COSC3500: High Performance Computing

Semester 2, 2021

n-body Gravity Simulation

Project Report - Milestone 2

by Tom Stephen
45811449

Contents

Chapter 1

Serial

1.1 Introduction

This project attempts to implement and later optimise an interacting particle system - specifically, gravitational interactions between massive bodies in space - using high performance computing methods. This chapter introduces the theory required, motivates this task, then demonstrates and benchmarks a serial implementation. Some basic analysis is conducted to verify that the implementation is performing correctly, and the time dependance on input parameters is presented. The next chapter is focused on how this implementation may be optimised using parallel computing techniques.

1.2 Background

Increasingly, the space surrounding Earth has become littered by satellites and "space junk" that any new space endeavor must avoid. Should this be an isolated system, where each satellite/space junk/object (henceforth referred to as a "body") only experiences an attractive force with the Earth, it would be relatively trivial to map the positions over time for all bodies analytically. However, gravitational interactions with the sun, moon, asteroids and other pieces of debris have a non-trivial effect on each other, and thus must be taken into account for a truly accurate predication of future positions. In any useful model, we would wish to track as many bodies as possible. This leads to performance difficulties as the force each body imparts on every other body must be calculated.

The gravitational attractive force of object 1 acting on object 2 is given by

$$F = G\frac{m_1 m_2}{r^2} \tag{1.1}$$

where m_1, m_2 are the masses of the interacting objects, and r is the distance between them. G is the gravitational constant, and is $G = 6.67 \times 10^{-11}\text{Nm}^2/\text{kg}^2$[1]. Conveniently, the force on object 1 from object 2 is equal to the force of object 2 from object 1, so we may cut down our computations. Once we know the total force acting on an object, F_{total}, we know the acceleration given by newtons second law of motion,

$$a = \frac{F_{\text{total}}}{m} \tag{1.2}$$

Eventually, we will wish to initialise an example solar system, comprising of a supermassive body (i.e, a star), orbited by some number of much smaller bodies (i.e., planets). The velocity of the orbiting bodies must be perpendicular to the main acceleration, i.e., towards the supermassive body, and the velocity must be such that the gravitational acceleration is equal to the centripetal acceleration. With supermassive body having mass M, orbiting body with mass m, distance r

and velocity v, we have the gravitational acceleration as

$$a = \frac{GM}{r^2} \tag{1.3}$$

[1]centripetal acceleration,

$$a = \frac{v^2}{r} \tag{1.4}$$

[1] Equating and solving for v,

$$\frac{v^2}{r} = \frac{GM}{r^2} \tag{1.5}$$

$$v_{\text{orbit}} = \sqrt{\frac{GM}{r}} \tag{1.6}$$

So, given some arbitrary r, we may calculate the orbital velocity.

1.3 Implementation

1.3.1 Algorithm

All tests in this report use a time range of 100s, with a step size of $h = 0.1$ so consistency. The number of bodies, n, is varied for each test.

Algorithm 1 Main Loop Algorithm

create $n-bodyarrays$
initialise n-bodies
simulate system
delete $n-bodyarrays$

A separate array is used for x position, y position, x, y velocities, x, y accelerations and body mass, where each index represents a unique body. It is tempting to structure the project using a body class or struct (which, in C++, is essentially a class), and then store the bodies in a dynamic vector. However, this adds a few levels of overhead - having to access classes is significantly slower than simply indexing an array. All calculations are done in-place, to reduce memory usage. Similarly, if selected, the current state is output after each time step, so that the arrays need only contain the current state, and not be a matrix including all previous states.

The function to initialise the bodies is not particularly complex. It simply sets the first body to be the "star", then loops through each of the remaining bodies, generating a random distance and angle from the center. The position is then calculated from this, and the orbital speed is calculated using equation 1.6.

The simulation function is also quite simple. At each time step, it calls the `time_step` function, and then outputs some data. The `time_step` follows the following algorithm,

Algorithm 2 Time Step Function

call `update_acceleration`
for each body $\in$ all bodies **do**
    $v_x \leftarrow v_x + h \cdot a_x$
    $v_y \leftarrow v_y + h \cdot a_y$
    $x \leftarrow x + h \cdot v_x$
    $y \leftarrow y + h \cdot v_y$
end for

and so, this is a simply implementation of Euler's method. Euler's method is arguably the most simple iterative method, whereby one updates the value of some function, say $f(x_n)$ to $f(x_{n+1})$ by adding the product of the derivative of the function, $f'(x_n)$ and the difference in the points, $x_{n+1} - x_n$. That is,

$$f(x+h) = f(x) + h * f'(x)$$

. As h becomes small, the updates each time are smaller and in turn more accurate - they approximate less of the acceleration and velocity changes.

Finally, the `update_acceleration` function. This is the most computationally intensive function by far. It looks at every pair of body to calculate the gravitational force that each exerts on one another, and then calculates accelerations experienced by each body, and appends that to the acceleration vectors. It's algorithm is as follows,

Algorithm 3 Update Acceleration Function

for $i \in 1 : n$ **do** ▷ Set all accelerations to 0
  $a_x[i] \leftarrow 0$
  $a_y[i] \leftarrow 0$
end for
for $i \in 1 : n$ **do**
  for $j \in (i+1) : n$ **do**
    $\text{dist} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ ▷ Calculate distance between bodies
    $F_x = -G \cdot m_i \cdot m_j \cdot (x_i - x_j) \cdot \text{dist}^{-3}$
    $F_x = -G \cdot m_i \cdot m_j \cdot (y_i - y_j) \cdot \text{dist}^{-3}$
    $a_x[i] \leftarrow a_x[i] + F_x/m_i, \quad a_x[j] \leftarrow a_x[j] - F_x/m_j$
    $a_y[i] \leftarrow a_y[i] + F_y/m_i, \quad a_y[j] \leftarrow a_y[j] - F_y/m_j$
  end for
end for

The first for loop only takes $\mathcal{O}(n)$ time, so does not contribute so much to the run time. The second for loop, however, runs $\mathcal{O}(n!)$ times! The operation inside the for loop are also relatively slow - particularly the square root and repeated floating point division.

1.3.2 Program Design

Most of the design choices are discussed above as they have been made for optimisation purposes. The project has been created in C++11, making use of a few built in libraries. C++ itself is used over C mostly for being able to handle command line arguments - there is no particular reason it couldn't be ported to C. Plotting is done in gnuplot, and animation is done with a Python script. Details for these are below.

1.3.3 Profiling and Optimisation

To determine which functions where the slowest in runtime, the C++ build command was run with the -pg flag, and without any optimisation flags. `gprof` was then used to produce a profile of the run time. The first few lines of the analysis are attached below, with simulation set up of 20 bodies, timestep 1s, final time 100,000s:

```
Each sample counts as 0.01 seconds.
    %   cumulative   self              self     total
   time   seconds   seconds    calls  ms/call  ms/call  name
   93.42      0.70     0.70   100001     0.01     0.01  update_acceleration(double*, double*, dou
    2.67      0.72     0.02   400004     0.00     0.00  euler(double*, double*, double, int)
    2.67      0.74     0.02   100001     0.00     0.01  time_step(double*, double*, double*, doub
    1.33      0.75     0.01   100001     0.00     0.00  kinetic_energy(double*, double*, double*)
    0.00      0.75     0.00   100001     0.00     0.00  write_state(double*, double*, double, dou
```

```
    0.00      0.75     0.00     2496     0.00     0.00  std::__detail::_Mod<unsigned long, 429496
    ...
```

Clearly, a majority of the run time is spent in the `update_acceleration` function, even though it has fewer calls than, for example, the `euler` function. Clearly, the bulk of the optimisation should be spent here. Since the amount of work done by `update_acceleration` scales nonlinearly with the number of bodies, let us example the effect of increasing the number of bodies to 250, but decreasing the final time to 1000s,

```
Each sample counts as 0.01 seconds.
    %   cumulative   self              self     total
   time   seconds   seconds    calls   s/call   s/call  name
   99.13      1.03     1.03     1001     0.00     0.00  update_acceleration(double*, double*, dou
    0.96      1.04     0.01     4004     0.00     0.00  euler(double*, double*, double, int)
    0.00      1.04     0.00     2496     0.00     0.00  std::__detail::_Mod<unsigned long, 429496
    0.00      1.04     0.00     2496     0.00     0.00  unsigned long std::__detail::__mod<unsign
    0.00      1.04     0.00     2492     0.00     0.00  std::__detail::_Mod<unsigned long, 624ul,
```

Increasing the number of bodies further increases the relative time spent inside of `update_acceleration` . If this program is to be used for large numbers of orbitting bodies, doing any optimisations elsewhere would be trivial compared to the potential optimisations awarded by decreasing the run time in `update_acceleration`. `update_acceleration` could be written in the following, relatively naive way:

```
1  void update_acceleration(double p_X[], double p_Y[], double p_M[], double p_aX
       [], double p_aY[]) {
2      // calculate accelerations
3      // set all accelerations to zero
4      for (int i = 0; i < nBodies; i++) {
5          p_aX[i] = 0;
6          p_aY[i] = 0;
7      }
8
9      // for each pair...
10     for (int i = 0; i < nBodies; i++) {
11         for (int j = 0; j < nBodies; j++) {
12             if (i != j) {
13                 // get distance between bodies
14                 double dist = sqrt(pow(p_X[i] - p_X[j], 2) + pow(p_Y[i] - p_Y[j
                       ], 2));
15                 // calculate component forces
16                 double fX = -G * p_M[i] * p_M[j] * (p_X[i] - p_X[j])/pow(dist,
                       3);
17                 double fY = -G * p_M[i] * p_M[j] * (p_Y[i] - p_Y[j])/pow(dist,
                       3);
18                 // append component accelerations to arrays
19                 p_aX[i] += fX / p_M[i];
20                 p_aY[i] += fY / p_M[i];
21             }
22         }
23     }
24 }
```

All of the accelerations are set to 0, then, for each of bodies, the force on the ith body is calculated, and accelerations set accordingly. However, we may apply some key optimisations to greatly reduce the work done:

- given the the force acting on object a due to b is equal to the force acting on object b due to a, we can reduce the for loops from looking at every pair (n^2 comparisons) to just have

each body look at the bodies past it in the index, then update both bodies accelerations ($\frac{n^2-n}{2}$ comparisons)

- a lot of the calculations done here are repeated twice - simply precalculating these should decrease runtime by a factor of roughly 2 for these sections

Implementing these improvements, we are left with

```
double update_acceleration(double p_X[], double p_Y[],
                           double p_M[], double p_aX[],
                           double p_aY[]) {
    // updates accelerations and returns total potential energy
    double PE = 0;
    // for each pair...
    double dX, dY, dist, coeff, fX, fY, inv_r_cube;
    for (int i = 0; i < nBodies; i++) {
        for (int j = i + 1; j < nBodies; j++) {
            // get distance between bodies
            dX = p_X[i] - p_X[j]; dY = p_Y[i] - p_Y[j];
            dist = sqrt(dX * dX + dY * dY); // cartesian dist
            // calculate component forces
            coeff = -G * p_M[i] * p_M[j];
            inv_r_cube = pow(dist, -3);
            fX = coeff * dX * inv_r_cube;
            fY = coeff * dY * inv_r_cube;
            // append component accelerations to arrays
            p_aX[i] += fX / p_M[i]; p_aX[j] -= fX / p_M[j];
            p_aY[i] += fY / p_M[i]; p_aY[j] -= fY / p_M[j];
            // increase potential energy value
            PE += coeff / dist;
        }
    }

    return PE;
}
```

Note that, in this version, the acceleration arrays had been set to 0 at a different point in the code.

The integration methods (Euler's method) step size h is linearly proportional to run time, and the choice of which is very dependant on the situation. Thus, trying to "optimise" the code by increase the step size does not really help the situation (although it does reduce calls to `update_acceleration`). Instead, optimisation flags were trialed, and the relation between number of bodies and run time determined experimentally.

1.4 Verification

Due to the nature of the problem, analytical solutions to initial conditions do not exist. Instead, to "verify" that the code is performing correctly, we confirm that the total energy in the system does not vary over time. Since this is a closed system, the total energy should remain constant with time, and a poor integration method would show increases in energy when the situations discussed above occur. This is calculated as follows

```
    fabs(fabs(final_TE - initial_TE)/initial_TE)
```

To examine the conditions in which the serial code works, the program was run with varying Δt, and the resulting change in energy recorded. This was repeated for a range of n-body counts. The results are plotted in figure 1.1.

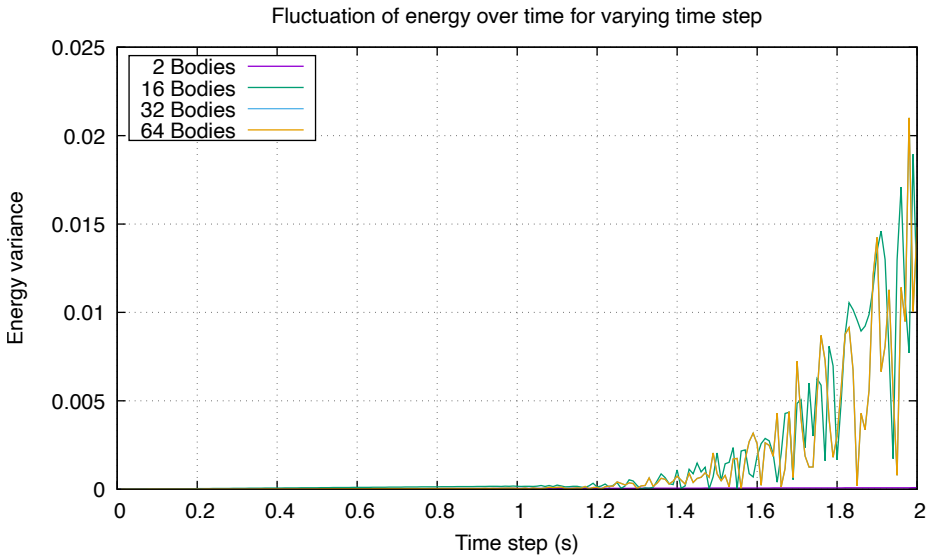

Figure 1.1: Comparison of how the energy varies over a 1000s simulation for 2, 16, 32 and 64 bodies.

Below approximately $\Delta t = 1$, there is essentially no change in energy over the run time (that is, within numerical error from calculating the energy components). Past $\Delta t = 0$, the energy variance fluctuates for each set of trials with more than 2 bodies. Consider that the number of potential collisions (or close passes of two bodies) increases with the number of bodies, and hence accelerations spike more often, and the numerical solution is less accurate, in turn leading to jumps in total energy. This is shown below is a pair of 'good' and 'bad' situations -

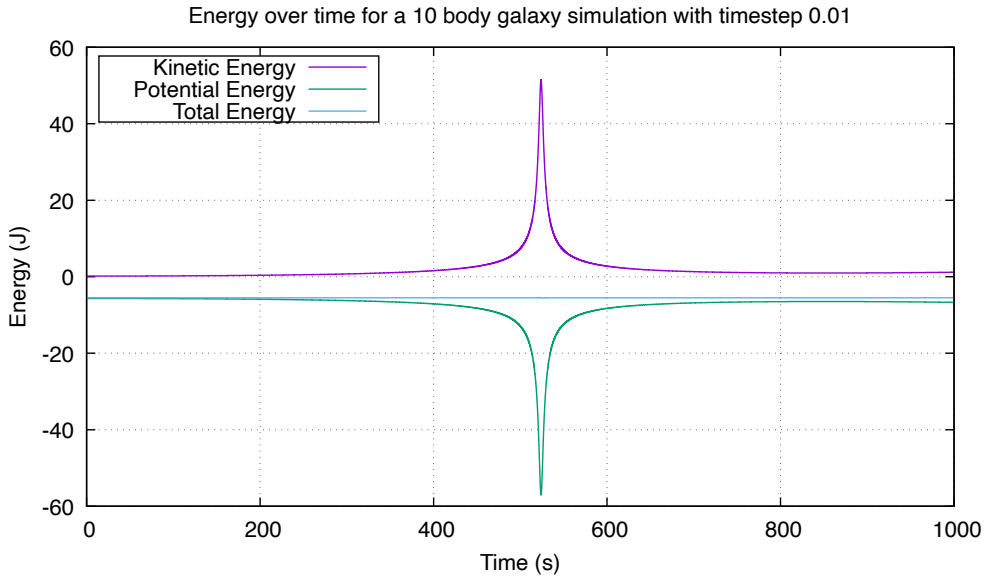

Figure 1.2: Kinetic, potential and total energy over time for a 10 body galaxy simulation, with a small timestep over 1000s

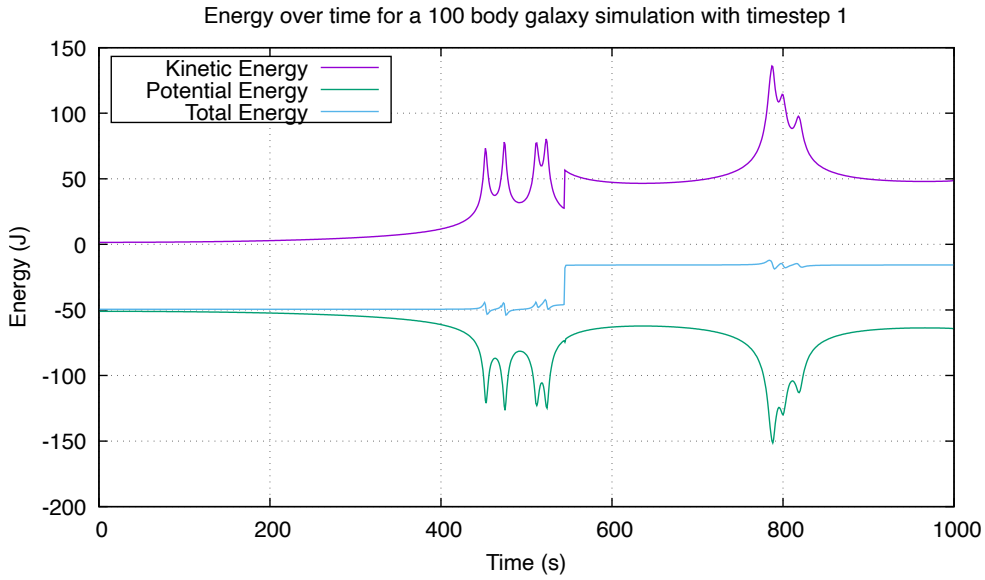

Figure 1.3: Kinetic, potential and total energy over time for a 100 body galaxy simulation, with a large timestep over 1000s

As you can see, the total energy in figure 1.2 is constant, and the trade-off between potential and kinetic energy at a close approach between two planets is smooth, as expect. In figure 1.3, the close approaches between pairs of planets cause discontinuities in the total energy, eventually leading to a significant change in the total energy of the system.

To complement the above results, some basic animations of bodies position over time were produced. Some of these animations are included in the presentation. These showed a couple trends:

- When bodies are far apart, the attractive force is very small, as expected. Conversely, when bodies are closer, the attractive becomes very strong, as expected.
- When two bodies become very close, the attractive force, and hence eventually relative velocity becomes very large. As we are using a fixed step size, this can lead to large jumps in position, perhaps to points that are unphysical. One example that has been observed is that two bodies become close, at one time step that are very close and hence experience a very strong force, and at the next time step are moved very far apart. Since the acceleration at the next time step is relatively low, the two bodies continue on at unphysical speeds.
- In contrast, this is seen very rarely for significantly smaller step sizes.

For simulations henceforth, when we are not examining the relationship between step size or final time and some other parameter, all simulations will be run to a final time of 1000s, with a step size of $\Delta t = 1$s.

1.5 Performance Results

To benchmark the performance of the program, the simulation was run many times with varying parameters, and timed using bash's `time` command. All tests were done with a final time of $1000s$, with a time step of $1s$. The number of bodies ranged from 10 to 1000. For each number of bodies, the program was run with the optimisation flags -O0, -O1, -O2 and -O3. This was run on `getafix`, and the `goslurm.sh` script is attached below. The flag number, number of bodies, and real time (i.e., wall clock time) were piped to a CSV file, and plotted using gnuplot to produce figure 1.4.

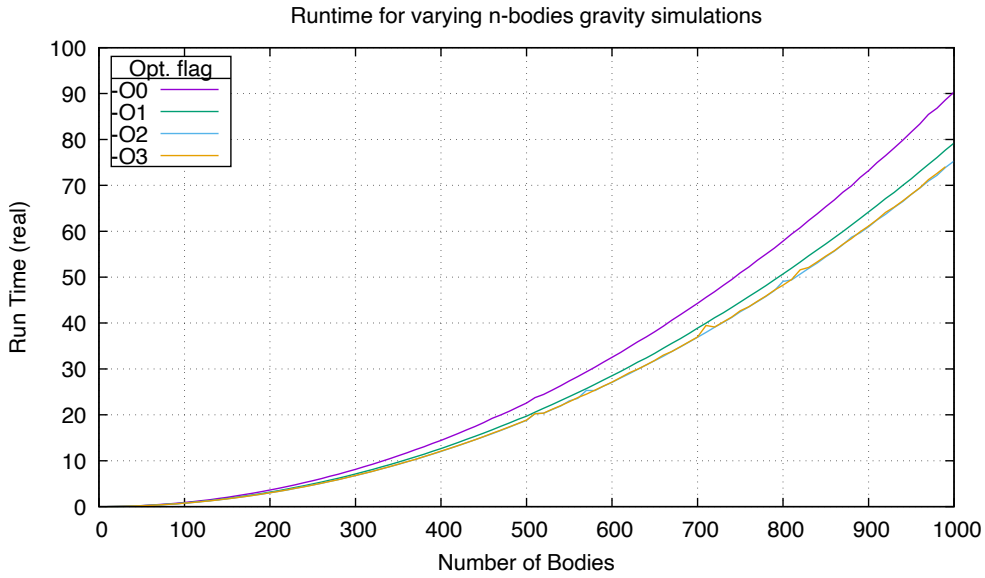

Figure 1.4: Plot of run-times for running a simulation with 1000 time steps, with a range of body counts, and using different compilation flags.

Clearly, the relationship between number of bodies and runtime follows the predicted pattern of $\mathcal{O}(n!)$. The optimisation flag -O0 performed the worst, and -O2 and -O3 performed similarly well for the best. -O3 took around 75 seconds for 1000 bodies, and -O0 took around 90 seconds for the same parameters, a difference of about 18%.

Given that this model could feasibly be used for a system with many more than n bodies, and the number of time steps here is very small, chosen only as an example, this implementation is currently too slow to be practical.

Chapter 2

Parallel

This chapter attempts to take the serial code created in the previous chapter and decrease the runtime using parallelisation techniques, particularly doing repeated calculations in parallel on a GPU using Cuda. First, there is a discussion of the limitations of parallising this problem, followed by details of the implementation and the verification procedure. Finally, the performance is compared to the serial implementation, particularly how the problem scales with number of bodies.

2.1 Parallelisation Strategies

A key limitation of the n-body problem is that the problem is that each state is dependant on the entirety of the previous state - for each new state, we must first calculate the acceleration of each body, which is dependant on the current location of every state, which can only be calculated at the end of the state. Hence, we cannot, for example, run the `update_acceleration()` function for two states simultaneously, and our parallelisation methods are constrained to only operate on the current state.

First, we shall focus on optimising the `update_acceleration()` function. As shown in the serial gprof, most of the runtime is spent in this function, especially for large numbers of bodies. Thankfully, the calculations for the new state are *only* dependant on the previous state, and as such we can calculate the acceleration for each body without waiting for the other bodies. All of the calculations done in `update_acceleration()` are relatively simple, and so we will have no issue running these using Cuda. It is worth noting, however, the some trickery is required to keep the number of comparisons low.

As discussed earlier, the absolute best way to decrease the runtime of the program is to decrease the number of comparisons made in `update_acceleration()`. This is done by only looking at unique pairs of bodies, rather than all bodies (that is, once we've looked at the pair $(1, 10)$, we don't need to look at $(10, 1)$). This was done using careful nested for loops, but for Cuda we'd rather do as little work on each thread at once. Thus, we precalculate the list of pairs of bodies to compare (are relatively instant process), and then use the index of the current Cuda thread to get the current pair to compare.

Beyond this, the implementation is relatively simple, keeping in mind to copy information to and from the device and host as infrequently as possible to save time.

2.2 CUDA Implementation

The implementation of Cuda to parallise the code is fairly straightforward for most of the code - the only functions affected are `main()`, `simulation()`, `time_step()` and, most importantly, `update_acceleration()`. Most of the changes are just a matter of allocating and freeing memory, and copying memory back and forth when needed. Three key changes, however: first, in `main()`,

```
// generate (i,j) pairs for body comparisons
int pairs = (nBodies * nBodies - nBodies) / 2;
int *Is = new int[pairs];
int *Js = new int[pairs];
int ind = 0;
for (int i = 0; i < nBodies; i++) {
    for (int j = i + 1; j < nBodies; j++) {
        Is[ind] = i; Js[ind] = j;
        ind++;
    }
}
```

This generates the list of (i, j) pairs, so that the same optimisation that significantly reduces the working set can still be used when working in parallel.

Next, in `time_step()`,

```
// recalculate acceleration arrays
update_acceleration<<<pairs / 32 + 1, 32>>>(p_X_device, p_Y_device,
                                            p_M_device, p_aX_device,
                                            p_aY_device, Is, Js,
                                            PE_device, pairs);
```

The call to `update_acceleration()` is careful to minimize the number of blocks and maximize the number of threads. There will be, at most, 31 excess threads initialized.

Finally, in `update_acceleration()`,

```
__global__ void update_acceleration(double p_X[], double p_Y[],
                                    double p_M[], double p_aX[],
                                    double p_aY[], int Is[], int Js[],
                                    double *PE_device, int pairs) {
    // updates accelerations and returns total potential energy
    // parameters:
    //     p_X, p_Y: (x,y) values for each body
    //     p_vX, p_vY: (x,y) velocites for each body
    //     p_aX, p_aY: (x,y) accelerations for each body
    //     p_M: mass of each body
    //     Is, Js: form the (i,j) pairs that need to calculated for each body
    //     pairs: integer number of pairs to calculate accelerations for
    //     PE_device: pointer to potential energy sum on GPU
    // output:
    //     no output - overwrites parameter arrays
    const int ind = blockIdx.x * blockDim.x + threadIdx.x;

    // for each pair...
    if (ind < pairs) {
        int i = Is[ind]; int j = Js[ind];

        [sic]

        atomicAdd(PE_device, PE);
    }
}
```

Alongside the earlier additions to code, the lines where the index is calculated to find the (i, j) pair allows the function to be split across as many threads as possible, and sending an even amount of work to each thread. If each thread dealt with a single body, threads would either be doing redundant calculations or the certain threads would have significantly more work than others, leading to many threads waiting.

The overall process by which the code works has not changed, so we should see very similar results to serial in the verification procedure.

2.2.1 Profiling

To profile, we'll run the code once with very few bodies and a long simulation time (20 bodies, timestep 1s, final time 100,000s):

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 37.50      0.03     0.03   100000   300.01   300.01  time_step(double*, double*, double*, double
 37.50      0.06     0.03                             cudart::getCudartError(cudaError_enum)
 12.50      0.07     0.01   100000   100.00   100.00  kinetic_energy(double*, double*, double*)
 12.50      0.08     0.01                             cudart::cuosCondCreateWithSharedFlag(pthrea
  0.00      0.08     0.00   100000     0.00     0.00  write_state(double*, double*, double, doubl
  ...
```

We see that the time is most distributed between the `time_step()` function and `kinet_energy ()` function, and some function used for dealing with memory on the device. This suggests that the CUDA code adds a significant overhead for very few bodies! To compare, we will run the simulation again with 256 bodies, timestep of 1s, and final time of 10,000s,

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 70.00      0.07     0.07    20000     3.50     3.50  time_step(double*, double*, double*, double
 10.00      0.08     0.01    10000     1.00     1.00  __device_stub__Z19update_accelerationPdS_S_
 10.00      0.09     0.01                             write_header(std::basic_ofstream<char, std:
 10.00      0.10     0.01                             simulate(double*, double*, double*, double*
```

Here, the bulk of the time is still spent in the `time_step` function, but now a significant portion is also spent in `update_acceleration`. Increasing the number of bodies has a very small effect on the time profile of the program - this suggests that the parallelisation techniques have been successful.

2.3 Verification

To verify that the Cuda implementation is performing correctly, the same tests as were performed for serial are used. The system is closed, so we expect to see energy conserved, and hence the y axis to stay near 0,

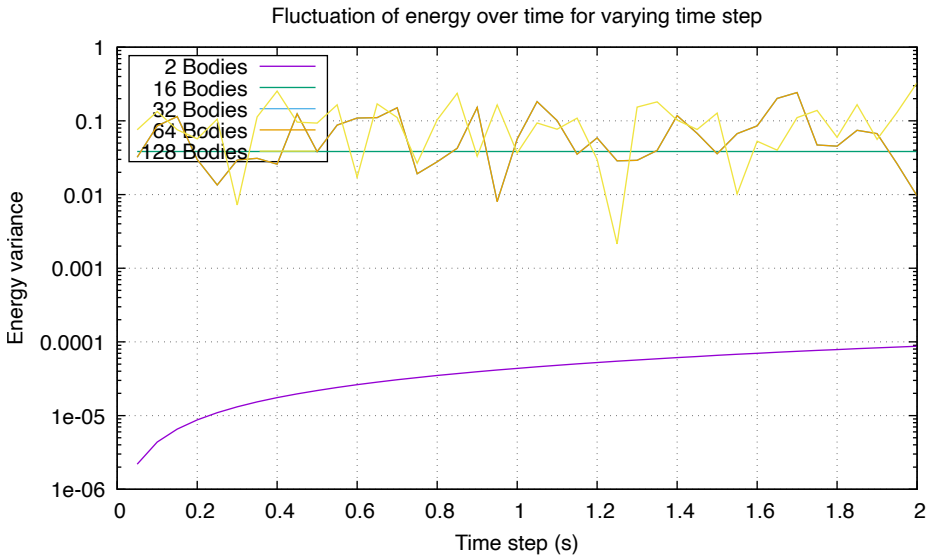

Figure 2.1: Comparison of how the energy varies over a 1000s simulation for 2, 16, 32, 64 and 128 bodies.

Clearly, the step size does not have a significant effect on any of the results - the 2 body system is spaced enough such that the error does not change by any significant amount. The 16 and 32 body systems perform similarly to the serial test, showing that step sizes under $\Delta t = 2$ are suitable. Past this, significant noise is seen in the errors of the 64 and 128 body systems, noticeably more severe in the 128 body system. This is as expected - as the number of bodies increases, so does the density of the bodies and the likelihood of "collisions" (or, at least, particles jumping very close together then far apart), and hence we see poorer performance here.

Next, we examine a "good" and "bad" simulation,

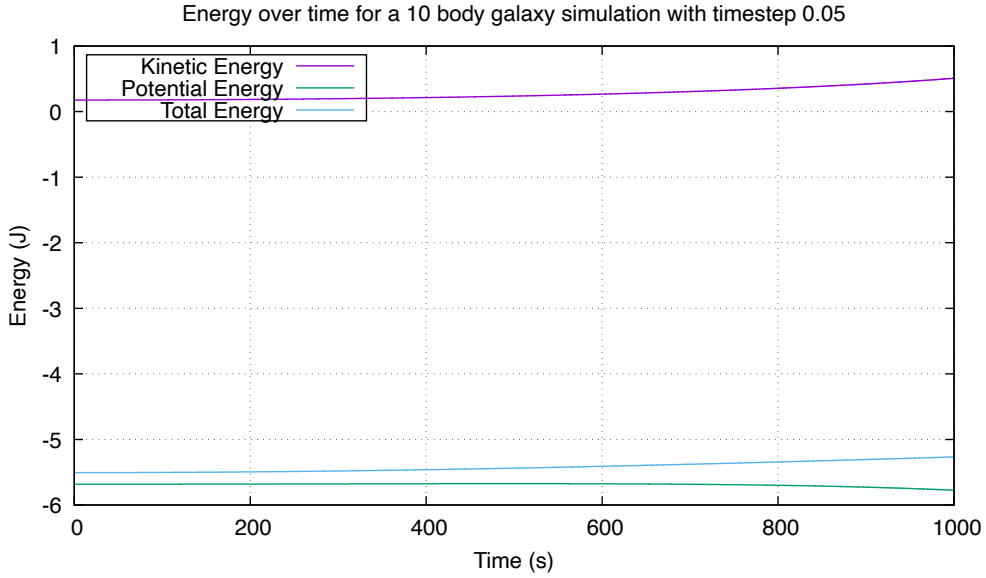

Figure 2.2: Kinetic, potential and total energy over time for a 10 body galaxy simulation, with a small timestep over 1000s

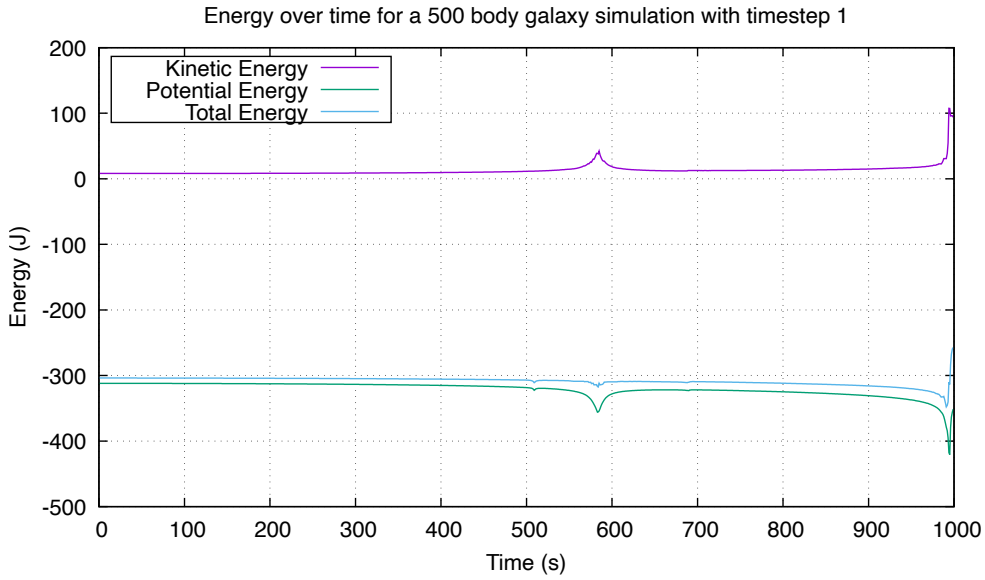

Figure 2.3: Kinetic, potential and total energy over time for a 500 body galaxy simulation, with a large timestep over 1000s

In both of the above scenarios, we slight change in total energy of the run time a constant total energy, a change of approximately 3% for the good situation and approximately 15% for the poor scenario, with a discontinuity near the end of the simulation. This is of course concerning, but considering the change is being observed, in the case of the good scenario, over 20 thousand iterations, I believe this to be an acceptable change. The more drastic change in the poor scenario may be attributed to the significantly larger timestep.

2.4 Experimentation

Finally, we wish to measure the time performance of the parallised code, particularly in comparison to the serial code. To do this, we will run tests with a time step of $\Delta t = 0.01s$, to a final time of $100s$, first purely for the Cuda code to determine the best compile flag to use, and then with the serial code to determine which scales better and quantify the difference.

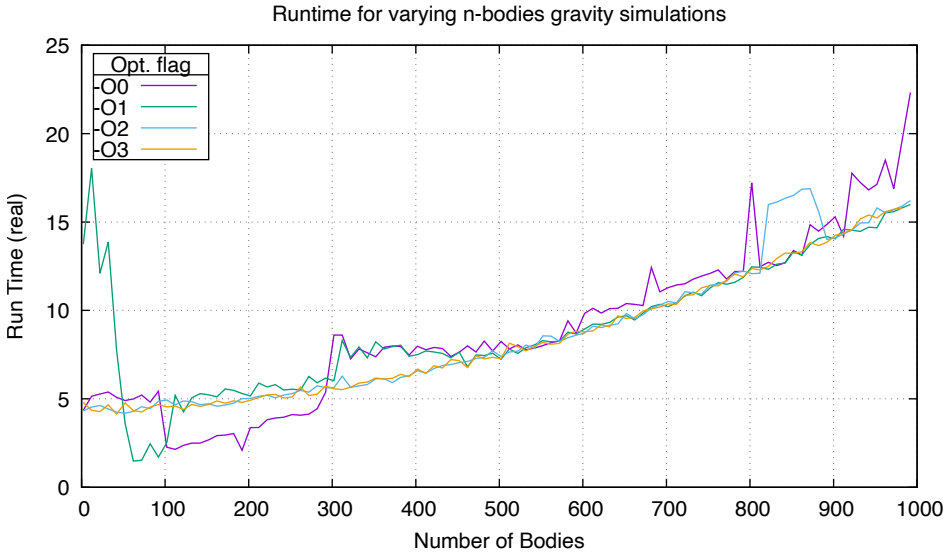

Figure 2.4: Effect of number of bodies of run time of parallel code, with varying optimisation flag

The plot of runtime against problem size in figure 2.4 is quite promising - in the best case, simulating 1000 bodies took only 15s, whereas it took closer to 90s in the serial case! Otherwise, we can observe that the choice of flag has a significant effect on the result. The runtime with optimisation flag -O3 is very consistent. -O2 follows the same trend closely, aside from a jump at 300 and 800 bodies. Flags -O0 and -O1 are relatively 'noisey' in comparison, providing wildly varying run time - in some case, much fast than the other flags, however typically worse. As we are most interested in how the code performs as the problem size becomes large, -O3 will be used henceforth.

Now, let us directly compare the performance of the parallel and serial code. We will produce 2 plots - in the first, we plot the run time as a function of the number of bodies. Then, we plot the run time as a function of the final time of the simulation, for a range of different numbers of bodies.

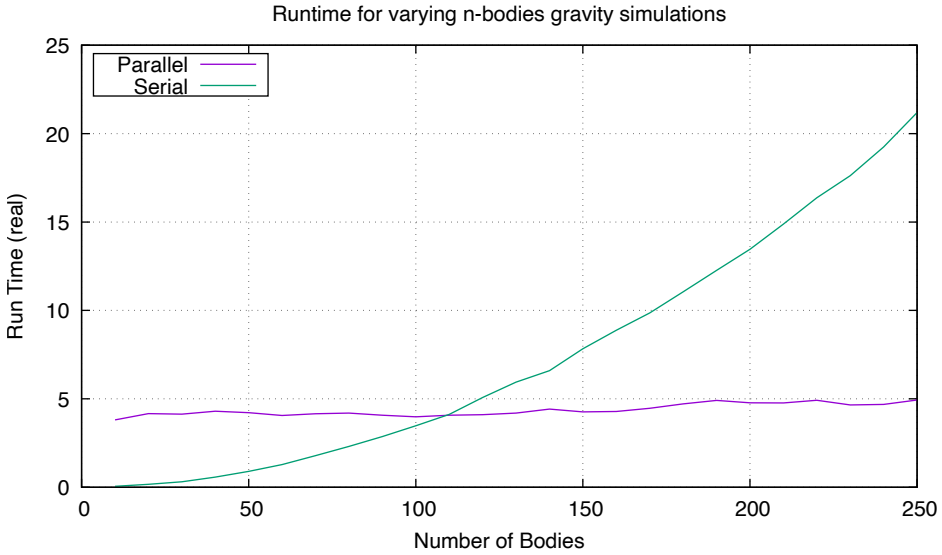

Figure 2.5: Comparison of the run time of the parallel and serial code, for fixed final time and step size (1000s and 1s), and varying number of bodies.

Figure 2.5 shows the run time of the serial code to grow faster than linearly as the number of bodies is increased, and the run time of the parallel code increases only very slightly. Interestingly, for small body counts, below 100, the serial code is faster - this is likely due to the significant overhead associated with running the parallel code (much more initialisation of memory, copying memory around, etc). Clearly, for simulations with many bodies, the overhead is negligible compared to the speed increase, and hence parallel becomes the faster code.

Finally, the compare the effect of changing run time, lets set 3 different body counts: 32, 128, and 256. We should see that the serial code performs better than the parallel code at 32 bodies, similarly at 128, and worse at 256. This is plotted in figure 2.6. As expected, the serial code is faster initially, however becomes slower than the parallel implementation as the final time increases. The rate at which this occurs is proportional to the number of bodies being simulated.

In figure 2.6, the parallel code appears to run in constant time relative to both number of bodies and the final time - in actual fact, it is more likely that the parallel code is so much faster in the `update_acceleration()` function as compared to the serial code that when the two programs are directly compared the parallel code appears to run `update_acceleration()` instantly.

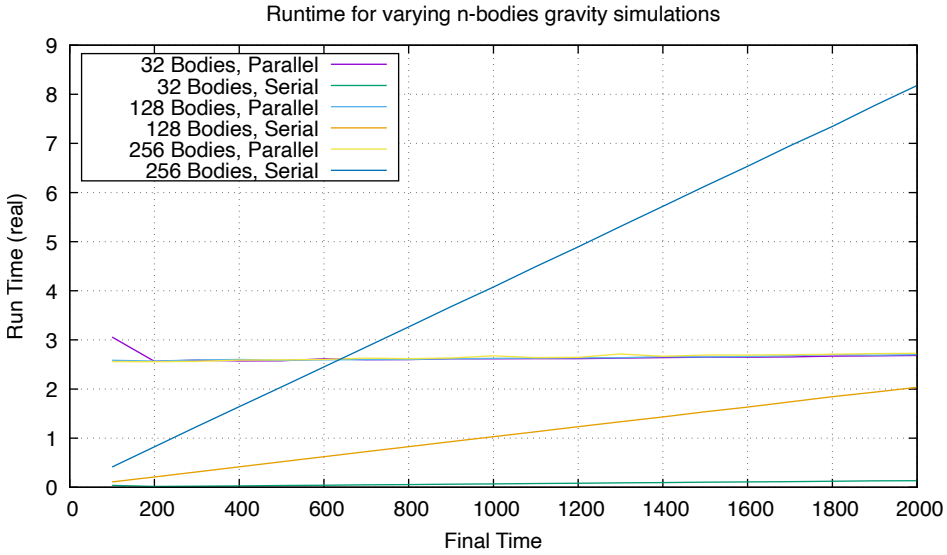

Figure 2.6: Plot of the relationship between run time and final time of simulation, compared between the parallel and serial implementations for a range of body counts.

2.5 Conclusion

This report aimed to propose a naive and later optimised solution to the n-body problem, where some number of bodies experience a gravity force on one another. The first chapter, where the serial implementation was discussed, showed how the amount of work done may be greatly decreased with some careful precalculation and choice of nested for loops. The profile analysis of this code showed that most of the run time was spent calculating the acceleration on each body, so the second chapter discusses how to optimise this function by running it in parallel on a GPU, using CUDA.

Although the opportunities for parallelisation of the simulation are limited - each state requires the entire previous state to be calculated - implementing the simulation on CUDA is successful at providing a very significant speed increase, particularly when simulating scenarios with a large number of bodies and/or a distant final time, despite the overhead that is introduced. This speed increase was so significant that the function to update accelerations was no longer the function where most of the run time was spent. Hence, with the improvements made, the code is now suitable to simulate systems with very many bodies (orders of magnitude than feasible with the naive solution).

Although the optimisation was successful, this report is not without fault or limitations. A more careful and realistic study into the propagation of bodies over time would have strengthened the verification sections - for example, comparing to a known system, like the Milky way, or the analytical solution to a 2/3-body problem. Also, the parallel code was only tested for scenarios that are feasible to test alongside the serial code, due to the serial code being prohibitively slow for large numbers of bodies. Finally, the use of Euler's method for iterating the velocities and position was chosen to minimise the number of calls to the `update_acceleration()` function, with the trade off of accuracy. Since the parallelised code has decreased the run time of `update_acceleration()` so far, a method such as kick-drift-kick or verlet integration (both or which would be required to make more calls to the `update_acceleration()` function) should be implemented and the effect on the verification studied.

Bibliography

[1] Randall D. Knight. *Physics for Scientists and Engineers*. Pearson, 2017.

Chapter 3

Appendix

3.1 Serial Bash Scripts

3.1.1 Serial Validation

Shell script to submit to `sbatch` and produce datafiles:

```
#!/bin/bash -l
#SBATCH --job-name=SerialValidation
###SBATCH --nodelist=smp-6-3
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=2G # memory (MB)
#SBATCH --time=0-00:30 # time (D-HH:MM)

FINAL_TIME=1000

make serial

# save example run to file that probably doesn't␣break
build/main_serial --nBodies 10 --finalTime ${FINAL_TIME} --timeStep 0.01 --
    filename output/serial_valid_good.dat
# save example run to file that probably does break
build/main_serial --nBodies 100 --finalTime ${FINAL_TIME} --timeStep 1 --
    filename output/serial_valid_bad.dat

echo "dt␣2body␣16body␣32body␣64body" > output/serial_valid.csv
for dt in $(seq 0.01 0.01 2); do
    err1=$( { build/main_serial --nBodies 2 --finalTime ${FINAL_TIME} --
        timeStep ${dt} --energyDiff; } 2>&1 )
    err2=$( { build/main_serial --nBodies 16 --finalTime ${FINAL_TIME} --
        timeStep ${dt} --energyDiff; } 2>&1 )
    err3=$( { build/main_serial --nBodies 32 --finalTime ${FINAL_TIME} --
        timeStep ${dt} --energyDiff; } 2>&1 )
    err4=$( { build/main_serial --nBodies 64 --finalTime ${FINAL_TIME} --
        timeStep ${dt} --energyDiff; } 2>&1 )
    echo "${dt}␣${err1}␣${err2}␣${err3}␣${err4}" >> output/serial_valid.csv
done
```

Plotting scripts: for the key plot,

```
set terminal pdf
```

```
set output "serial_valid.pdf"

set xlabel "Time step (s)"
set ylabel "Energy variance"
set title "Fluctuation of energy over time for varying time step"
set grid

set key left top
set key box

plot 'serial_valid.csv' every ::1 using 1:2 with lines title "2 Bodies",\
     'serial_valid.csv' every ::1 using 1:3 with lines title "16 Bodies",\
     'serial_valid.csv' every ::1 using 1:4 with lines title "32 Bodies",\
     'serial_valid.csv' every ::1 using 1:4 with lines title "64 Bodies"
```

For the 'good' scenario,

```
set terminal pdf
set output "serial_valid_good.pdf"

set xlabel "Time (s)"
set ylabel "Energy (J)"
set title "Energy over time for a 10 body galaxy simulation with timestep 0.01"
set grid

set key left top
set key box

plot 'serial_valid_good.dat' every ::1 using 1:2 with lines title "Kinetic
    Energy",\
     'serial_valid_good.dat' every ::1 using 1:3 with lines title "Potential
        Energy",\
     'serial_valid_good.dat' every ::1 using 1:4 with lines title "Total Energy
        "
```

For the 'bad' scenario,

```
set terminal pdf
set output "serial_valid_bad.pdf"

set xlabel "Time (s)"
set ylabel "Energy (J)"
set title "Energy over time for a 100 body galaxy simulation with timestep 1"
set grid

set key left top
set key box

plot 'serial_valid_bad.dat' every ::1 using 1:2 with lines title "Kinetic
    Energy",\
     'serial_valid_bad.dat' every ::1 using 1:3 with lines title "Potential
        Energy",\
     'serial_valid_bad.dat' every ::1 using 1:4 with lines title "Total Energy"
```

3.1.2 Parallel Validations

Shell script to submit to `sbatch` and produce datafiles:

```
#!/bin/bash -l
#SBATCH --job-name=ParallelValidation
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --partition=gpu
#SBATCH --gres=gpu:1
#SBATCH --mem-per-cpu=8G # memory (MB)
#SBATCH --time=0-00:30 # time (D-HH:MM)

FINAL_TIME=1000

make cuda

module load gnu
module load cuda

# save example run to file that probably doesn't␣break
build/main_cuda --nBodies 10 --finalTime ${FINAL_TIME} --timeStep 0.05 --
    filename output/cuda_valid_good.dat
# save example run to file that probably does break
build/main_cuda --nBodies 500 --finalTime ${FINAL_TIME} --timeStep 1 --filename
     output/cuda_valid_bad.dat

echo "dt␣2body␣16body␣32body␣64body␣128body" > output/cuda_valid.csv
for dt in $(seq 0.05 0.05 2); do
    err1=$( { build/main_cuda --nBodies 2 --finalTime ${FINAL_TIME} --timeStep
        ${dt} --energyDiff; } 2>&1 )
    err2=$( { build/main_cuda --nBodies 16 --finalTime ${FINAL_TIME} --timeStep
         ${dt} --energyDiff; } 2>&1 )
    err3=$( { build/main_cuda --nBodies 32 --finalTime ${FINAL_TIME} --timeStep
         ${dt} --energyDiff; } 2>&1 )
    err4=$( { build/main_cuda --nBodies 64 --finalTime ${FINAL_TIME} --timeStep
         ${dt} --energyDiff; } 2>&1 )
    err5=$( { build/main_cuda --nBodies 128 --finalTime ${FINAL_TIME} --
        timeStep ${dt} --energyDiff; } 2>&1 )
    echo "${dt}␣${err1}␣${err2}␣${err3}␣${err4}␣${err5}" >> output/cuda_valid.
        csv
done
```

Plotting scripts same as above with relevant adjustments.

3.2 Performance Analysis

3.2.1 Serial Flags

```
#!/bin/bash -l
#SBATCH --job-name=GalaxySimTS
###SBATCH --nodelist=smp-6-3
#SBATCH --nodes =1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1G # memory (MB)
#SBATCH --time=0-00:01  # time (D-HH:MM)

export TIMEFORMAT='%R,%U,%S'
N_BODIES=$(seq 2 10 100)
# N_BODIES=$(seq 2 10 1000)
FLAGS=(0 1 2 3)
FINAL_TIME=100
TIME_STEP=0.1

echo "flag,nBodies,rtime,utime,stime" > out.csv

for flag in ${FLAGS[@]}; do
    g++ -std=c++11 -O${flag} -Wall main.cpp -o main
    for n in ${N_BODIES[@]}; do
            t=$( { time ./main --nBodies ${n} --finalTime ${FINAL_TIME} --
                timeStep ${TIME_STEP} --noSummary; } 2>&1 )
            echo "${flag},${n},${t}" >> out.csv
    done
done
```

3.2.2 Parallel Flags

```
#!/bin/bash -l
#SBATCH --job-name=ParallelValidation
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --partition=gpu
#SBATCH --gres=gpu:1
#SBATCH --mem-per-cpu=8G # memory (MB)
#SBATCH --time=0-02:00  # time (D-HH:MM)

export TIMEFORMAT='%R,%U,%S'
# N_BODIES=$(seq 2 20 100)
N_BODIES=$(seq 2 10 1000)
FLAGS=(2 3)
FINAL_TIME=100
TIME_STEP=0.01

module load gnu
module load cuda

echo "flag,nBodies,rtime,utime,stime" > output/cuda_flag_parttwo.csv
echo "Starting␣for␣loop"
```

```
24
25 for flag in ${FLAGS[@]}; do
26     make cuda OLEVEL=-O${flag}
27     for n in ${N_BODIES[@]}; do
28             t=$( { time build/main_cuda --nBodies ${n} --finalTime ${FINAL_TIME}
                       --timeStep ${TIME_STEP} --noSummary; } 2>&1 )
29             echo "${flag},${n},${t}" >> output/cuda_flag_parttwo.csv
30     done
31     echo "flag␣-O${flag}␣done!"
32 done
```

3.2.3 Misc

n-body comparison

Final time comparison

```
1  #!/bin/bash -l
2  #SBATCH --job-name=ParallelVsSerial
3  #SBATCH --nodes=1
4  #SBATCH --ntasks=1
5  #SBATCH --ntasks-per-node=1
6  #SBATCH --cpus-per-task=1
7  #SBATCH --partition=gpu
8  #SBATCH --gres=gpu:1
9  #SBATCH --mem-per-cpu=8G # memory (MB)
10 #SBATCH --time=0-02:00 # time (D-HH:MM)
11
12 export TIMEFORMAT='%R,%U,%S'
13 N_BODIES=(32, 128, 256)
14 FINAL_TIMES=$(seq 100 200 5000)
15 TIME_STEP=1
16
17 module load gnu
18 module load cuda
19
20 make all
21
22 echo "nBodies,finalTime,rtime_para,utime,stime,rtime_seri,utime,stime" > output
       /run_time_comp.csv
23 echo "Starting␣for␣loop"
24
25 for n in ${N_BODIES[@]}; do
26     for FINAL_TIME in ${FINAL_TIMES[@]}; do
27         t_para=$( { time build/main_cuda --nBodies ${n} --finalTime ${FINAL_TIME
              } --timeStep ${TIME_STEP}; } 2>&1 )
28         t_seri=$( { time build/main_serial --nBodies ${n} --finalTime ${
              FINAL_TIME} --timeStep ${TIME_STEP}; } 2>&1 )
29         echo "${n},${FINAL_TIME},${t_para},${t_seri}" >> output/run_time_comp.
              csv
30     done
31     echo "\n\n"
32 done
33 echo "done!"
```

3.3 Main Source Code

3.3.1 Serial

```
/*
Galaxy Simulation - COSC3500
Tom Stephen, 45811449

Program to simulate gravity interaction between N bodies.

This is the *serial* implementation.

See README.md for instructions on usage.
*/

#include <cstdlib>
#include <iomanip>
#include <iostream>
#include <math.h>
#include <cstring>
#include <random>
#include <chrono>
#include <ctime>
#include <fstream>

#define G 1
#define PI 3.1415

int nBodies = 2;
double tf = 1000;
double h = 1.0;
bool energy_diff = false;
bool verbose = false;
std::string filename;

#include "system_init.cpp"
#include "debugger_tools.cpp"
#include "ode_solve.cpp"
#include "helper_functions.cpp"

double update_acceleration(double p_X[], double p_Y[], double p_M[], double
    p_aX[], double p_aY[]);
double time_step(double p_X[], double p_Y[], double p_vX[], double p_vY[],
    double p_aX[], double p_aY[], double p_M[]);
void simulate(double p_X[], double p_Y[], double p_vX[], double p_vY[], double
    p_aX[], double p_aY[], double p_M[], std::string filename);

int main(int argc, char* argv[]) {
    // parse command line arguments
    parse_args(argc, argv);

    // create arrays
    double *planet_X = new double[nBodies];
    double *planet_Y = new double[nBodies];

```

```
49      double *planet_vX = new double[nBodies];
50      double *planet_vY = new double[nBodies];
51
52      double *planet_aX = new double[nBodies];
53      double *planet_aY = new double[nBodies];
54
55      double *planet_M = new double[nBodies];
56
57      // initialise planets
58      init_N_orbitting_planets(planet_X, planet_Y, planet_vX, planet_vY, planet_M
            , nBodies);
59
60      // simulate system
61      simulate(planet_X, planet_Y, planet_vX, planet_vY, planet_aX, planet_aY,
            planet_M, filename);
62
63      // delete arrays
64      delete [] planet_X;
65      delete [] planet_Y;
66
67      delete [] planet_vX;
68      delete [] planet_vY;
69
70      delete [] planet_aX;
71      delete [] planet_aY;
72
73      delete [] planet_M;
74
75      exit(0);
76  }
77
78  void simulate(double p_X[], double p_Y[],
79                double p_vX[], double p_vY[],
80                double p_aX[], double p_aY[],
81                double p_M[], std::string filename) {
82      // simulates n-body system with gravity forces
83      // parameters:
84      //     p_X, p_Y: (x,y) values for each body
85      //     p_vX, p_vY: (x,y) velocites for each body
86      //     p_aX, p_aY: (x,y) accelerations for each body
87      //     p_M: mass of each body
88      //     filename: name & location to write the output file
89      //
90      // No returns, just output file and printing relative change in total
            energy.
91
92      // open output file
93      std::ofstream outfile;
94      outfile.open(filename);
95      // headerline
96      write_header(outfile);
97      double PE, KE, TE = 0.0, initial_TE = 0.0, final_TE;
98      // do simulation until time reaches final time `tf`
99      for (double time = 0.0; time <= tf; time += h) {
100         // step forwards by `h`
```

```
        PE = time_step(p_X, p_Y,
                       p_vX, p_vY,
                       p_aX, p_aY,
                       p_M);
        // calculate energies
        KE = kinetic_energy(p_vX, p_vY, p_M);
        TE = PE + KE;
        if (time == 0.0) {
            initial_TE = TE;
        }
        // output current state
        write_state(p_X, p_Y, time, KE, PE, TE, outfile);
    }
    outfile.close();

    // display relative change in total energy
    if (energy_diff) {
        final_TE = TE;
        std::cout << fabs(fabs(final_TE - initial_TE)/initial_TE) << std::endl;
    }
}

double time_step(double p_X[], double p_Y[], double p_vX[], double p_vY[],
    double p_aX[], double p_aY[], double p_M[]) {
    // Function to progress simulation forwards by 1 timestep.
    // This requires calculating the acceleration on each body,
    // then iterating positions and velocities accordingly.
    // parameters:
    //     p_X, p_Y: (x,y) values for each body
    //     p_vX, p_vY: (x,y) velocites for each body
    //     p_aX, p_aY: (x,y) accelerations for each body
    //     p_M: mass of each body
    // output:
    //     PE: gravitational potential energy of system
    //     overwrites p_*[] parameters

    // set all accelerations to zero
    for (int i = 0; i < nBodies; i++) {
        p_aX[i] = 0;
        p_aY[i] = 0;
    }

    // recalculate acceleration arrays
    double PE = update_acceleration(p_X, p_Y, p_M, p_aX, p_aY);

    // update velocities
    euler(p_vX, p_aX, h, nBodies);
    euler(p_vY, p_aY, h, nBodies);
    // update positions
    euler(p_X, p_vX, h, nBodies);
    euler(p_Y, p_vY, h, nBodies);

    return PE;
}

```

```
155 double update_acceleration(double p_X[], double p_Y[],
156                           double p_M[], double p_aX[],
157                           double p_aY[]) {
158     // updates accelerations and returns total potential energy
159     // parameters:
160     //     p_X, p_Y: (x,y) values for each body
161     //     p_vX, p_vY: (x,y) velocites for each body
162     //     p_aX, p_aY: (x,y) accelerations for each body
163     //     p_M: mass of each body
164     //     PE: gravitational potential energy of system
165     //     overwrites parameter arrays
166 
167     // updates accelerations and returns total potential energy
168     double PE = 0;
169     // for each pair...
170     double dX, dY, dist, coeff, fX, fY, inv_r_cube;
171     for (int i = 0; i < nBodies; i++) {
172         for (int j = i + 1; j < nBodies; j++) {
173             // get distance between bodies
174             dX = p_X[i] - p_X[j]; dY = p_Y[i] - p_Y[j];
175             dist = sqrt(dX * dX + dY * dY); // cartesian dist
176             // calculate component forces
177             coeff = -G * p_M[i] * p_M[j];
178             inv_r_cube = pow(dist, -3);
179             fX = coeff * dX * inv_r_cube;
180             fY = coeff * dY * inv_r_cube;
181             // append component accelerations to arrays
182             p_aX[i] += fX / p_M[i]; p_aX[j] -= fX / p_M[j];
183             p_aY[i] += fY / p_M[i]; p_aY[j] -= fY / p_M[j];
184             // increase potential energy value
185             PE += coeff / dist;
186         }
187     }
188 
189     return PE;
190 }
```

3.3.2 Parallel

```
1  /*
2  Galaxy Simulation - COSC3500
3  Tom Stephen, 45811449
4  
5  Program to simulate gravity interaction between N bodies.
6  
7  This is the *parallel* implementation.
8  
9  See README.md for instructions on usage.
10 */
11 
12 #include <cstdlib>
13 #include <iomanip>
14 #include <iostream>
15 #include <math.h>
16 #include <cstring>
17 #include <random>
```

```
18 #include <chrono>
19 #include <ctime>
20 #include <fstream>
21
22 #define G 1
23 #define PI 3.1415
24
25 int nBodies = 2;
26 double tf = 1000;
27 double h = 1.0;
28 bool energy_diff = false;
29 bool verbose = false;
30 std::string filename;
31
32 #include "system_init.cpp"
33 #include "debugger_tools.cpp"
34 #include "ode_solve.cpp"
35 #include "helper_functions.cpp"
36
37 void checkError(cudaError_t e);
38 __global__ void update_acceleration(double p_X[], double p_Y[], double p_M[],
       double p_aX[], double p_aY[], int Is[], int Js[], double *PE_device, int
       pairs);
39 void time_step(double p_X[], double p_Y[], double p_vX[], double p_vY[], double
        p_aX[], double p_aY[], double p_M[], double p_X_device[], double
       p_Y_device[], double p_aX_device[], double p_aY_device[], int Is[], int Js
       [], int pairs, double *PE_device);
40 void simulate(double p_X[], double p_Y[], double p_vX[], double p_vY[], double
       p_aX[], double p_aY[], double p_M[], std::string filename, double
       p_X_device[], double p_Y_device[], double p_aX_device[], double p_aY_device
       [], double p_M_device[], int Is[], int Js[], int pairs, double *PE_device);
41
42 int main(int argc, char* argv[]) {
43     // parse command line arguments
44     parse_args(argc, argv);
45
46     // create arrays on host
47     double *planet_X = new double[nBodies];
48     double *planet_Y = new double[nBodies];
49
50     double *planet_vX = new double[nBodies];
51     double *planet_vY = new double[nBodies];
52
53     double *planet_aX = new double[nBodies];
54     double *planet_aY = new double[nBodies];
55
56     double *planet_M = new double[nBodies];
57
58     // ...and on device
59     double *planet_X_device;
60     double *planet_Y_device;
61     double *planet_aX_device;
62     double *planet_aY_device;
63     double *planet_M_device;
64
```

```
65      double *PE_device;
66
67      checkError(cudaMalloc(&planet_X_device, nBodies * sizeof(double)));
68      checkError(cudaMalloc(&planet_Y_device, nBodies * sizeof(double)));
69      checkError(cudaMalloc(&planet_aX_device, nBodies * sizeof(double)));
70      checkError(cudaMalloc(&planet_aY_device, nBodies * sizeof(double)));
71      checkError(cudaMalloc(&planet_M_device, nBodies * sizeof(double)));
72      checkError(cudaMalloc(&PE_device, sizeof(double)));
73
74      // initialise planets
75      init_N_orbitting_planets(planet_X, planet_Y, planet_vX, planet_vY, planet_M
            , nBodies);
76      // copy initialised states to device
77      checkError(cudaMemcpy(planet_X_device, planet_X, nBodies * sizeof(double),
            cudaMemcpyHostToDevice));
78      checkError(cudaMemcpy(planet_Y_device, planet_Y, nBodies * sizeof(double),
            cudaMemcpyHostToDevice));
79      checkError(cudaMemcpy(planet_M_device, planet_M, nBodies * sizeof(double),
            cudaMemcpyHostToDevice));
80
81      // generate (i,j) pairs for body comparisons
82      int pairs = (nBodies * nBodies - nBodies) / 2;
83      int *Is = new int[pairs];
84      int *Js = new int[pairs];
85      int ind = 0;
86      for (int i = 0; i < nBodies; i++) {
87          for (int j = i + 1; j < nBodies; j++) {
88              Is[ind] = i; Js[ind] = j;
89              ind++;
90          }
91      }
92      // ...and store these pairs on the device
93      int *Is_device;
94      int *Js_device;
95      checkError(cudaMalloc(&Is_device, pairs * sizeof(int)));
96      checkError(cudaMalloc(&Js_device, pairs * sizeof(int)));
97      checkError(cudaMemcpy(Is_device, Is, pairs * sizeof(int),
            cudaMemcpyHostToDevice));
98      checkError(cudaMemcpy(Js_device, Js, pairs * sizeof(int),
            cudaMemcpyHostToDevice));
99
100     // simulate system
101     simulate(planet_X, planet_Y, planet_vX, planet_vY, planet_aX, planet_aY,
            planet_M, filename, planet_X_device, planet_Y_device, planet_aX_device,
             planet_aY_device, planet_M_device, Is_device, Js_device, pairs,
            PE_device);
102
103     // free memory
104     delete [] planet_vX;
105     delete [] planet_vY;
106
107     checkError(cudaFree(planet_X_device));
108     checkError(cudaFree(planet_Y_device));
109     checkError(cudaFree(planet_aX_device));
110     checkError(cudaFree(planet_aY_device));
```

```
111     checkError(cudaFree(planet_M_device));
112     checkError(cudaFree(PE_device));
113 
114     exit(0);
115 }
116 
117 
118 void simulate(double p_X[], double p_Y[],
119               double p_vX[], double p_vY[],
120               double p_aX[], double p_aY[],
121               double p_M[],
122               std::string filename,
123               double p_X_device[], double p_Y_device[],
124               double p_aX_device[], double p_aY_device[],
125               double p_M_device[],
126               int Is[], int Js[], int pairs,
127               double *PE_device) {
128     // simulates n-body system with gravity forces
129     // parameters:
130     //     p_X, p_Y: (x,y) values for each body
131     //     p_vX, p_vY: (x,y) velocites for each body
132     //     p_aX, p_aY: (x,y) accelerations for each body
133     //     p_M: mass of each body
134     //     filename: name & location to write the output file
135     //     p_*_device: as above, but stored on the GPU
136     //     Is, Js: form the (i,j) pairs that need to calculated for each body
137     //     PE_device: pointer to potential energy sum on GPU
138     //
139     // No returns, just output file and printing relative change in total
            energy.
140 
141     // open output file
142     std::ofstream outfile;
143     outfile.open(filename);
144     // headerline
145     write_header(outfile);
146 
147     double PE, KE, TE, initial_TE, final_TE;
148     // do simulation until time reaches final time `tf`
149     for (double time = 0.0; time < tf; time += h) {
150         // step forwards by `h`
151         time_step(p_X, p_Y, p_vX, p_vY, p_aX, p_aY,
152                   p_M_device, p_X_device, p_Y_device, p_aX_device, p_aY_device,
153                   Is, Js, pairs, PE_device);
154         // calculate energies
155         KE = kinetic_energy(p_vX, p_vY, p_M);
156         checkError(cudaMemcpy(&PE, PE_device, sizeof(double),
                cudaMemcpyDeviceToHost));
157         TE = PE + KE;
158         if (time == 0.0) {
159             initial_TE = TE;
160         }
161         // output current state
162         write_state(p_X, p_Y, time, KE, PE, TE, outfile);
163     }
```

```
164     outfile.close();
165 
166     // display relative change in total energy
167     if (energy_diff) {
168         final_TE = TE;
169         std::cout << fabs(fabs(final_TE - initial_TE)/initial_TE) << std::endl;
170     }
171 }
172 
173 void time_step(double p_X[], double p_Y[], double p_vX[], double p_vY[], double
      p_aX[], double p_aY[], double p_M_device[], double p_X_device[], double
     p_Y_device[], double p_aX_device[], double p_aY_device[], int Is[], int Js
     [], int pairs, double *PE_device) {
174     // Function to progress simulation forwards by 1 timestep.
175     // This requires calculating the acceleration on each body,
176     // then iterating positions and velocities accordingly.
177     // parameters:
178     //     p_X, p_Y: (x,y) values for each body
179     //     p_vX, p_vY: (x,y) velocites for each body
180     //     p_aX, p_aY: (x,y) accelerations for each body
181     //     p_M: mass of each body
182     //     p_*_device: as above, but stored on the GPU
183     //     Is, Js: form the (i,j) pairs that need to calculated for each body
184     //     PE_device: pointer to potential energy sum on GPU
185     // output:
186     //     no output - overwrites p_*[] parameters
187 
188     // set all accelerations to zero
189     for (int i = 0; i < nBodies; i++) {
190         p_aX[i] = 0;
191         p_aY[i] = 0;
192     }
193     checkError(cudaMemcpy(p_aX_device, p_aX, nBodies * sizeof(double),
          cudaMemcpyHostToDevice));
194     checkError(cudaMemcpy(p_aY_device, p_aY, nBodies * sizeof(double),
          cudaMemcpyHostToDevice));
195 
196     // set potential energy sum to 0
197     double reset_PE = 0;
198     checkError(cudaMemcpy(PE_device, &reset_PE, sizeof(double),
          cudaMemcpyHostToDevice));
199 
200     // recalculate acceleration arrays
201     update_acceleration<<<pairs / 32 + 1, 32>>>(p_X_device, p_Y_device,
          p_M_device, p_aX_device, p_aY_device, Is, Js, PE_device, pairs);
202 
203     // copy accelerations back to host
204     checkError(cudaMemcpy(p_aX, p_aX_device, nBodies * sizeof(double),
          cudaMemcpyDeviceToHost));
205     checkError(cudaMemcpy(p_aY, p_aY_device, nBodies * sizeof(double),
          cudaMemcpyDeviceToHost));
206 
207     // update velocities
208     euler(p_vX, p_aX, h, nBodies);
209     euler(p_vY, p_aY, h, nBodies);
```

```
210     // update positions
211     euler(p_X, p_vX, h, nBodies);
212     euler(p_Y, p_vY, h, nBodies);
213 
214     // copy positions back to device
215     checkError(cudaMemcpy(p_X_device, p_X, nBodies * sizeof(double),
            cudaMemcpyHostToDevice));
216     checkError(cudaMemcpy(p_Y_device, p_Y, nBodies * sizeof(double),
            cudaMemcpyHostToDevice));
217 }
218 
219 __global__ void update_acceleration(double p_X[], double p_Y[],
220                                     double p_M[], double p_aX[],
221                                     double p_aY[], int Is[], int Js[],
222                                     double *PE_device, int pairs) {
223     // updates accelerations and returns total potential energy
224     // parameters:
225     //     p_X, p_Y: (x,y) values for each body
226     //     p_vX, p_vY: (x,y) velocites for each body
227     //     p_aX, p_aY: (x,y) accelerations for each body
228     //     p_M: mass of each body
229     //     Is, Js: form the (i,j) pairs that need to calculated for each body
230     //     pairs: integer number of pairs to calculate accelerations for
231     //     PE_device: pointer to potential energy sum on GPU
232     // output:
233     //     no output - overwrites parameter arrays
234     const int ind = blockIdx.x * blockDim.x + threadIdx.x;
235 
236     // for each pair...
237     if (ind < pairs) {
238         int i = Is[ind]; int j = Js[ind];
239         double dX, dY, dist, coeff, fX, fY, inv_r_cube;
240 
241         // get distance between bodies
242         dX = p_X[i] - p_X[j]; dY = p_Y[i] - p_Y[j];
243         dist = sqrt(dX * dX + dY * dY); // cartesian dist
244         // calculate component forces
245         coeff = -G * p_M[i] * p_M[j];
246         inv_r_cube = pow(dist, -3);
247         fX = coeff * dX * inv_r_cube;
248         fY = coeff * dY * inv_r_cube;
249         // append component accelerations to arrays
250         p_aX[i] += fX / p_M[i]; p_aX[j] -= fX / p_M[j];
251         p_aY[i] += fY / p_M[i]; p_aY[j] -= fY / p_M[j];
252         // increase potential energy value
253         double PE = coeff / dist;
254 
255         atomicAdd(PE_device, PE);
256     }
257 }
258 
259 void checkError(cudaError_t e)
260 {
261    if (e != cudaSuccess)
262    {
```

```
263       std::cerr << "CUDA␣error:␣" << int(e) << "␣:␣" << cudaGetErrorString(e)
              << '\n';
264       abort();
265     }
266 }
```

3.3.3 Helper functions

General

```
1  void parse_args(int argc, char* argv[]) {
2     for (int i = 1; i < argc; ++i) {
3         if(!strcmp(argv[i], "-h") || !strcmp(argv[i], "--help") ){
4           std::cout << "Galaxy␣Simulation␣-␣for␣COSC3500,␣by␣Tom␣Stephen" << std
                ::endl
5                     << std::endl
6                     << "Usage:␣./main␣[arguments]␣␣␣␣␣␣␣␣␣␣␣␣␣␣run␣some␣simulation
                          " << std::endl
7                     << "\tor:␣vim␣[arguments]␣>␣[filename]␣␣␣␣output␣results␣to␣a
                          ␣data␣file" << std::endl
8                     << std::endl
9                     << "Arguments:" << std::endl
10                    << "\t--nBodies␣␣␣␣␣␣␣␣␣Set␣number␣of␣bodies␣to␣simulate" <<
                          std::endl
11                    << "\t--finalTime␣␣␣␣␣␣␣Set␣time␣to␣simulate␣to" << std::endl
12                    << "\t--timeStep␣␣␣␣␣␣␣␣Set␣time␣step␣(`h`)␣for␣integration␣
                          method" << std::endl
13                    << "\t--energyDiff␣␣␣␣␣␣Show␣change␣in␣total␣energy␣across␣
                          simulation" << std::endl
14                    << "\t--time␣␣␣␣␣␣␣␣␣␣␣␣Provide␣the␣current␣time␣to␣the␣
                          program" << std::endl;
15          exit(0);
16        } else if (!strcmp(argv[i], "--nBodies")) {
17            nBodies = atoi(argv[++i]);
18        } else if (!strcmp(argv[i], "--finalTime")) {
19            tf = atof(argv[++i]);
20        } else if (!strcmp(argv[i], "--timeStep")) {
21            h = atof(argv[++i]);
22        } else if (!strcmp(argv[i], "--energyDiff")) {
23            energy_diff = true;
24        } else if (!strcmp(argv[i], "--filename")) {
25            filename = argv[++i];
26        }
27    }
28 }
29
30 void write_state(double p_X[], double p_Y[], double t, double KE, double PE,
       double TE, std::ofstream& output) {
31    // function to write current state to file.
32    // writes current time, component and total energies,
33    // and the (x,y) position of each body.
34    output << t << "␣" << KE << "␣" << PE << "␣" << TE;
35    for (int i = 0; i < nBodies; i++) {
36        output << "␣" << p_X[i] << "␣" << p_Y[i];
37    }
38    output << "\n";
```

```
}

void write_header(std::ofstream& output) {
    // function to write the header line out the output file
    output << "t␣KE␣PE␣TE";
    for (int i = 0; i < nBodies; i++) {
        output << "␣p" << i << "x␣p" << i << "y";
    }
    output << "\n";
}

double kinetic_energy(double p_vX[], double p_vY[], double p_M[]) {
    // function to calculate the total kinetic energy of the system.
    // parameters:
    //     p_vX: x velocities of each body
    //     p_vX: y velocities of each body
    //     p_M: mass of each body
    double KE = 0; // initialise sum
    double v_sq; // declare v^2
    for (int i = 0; i < nBodies; i++) {
        // calculate v^2
        v_sq = pow(p_vX[i],2) + pow(p_vY[i],2);
        // increase sum energy
        KE += 0.5 * p_M[i] * v_sq;
    }
    return KE;
}
```

Debugger

```
#include <cstdlib>
#include <iostream>

void disp_planets(double planet_X[], double planet_Y[], double planet_vX[],
    double planet_vY[], double planet_M[], int nBodies) {
    // usage: disp_planets(planet_X, planet_Y, planet_vX, planet_vY, planet_M,
        nBodies);
    for (int i = 0; i < nBodies; i++) {
        std::cout << "Planet␣" << i << std::endl;
        std::cout << "(x,y)␣=␣(" << planet_X[i] << "," << planet_Y[i] << ")" <<
            std::endl;
        std::cout << "(vx,vy)␣=␣(" << planet_vX[i] << "," << planet_vY[i] << ")"
             << std::endl;
        std::cout << "m␣=␣" << planet_M[i] << std::endl;

        std::cout << std::endl;
    }
}
```

Initialisation

```
#include <cstdlib>
#include <math.h>
#include <random>

#define PI 3.1415
```

```
#define G 1

void init_N_orbitting_planets(double planet_X[], double planet_Y[], double
    planet_vX[], double planet_vY[], double planet_M[], int nBodies) {
    // massive star at origin (index 0). n-1 planets orbit it.
    // Orbital velocity calculated for each, moving in clockwise direction
    // system parameters
    double mass = 1;
    double dist_max = 5000;
    double dist_min = 500;
    double star_mass = 1000;

    // set star values
    planet_X[0] = 0; planet_Y[0] = 0; planet_vX[0] = 0; planet_vY[0] = 0;
    planet_M[0] = star_mass;

    // setup random number generator
    int seed = 69;
    std::mt19937 generator;
    std::mt19937 new_gen;
    generator.seed(seed);
    new_gen.seed(seed + 1);
    std::uniform_real_distribution<double> distribution(0.0,1.0);
    std::uniform_real_distribution<double> new_dist(0.25,0.25);

    // for each remaining planet...
    double dist, angle, orb_speed;
    for (int i = 1; i < nBodies; i++) {
        // generate position + velocity for new body
        planet_M[i] = mass;

        // get random height in [0, dist_bound]
        dist = dist_min + (dist_max - dist_min) * distribution(generator);

        // get random angle in [0, 2pi]
        angle = 2 * PI * distribution(generator);

        // get and set x and y to suit
        planet_X[i] = dist * cos(angle);
        planet_Y[i] = dist * sin(angle);

        // calculate orbital velocity
        // orb_speed = sqrt(G * star_mass / dist);
        // orb_speed = sqrt(G * star_mass / dist) * 0.5;
        orb_speed = sqrt(G * star_mass / dist) * new_dist(new_gen);
        // orb_speed = 0;

        // get and set vx and vy to suit
        planet_vX[i] = orb_speed * cos(angle + PI / 2);
        planet_vY[i] = orb_speed * sin(angle + PI / 2);
    }
}
```

Iteration

```
void euler(double x[], double dX[], double dt, int n) {
```

```
2     for (int i = 0; i < n; i++) {
3         x[i] += dt * dX[i];
4     }
5 }
```