Final Report: N-Body Simulation

Sigurjon Agustsson, Saga David, Constance Ferragu

June 17, 2022

Link to the project: `https://github.com/constanceferragu/concurrent_n_body`

1 Introduction

In this project, we aim to do a working simulation of N bodies, first by implementing it sequentially and then we parallelise it. Lastly we implemented a more advanced simulation algorithm, the *Barnes-Hut* algorithm and discussed ways of parallelizing it.

2 The Body class

The first step of the project was to create a class `Body` to represent the bodies. It has the attributes x and y (coordinates), mass, velocity and a name. It has a print function to display it's attributes in the console and an "apply force" function which given force in x and y directions, updates the body's position and velocity

3 Sequential simulation

The first algorithm to implement is the one described in the project report. Our first setup was the Earth-Moon system. We decided to use the exact values of the Earth and Moon for mass, position and velocity, because it made it easy to debug. Knowing that the moon makes one orbit around the Earth in 4 weeks allowed us to easily see if our code was working or not. The timestep we use is 1 second, x and y coordinates are given in meters and velocity in meters per second. Then making the Moon do one orbit around the Earth was as simple as running the algorithm for 2,419,200 seconds which is the number of seconds in 4 weeks. Of course, since the Moon's orbit is more complex than a circle we knew that our simulation would not be completely accurate, but it would be interesting to see how far off it was.

The idea of the algorithm is the following: At each timestep we need to update the position of the bodies. To do so we have two for loops. First we have a double for loop over all the bodies to compute the forces between each body and all the other bodies. The function `get_force_components` computes the x and y components of the force between bodies B_i and B_i. We decided to split the force into x and y components immediately because those are the values we are interested in when it comes to update the position of the bodies later. Importantly, since the forces between bodies B_i and B_i are equal but opposite, we do not have to complete the for loop over all of them, it suffices to iterate over all the bodies B_i and then over the bodies B_i such that $j \geq i$ and then fill in two values of the force matrix at the same time. So, the upper triangle of the force matrices is the same as the lower triangle, but with the opposite sign. This way we do not have to compute the forces between B_i and B_i twice. The form of the force

matrix is the following:

$$F_{4\times4} = \begin{pmatrix} 0 & F_{12} & F_{13} & F_{14} \\ F_{21} & 0 & F_{23} & F_{24} \\ F_{31} & F_{32} & 0 & F_{34} \\ F_{41} & F_{42} & F_{43} & 0 \end{pmatrix} = \begin{pmatrix} 0 & F_{12} & F_{13} & F_{14} \\ -F_{12} & 0 & F_{23} & F_{24} \\ -F_{13} & -F_{23} & 0 & F_{34} \\ -F_{14} & -F_{24} & -F_{34} & 0 \end{pmatrix}$$

At this point we have successfully filled the force matrices for x and y forces between all the bodies, and then it remains to update the positions of the bodies. Then we iterate again over all the bodies, and for each body we add up the forces applied on it by all the other bodies. Once we have the total forces applied on the body B_i in x and y directions respectively, we can update the position. To do that, call the function `apply_force` of the body class which updates the x and y position given forces in each direction and some timestep dt.

What was not clear to us in the beginning, but seems obvious now, is that there is no need to have these two for loops separated. There is no reason to compute the forces first and then doing another iteration for applying them, but in the early stages of the project, some physics misunderstanding led us to believe that we could not update the position of body B_i correctly unless by using the total force applied on B_i by all the other bodies, but this is false: updating the position incrementally by the force applied by a single body at a time would be the same as updating it by the total force of all the bodies.

In any case, the loss in performance because of this is not significant since executing a for loop twice has asymptotically the same cost as doing it once.

4 Parallelisation

When parallelising, we kept the two codes as similar as possible. The idea is to parallelise the step where we compute the force matrices. The key observation is that while we are computing the force between bodies say B_1 and B_2, the bodies B_3 and B_4 are sitting idly by, not used at all. We seek to exploit this fact by computing the forces between B_1 and B_2, and B_3 and B_4 in parallel and saving them in the force matrix.

Let's consider F to be a force matrix, and let us divide it into four smaller matrices. Then we can look at it as follows:

$$F = \begin{pmatrix} A & C \\ -C & B \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & C_{11} & C_{12} \\ A_{21} & A_{22} & C_{21} & C_{22} \\ -C_{11} & -C_{12} & B_{11} & B_{12} \\ -C_{21} & -C_{22} & B_{21} & B_{22} \end{pmatrix}$$

We want to let each part A, B and C of the matrix be computed by a separate thread. There is a catch however: we cannot compute parts share any bodies at the same time. For instance, both part A and part C use the bodies B_1 and B_2 and both part B and part C use the bodies B_4 and B_4. Thus we can only compute simultaneously the forces in parts A and B, since they don't share any bodies. The advantage of this approach is that we don't need to use any locks since at no time are there threads updating or accessing the same value at the same time.

The rest of the implementation is the same as the one for sequential simulation.

5 Performance

When we compare the two algorithms, we run the algorithms with 10, 20, and 40 randomly generated bodies respectively and simulate one day. We find that the sequential algorithm performs the best.This is not unexpected, since on our machines with the relatively light tests,

the cost of launching new threads is too high for it to increase performance. However, we do observe some interesting trends. The Parallel algorithm with 4 threads is almost constant in time, with little increase when the input gets larger. Also, when using only 1 thread in the parallel algorithm, it is still much slower than the original sequential algorithm. This is likely due to the fact that we do not do the computations in the main thread of execution, but launch one extra thread, so effectively it is like having 2 threads which will inevitably slow the code down. As we run with more and more threads, this fact does not affect the performance.

Table 1: Comparison of sequential and parallel algorithms.

N bodies	**Sequential**	**Parallel** 1 thread	**Parallel** 2 threads	**Parallel** 4 threads
10	0.53	12.27	28.14	88.56
40	6.88	29.99	37.72	76.69
80	26.9	73.61	64.11	90.31

6 Visualization

When implementing the algorithms, it was important to be able to test their correctness by being able to visualize the bodies created and how they moved with time. We followed three different approaches that we will detail in the following.

6.1 Printing

When we were still working on the thread free implementation of the main algorithm, we wrote the function `visualise_bodies()` which prints a plane representing the positions of the bodies at each time. Figure 1 shows what the function gave for the last positions of random bodies. In order to test, we print the positions at each time frame and check that they are moving through time. The values we take for the bodies' position, mass and velocities are realistic, e.g. when we test on the Earth and the Moon the values we choose are their actual values. In order for us to visualise the bodies on a (-1,1)-plane, we normalize the values and print the newly computed positions.

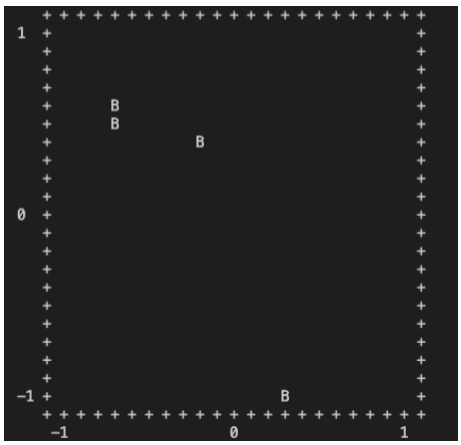


Figure 1: Resulting graph that shows the final positions of bodies after a simulation

6.2 Magick++

Once we saw that the algorithm was working well with the `visualise_bodies()` function, we moved on to a visualization as it was asked in the project description by using Magick++. The goal was to create a gif animation of the simulation by creating a canvas and drawing on it at the various bodies' positions computed. We encountered some issues with this library on Windows but could download it and run some examples on Mac. Unfortunately, while some simple examples could be ran smoothly, an important delegate called 'XC' was missing and after hours of deinstalling, reinstalling and searching on the internet, we decided to move on to another library which you recommended to us called SFML, which we will describe in the next section.

6.3 SFML

SFML (simple and fast multimedia library) provides a lot of functions that permit us to create a window to represent the canvas where we can draw the bodies.

As explained in previous sections, the code contains the N-body class and a main loop, which uses functions to compute the forces, apply them and compute new positions. A challenge was to adapt to that code in order to create the visualization. To explain these challenges note that:

- There was an existing while loop going through and two for loops going through the bodies with the code from other sections
- The visualization code requires a loop on it's own, as we want to create a window that runs through time.
- We do not want to print all of the positions over the time considered, as it could run for a while and would show too many steps that are not important for the project.

To work on all of these challenges, we came up with a structure for storing the positions in a vector and be able to use that vector in the visualization loop.

We have (see figure 2):

- A vector that contains vectors of pairs: `vec_of_times`.
- A vector of pairs which contains the positions of all bodies at time t: `pos_at_time`
- Pairs that represent the coordinates (x, y) of a body.

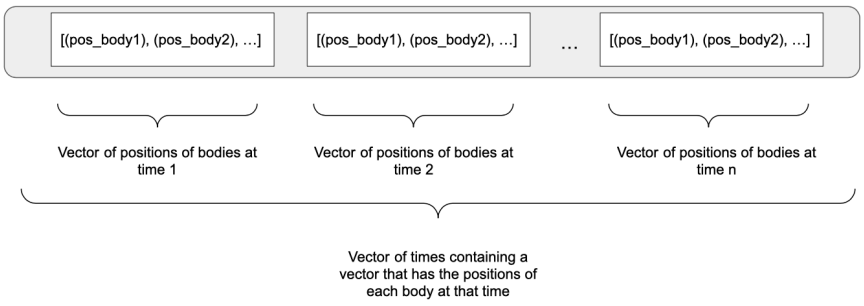


Figure 2: A visual representation of the vectors used

At each time, we create a vector `pos_at_time`, which we fill by running through each body and computing their new position. Note that we only add positions that we will want to draw on the canvas.

At the end of the while loop that goes through time, we thus have a full vector `vec_of_times`, which we use in the visualization loop to draw each point on the window.

In case there are some problems running the library on your computer, we provided recordings of some examples of renderings which show the bodies movement with respect to each other through time.

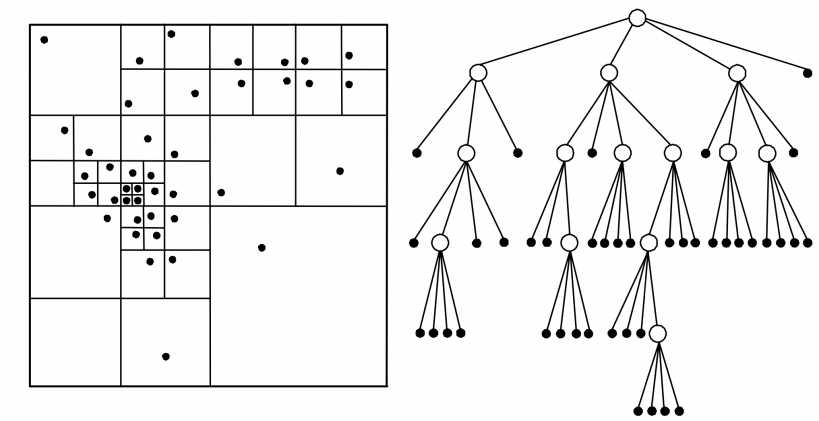
On a side note, SFML was a lot easier to use than Magick++ and many tutorials and youtube videos were available. If you think it produces good enough results, it would be nice for coming years to suggest this library in the project description (instead of Magick++ or in addition to it).

7 Barnes-Hut Algorithm

The idea behind the Barnes-Hut Algorithm is to group bodies that are "close together" (this is decided by a threshold value) and approximate this as one body. This singular body (which encapsulates multiple bodies) has a total mass equal to the sum of these bodies, and its position is the weighted average position of these bodies.

For the implementation of this Algorithm, we started by reading the article provided in the project description, and then followed the friendlier description for the implementation (given in the project outline).

The first step of the Algorithm is to build the Barnes-Hut tree to represent the spatial domain of our bodies. The main idea is to recursively cut the spatial domain into 4 sub-quadrants, until one or zero bodies are in each sub-domain.



We first implemented a node of the Barnes Hut tree. There are three different kind of nodes which we all represent in the same class:

1. An external node with a body. This node represents a final sub-domain (does not have any sub-quadrants) containing a body.
2. An external node without a body. This node represents a final sub-domain not containing a body.
3. An internal node. This node does not contain a body itself, however the spatial domain it represents contains more than 1 body. This node has 4 children representing each quadrant, and will hold the center of mass coordinates and total mass of the bodies in its quadrants.

We differentiate between the nodes with the `external` and `contains_body` attributes.

Then, we implemented the Barnes Hut Tree from the Barnes Hut Node. The main work to be done was to implement the add_bodies function. We followed the implementation detailed here.

7.1 How we would have parallelized Barnes-Hut

As a final step of our project, we discussed the possible ways of parallelizing the Barnes Hut Algorithm.

In a first step we thought of parallelizing the Barnes-Hut Tree. We would have done this as we did with the BST in our course. In this case we would have had to switch to a non-recursive algorithm for the add_bodies function. We would need extra functions such as contains, and a function that finds the position of insertion for each body. In this case we would also need to introduce Atomic variables or Locks. One big question we had was: Is it possible to parallelize the recursive algorithm we are currently using ? An iterative algorithm would increase the number of traverses of the tree, thus we questioned how big the trade-off would be between keeping a recursive algorithm or having a parallelized iterative algorithm.

In a second way, we thought of parallelizing the algorithm that updates the body positions iteratively. In this case, since the branches of the tree corresponds to grouping bodies that are close together, we would distribute branches onto the threads. We noted that this would save a lot of computation time.

Our final conclusions were that parallelizing the construction of the tree (add_bodies function) does not seem highly necessary in this setting, however, paralellizing the computation of the forces and the relocation of the bodies would save a lot of computation time. Hence, the next step of our project would be to parallelize the second part of the Barnes Hut Algorithm.