CPS 376 Final Project Paper

Kathy Breczinski

December 2021

1 Introduction to the N-Body Problem

The N-Body problem attempts to solve or calculate the interactions between all N number of particles or bodies and their anticipated positions in space based on the gravitational pulls of all other particles for all of future time. The two-body problem has been solved, as has a restricted version of the three-body problem, but the N-body problem has no closed-form solution like the two-body problem does. This is due to the chaotic interactions between each of the particles as they are all moving at once, and these changes affect each particle differently [5]. However, the interactions between N number of bodies can be simulated using a large number of various algorithms.

2 A Detailed Description

There are many variables that need to be kept track of in order to accurately calculate the metrics necessary to know the next position of each particle. It is important to know how many particles there are. Additionally, the mass of each particle needs to be kept track of. Other important values are the velocities, positions, and accelerations of each particle. Because the particles are in space, they will have an x, y, and z coordinate to simulate movement in a three dimensional space. Because of this, there will be an x-component, y-component, and z-component velocity stored for each particle. There will also be an acceleration for each coordinate of each particle. Because this problem is happening in a set amount of time, there needs to be a starting time, usually 0. There also needs to be an end time and a time step. The end time can be whatever is preferred, but the time step has to be small enough that the particles won't be jumping to unexpected positions. If the time step is too large, the energy in the system won't be conserved. Since the N-Body problem is a closed system where no mass is exiting or entering the system because there are only N number of particles in the system, the energy must be conserved. The final variable is the gravitational constant, which is $6.67 * 10^{-11}$ [1].

In order to calculate the motions and positions of each particle for each time step, the accelerations for each x, y, and z component need to be calculated. Additionally, the velocities must be updated, and the positions will be updated using the calculated accelerations and velocities. The initial velocities and positions for each x, y, and z component can be calculated randomly [1].

The formula to calculate acceleration is taken from Newton's second law of motion, which describes the relationship between an object's acceleration and the net force acting upon the object. His second law of motion is defined as mass multiplied by acceleration equals the sum of forces on the mass [1],[5], [6].

$$F_{net} = m * a$$

To calculate the acceleration for a particle, i, the equation for force must be defined [5]. This can be described as

$$F_{ij} = \frac{Gm_i m_j (q_j - q_i)}{\|q_j - q_i\|^3}.$$

Here, G is the gravitational constant, $m_{i,j}$ are the masses of the two objects or particles, and $q_{i,j}$ are the positions of the objects. $\|q_j - q_i\|^3$ is the magnitude of the distance [5].

These two equations can be combined to solve for the acceleration of each particle. Replace F_{net} with the summation of the second force formula to calculate force to get [1]

$$a_i = G \sum_{j \neq i} m_j \frac{r_j - r_i}{\|r_j - r_i\|^3}.$$

The variables m from the F_{net} equation and m_i from the F_{ij} equation can be cancelled out when combining the equations to get the above acceleration equation. This is because m from the F_{net} equation is technically m_i since i represents the current particle for which acceleration is being calculated. An additional variable is included in this equation. This variable is called softening, and it is important in situations where two particles are too close together, which causes the force to go to infinity. In this implementation, softening is 0.01 and it is added when calculating the magnitude. The formula for calculating the magnitude of the positions changes to [1]

$$\sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2 + softening^2}$$

Velocity is also needed to calculate the motions and positions of each particle. In this implementation, the leapfrog method for time stepping is being used. This is to ensure that accuracy is maintained, and the total energy of the system is preserved. The leapfrog method can be described as "kick-drift-kick" with half step kicks. So, the time step would be half of its value for each time it is being used to calculate velocity. The formula used for velocity is [1], [2], [4]

$$v = u + a * t$$

where u is the initial velocity, a is the acceleration, and t is the time or time step.

3 The Algorithm and Implementation

Before calculating each new position for each particle, everything must be initialized. A number of particles, N, must be chosen. In this implementation, $N = 100$. The velocities and positions can be randomly chosen. To keep the system simple, they start out as any number between -2 and 2. The masses are also initialized. In this project, they are a number of different values. They can all be different values, the same values, one can be much larger than the others, and many other scenarios. All of these different situations will affect how the particles interact with each other in distinct ways since mass is part of the acceleration equation. The time step can be anything as long as it is small enough. In this implementation, it stays at 0.01. The end time can also be any value. In this implementation, it is 10. This means the algorithm will run for 10/0.01 or 1000 steps for each particle. The gravitational constant, G remains at $6.67 * 10^{-11}$.

After everything has been initialized, the accelerations must also be initialized. The formula to calculate the accelerations is used to do so, and the initialized positions and masses and N and G are used.

The algorithm to calculate the next positions and motions of the particles then runs for $numSteps = endTime/timeStep$ times. Within this loop, the velocity is first recalculated for the first half step of the leapfrog method, where

$$velocity = velocity + acceleration * (\frac{dt}{2.0})$$

for each x, y, and z component of each particle, and dt is the time step. Then, the position is updated, where

$$position = position + velocity * dt$$

for each component of each particle. The second half step is then calculated, where

$$velocity = velocity + acceleration * \frac{dt}{2.0}$$

for each component of each particle [1].

The C++ and CUDA implementation require some additional components to be fully implemented. In this implementation, there is one thread per particle, so there are 100 threads in 1 block. The initialization for the constant, single values happens in the main function. The masses are initialized in the host function called from main. The velocities and positions are initialized with random values between -2 and 2 in a CUDA kernel. The acceleration is calculated in a separate kernel. The process of calculating the motions and positions of each particle are split between three CUDA kernels and called from the host function in a loop that runs for the calculated number of steps, 1000 in this implementation. The first step is the first half step of the leapfrog method and recalculating the positions. The second step is recalculating the accelerations. The third step is the second half step of the leapfrog method. Because three separate kernels are called, the threads must wait until each thread has finished before continuing to the next step. This is because the data

in various variables has changed in each step because the threads are updating information about its particle in at least one of the variables. An additional variable keeps track of each of the previous positions of each particle, which can then be used in a Python simulation to visualize each particle's movement.

4 Why Parallelization Makes Sense

This problem can and should be parallelized because it can take a long time to run a simulation depending on how complicated the equations used are and how many steps are in the algorithm. Additionally, the more particles there are and the higher the end time is, the simulation can run for a signification amount of time. Another reason to consider is that all particles should move at the same time since the positions of each particle are important in calculating the acceleration for each particle. If one particle moves before all other particles have used its previous position to calculate its next motions and movements, the system will become

5 Challenges Encountered

Because this problem is parallelized, the classic problems that come with parallel programming have occurred. The largest is a synchronization issue because three separate kernels are being called from the host function, which necessitates the use of cudadevicesynchronize(). This has to be used since there is shared data between the threads, such as the positions of each particle. This is necessary for the calculation of the accelerations. Although synchronization has the potential to increase run time, in this case it did not. Other challenges encountered were related to the math and physics of the problem. The simplest formulas were used for this implementation, but there are more complicated formulas that involve integration and differentiation that can alternately be used. Additionally, there are more complicated algorithms, such as the Barnes-Hut algorithm, that can be used to speedup the simulation even more [3].

Another challenge encountered was the issue of whether the positions and motions were changing based on different masses. After trying three different scenarios, it did not seem like the positions were different. The first scenario had all masses initialized to 0.2. The second had two masses initialized to 5 and the rest to 0.01*i. The third had all masses calculated as 0.2*i. Since the C++/CUDA implementation will generate the same random numbers each time and the Python implementation won't do this, it is difficult to compare whether the problem is because of an incorrect aspect of the C++/CUDA implementation specifically, or an issue with the formulas and equations used.

6 Results

The C++/CUDA implementation is significantly faster than the Python version. In this implementation, the number of particles compared were 10, 25, 50, 75, and 100. Initially,

the comparisons were going to be 10, 100, 1000, 10000, but the Python implementation was too slow to run 1000 particles. See Figure 1.

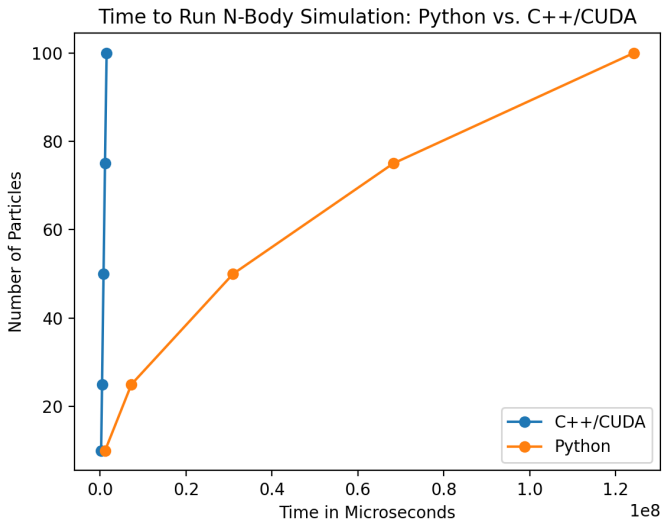

Figure 1: Time Comparison between Python and C++/CUDA Implementation of N-Body Simulation

Additionally, a simulation using Python helped visualize the motions of the particles.

7 References

[1] P. Mocz, *Create Your Own N-body Simulation (With Python),* Sept. 13, 2020. Accessed on: Dec. 17, 2021. [Online]. Available: https://medium.com/swlh/create-your-own-n-body-simulation-with-python-f417234885e9

[2] lemon, *Which timestep should I use for a N-body simulation of the Solar system?*. May. 28, 2016. Accessed on: Dec. 17, 2021. [Online]. Available: https://physics.stackexchange.com/questions/258456/which-timestep-should-i-use-for-a-n-body-simulation-of-the-solar-system

[3] *"Barnes–Hut simulation,"* Dec. 11, 2020. Accessed on: Dec. 17, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Barnes%E2%80%93Hut_simulation.

[4] *"Numerical model of the Solar System,"* Oct. 26, 2021. Accessed on: Dec. 17, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Numerical_model_of_the_Solar_System

[5] *"n-body problem,"* Oct. 4, 2021. Accessed on: Dec. 17, 2021. [Online]. Available: https://en.wikipedia.org/wiki/N-body_problem

[6] *"Newton's laws of motion,"* Dec. 13, 2021. Accessed on: Dec. 17, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Newton%27s_laws_of_motion#Newton%27s_second_law