Advanced Computational Physics: Direct N-body Simulation

Paul Hayes
Level 7 Computational Physics, School of Physics, University of Bristol.
(Dated: March 1, 2019)

I. INTRODUCTION

Direct N-body simulations may be used to investigate the dynamical evolution of large scale systems under the influence of physical forces, usually gravity. Examples include the time evolution of a number of planetary orbits or, on a greater scale, that of a galaxy[1].

Applying Newtons Law of Gravity to an N-body system of particles gives the total force on particle i of mass m_i, due to the interaction of $N-1$ other particles, as:

$$\mathbf{F}_i = m_i\ddot{\mathbf{r}}_i = -G\sum_{j=1;j\neq i}^{N}\frac{m_j(\mathbf{r}_i-\mathbf{r}_j)}{\mid \mathbf{r}_i-\mathbf{r}_j \mid^3}. \tag{1}$$

Here G is the gravitational constant and $\mathbf{r}_i$ is the position of particle i. This problem is a set of non-linear second order ordinary differential equations. Given the initial conditions at time t_o for mass, m_i, position, $\mathbf{r}_i$ and velocity, $\mathbf{v}_i$ on each particle, there exists an exact solution only up to a total of two interacting bodies, where larger N require numerical integration methods [2].

Computational Method

One example of an integrator that can be used to solve EQ.1 directly is the leapfrog-verlet method, where velocities one-half timestep out of synchrony with positions are used to update a particle's position [3]. First, the positions, $\mathbf{r}_i^{t+\frac{1}{2}}$, at one half of the timestep, Δt, are evaluated from the velocities $\mathbf{v}_i^t$:

$$\mathbf{r}_i^{t+\frac{1}{2}} = \mathbf{r}_i^t + \mathbf{v}_i^t.\frac{\Delta t}{2}, \tag{2}$$

and then the accelerations, $\mathbf{a}_i^{t+\frac{1}{2}}$, using these positions updated at half a timestep,

$$\mathbf{a}_i^{t+\frac{1}{2}} = G\sum_{j=1;j\neq i}^{N}\frac{m_j(\mathbf{r}_i^{t+\frac{1}{2}}-\mathbf{r}_j^{t+\frac{1}{2}})}{\mid \mathbf{r}_i^{t+\frac{1}{2}}-\mathbf{r}_j^{t+\frac{1}{2}} \mid^3}. \tag{3}$$

Then the ith particle is evolving according to:

$$\mathbf{v}_i^{t+1} = \mathbf{v}_i^t + \mathbf{a}^{t+\frac{1}{2}}.\Delta t, \tag{4}$$

$$\mathbf{r}_i^{t+1} = \mathbf{r}_i^t + (\mathbf{v}_i^t + \mathbf{v}_i^{t+1}).\frac{\Delta t}{2}. \tag{5}$$

In determining the accelerations of each particle in an N-body system, the interactions due to all other particles must be calculated. From EQ.3, this requires $\frac{1}{2}N(N-1)$ calculations per timestep. With a computational complexity of $\mathcal{O}(N^2)$, this is the most computationally expensive part. However, the calculation of each particle's acceleration is independent of any of the calculations for other particles. This therefore makes it possible to parallise, where an increase in compute times at higher number of bodes should be observed as the load of this calculation can be distributed between a number of CPUs.

The Solar system

The N-body simulation used in this case is the solar system, comprised of a series of planets and the Asteroid Belt. This is an ideal model to use for the evaluation of different parallisation methods, as the solar system has a large number of asteroids orbiting its belt for the required high number of bodies to observe parallisaiton speedups, with the initial conditions of these bodies known accurately, such that a check on the simulation's accuracy is straightforward. Using known initial positions of the planets and asteroids, these are used to give values of the initial velocities, $\mathbf{v}_i^o$, assuming keplarian orbits, by balancing the centripetal force with the Gravitational force to give:

$$\mathbf{v}_i = \sqrt{\frac{GM}{r_i}}. \tag{6}$$

Here, M is the solar mass and r_i is the distance between the two bodies. In order to evaluate the accuracy of this simulation, a conservation check on a physical quantity should be imposed. In such an N-body simulation of the solar system, a number of physical quantities should be conserved. Out of these, the most straightforward and accessible to calculate are total energy, total momentum and total angular momentum of the system. However, as the total energy calculation is a difference between two large numbers, the total kinetic energy and potential energy, experience has rendered it the most sensitive method for evaluating accuracy [2]. It is therefore used as the conservation check.

The total energy of the system is given by,

$$E = \frac{1}{2}\sum_{i=1}^{N} m_i\mathbf{v}_i^2 - \sum_{i=1}^{N}\sum_{j>i}^{N}\frac{Gm_im_j}{\mid \mathbf{r}_i-\mathbf{r}_j \mid}. \tag{7}$$

Such an evaluation of the total energy of the system should show conservation over a large enough time period.

Initial results

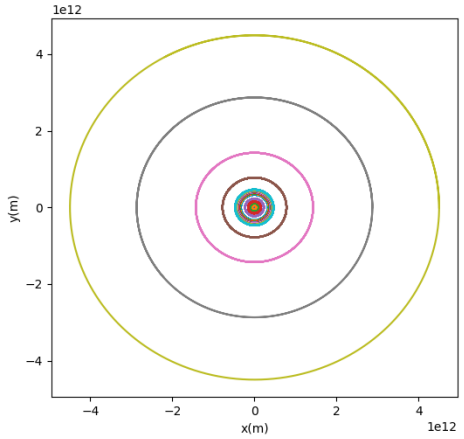

FIG. 1. Plot in the x-y plane of the orbital motion of the planets in the solar system and asteroid belt with 1000 asteroids.

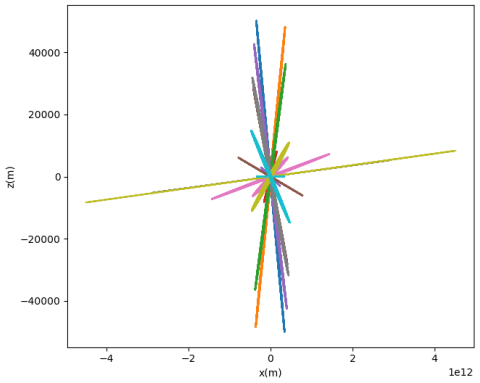

FIG. 2. Plot in the x-z plane of the orbital motion of the planets in the solar system and asteroid belt with 1000 asteroids.

A plot of the planets and asteroids orbiting about a centre of mass, the sun, is seen in FIG.1 and FIG.2, with a plot in the z direction used to highlight the use of three dimensions in this simulation. The orbits plotted in FIG.1 are circular, with no observable deviation from their orbits. This confirms the accuracy of this simulation and shows the expected orbits for the initial conditions used. Due to the initial conditions of the bodies, with the sun stationary at the origin, there will be a net momentum initially. Therefore, it was necessary to give the sun a net velocity such that is opposes the momentum of the other bodies. Without this calculation, there is a net drift of the sun in a direction, as this is not present in FIG.1, this shows this method was sufficient to give the desired accurate orbits.

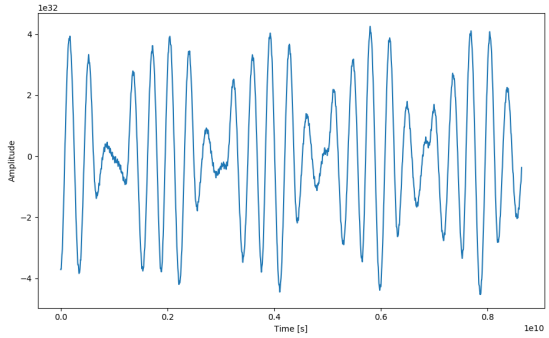

FIG. 3. Plot of the oscillation of the total energy of the system as a function of time, offset such that the average value is 0, for the FFT plot.

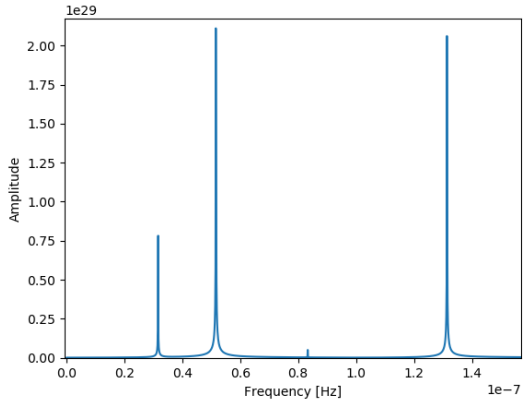

FIG. 4. FFT of the oscillation of the total energy of the system, showing the frequency spectrum, with observed peaks at orbital frequency of bodies.

FIG.3 is a plot of the total energy of a system of 100 bodies, including all the planets and a number of asteroids, orbiting the sun, with the energy offset such that the average value is at zero, in order that an FFT can be taken. Careful inspection of the plot reveals a period of oscillation of the peaks. Taking the FFT of this plot gives the spectrum seen in FIG.4. Here, the sharp peaks give frequencies of $3.17 \times 10^{-8} s^{-1}$, $5.15 \times 10^{-8} s^{-1}$ and $1.31 \times 10^{-7} s^{-1}$, which are in fact the inverse of the orbital periods of the Earth, Venus and Mercury, respectively, of 365.25, 224.7 and 88 days. This confirms that the simulation is working effectively and accurately.

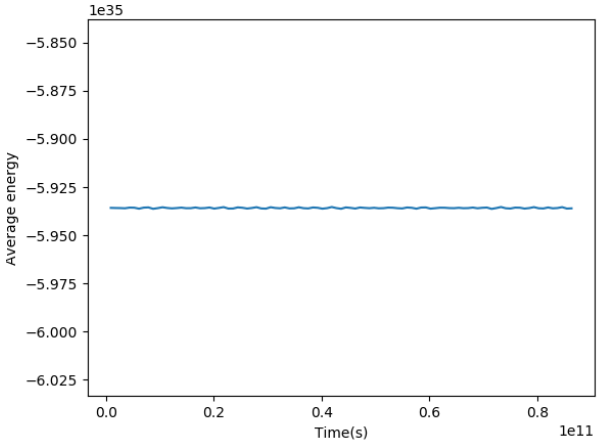

FIG. 5. Plot of the oscillation of the average energy of the system over a period of time.

Seen in FIG.5 is the average energy of the system over a period of time. It is clear that any deviation from a constant energy is negligible. In fact, there should be some systematic error associated with simulating such a system, where the average energy will overall increase [2]. However, it is clear in this plot that the simulation has not run for a sufficient period to reveal such an error. Due to lack of time, this increase in total energy has not be identified, but should be expected at a higher time. It is also clear in this plot that the total energy is always negative, which is what is expected for a closed orbit.

II. CONSIDERATIONS OF PARALLELISATION METHODS

1. Linear Cython

Using Python in simulations of this kind is not desirable due to the high relative computational time compared to that of a compiled language such as C. For example, in Python data types are not specified by the user but by context, which creates a large overhead to the program, whereas using a compiled language can be 100x faster than this. Cython, on the other hand, gives the flexibility of Python but with the added benefit of being able to compile code into highly efficient binary, giving speeds comparable to compiled languages. In Cython, using the cdef key word to define variables can lead to potential speed ups of 3x, especially if they are used inside loops. A further speed up can be observed using cimport to make use of the C maths library to give computational speeds comparible to that of C. This was observed when using the square root function inside the force calculations. Changing from using the Numpy square root to the C library equivalent gave speed ups of up to 2x.

Further speedups can be observed with boundscheck set to False, which turns off double checking when accessing memory outside of an array when indexing outside of it and cdivision set to True, which converts all divides into C divides by default.

2. OpenMP with Cython

OpenMP can be used to distribute the workload of calculating data between a number of computational cores. It can be used in loops where each iteration is independent of others, and where there is no data dependence inside a loop. In this simulation it is used to parallise the integrator and force calculations. As the force calculations are the most computationally expensive part, this is an effective way of improving efficiency. The force calculations may be paralellised as they are an independent data set. The calculation of forces uses a nested loop. It has been found that using a parallel loop for both of these loops is in fact slower than just the outermost loop, with speeds up to 3x slower. This is straightforward to explain as simply the innermost loops being parallelised N times, which greatly increases the communication time between CPUs, and thus reducing potential speed ups. Therefore the parallel loop was used in the outermost loop wherever possible.

Scheduling in OpenMP

The number of iterations assigned to each compute core can be controlled using the ”schedule” clause in OpenMP. There are three options for the type of scheduling, static, dynamic and guided. In static, the number assigned is determined at the compile time and remains fixed. This is suitable if each of the chunks distributed are of the same length. In dynamic, the exact decomposition of the array is determined at run time and blocks are carried out by a worker whenever it becomes free. In guided, the scheduling will gradually reduce the size of a block to give even load balancing towards the end of a loop. The default is dynamic, which was determined to be the fastest method for the calculation of the forces at each timestep, and was therefore used throughout. The static scheduling was not applicable unless there was an exact division of the size of the loop between the workers, which will be rare.

3. MPI with Python

Whereas OpenMP uses a shared memory system to distribute workload between CPUs, MPI communicates explicitly between CPUs and uses a distributed memory system, sending and receiving chunks of independent data to and from specific tasks, shared and distributed memory will be discussed in more detail later. Careful consideration is needed of the design of inter-task communication. The benefit of parellising has to be balanced with the cost the overhead of synchronous communication between tasks, whereby tasks are waiting instead of being able to perform work. Asynchronous

communication should be used wherever possible, where tasks transfer data independently and therefore work can be done independent of communication to give the most efficiency. Force calculations may be performed asynchronously as each chunk sent to a task is not dependent on another, therefore there is no wait time. However, timesteps are synchronous as positions cannot be updated until all force calculations have been performed. However, the potential time where tasks are idle should not have a large effect on the compute times at larger N.

To reduce the computational cost of not using Cython with MPI, Numpy was used extensively and wherever possible. This is because the Numpy package utilises BLAS, vectorisation and some C. Speed ups compared with normal Python were seen of up to 3x. Effective use of Numpy can provide up to 100x speedups.

Furthermore, as the cost of distributing data between tasks is so great, care was taken to simply send only the minimum information required for each task to calculate a given force. This can be achieved by defining masses at the master task and initially distributed only once, sending only the positions necessary for each task to calculate a chunk of the forces and only ever computing the updated positions once at the end of a timestep at the master task. One chunk of work was left for the master to perform, so that it was not a wasted worker, this gave an observable speedup.

Shared vs Distributed memory

In OpenMP, all the processors share the same address space and have access to shared global memory pool for computation. Shared memory systems are fast, with high bandwidth, the data rate that can be achieved once communication between tasks has begun, and low latency, the initialisation period before a transfer of a chunk of memory[4]

For MPI, each processor has its own private address space and must communicate between each other to exchange data across processors. In this case, there is a higher latency than for a shared memory system. However for large enough models, the effect of latency is diminished significantly. The BlueCrystal supercomputer used for the computation in this simulation is a distributed memory system. At high enough N, MPI should be the faster parallisation method. This is a consequence of being able to use its own private memory space and thus the effects of latency are decreased at higher N, whereas OpenMP still needs to access shared memory at a rate that does not decrease. However, due to efficiency limitations using Python with MPI, compared to Cython or C with OpenMP, the observed timings for MPI are in fact slower than OpenMP at higher N.

III. Results

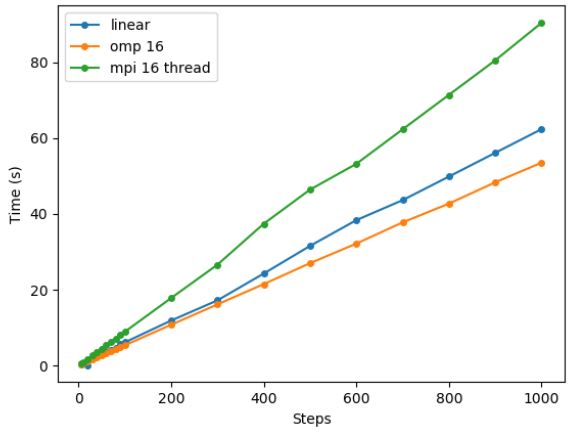

FIG. 6. The computational time as a function of the number of timesteps used, at 500 particles. Differences between linear OpenMP and MPI can be seen, with all plots showing clear linearity.

FIG.6 is a plot of the variation in execution time for a number of different timesteps, at fixed particle number of 500. A clear $\mathcal{O}(N)$ relation is seen for each of the three methods. It also shows that OpenMP is the fastest to execute at each timestep, with MPI the slowest.

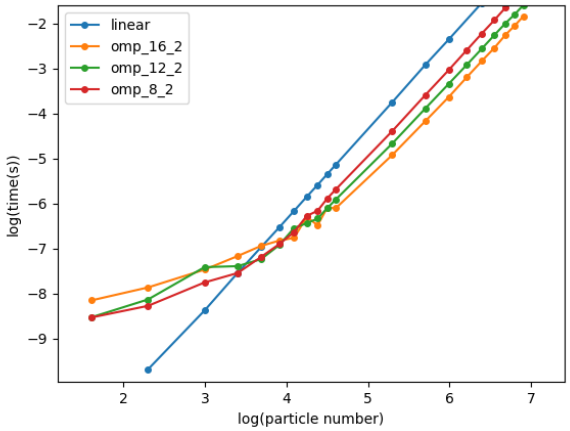

FIG. 7. Log-log plot of the computational time versus the particle number for different number of threads used in OpenMP, and compared to serial cython code to reveal any "crossover".

The log-log plot in FIG.6 shows the expected $\mathcal{O}(N^2)$ relation for the time to compute N particles, with a gradient of each of the lines close to a value of 2 at a sufficiently high N. The linear plot with no implementation of parallelisation,

seen in blue, has a gradient of 1.99 $\pm$ 0.01, which is consistent at lower particle numbers. Interestingly, for OpenMP plots using various compute cores, there is a section at lower particle numbers where it is clearly slower than serial code, with a "crossover" at given particle number. This is due to the computational cost of initialising the communication between the cores becoming the limiting factor in the efficiency. Theoretically, the crossover points should show a $N = k\frac{N}{p}$ relation, where k is some constant. This is because the parallelised code should show a $\frac{1}{p}$ trend as it is linear in this region, with the serial plot a $\mathcal{O}(N^2)$ trend, and they are equal at the intercept. However, the data seen in FIG.6 becomes inconsistent around this point, as the communication times begin to dominate, and reliable intercepts are not discernible at such low particle numbers. An average crossover between the different number of threads is 39 particles, rounded to the nearest whole particle.

The gradient of the OpenMP plots after the crossover point are observed to decrease with the number of computer cores used. As there is a $T = CN^2$ relation, at high enough N, with the logarithm of N and T plotted this C is seen in the intercept. The relative computational speed of C for each of the different thread numbers and serial code can be calculated by taking the exponential of the intercepts of the y axis. The time per force calculation is seen to be 2.2 times faster between linear and 8 threads on OpenMP, and speedups between increasing the number of threads by 4 as 1.28 times faster.

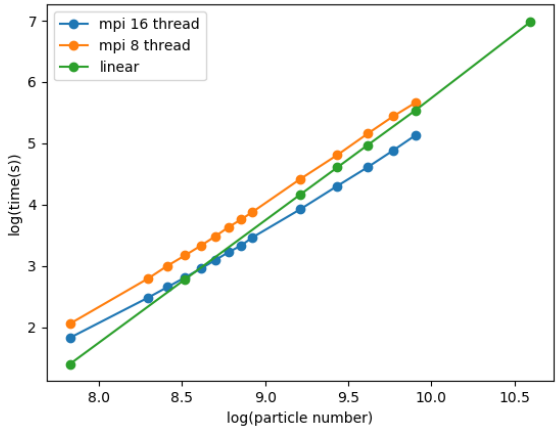

FIG. 8. A log-log plot of the computational time as a function particle number, using different number of compute cores in MPI, with a comparison to serial cython code to evaluate any speedups of parallisaiton.

The data for the compute time as a function of particle number for MPI, compared with serial code, is seen in FIG.8. In this case, at lower particle numbers, the serial code is quicker, up to a higher number than for OpenMP. The crossover point for this case is also at a higher particle number than for OpenMP, with 5431 particles at 16 threads, and a predicted 24343 particles for 12 threads. There is greater and more defined difference between these two values than that seen in FIG.7. This is because the crossover points in OpenMP are effected predominately by the serial regime, whereas MPI does not suffer from this around the crossover point. With more time, these crossover points can be found and compared with the intercept point of $N = \frac{k}{\sqrt{p}}$, different to that of OpenMP as it is in the $\mathcal{O}(N^2)$ regime, and a value for the constant k determined. This can be used to predict where MPI will be faster than serial code for a given particle number and number of compute cores.

MPI vs OpenMP

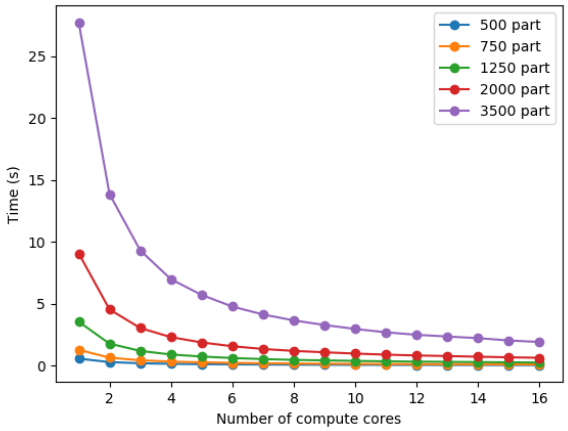

FIG. 9. The computation time taken as a function of the number of compute cores used in OpenMP, with differing numbers of particles.

FIG.9 is a plot of how the computation time varies as a function of threads for a series of particle numbers. Clearly, there the time scales with $\mathcal{O}(p^{-1})$, where p is the number of processors, at each of the different particle numbers.

To investigate FIG.9 further, Amdahl's law can be used to determine the percentage of the code that is sequential and parallel at different particle numbers [5]. The time total time for a computation, T_p, is given by:

$$T_p = T_1(F_s + \frac{F_p}{p}). \tag{8}$$

Here, T_1 is time for 1 processor's execution and fractions F_p and F_s are the parellel and sequential code respectively, where F_p + F_s = 1. Taking these 5 data sets, and plotting a fitting function of the form of EQ.8 with F_p = 1- F_s, 5 different values for the sequential code can be determined. These are plotted in FIG.10.

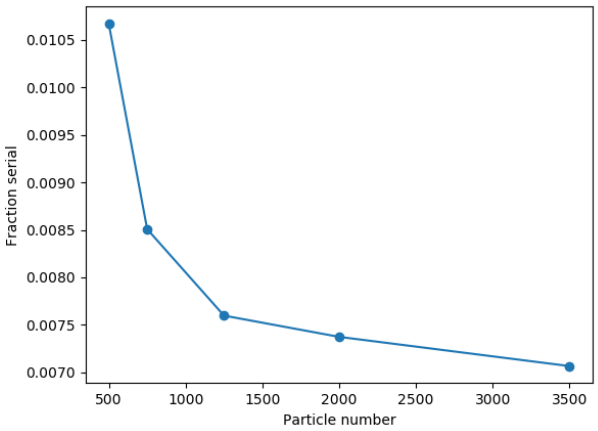

FIG. 10. The fraction of serial code as a function of the particle numbers used.

From FIG.7, it is known that at lower particle numbers, the cost of initialising the communication dominates the computational time, where it obeys a more linear relationship with particle numbers. This is what is being shown in FIG.10, where the higher the particle numbers used, the lower the serial code seen, where it is expected there will be a minimum fraction of serial code that is always present, as this cannot be eliminated completely.

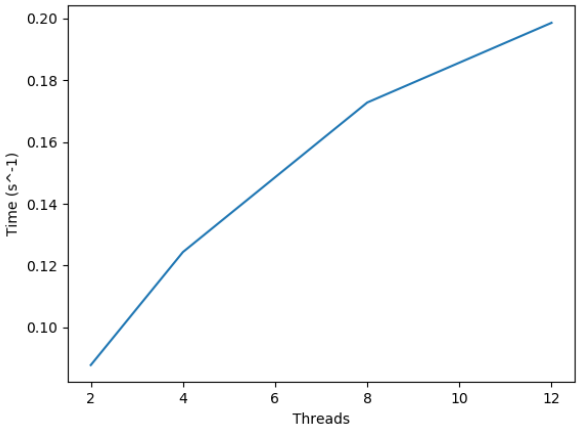

FIG. 11. The inverse of the compute time taken as a function of the number of threads used for MPI.

FIG.12 shows a set of data for the variation in compute time against the number of compute cores for MPI, with the times inverted to reveal the expected relation of inverse of the number of processors. The exact relation for the number of compute cores will be a constant + $\frac{1}{p}$, however, at large enough N the constant will become negligible and the number of processor dominates. However, the graph is not perfectly linear, as large enough particle number has not be reached to observe this. Further investigations may be done to give a plot similar to that in FIG.10 to reveal the change in fraction in serial code with particle number.

IV. CONCLUSIONS

In conclusion, parallelisation methods of OpenMP and MPI were used to compare execution times with serial code for varying model sizes. OpenMP gave execution times lower than serial code, after an initial zone where there is a computational cost of initialising communication between cores. MPI was observed to be faster than the serial code at a certain particle number, but this was greater than that of OpenMP. Further study should seek to see if there is a crossover point between MPI and OpenMP, where MPI becomes more efficient. This should be expected as MPI will become more effective in comparison to OpenMP at higher particle numbers, due to the reasons given previously. Therefore, for large N-body simulation of the solar system, it is predicted that MPI should be used to give the lowest compute times, which is what is desired in such simulations.

REFERENCES

[1] Lacey, C. and Cole, S., 1994. Merger rates in hierarchical models of galaxy formationII. Comparison with N-body simulations. Monthly Notices of the Royal Astronomical Society, 271(3), pp.676-692.

[2] Gravitational N-Body Simulations, Tools and Algorithms, S. J. Aarseth, Cambridge University Press (2003).

[3] William H. Press, Saul A.Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes: The Art of Scientific Computing, Cambridge University Press, Third Edition (2007).

[4] Bershad, B.N. and Zekauskas, M.J., 1991. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors.

[5] Hill, M.D. and Marty, M.R., 2008. Amdahl's law in the multicore era. Computer, 41(7), pp.33-38.