38611
	41	91	259	1478	3009	17300	38252
	20	109	231	1564	3391	17314	38282
	31	118	258	1343	3263	17396	37994
	17	106	257	1402	3019	16571	36556
	20	115	239	1348	3295	17229	38197
	28	108	245	1512	3681	17577	38183
	77	136	238	1386	3420	17155	38080
	70	105	227	1444	3960	17190	40398
	62	137	237	1483	3956	17126	38103
	44	102	321	1628	3682	17461	39061
	33	119	362	1389	3473	17262	38450
	30	100	252	1435	3405	17510	38274

	21	86	277	1375	3407	17360	39182
	119	103	255	1516	3634	16486	36750
	46	88	252	1470	3467	17152	37954
	33	85	275	1401	3794	17155	37778
	21	89	262	1397	4241	17349	38571
	28	114	244	1777	3369	17432	38446
	25	94	259	1424	3997	17371	38973
	22	96	268	1516	3081	17142	38510
	19	97	248	1397	3602	17351	38885
	21	101	245	1724	3989	17459	38327
	22	136	252	1457	3638	17259	38419
	17	107	268	1480	3999	18306	36795
	63	105	263	1383	3080	17655	38531
	41	105	208	1473	3864	17734	38174
	30	103	271	1386	3778	17548	44735
	35	105	245	1430	3608	17795	38442
	20	104	326	1454	3985	17644	38221
	51	119	236	1539	3015	17561	38699
	71	116	245	1417	4025	17783	38135
	40	121	238	1473	2993	17389	38043
	22	121	288	1436	3512	17986	38242
	17	116	318	1515	3970	16979	38501
	18	128	348	1629	4040	17887	38321
	53	156	369	1686	3765	17547	38296
	56	139	242	1437	4012	17599	40081
	43	123	319	1552	4024	17724	38591
	26	117	253	1610	3867	17659	39856
	28	126	313	1479	3433	17536	39584
	20	107	237	1511	3802	17706	38054
	23	137	336	1450	3597	17803	38890
	21	110	258	1662	3603	17986	38803
	14	105	326	1481	3932	17147	36819
	17	114	261	1440	4052	17616	38231
	23	125	330	1568	4041	17425	38257
	74	137	267	1604	3622	17422	38636
	74	146	265	1395	3887	17865	38330
	74	114	295	1517	3776	18094	39554
	36	110	263	1706	3396	17822	39100
	51	98	262	1456	3725	17590	38322
	33	109	250	1398	3861	17598	38670
	123	114	326	1616	3800	17716	38649

Table 7: Recorded Times for Barnes-Hut with theta =0.75

N	10	50	100	500	1000	5000	10000
Time (µs)	33	127	222	1134	2280	12597	25817
	21	68	164	1198	2175	13051	27521
	24	107	207	1113	2431	13006	27371
	15	114	175	991	2152	12760	27809
	15	320	219	995	2221	12725	26860
	35	86	273	1145	2385	13343	27611
	37	270	335	1054	2338	13715	28378
	23	176	216	1055	2509	13785	28440
	114	133	228	1217	2298	13419	28596
	49	110	488	1104	2228	13960	28347
	70	460	457	1129	2220	13310	26435
	31	216	323	1181	2218	12771	28190
	31	182	248	1022	2230	12960	28824
	32	138	233	1049	2250	12874	29880
	29	152	229	1067	2302	12759	36871
	43	93	543	1042	3016	13038	28682
	26	420	390	1343	2241	13568	28743
	26	290	230	1121	2250	13631	28078
	29	86	299	1049	2226	13439	28173
	21	347	297	1071	2245	13563	28331
	28	179	185	1125	2332	13319	26786
	36	156	224	1170	2278	13623	28389
	29	324	232	1101	2319	13199	28550
	91	218	227	1081	2289	13362	29106
	58	87	801	1099	2269	13882	28875
	36	105	492	1117	2321	13245	28574
	23	358	342	1306	2293	13529	28381
	15	205	300	1092	2377	13181	28257
	15	378	265	1123	2388	13506	28339
	16	184	228	1277	2243	13231	28269
	39	153	252	1146	2352	13815	26694
	52	128	295	1145	2333	14064	28476
	116	105	226	1124	2310	13572	28424
	39	94	905	1139	2239	13479	28327
	37	291	463	1107	2304	13901	28176
	37	202	348	1224	2289	13606	28401
	20	158	370	1078	2278	13600	27895
	23	267	256	1229	2339	13927	30360
	15	257	216	1053	2442	15305	30178
	24	138	258	1073	2282	13754	29678
	9	287	207	1136	2454	12956	27470
	17	244	261	1215	2320	13522	28655
	17	161	234	1107	2339	13379	29070

	24	289	283	1111	2435	13506	29629
	14	83	251	1164	2413	17597	36915
	23	227	208	1181	2455	13188	28727
	19	157	266	1062	2385	13408	29328
	26	117	209	1142	2496	13598	29783
	20	105	214	1092	2276	13837	29867
	22	110	223	1094	2279	14602	34311
	14	83	250	1190	2316	13684	29326
	11	101	204	1089	2627	14613	30112
	21	397	197	1221	2709	14020	34356
	22	232	231	1220	2333	13769	30146
	18	203	174	1135	2630	13831	28725
	25	160	268	1168	2427	14404	29668
	56	96	205	1304	2303	14562	28984
	49	86	238	1127	2999	13597	29311
	46	367	214	1131	2252	14297	28349
	15	293	240	1215	2364	14026	28985
	29	157	233	1198	2472	13303	28164
	33	286	252	1142	2359	13849	28572
	16	202	263	1120	2300	13920	29169
	19	132	613	1094	2292	13520	29409
	16	288	229	1327	2515	13750	28713
	18	169	218	1113	2404	13941	29213
	71	153	260	1085	2425	14375	29525
	55	136	206	1134	2475	13871	29102
	62	112	254	1156	2370	13618	29941
	35	579	205	1106	2360	13835	28873
	23	184	209	1104	2259	13400	28466
	20	138	225	1180	2337	14227	29317
	40	115	220	1205	2352	14000	35362
	27	116	205	1480	2345	14471	28898
	25	100	217	1111	2569	15193	28405
	19	444	225	1231	2482	14427	29688
	20	348	207	1217	2411	14250	28691
	23	299	239	1100	2546	14266	28932
	18	175	225	1088	2389	14131	29708
	21	233	314	1212	2359	13960	28242
	51	264	214	1117	2380	13331	26555
	19	204	241	1139	2319	15139	29865
	15	152	217	1120	2339	13952	28484
	17	382	233	1268	2589	13784	28519
	21	181	210	1121	2561	13454	28260
	15	115	239	1124	2611	14230	28485
	22	153	209	1229	2510	14121	28458

	16	410	241	1127	2426	14350	28115
	26	196	227	1125	2770	14098	28773
	13	233	281	1244	2409	14384	27708
	20	299	230	1119	2654	13154	26474
	16	184	348	1096	2645	13856	28268
	20	248	232	1347	2433	13662	28910
	25	113	241	1227	2600	13840	28389
	24	186	245	1136	2360	13725	28849
	20	238	253	1150	2503	15680	28362
	18	182	218	1181	2406	13894	28230
	24	121	214	1147	2507	14525	28529
	22	303	229	1140	2512	15470	28316
	20	193	206	1159	2404	13646	28575

Table 8: Recorded Times for Barnes-Hut with theta =1.0

N	**10**	**50**	**100**	**500**	**1000**	**5000**	**10000**
Time (μs)	19	112	175	1581	1806	9714	19882
	14	72	290	1317	1787	10790	19399
	20	103	218	1158	1845	10005	20716
	15	83	176	1167	1737	10373	19687
	22	92	176	887	1704	9526	19766
	32	66	179	956	2039	10387	20181
	14	99	266	899	1787	11322	20096
	16	72	182	905	1801	9997	20729
	19	95	599	1196	1768	10460	20809
	18	70	361	1129	1851	10111	20514
	15	84	195	1008	1767	9474	19164
	20	74	247	1180	2508	10190	21398
	20	104	195	924	1829	10177	20929
	18	75	260	957	2140	10209	21047
	24	105	224	934	1826	10149	20560
	17	104	251	986	2124	9927	20929
	21	73	240	922	1803	10368	20510
	16	95	683	1078	1860	10082	20396
	18	81	463	1031	1812	10295	21172
	21	93	317	981	1789	10101	20276
	17	81	287	1025	1791	10423	18986
	29	103	202	962	1823	10086	20919
	33	104	202	983	1774	9785	20823
	23	102	180	965	1792	10762	20499
	21	79	197	1178	1830	9904	21120
	22	96	313	1046	1784	10539	20268
	20	95	181	1020	1799	14686	20467
	26	72	215	1040	1780	10012	20395
	23	86	199	997	1793	9717	20916

	27	106	212	1008	1807	10269	20318
	16	84	211	1098	1771	9604	18807
	22	83	210	962	2151	10464	20339
	21	107	851	938	1795	9873	20247
	24	73	435	993	1790	9676	20336
	27	159	277	1099	1789	9927	20986
	26	124	317	1064	1812	10153	20334
	18	83	199	983	1824	10211	20561
	22	115	199	923	1814	9965	20353
	27	98	178	1044	1843	9878	20460
	12	115	238	1053	1810	9993	20470
	24	76	276	1079	1849	9776	19100
	21	108	639	966	1821	9791	20346
	18	82	345	989	1839	9877	21876
	15	124	286	972	1852	10085	20564
	70	84	221	1024	1890	10044	20378
	96	119	209	1060	1849	10190	20322
	87	114	177	956	1792	10183	20653
	72	122	1005	922	1792	10385	20912
	61	120	341	956	1825	10260	20472
	38	82	261	1037	1801	10259	21031
	26	108	287	965	1810	9758	19003
	17	75	204	1036	1781	9820	20459
	23	109	195	1001	1748	9979	20245
	30	96	187	1275	1795	10065	20650
	38	136	168	1087	1780	9780	20590
	61	86	204	981	1822	10473	20569
	63	120	685	1084	1771	10147	20436
	35	92	413	1186	2902	10554	20537
	26	110	264	945	1832	10322	20206
	38	85	181	1028	2073	10192	20550
	27	122	222	949	1845	11648	19012
	23	118	219	969	1795	10437	19886
	24	86	198	1061	1788	10080	20438
	19	127	185	1056	1767	9889	20093
	27	88	202	1012	1777	9972	20926
	32	109	668	1041	1788	10644	20507
	30	86	391	1023	1920	10007	20553
	25	125	276	987	1852	10033	20996
	26	92	266	1054	2011	10079	21497
	56	114	177	1089	1846	10012	21100
	54	94	182	962	1840	9503	19825
	55	126	415	1030	1835	10635	25901
	54	174	328	1053	1870	10236	21076

	27	186	258	1044	1884	10182	25104
	24	143	231	1093	1873	10349	20944
	19	93	191	946	1896	11877	21574
	20	163	199	1993	1865	10446	21951
	14	97	179	973	1921	10021	22234
	14	89	196	1318	1830	10449	20835
	32	86	190	1112	1874	9885	21538
	18	150	647	1003	1906	9777	19777
	18	94	356	1197	1918	10240	21322
	16	191	283	968	1832	10619	21405
	16	95	378	998	1840	10338	21335
	22	108	191	1203	1891	10276	22032
	22	123	209	970	1942	10478	21171
	23	148	204	980	1860	10254	20956
	19	138	391	1088	1869	10892	21550
	70	110	192	1096	1919	10621	21336
	59	129	743	1069	1826	11216	21385
	45	103	421	1219	1870	10002	20261
	61	112	203	1170	1830	10130	22206
	31	85	204	986	1899	10085	22001
	29	103	198	1012	1921	10134	22161
	33	79	183	992	1908	10105	21630
	29	109	714	1055	1891	10363	22496
	32	88	392	1154	1853	10432	22320
	36	118	283	986	2018	10737	22412
	87	85	273	1042	1902	11041	21705
	86	84	259	1087	1851	10675	22521

Table 9: Recorded Times for Barnes-Hut with theta = 1.5

Appendix B – Barnes-Hut Tree diagram

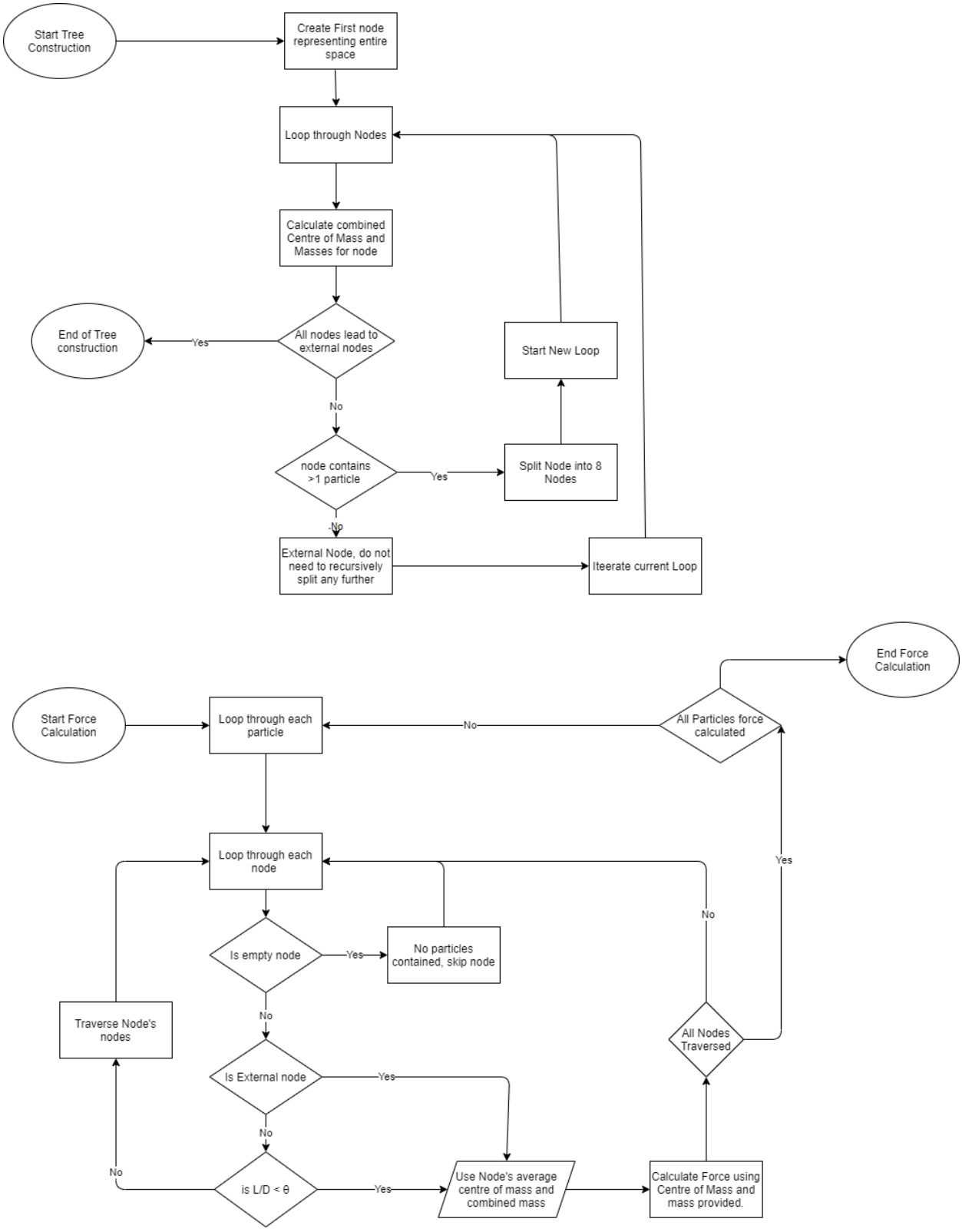

Figure 25: Barnes-Hut Tree Construction and Force Calculation Diagram

Appendix C – Artefact Input and Output

All input variables can be changed in the file setup.txt

All variables can be changed using Key input

m - change method

n - increase n

b - decrease n

l - increase timestep

k - decrease timestep

p - increase theta

o - decrease theta

q - decrease number of runs before output

w - increase number of runs before output

r - restart sim and apply changes

.csv files created in output folder.