Numerical Algorithms

Space Bodies Summative

Compiling the code

Serial execution locally:
g++-4.9 -O3 spacebodies.cpp -std=c++11 -o spacebodies

Serial execution on hamilton:
icc -O3 -DHAMTIME spacebodies.cpp -std=c++11 -o spacebodiesserial

Parallel execution locally:
g++-4.9 -O3 -fopenmp -DPAR spacebodies.cpp -std=c++11 -o spacebodies

Parallel execution on hamilton:
icc -O3 -DHAMTIME -DSIMTIME -openmp -DPAR spacebodies.cpp -std=c++11 -o spacebodiesparallel

Flags:
-DPAR - Signifies that the code will be run in parallel (for clock accuracy)
-DSIMTIME - Time simulation - turns off outputs
-DFOOBUG - Debug mode

Running the compiled executable

The code takes a variable number of arguments:

spacebodies [double <collision distance>
[int <time steps>
[double <time step size>
[int <plot every k steps>
[int <number of bodies>
[int <maximum mass>
[double <density>
[int <maximum velocity>]]]]]]]]

The default values are equivalent to:
spacebodies 0.1 1000000 0.001 100 50 13 2 5

The submission also includes a make file with various execution modes and their respective flags predefined (normal, timed and debug for parallel and non-parallel implementations).

The role of these arguments in the program's execution will be further explained later.

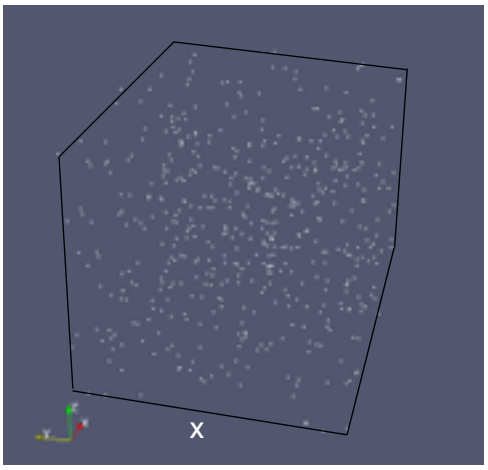
Extending the code

The code included is programmed to create a user defined number of bodies (n) with randomised parameters based on user defined arguments.

The sample code has been adapted so that each body in dynamic, with the code resolving forces and movement between all bodies. Collision of bodies is resolved through a variable collision distance threshold.

If two bodies are calculated to be within collision distance of one another, a collision is resolved. The first body moves to halfway between the pair's original position and the mass, velocities and forces are resolved appropriately, and stored within this body's struct. The other is flagged as having collided through a pointer to it's pair, and is subsequently ignored from calculations. This means that after each collisions, accumulatively fewer calculations are made each update, means execution speeds up.

The mass of each particle is randomised between 0 and the maximum mass, and similarly the maximum velocity, and the direction of that velocity, is defined by a maximum velocity argument.



Spawned particles are given randomised positions within a defined range. As can be seen here, particles are spawned randomly within a cube volume defined by the density value and the number of bodies. This density argument allows consistent body interactions as the number of bodies spawned scales (ensuring some degree of consistency of collision frequency as n scales).

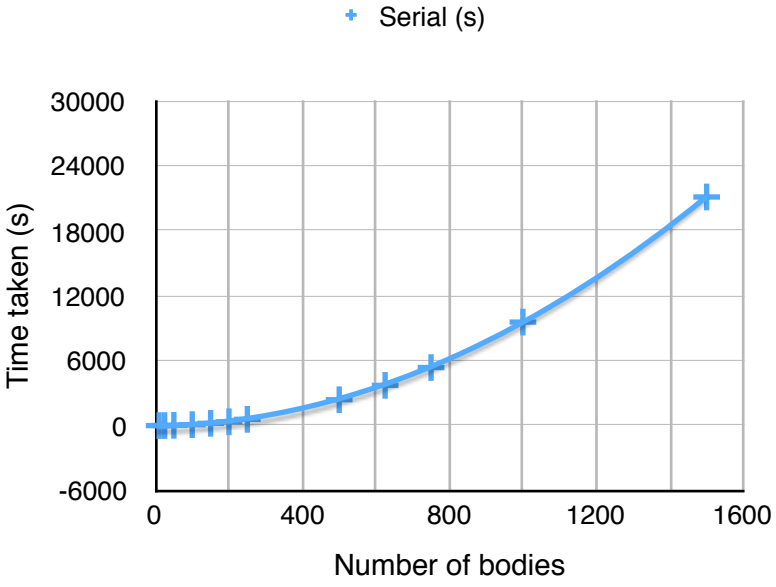
In the image to the left, the magnitude of x represents the maximum dimensional variance of initial body positions. This value is calculated by the formulae:

$$x = no\ of\ bodies\ /\ \sqrt[3]{density}$$

Runtime

Logically, as more bodies are introduced to a simulation, the run time increases. There are two nested for loops in each call of the update method, with each at worst case (collisions decrease the loop size) O(n^2). The run time therefore increases polynomially.

This is evident in the graph to the right, which plots taken taken against the number of bodies (n), run on Hamilton. Although the algorithm generates outputs rapidly with small numbers of bodies, execution becomes prohibitively slow, taking hours by the 1000-body mark.



Time stepping

The time step is a variable that varies the quantisation of the simulation's calculations. Large time steps mean few calculations to simulate the same period of time, but each calculation is less accurate, with errors compounding over time.

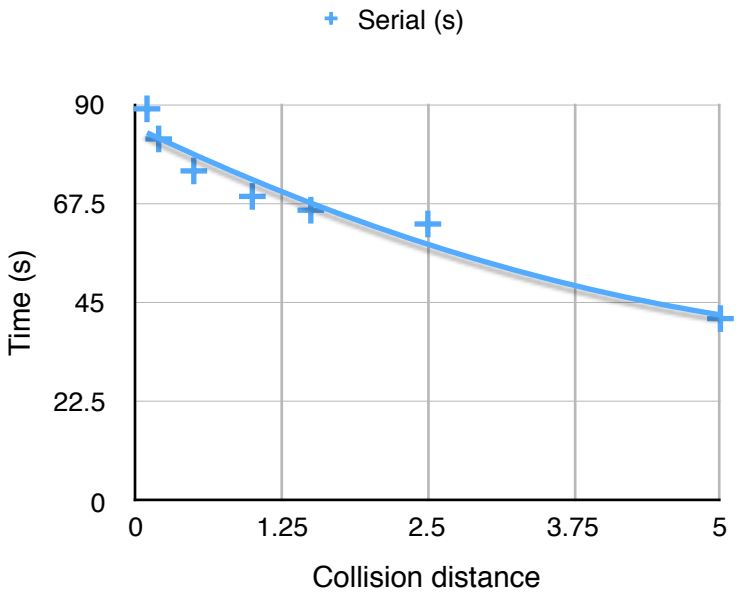
When the time step is particularly large, bodies begin to large jumps and pass one another without colliding, or passing near one another without their interacting forces being appropriately resolved. The collision distance can be increased to compensate somewhat for the fidelity of the time-step but this is far from ideal.

Small time steps mean high fidelity and high accuracy of calculation, but the simulation takes longer to run. As time steps increase, accuracy increases with diminishing returns.

Floating point calculations introduce calculation errors, and therefore increasing the time-step means a potential increase the cumulative effect of this error on the results. However, this error is so small it is insignificant in this context.

Performance of algorithm over time

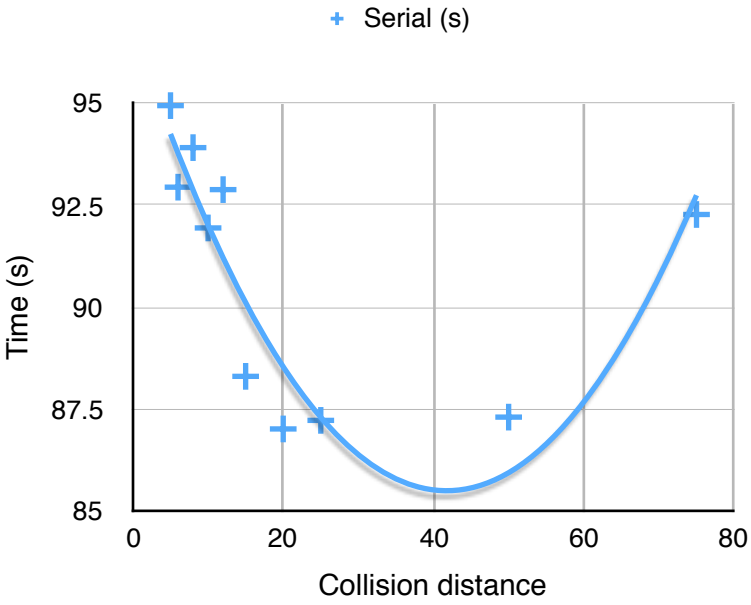
A key change in performance overtime lies in the number of collisions that occur. As collisions occur the number of bodies (n) the calculation runs on decreases, and due to the polynomial nature of the run time a small change in n can lead to significant speed up. This can be seen in the graph to the right, where an increase in collision distance leading to and increase in collisions in turn leads to a large speedup. This effect is less noticeable in parallelised code.



Time stepping, as mentioned earlier, is key to calculation accuracy. When body masses get extremely large, the increased forces lead to problems. When seeding bodies with high mass, initially time speeds up as more collisions occur. However, they then begin to create slow the execution down (see right), as the large masses create higher velocities resulting in a relative unsuitability of time step. The high velocities mean the bodies will make large jumps missing collisions and introducing errors as previously explored.

This process can also occur as the program progresses and collisions occur. This means that the algorithm can perform worse over time as the time-step's suitability worsens.

Relating time-step to an individual body, related to its mass and velocity, would help to maintain consistent simulation performance over time.

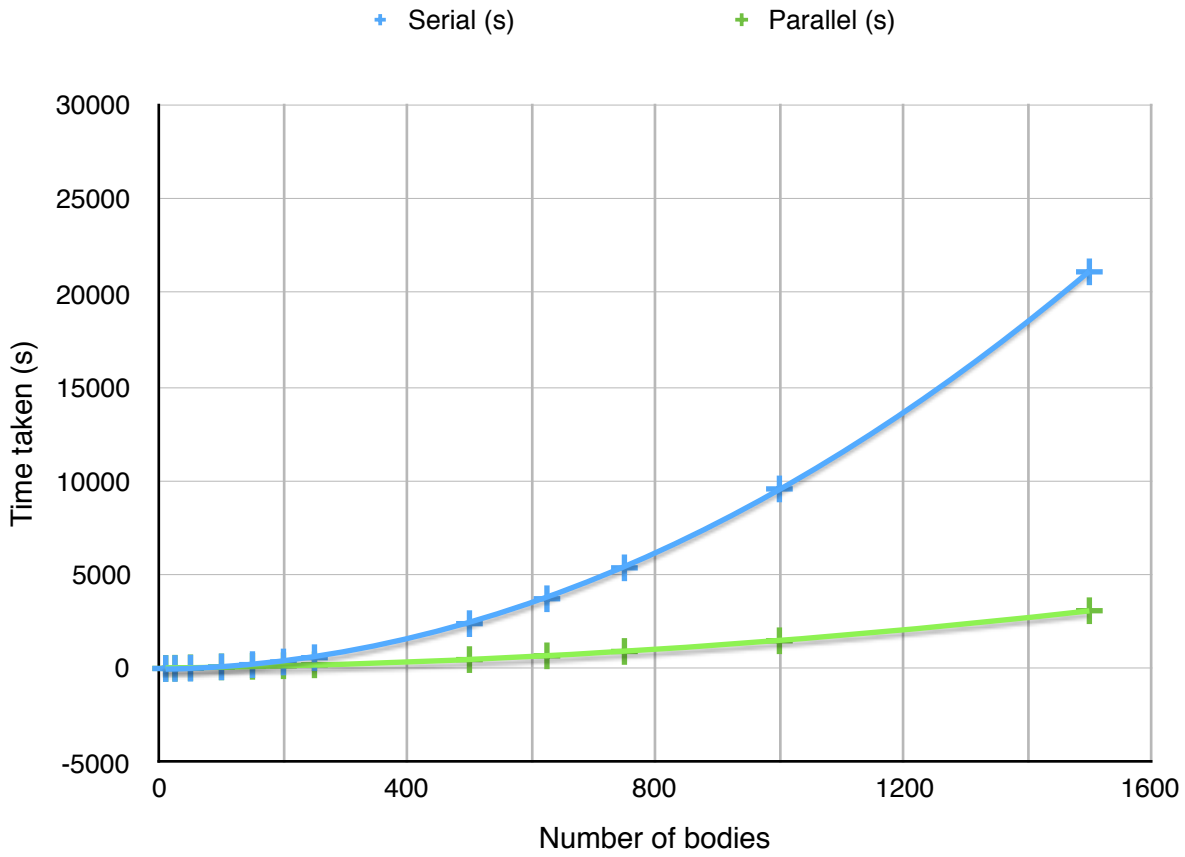


Step 2: Performance Engineering

I decided to improve the code's execution through parallelisation. Shared memory parallelisation involves dividing a problem up into component parts, solving those components separately, and bring the results back together before continuing. This allows the utilisation of multiple processors, and with the correct implementation of parallelisation, huge time savings can be made. OpenMP is an easy to use pre-compilation API which I opted to use to implement parallelisation.

This parallelisation is achieved by preceding a suitable loop with *#pragma omg parallel for,* which tell the compiler to attempt to split the loop instances as separate tasks which can be processed in parallel. Suitability relies in the non-interdependency of the calculations. Body force calculations can be made in parallel, and in a separate instance these forces can be resolved. However, these two processes must be divided. Collision resolution cannot be parallelised.

Below is a graph plotting a comparison of serial and parallel run times, run on Hamilton with 16 cores and the input arguments: *0.1 1000000 0.001 100 n 13 2 5.*



The dramatic time savings conferred by parallelisation are clear. Though both grow polynomially, the run time for the parallel code does so by a significantly smaller magnitude. These huge time savings mean that the code will scale well and run in a very reasonable time frame even with thousands of bodies, where the serialised code takes a prohibitively long time.

The clear divergence is because parallelisation reduces run time by a magnitude of the cores available to the code's execution - in this case 16. Theoretically, if the number of cores used scaled with n rather than remaining fixed linear run time growth could be achieved, but in reality the number of cores to us is very limited.

It is important to note that when dealing with trivially small numbers of bodies, the overhead associated with preparing tasks for parallelisation results in longer execution times, but beyond ~100 bodies the parallel code out performs the serialised code.

Because of a combination of side effects and race conditions (e.g. creating arbitrary bodies and adding them to the vector via push_back) the opportunities for parallelisation are somewhat limited. Having said this, some situations that would initially appear to lend themselves to parallelisation (such as the aforementioned creation of bodies) are relatively trivial to compute in a linear fashion and as a result are not worth worrying about.

Parallelisation in this context is particularly limited by the n-body problem (https://en.wikipedia.org/wiki/N-body_problem), and hence only certain parts of the code can be run in parallel.

Example outputs

1000 bodies
https://www.youtube.com/watch?v=XWX4irSFM8Y

4 bodies
https://www.youtube.com/watch?v=MmfjGQsJF7U

Many bodies, low density, high mass
https://www.youtube.com/watch?v=d3xSrnBW-Mg

High velocity - time step too high
https://www.youtube.com/watch?v=mju-LSjhgFo&feature=youtu.be

Low collision distance