Simulating Celestial Orbits Using a Direct Approach to Newtonian Gravity

R. Beale
Advanced Computational Physics, Department of Physics, University of Bristol
(Dated: February 12, 2020)

This report aimed to explore methods of optimisation using a direct approach to modelling a solar system using Newtonian gravity. The data of the planets of the solar system were entered into Python and an increasing number of asteroids were randomly generated to produce arrays of 10-265 particles. The gravitational force was found between every particle and put into SUVAT equations to find the new position of each particle at every timestep. Cython and vectorisation were then employed on the original serial programme to produce maximum speed-ups of 11.3x and 9.5x, respectively. The code was then parallelised using OpenMP and MPI libraries to produce maximum speed-ups of 12.3x and 12.9x, respectively.

1. INTRODUCTION

Newton constructed the mathematical framework required to find the equation of motion between two objects due to gravity. Newton first described the 'N-Body Problem' in his pivotal 'Principia' where he alluded to the unsolvable nature of it [1]. Four centuries later, an analytical solution for large N-systems is unattainable via the original method of first integrals [2]. Other pioneers, notably Poincare, attempted to solve the N-body problem via a series expansion (which must be valid for all time). Whilst Poincare's initial attempts did not yield a solution, his groundbreaking work into chaotic systems laid the foundations of the field [2]. More recent studies into this elusive problem have produced convergent series, for example, in 1991 Qiu-Dong's paper published a global solution [3]. However, although solutions have been found, the series are astonishingly complex and and not practical for real world applications. Consequently, the most common method of approaching N-body problems is numerical integration.

Numerical solutions utilise computer algorithms to approximate the solution to differential equations. By using an increasing number of iterations, the numerical approximation will tend to the true value in an exponential manner [4]. Numerical solutions are an increasingly important tool for scientific study but come with the inevitable time and computational power cost. Moreover, direct approaches of N-body simulations do not approximate the systems. Hence, for each iteration in a gravitational simulation, the force on each body must be calculated from the net force of every other body. This requires substantial computational resources, however, they also produce the most accurate simulations [5]. Therefore, producing efficient algorithms is essential to maximise the effectiveness of numerical solutions.

2. THEORY

This programme aims to simulate the orbits of bodies in our solar system for an increasing number of randomly generated asteroids. To minimise computation time and aid in simplicity, some approximations have been made, namely: all the bodies exist on the same plane, hence, their z position and velocity is omitted and the bodies are treated as point like particles.

This simulation employs Newtonian gravity to find the force, F, applied to the i^{th} body. As gravity acts over infinite distances, the individual forces between every body must be found. This was done in the x and y decomposition.

$$\mathbf{F}_{\mathbf{x},i} = \sum_{j=1}^{N} G \frac{M_i M_j (\mathbf{x}_j - \mathbf{x}_i)^2}{((\mathbf{x}_j - \mathbf{x}_i)^2 + (\mathbf{y}_j - \mathbf{y}_i)^2)^{3/2}} \tag{1}$$

where G is the gravitational constant and M is the mass.

After calculation of the force, the acceleration can be determined using Newtons second law. The initial position and velocity of the planets were taken from NASA data (and the asteroid variables were randomly generated). The 'SUVAT' equations were then employed to produce a new position and velocity. This process was completed iteratively to simulate the trajectory of the bodies in the solar system. This can only approximate the analytical solution of said bodies. However, decreasing the time step and increasing the number of iterations will cause the approximation to asymptotically approach the 'true' orbit.

3. METHODS OF SINGLE CORE SPEED OPTIMISATION

Initial attempts at the simulation were completed with Python on a single core using Blue Crystal Phase 3.

3.1 Vectorisation

The NumPy module allows for vector algebra, computing many operations simultaneously. The most time costly operation is the gravitation function, followed by the SUVAT equations (99.10% and 0.89% of total computation time, respectively). This is due to the nested loop operations required to find the force on each body felt from every other body and the element-wise equations necessary to produce the new **x** or **y** velocity and position.

A major proportion of the computational time within the gravitation function was finding the distances between every particle. By creating one horizontal, one vertical NxN array containing the **x** and **y** positions of each body, the distance between each body can be determined concurrently.

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & \dots \\ x_1 & x_2 & x_3 & x_4 & \dots \\ x_1 & x_2 & x_3 & x_4 & \dots \\ x_1 & x_2 & x_3 & x_4 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} - \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & \dots \\ x_1 & x_2 & x_3 & x_4 & \dots \\ x_1 & x_2 & x_3 & x_4 & \dots \\ x_1 & x_2 & x_3 & x_4 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

$$= \begin{pmatrix} x_1 - x_1 & x_1 - x_2 & x_1 - x_3 & x_1 - x_4 & \dots \\ x_2 - x_1 & x_2 - x_2 & x_2 - x_3 & x_2 - x_4 & \dots \\ x_3 - x_1 & x_3 - x_2 & x_3 - x_3 & x_3 - x_4 & \dots \\ x_4 - x_1 & x_4 - x_2 & x_4 - x_3 & x_4 - x_4 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \quad (2)$$

This is repeated for the y axis and Nx1 vectors of M_i and M_j are multiplied. Subsequently, the summation from Equation 1 is removed and NxN arrays replace the current subscripted variables. The resulting NxN array produces the force on each body from every body simultaneously. Similarly, the vectorised code meant the nested loops within the SUVAT equations could be eliminated; the new position and velocity of each body could be found simultaneously.

3.2 Cython

The elegance and adaptability of Python code has no doubt been the rationale for its ubiquity among scientific computing. Combined with Python's ability to interface with C/C++ and Fortran it is clear why it is such a popular language [6]. However, Pythons inefficiency and sluggish speeds continue to hinder its use. The introduction of Cython leverages the high performance of C to optimise python code. Part of C's speed is owed to compiling code. This involves building an executable in machine code; this low level language is much faster for the hardware within your computer to read [7]. Cython is an extension of Python that allows the compiling of programmes into machine code, achieving significant speed-ups [8].

Cython additionally enables one to C-define variables and also make use of C-imported functions. The advantage of C-defining lies in Pythons inefficient use of loops. At the start of a function, Python determines what data type each variable is and then performs the operation. However, it will do this with every iteration of the loop, resulting in unnecessary extra computation time. By C-defining the variables at the start of a function, Cython removes the need to repeatedly perform this operation. Moreover, utilising C-imported functions such as sqrt() employs the fully optimised functions within C and produces marginal speed-ups.

From Figure 1, it can be seen that introducing C-defined variables and C-imported functions produces substantial speed-ups for smaller array sizes, 10.0x in this instance. However, as the array sizes increase, the speed-up gradually diminishes. By considering the percentage of time declaring variables with regards to the total time within the loop, it can be seen that this effect decreases in significance as the number of operations within the loop increases.

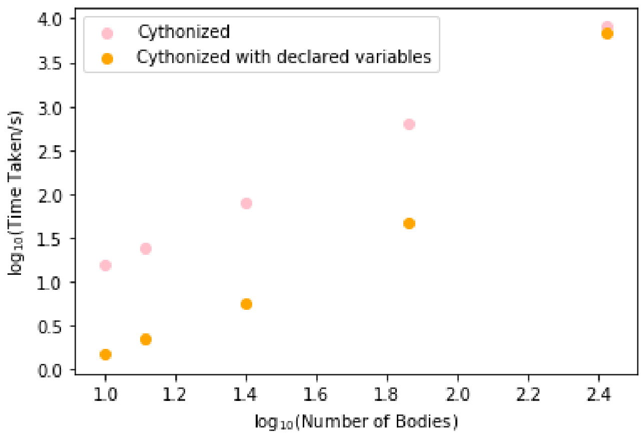

FIG. 1. Time taken to run the gravitational simulation with increasing numer of bodies for two different Cythonised programmes

3.3 Comparison of Methods

Each method was timed for increasing number of bodies and plotted in Figure 2. Initially, the original code was simply 'Cythonized' which rendered modest speed-ups of 1.13x for 10 bodies but with worsening effect as number of bodies increases (<0.01x speed-up for 265 bodies). However, upon further optimisation with C-defined functions, speed-ups of 11.3x were observed for 10-body arrays. Similarly, the optimised Cython code decreased in efficiency as the number of bodies increased. Conversely, vectorisation produced greater speed-ups for larger array sizes; 10 body arrays were 4.92x faster whereas 256 bodies achieved 9.50x speed-up.

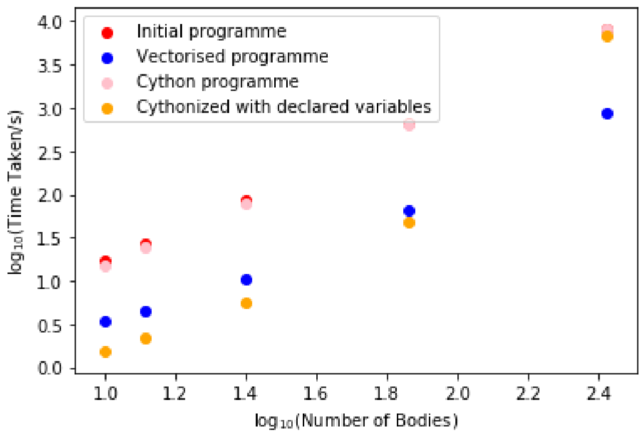

FIG. 2. Comparison of single core speeds of the gravity simulation for different optimisation methods

4. PARALLEL PROCESSING

In recent years the improvement in performance of processors has curtailed; mainly due to chips inability to increase the number of cycles per second [9]. Consequently, improvements in computer performance requires parallel computing. Parallel processors consist of multiple processing units, which communicate via interconnection networks, and a software capable of managing the hardware [10]. One of the key challenges of high performance computers is their ability to manage memory between cores, a function controlled by the interconnection network. Memory can be managed via distributed memory or shared memory. The former involves individual cores containing their own memory and relying on communication to share data between them. In the latter, individual cores lie on a chip and each core can access the same memory [11].

The division of work between different processors can introduce significant reductions in computation time, known as speed-up, S_p.

$$S_p = \frac{T_1}{T_n} \tag{3}$$

where T_1 and T_n is the time taken for one and n processors to run, respectively. Furthermore, as processor number increases, efficiency, E_p may decrease, where there is increased idle time in the cores.

$$E_p = \frac{S_p}{n} \tag{4}$$

Efficiency is likely to decrease as the number of processors increases as processors will try and claim the same finite computing resources. Furthermore, the communication between cores in distributed memory will result in extra computation not previously required. As the number of cores increases, the cost of communication will eventually outweigh the extra processing power and any further CPU's added will increase the total computation time [12]. The programmer may alleviate some of the decrease in efficiency by structuring software to ensure the workload between cores is balanced, ensuring the whole programme is not waiting on one core to finish its portion [13]. Perfect efficiency is possible but very unlikely as it requires algorithms that involves no intercommunication of data, such programmes are known as embarrassingly parallel.

4.1 MPI

The Message Passing Interface (MPI) model allows for the communication of memory between nodes. Typically, the user will assign a 'master' core which will be the lead communicator for the other 'worker' cores. There are two main methods of communication: point to point, where the master sends data to individual threads, and collective where the master sends the same data to all cores. Furthermore, the messages can be synchronous or asynchronous, as decided by the programmer.

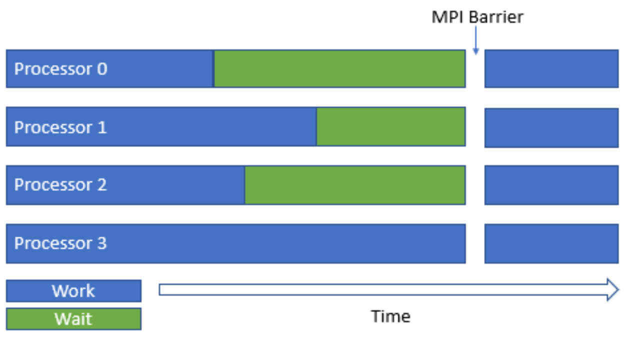

FIG. 3. Demonstrating waiting times due to blocking communication in MPI programmes

Synchronous communication implements a barrier at specific places to ensure all threads are at the same point of execution. This has the benefit of ensuring that threads have the relevant updated data to proceed with their respective task e.g. positional data in each time step [14].

However, with synchronicity comes a cost in time management. The waiting periods, as shown in Figure 3, could be running useful functions. If possible, asynchronous communication will produce quicker computation time but one must ensure it will not give erroneous results. One way to alleviate waiting periods is to ensure the aforementioned load balancing of work is optimised.

This programme required the use of blocking communication as each time-step used updated values of position and velocity to find the new position and velocity. Asynchronous communication would lead to one thread beginning before another had finished and data being used from different points in time.

4.2 OpenMP

The advantage of shared memory systems is the speed at which they can access data. Unlike distributed memory systems there is no 'master' core which provides the data for the other cores to work with. The time costs of acquiring data in the shared memory model is a fraction of the distributed memory model as there is not the same communication overheads. Their time constraint is the time taken for data to be transferred across the physical distance between the processor and memory location [15]. An important distinction is that shared memory systems are limited in size. Every processor on the same cluster will require RAM. Therefore, the shared memory must be able to account for the fact that at peak times of use, each processor will only be able to access $^1/_N$ of the total available.

OpenMP is not compatible with Python but is with Cython. Importing 'prange' from Cythons parallel library allows one to change normal loops into a parallel loop; handing out different parts of the loop to different processors. However, the

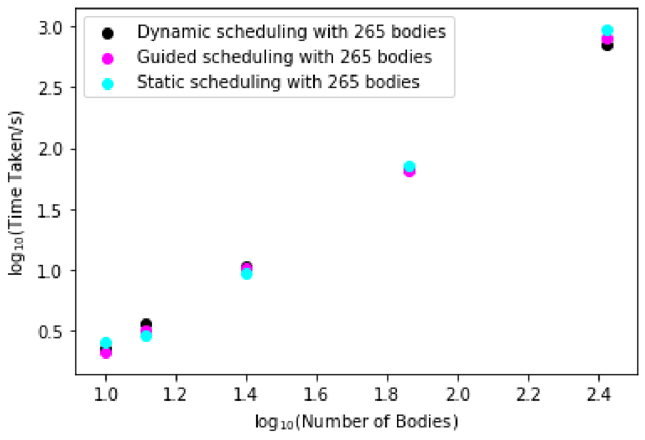

FIG. 4. Time taken for the OpenMP programme to run on 16 cores with varying scheduling protocols

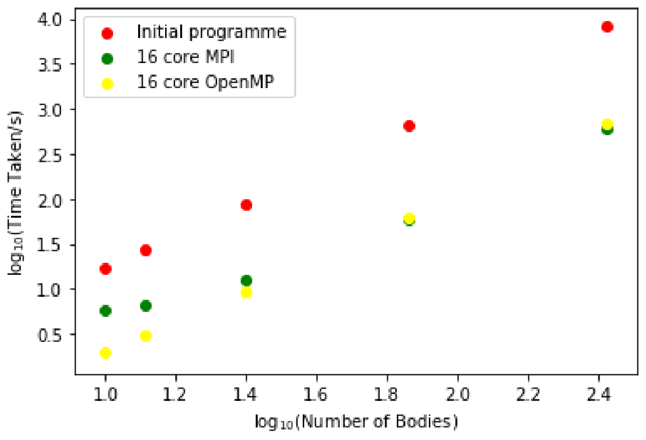

FIG. 5. Comparison of timings for initial single core and OpenMP and MPI multi-core programmes

Global Interpreter Lock (GIL) within Cython prevents the programme having more than one thread active at one time [16]. By using the option 'nogil=True', the GIL can be turned off, enabling the use of multithread processing.

Within a prange loop, data can be handed out under different strategies in a process known as scheduling. By default, the loop hands out data with a 'static' schedule; the number of iterations within the loop is divided by the number of processors and one chunk is given to each thread [17]. This can produce non-optimal results if the run time for one thread is longer than another and will result in load unbalance. To combat this effect, one can opt to use 'dynamic' scheduling. In this method each thread is given a single iteration and when a thread has completed its work, it gets handed another one until all the work has been completed. A further method is known as 'guided'. This is similar to dynamic but the chunksize of iterations starts off large and then decreases in size upon each successive hand out [18]. Figure 4 demonstrates that there was little effect on the introduction of scheduling. The maximum speed-up seen was with 265 bodies and dynamic scheduling, displaying a 1.30x decrease in run time. The lack of change suggests that the code is well load balanced. This is due to the fact the introduction of optimisations aimed at minimising waiting times had little effect which suggests the waiting times must have been small.

4.3 Comparison of Methods

Figure 5 demonstrates the significant speed-ups for the multi-core over single core programmes. Namely, for 10 bodies and 16 cores, the speed-ups were 4.32x and 2.80x for OpenMP and MPI, respectively. Whereas 265 bodies produced 8.77x and 12.86x for OpenMP and MPI, respectively. At smaller size arrays, the time cost of communication between nodes in MPI is significant and OpenMP is almost 3x faster. For smaller arrays, it may be time-effective to reduce the number of cores, thereby reducing the associated cost of communication (whilst still benefiting from some parallelism). Figure 6 displays the timings for a 10 body array as the number of cores increase.

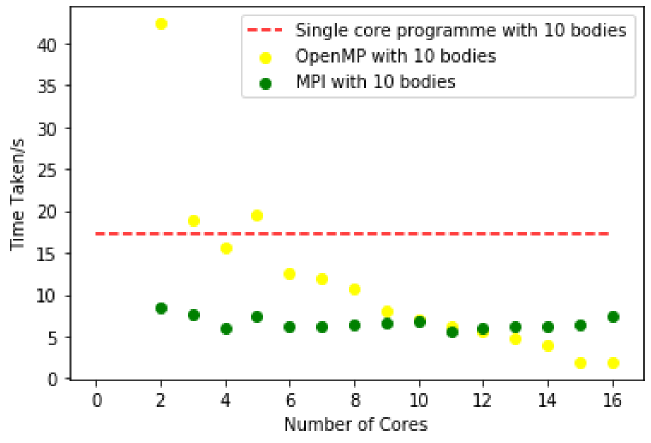

FIG. 6. Time taken for varied number of cores, with MPI and OpenMP programmes, to solve a 10 body gravity simulation

Although there is indeed speed-up through the use of multithreading, it is evidently not ubiquitous. Using OpenMP for up to 4 cores, results in small changes or even slower speeds. This is likely due to increased overhead when implementing OpenMP, such as thread startup and thread library startup [19]. However, a general downward trend in time taken is observed. Conversely, the MPI programme were all faster but show very little correlation between timing and number of cores, with timings displaying a small spread near to its average value. It may be inferred that increased overhead due to increased communication is approximately equal to the benefits of increased number of cores and as such provides no net benefit. Indeed it would likely be a waste of resources to increase the number of cores past four for small MPI programmes. See Table I for a table of speed-ups and efficiency.

	No of cores	2	4	6	8	10	12	14	16
OpenMP	Speed-up	0.1	0.407	1.104	1.368	1.624	2.444	3.031	4.324
	Efficiency	0.14	0.204	0.276	0.228	0.203	0.244	0.253	0.309
MPI	Speed-up	0.13	2.020	2.829	2.762	2.693	2.525	2.896	2.795
	Efficiency	0.0080	1.010	0.707	0.460	0.337	0.253	0.241	0.200

TABLE I. Comparison of speed-up and efficiency for OpenMP and MPI programmes as number of cores increas for a 10 body system

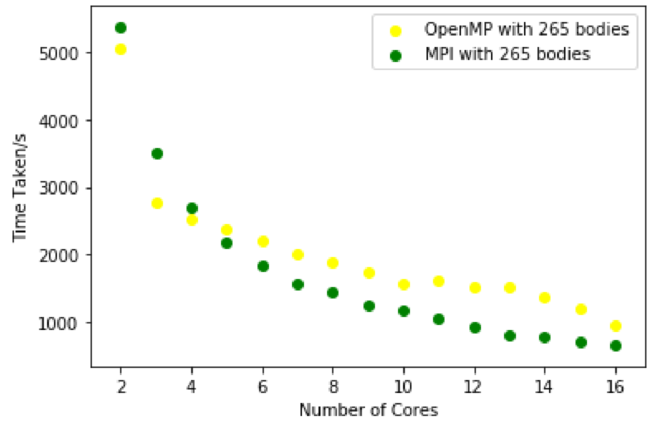

FIG. 7. Time taken for varied number of cores, with MPI and OpenMP programmes, to solve a 265 body gravity simulation

The initial introduction of MPI gave an embarrassingly parallel result. However, this may be due to fluctuations in time taken for the programme to run; efficiency would likely be less than 1 upon calculation over more repeats.

Figure 7 displays the different timings for a 265 body problem. This situation produced faster computational times in every case. It can subsequently be inferred that parallelism is more effective on larger array sizes. See Table II for a comparison of speed-ups and efficiency. The points were then fitted with an allometric function allowing the prediction of run times for different numbers of processors. OpenMP follows a $t \approx 7800n^{-0.7}$ and MPI follows a $t \approx 11000n^{-0.1}$.

	No of cores	2	4	6	8	10	12	14	16
OpenMP	Speed-up	1.637	3.280	3.765	4.403	5.259	5.435	6.074	12.255
	Efficiency	0.818	0.820	0.627	0.550	0.526	0.453	0.434	0.766
MPI	Speed-up	1.534	3.068	4.477	5.771	7.098	8.968	10.527	12.862
	Efficiency	0.767	0.767	0.746	0.721	0.710	0.747	0.752	0.804

TABLE II. Comparison of speed-up and efficiency for OpenMP and MPI programmes as number of cores increase for a 265 body system

5. ANALYSIS

The aforementioned accuracy of direct methods is accompanied by large computational times. Specifically, one expects the run times to scale with N^2 [20]. As such, the data for run times against size of array for each programme was fit with an order two polynomial function. The reduced chi-squared indicated a good fit in every programme. A table presenting the coefficients of the N and N^2 terms are presented in Table III.

Programme	N	N^2
Original	0.59	0.12
Cython with Cdef	-11	0.14
Vectorised	-0.054	0.13
MPI (16 cores)	0.18	0.0080
OpenMP (16 cores)	0.19	0.0089

TABLE III. Coefficients of the second order polynomial fit of how the computational times scale with problem size

This allows the different methods to be extrapolated into the larger N region. The equations of fit suggest that MPI with 16 cores will give the smallest run times in this limit. For example, taking N to be 10^3 gives $\approx$ 8200s whereas the original programme gives $\approx$ 120000s. At this scale, the run times for the original programme are too long for any physical experiment but that of the MPI programme is only of the order of 2 hours. Moreover, MPI programmes can increase the number of threads used with ease, further decreasing the run time. However, upon increasing the problem size to 10^4 bodies, the computational times for every method becomes too long. This demonstrates direct method's inherent limitation of handling large problem sizes. Simulating the gravitational attraction of large body systems requires the use of indirect methods such as Barnes-Hut which scale as $\mathcal{O}(n \log n)$ [21].

The efficiency of the MPI programme showed little change as the number of cores was increased. This suggests the programme is strongly scalable under the current processor range [22]. By increasing the number of processors further, the scalability can be investigated in this limit. Figure 8 demonstrates the decrease in efficiency as the the number of bodies is increased. Initial increases in the number of cores produces

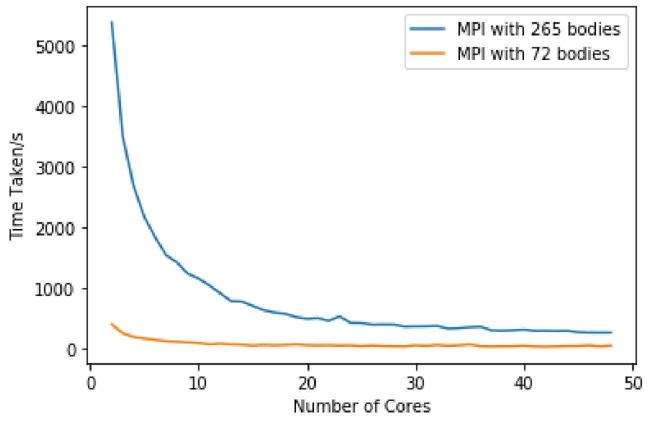

FIG. 8. Demonstrating the diminishing effect on run times when increasing the number of processors for a fixed problem size

significant reductions in the run time, i.e a strongly scalable problem. As the number of processors decreases further the gradient decreases, suggesting a diminishing efficiency. The 72 body problem, levelled out at a much quicker rate which implies a poor scalability.

A notable point to be made was the substantial slow-down accompanying the introduction of 2-core OpenMP. Indeed, one would expect OpenMP to be quicker than MPI in every instance (up to 16 cores) as there is no communication overhead required in shared memory systems. A possible rationale for this observation is the part of the code that is parallelised. Once the GIL is released, many Pythonic operations are inhibited. To allow OpenMP to work, the parallel loop was moved to the innermost part of the code to minimise the number of operations that needed to be completed with the GIL released [19]. Conversely, in MPI, no such issues were encountered meaning the parallel loop could be further out and incorporate more operations. Amdahl's law state that the performance improvement due to parallel processing is limited by the sections that must be completed serially [23]. The position of the parallel loop in OpenMP meant that a larger proportion of the code is run on a single processor. Therefore, one would expect the speed-up of OpenMP to be limited and produce slower run times.

CONCLUSION

This paper aimed to investigate different methods of solving a direct N-body gravity simulation. It was found that small array sizes (≈ 10) benefited from the introduction of single core optimisations. Namely, using Numpy arrays to perform multiple operations at once (known as vectorisation) and implementing Cython, an extension of python that allows the user to utilise the faster processing speeds of C. The introduction of parallel processing produced mixed results. OpenMP models were initially significantly slower than the original programme but produced faster run times as the number of cores increased. MPI was consistently faster than the original programme but with very little change as the number of cores increased. Larger array sizes benefited far more from the introduction of multi-core computing with significant speed-ups seen for both OpenMP and MPI models. It was found that the quickest computational times were produced with Cython (combined with declared variables) for small array sizes and 16 core MPI programme was superior for ≥ 265 arrays.

REFERENCES

[1] Stephen Hawking. *On the shoulders of giants: The great works of physics and astronomy.* 2002.

[2] Florin Diacu. The solution of the n-body problem. Technical report, 1995.

[3] Wang Qiu-Dong. The global solution of the n-body problem. *Celestial Mechanics and Dynamical Astronomy*, 50(1):73–88, 1990.

[4] Nicolaas Govert De Bruijn. *Asymptotic methods in analysis*, volume 4. Courier Corporation, 1981.

[5] Michele Trenti and Piet Hut. N-body simulations (gravitational). *Scholarpedia*, 3(5):3930, 2008.

[6] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31, 2011.

[7] McGregor Jim, McGregor Richard, and Watt Alan. Simple c, 1997.

[8] Ilmar M Wilbers, Hans Petter Langtangen, and Åsmund Ødegård. Using cython to speed up numerical python programs. *Proceedings of MekIT*, 9:495–512, 2009.

[9] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. Lulu. com, 2013.

[10] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced computer architecture and parallel processing*, volume 42. John Wiley & Sons, 2005.

[11] Michel Syska. Interconnection networks for parallel processing: Graphs, models and algorithms. 1995.

[12] Norihisa Suzuki. *Shared Memory Multiprocessing*. MIT Press, 1992.

[13] Derek L Eager, John Zahorjan, and Edward D Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[14] Thomas Sterling, Maciej Brodowicz, and Matthew Anderson. *High performance computing: modern systems and practices*. Morgan Kaufmann, 2017.

[15] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[16] Serge Guelton, Pierrick Brunet, and Mehdi Amini. Compiling python modules to native parallel modules using pythran and openmp annotations. *Python for High Performance and Scientific Computing*, 2013, 2013.

[17] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of openmp task scheduling strategies. In *International Workshop on OpenMP*, pages 100–110. Springer, 2008.

[18] RW Green. Openmp* loop scheduling. *Intel Corporation*, 29, 2014.

[19] Paul Lindberg. Performance obstacles for threading: How do they affect openmp code. *Intel Software Developer Zone: https://intel. ly/2McZIG8*, 2009.

[20] Michele Trenti and Piet Hut. Gravitational n-body simulations. *arXiv preprint arXiv:0806.3950*, 2008.

[21] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[22] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011.

[23] João Manuel Paiva Cardoso, José Gabriel de Figueiredo Coutinho, and Pedro C Diniz. *Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation*. Morgan Kaufmann, 2017.