Direct Method Gravitational N-body Simulation
Exploring the Benefits of Parallel Code

T. du Sautoy
Advanced Computational Report, School of Physics, University of Bristol.
(Dated: February 8, 2018)

This report demonstrates how parallel programming can significantly improve gravitational n-body simulations. The direct method gravity model was simulated and OpenMP/MPI parallel interfaces were used to optimise the code. Significant speed ups were achieved with both parallel systems. OpenMP displayed speed ups of up to $5\times$ and MPI achieved a more modest $2\times$ speed up.

INTRODUCTION

Computing power has consistently followed Moore's law for the last 40 years. However the trend, as seen in Figure 1 is beginning to reach its limits. As the size of transistors reach the fundamental barriers of the atomic level and individual transistor speeds are limited by gate capacitance, optimising compute core performance has become ever more dependant on clever software development to utilise multiple cores. This is referred to as parallel programming and this report will look at the various interfaces available for creating parallel programs.

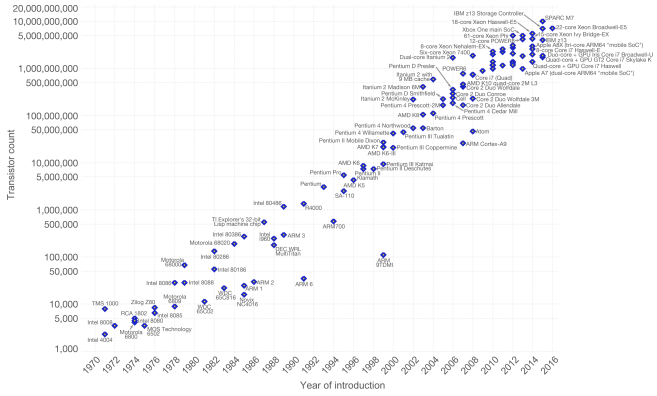

FIG. 1. Gordon Moore predicted in his 1965 paper that the number of transistors on an integrated circuit chip will double every year (since revised to approximately two years) [1]. Computing performance which also includes improvements in individual transistor operating speeds is said to double every 18 months [2]. Image Credit: Roser and Ritchie [3].

The problem that will be studied in this report is the N-body simulation under gravitational forces using the direct method. Many areas in Physics require heavy computational simulations which can be optimised with parallel code. Astrophysicists use simulations as a tool to study the evolution of cosmological processes with the number of particles in the tens of thousands. With serial code the compute time for such large systems is simply impractical. Using the direct method to simulate the gravitational N-body problem the computational complexity scales with N^2. Parallelising certain aspects of the simulations can distribute the work required of a single processor across multiple and dramatically improve the speed of the code leading to faster run times and the possibility to simulate larger systems like that in Figure 2.

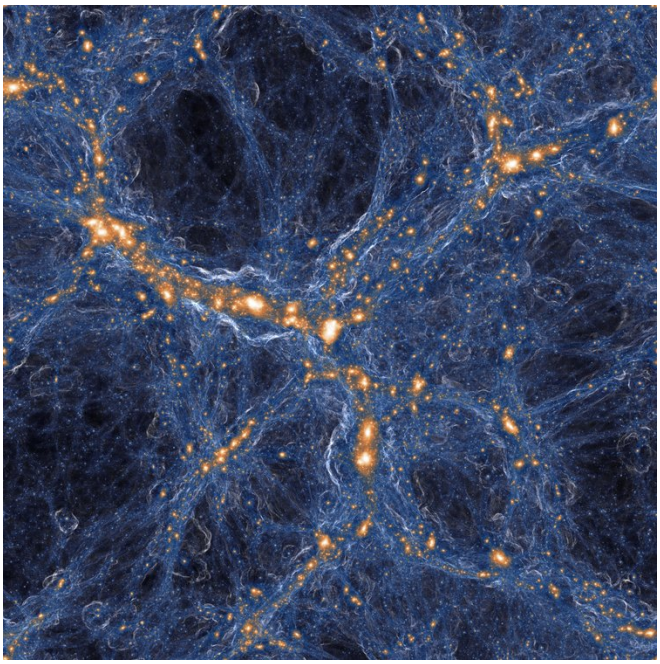

FIG. 2. Images such as this have been produced by the IllustrisTNG Project using Hazel Hen, Germany's fastest computer, to understand what drives the formation of galaxies. The image shows the formation history (blue) of the gravitationally collapsed bodies (orange and white). [5]

Four sets of code have been written for this report. The original program was written in Python, a flexible, dynamic, and simple language with fast development but poor computing speed. This was then converted to Cython, a language referred to as a blend of C, C++ and Python [4] to achieve speeds comparable to those of static languages. The third script used the Cython optimised code to apply OpenMP thread based parallelism. The final script optimised the Python code using MPI, a distributed memory parallel system. Analysis of these programs were run using the University of Bristol's High Performance Computing Machine, BlueCrystal Phase 3. The aim of this report is to discuss the background and theory behind

the four programs, the method behind the direct gravity simulation for each parallel system, and discuss the results obtained from the analysis using BlueCrystal.

BACKGROUND AND THEORY

Python:

Python is considered to be in the top 5 most widely used programming languages in the world [6]. Python is an interpreted language optimised for developer productivity. It is an incredibly useful tool for when speed of development matters owing to it being a simple language with dynamic typing. Dynamic typing means the data types are determined by context. Running a Python script requires the program to be converted into a set of byte codes which are then passed through an interpreter. This makes Python a much more flexible language than static type languages such as C where everything has to be defined before use. As a result Python code is often quicker to write as there is greater margin for error in syntax. There are also no compile steps and modules are imported at run time so programs can be run, changed, and rerun to debug and improve code rapidly.

However, for what Python enjoys in rapid development time, it lacks in computational speed. Data type interpretation slows down Python significantly in comparison to compiled program languages. Running large computations on serial Python code can take an order of $100\times$ longer than C compiled code. When deciding the language to use for a project, it is important to determine which of the development or run time costs are larger. However there is an alternative, a middle ground between Python and C/C++ known as Cython.

Cython:

Cython combines the power of C and Python. Cython code has to be compiled into binary which can produce code with speeds comparable to that of C/C++. The language is a derivative of Python. It was started as an offshoot from Pyrex which had a similar goal to what Cython has achieved, namely the speeding up of Python code. Cython is becoming an increasingly well documented language and many projects are beginning to utilise the efficiencies that come with writing in Cython. NumPy contains 5000 lines of Cython, and Sage, the main driver behind the development of Cython, uses 477,000 lines of Cython [7]. As well as improving the speed of serial code by upwards of $100\times$, Cython also adds the possibility to apply thread based parallelism with OpenMP.

OpenMP:

Built into Python is the global interpreter lock (GIL). The GIL forces only one thread to execute python byte codes at any one time. As it is built into Python many Python objects and functions are dependent on the guidelines of the GIL. Naturally if the code is limited to one thread, thread based parallelism becomes impossible. However, Cython is compiled so avoids running with byte code. If sections of code can be entirely compiled into C the GIL can be turned off and multiple threads can run the code simultaneously. To convert Python to code that can be compiled into pure C it is necessary to "c" define (`cdef`) all variables and `cimport` all relevant C functions. If one is certain of the correct operation of their code, turning off certain safety checks such as index bound checking and division by zero can produce even more C like code.

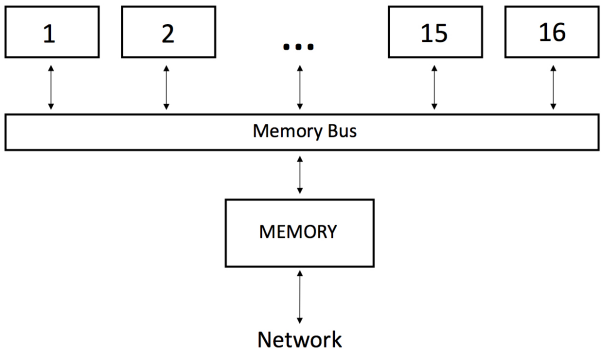

FIG. 3. Shared Memory: Every processor (numbered 1-16) has access to the same pool of shared memory and can run simultaneously on different sections of the same data, communicating via the shared memory. Private variables can also be declared which exist separately on each processor and will not be overwritten.

OpenMP works with shared memory as seen in Figure 3. The code runs on a master thread which when called upon will split the work of the code amongst available processors. A visual representation of this can be seen in Figure 4. The main function that can be used with OpenMP for Cython is the `prange` loop.

Unlike `range` in Python, `prange` can only be used in a `for` loop. The `prange` function must be imported from the `cython.parallel` library using `cimport`. `Prange` turns off the GIL and splits the code contained within the `for` loop over the number of threads specified in the function. Various scheduling types can be assigned to determine how the iterations are split across threads. These include Static, Dynamic, and Guided. Static splits the iterations equally at compile across all threads. The chunk size that each thread has to compute is automatically set as the total iterations divided by the number of threads. This scheduling type is ideal for evenly distributed work loads. Dynamic scheduling distributes chunks of data to threads at run time as they become available. The automatic chunk size is set to one. This is ideal for unevenly distributed work loads. Splitting the iterations incurs a runtime overhead so there will be an optimum chunk size dependant on the total number of

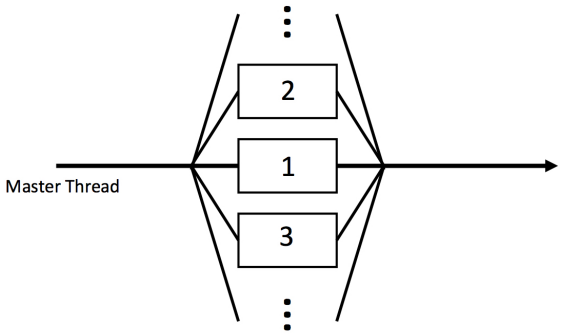

FIG. 4. Visual representation of a thread based parallel program that splits work across multiple processors.

iterations. The final scheduling type is Guided. Here chunks are distributed dynamically with the size proportional to the remaining iterations divided by the number of threads. This is ideal for problems with heavier computations required for the latter iterations.

MPI4PY:

The Message Passing Interface (MPI) is a message passing system used for communication across processors. The processes all run the same same code using MPI's point to point communication to exchange data between each processes private memory. Unlike OpenMP, MPI operates using distributed memory like that seen in Figure 5.

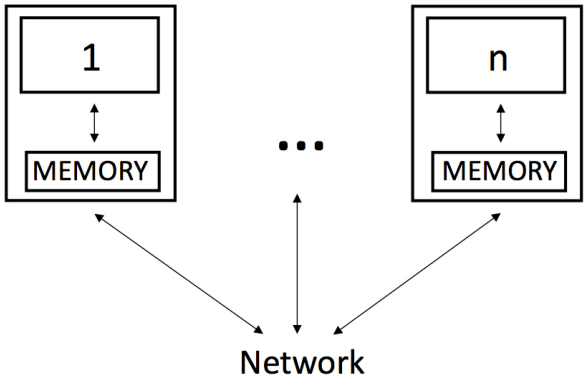

FIG. 5. Distributed Memory: Each process (1-n) has its own private memory and thus any calculations computed by one process must be packaged up and sent as a message to the next process that requires the data. MPI facilitates this communication between processors.

Communication is important when using MPI and it is necessary to understand the dependencies of the problem when writing process based parallel code. There are two types of point to point communication. Blocking and non-blocking message passing. Blocking messages will send out a message from one process to another process and all other work must wait for the communication to complete before continuing with the code. Non-blocking uses an area of memory as a send buffer to allow processing to continue immediately. Messages can also be passed using collective communication. This is another form of blocking message however is highly efficient when you have data that needs to be shared between more than two tasks.

MPI code is generally split into a Master-Worker architecture. This is where a Master process will distribute data to relevant Workers, and ideally perform some work too so not to be wasted waiting around. The Workers should each be working on a different part of the problem or data set and communicating their data to other Workers or back to the Master. It is important to understand the dependencies of each Worker when the problem is being tackled.

Communication between processes also incurs a cost. If Blocking messaging is used tasks can be left waiting around until communication has been completed. There is also a time cost associated with sending a message. Latency refers to the time taken to send a 0 byte message between processes. Bandwidth is the amount of data communicated per unit time. To optimise MPI code small messages should be packaged into larger blocks so as to avoid overhead due to latency delays. This way bandwidth can to optimised and the speed of the code improved.

METHOD

N-body Gravity - Direct Method:

The underlying dynamical model of the gravity simulation is Newtons force law. The force acting on particle i from j is given by:

$$\overrightarrow{F_{ij}} = -G\frac{m_i m_j}{|r_{ij}|^3}(\vec{r_i} - \vec{r_j}) \tag{1}$$

Where G is the gravitational constant, m_i and m_j are the masses of particles i and j, r_{ij} is the absolute distance between particles and $(\vec{r_i} - \vec{r_j})$ is the unit vector distance. Finding $\overrightarrow{F_{ij}}$ for every particle will build up a matrix array of size nxn (where n is the number of particles) from which the total force on each particle can be found. To find the total force on particle i due to its interactions with all other bodies we sum across j. This is given by:

$$\vec{F_i} = \sum_{i \neq j} -G\frac{m_i m_j}{|r_{ij}|^3}(\vec{r_i} - \vec{r_j}) \tag{2}$$

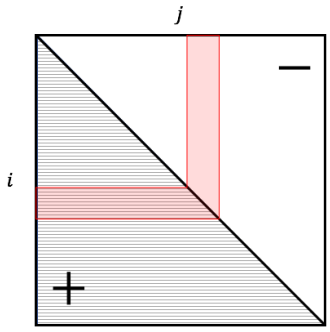

FIG. 6. In the serial code the nxn force matrix is built up using a two dimensional array. To save on computation Newtons third law is applied. This means only the grey shaded region of the 2D array has to be calculated. This is built up by iterating over 0 to i rows and for each row iterating up to the point where $i = j$. Once a row has been calculated it can be mirrored as indicated by the red region and the full nxn array formed.

The problem becomes a discrete integration over a specified time step. Once initial conditions are specified for the positions of the bodies the total force array is found for each time step and the positions and velocities of the particles are updated using the calculated total force. For simplicity and efficiency the code splits the x, y, and z dimensions into a separated structure of arrays and calculates the total force in each dimension seperatly. The velocities and positions are updated using $\vec{a_i} = \vec{F_i}/m_i$ and the equations of motion:

$$\vec{V_D} = \vec{U_D} + \vec{a_D}t \tag{3}$$

$$\vec{pos}_D = \vec{V_D}t + \frac{1}{2}\vec{a_D}t^2 \tag{4}$$

Each set of equations (1, 2, and 3/4) are contained within for loops that iterate around i and j to build up the respective arrays of data. These sets are contained within a while loop which steps with a specified time step which can be optimised for code speed vs accuracy of data. The larger the time step the less accurate the simulation. This while loop is then contained within an outer while loop used for printing to file which can be varied to increase the total run time of the simulation. The benefit of using two nested while loops is that data resolution and the total number of data points printed can be varied separately.

The direct method uses no approximations so every point in the nxn force array has to be calculated. Using Newtons law of equal and opposite forces it is possible to say that $F_{ij} = -F_{ji}$ and thus we reduce the number of calculations required to build up the nxn force array by a factor of two as seen by the grey shaded region of Figure 6.

Equation 1 poses a problem when the distances between two bodies approaches zero. The force will tend to infinity and the resulting velocities will be unphysical. A method for overcoming this is introducing a softening factor (ϵ) [8–10].

$$\vec{F_i} = \sum_{i \neq j} -G\frac{m_i m_j}{(|r_{ij}|^2 + \epsilon^2)^{3/2}}(\vec{r_i} - \vec{r_j}) \tag{5}$$

Where $\epsilon > 0$. This softening factor can be varied depending on the initialisation conditions such that the simulation maintains stability and bodies do not fly off at huge velocities.

As the number of particles (n) being modelled in the simulation increases the complexity scales with n^2. Serial code has to iterate through each row sequentially in order to build up the nxn matrix seen in Figure 6. In order to parallelise our problem we have to identify sections of our code that do not have conflicting dependencies. The while loops cannot be parallelised as the next iteration is dependent on the previous steps calculations. Conveniently though the greatest overhead, the nxn matrix, has no dependencies so it is possible to split the work load across parallel processors.

Parallelising with OpenMP:

The largest overhead in the code was in calculating the elements of the nxn force array. `Prange` was used to parallelise building the nxn array across multiple threads. Each thread was given a certain bit of the data to calculate. Static, Dynamic, and Guided scheduling was used on the matrix as seen in Figure 7, 8, and 9. `Prange` was not used on the loops used to calculate equation sets 2 and 3/4 as the time cost was negligible in comparison to the nxn array. Parallelising these loops also decreases efficiency for smaller simulations as the time cost associated with turning off the GIL outweighed the efficiency improvements of parallelising.

Parallelising with MPI:

To parallelise with MPI the problem was split between Master and Worker processes. The Workers have been assigned the task of calculating sections of the nxn force array as seen in Figure 20. The Master process has the job of sending out the correct bits of data to each Worker i.e. the relevant positions and counters. The counters ensure that each worker is only calculating the section of the nxn array assigned to it by the Master. Each section is equal to the total number of particles divided by the number of processes used. The Master also has the task of calculating the total force on each particle and updating the positions. This ensures it is only sitting idle until Worker 1 is ready to send back its block of data.

The code is set up such that Worker 1 has the shortest compute time and will send back the completed section to the Master first. Once the Master has received Worker 1's data it

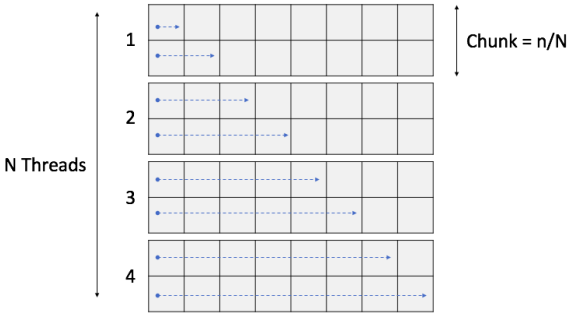

FIG. 7. STATIC: Static scheduling split the rows of the nxn array across a number of threads. The chunk size the threads receive are equal to the number of particles being modelled divided by the number of threads. In this visualisation 4 threads are used. It is clear for our problem that there is a significant imbalance between the work thread 4 has to complete and compared to that of thread 1.

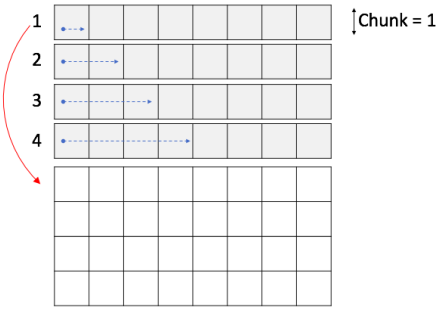

FIG. 8. DYNAMIC: Dynamic scheduling automatically sets chunk size to 1. Each thread will receive one row and once it has finished computing the row it will move on to the next available row to compute as indicated by the red arrow. This is ideal for problems such as computing the nxn array as the load will be balanced equally across all threads. There is however a cost in sending many small bits of data which can be optimised by varying chunk size.

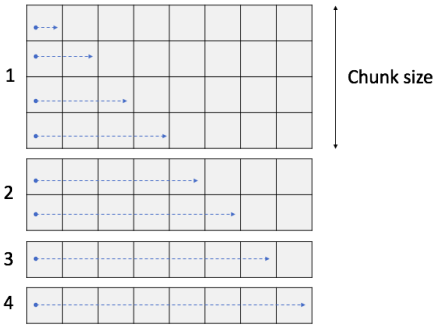

FIG. 9. GUIDED: Guided scheduling splits the work across the threads dynamically with the chunk size proportional to the remaining rows divided by the number of available threads. For a problem such as computing the nxn array where the later rows are more computationally demanding this method could see improvements on other types of scheduling.

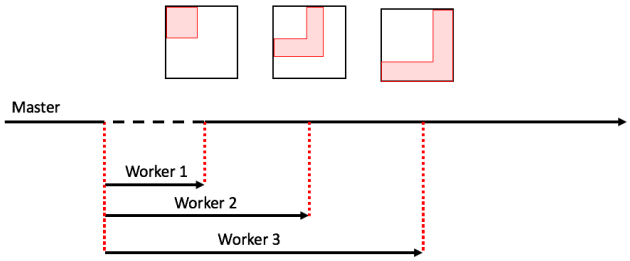

FIG. 10. MPI: The Master starts by sending out the position data and counters (indicated by dotted red lines). In this example 3 Workers are used. The first Worker receives a counter indicating its work load is $0 \leq i < n/3$, the second worker receives, $n/3 \leq i < 2n/3$ and the final worker receives the rest of the work. This is a similar load schedule to OpenMP's Static scheduling. Each Worker sends back a block of data as indicated by the red highlighted squares above. With these the Master can compute the total force on each particle and update the positions/velocities in parallel to the Workers computing.

will begin summing over that block of data to update the total force. Once this is complete for Worker 1's block the Master will request Worker 2's block of data and begin summing the total force again. The Master will skip over the columns already summed and update the total force without overwriting any data. Once the total force is found for each particle and positions/velocities are updated the program will repeat from the start for the next time step.

ANALYSIS

All parallel analysis was conducted on the University of Bristol's HPC, BlueCrystal. The HPC returned more precise data than a regular 2 core laptop and naturally had many more processors available for use. Each test was limited to a run time of 10 minutes. The initialisation conditions were randomised but the range of values used were kept constant across all tests. This meant that the number of particles being modelled could easily be varied while run time was kept reasonably constant. The masses used ranged between 10^{23} kg (Moon size) and 10^{24} kg (Earth size). Initial velocities were set in a range of ± 100 m/s and initial positions were confined to a 3D space with sides of 6×10^8 m.

Figure 11 and 12 show the time improvements associated with Cythonising the serial code and Figure 13 details the specific speed ups that come with each Cython change when testing 10 particles. These tests were run on a 2 core Intel i5 powered laptop as there was no benefit to using the HPC (other than precision).

The analysis of the OpenMP parallel programs can be seen in Figures 14-19. The nxn force array was the only part of the code parallelised for this analysis as this incurred over 90% of the computational overhead without parallelising. Figure 14 shows the percentage of total run time spent on computing the nxn force array for a various number of particles using 16 threads and a dynamic `prange` schedule. Figures 15-18 show the effect on efficiency of increasing the number of threads for various numbers of particles. The data taken for dynamic scheduling used the automatic chunk size of 1. Figure 19 displays the analysis of varying chunk size for dynamic scheduling and the time improvements associated over a various number of threads. This data was all conducted using a simulation of 1000 particles over 1000 time steps of length 100 seconds.

The analysis of MPI code was limited to just one program due to time constraints. The effect of using various numbers of processes and nodes was investigated for 100 particles. The data collected is seen in Figure 20.

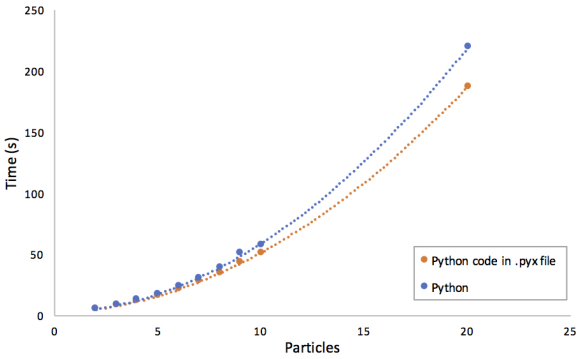

FIG. 11. The orange set of data points represent the total time taken for the Cython compiled Python code. A distinct improvement can be seen with speed ups of up to 15% seen when simulating for 20 particles. This speed up is achieved due to code being converted to the faster binary form through the Cython compiler.

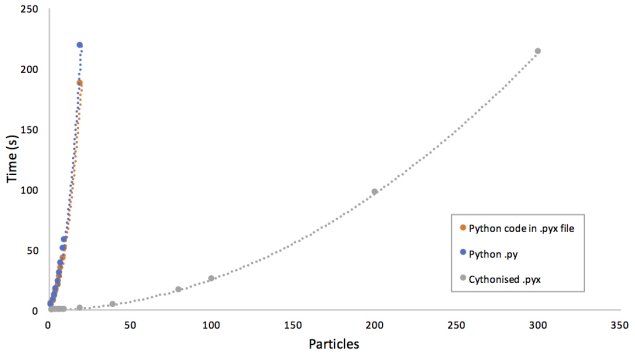

FIG. 12. This figure shows the dramatic performance improvements from fully Cythonising the program. It is possible to scale the simulation to 15X the size of the Python program with similar run times.

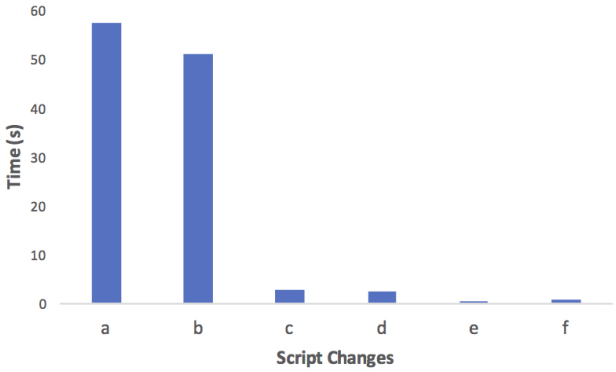

FIG. 13. Depicted are the time improvements for each serial program change when simulating 10 particles. (a) Pure Python, (b) Cython compiled Python program, (c) `cdef` all variables and arrays, (d) Using C library `sqrt`, (e) Replacing numpy functions with equivalent `for` loops, (f) replacing Python ** with C library `pow`

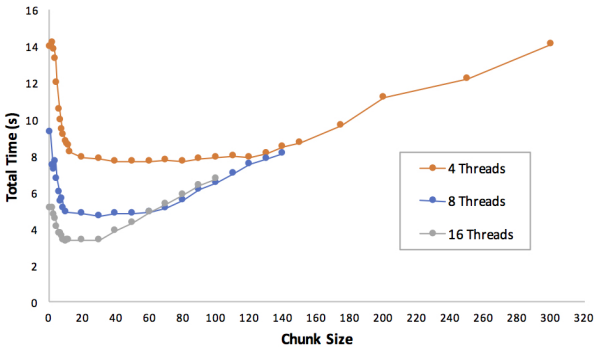

FIG. 19. Varying chunk size for dynamic scheduling. Analysis was conducted on a 1000 particle simulation for a range of threads. For each number of threads there is an optimum chunk size. 4 Threads: optimised with a chunk size between 13 and 120, 8 Threads: between 10 and 60, 16 Threads: between 9 and 30.

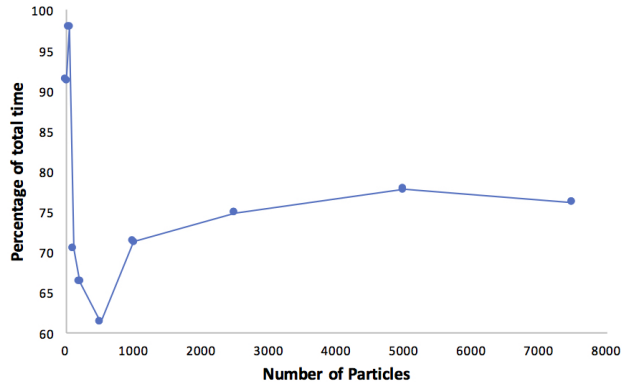

FIG. 14. Figure shows the percentage of total run time spent computing the nxn force array when operating on 16 threads for a dynamic schedule with chunk size equal to 1. Below 100 particles the percentage is high, likely due to costs in frequently accessing the shared memory. Above 500 particles the percentage begins to increase again due to the increasing complexity of the simulation.

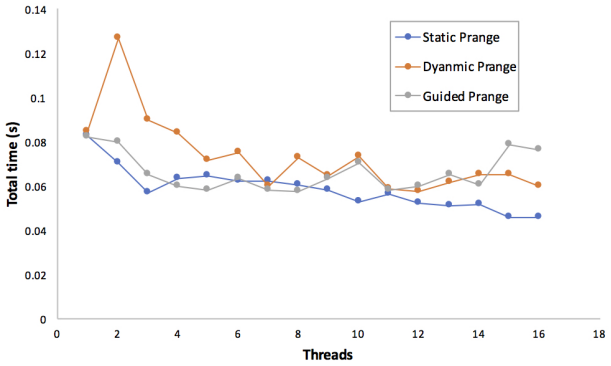

FIG. 16. 60 Particles: The trend seen for time improvements over an increasing number of threads begins to break down when simulating 60 particles and below. Some similarities can be seen with 1000 particles. Dynamic scheduling still increases in time between 1 and 2 threads and the total time does decrease somewhat between 1-16 threads.

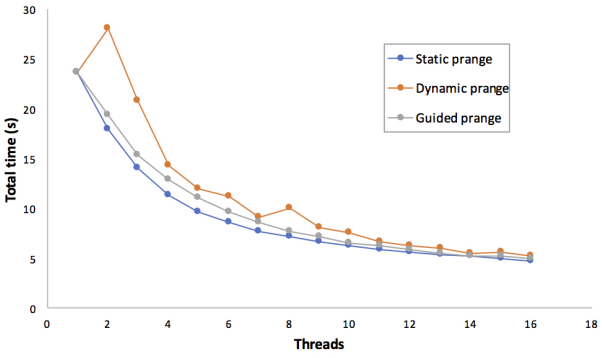

FIG. 15. 1000 Particles: Time improvements for increasing number of threads when simulating 1000 particles with various scheduling types. Dynamic scheduling is the slowest owing to the chunk size being set at 1. This will result in overheads due to threads accessing the shared memory frequently. There is a time increase between 1 and 2 threads which is representative of this. As the number of threads are increased each schedule type follows a similar trend.

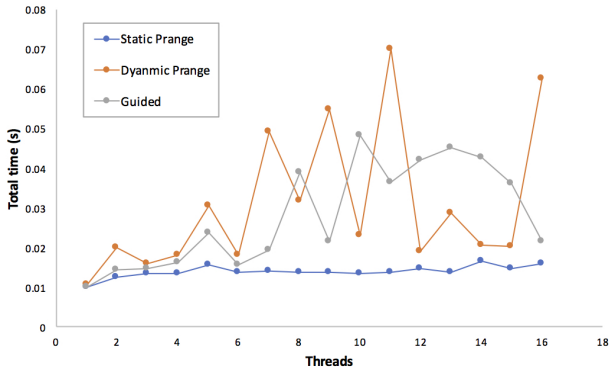

FIG. 17. 20 Particles: The trend completely breaks down around 20 particles and dramatic effects can be seen for guided and dynamic scheduling. Static scheduling doesn't exhibit such wild changes across the number of threads.

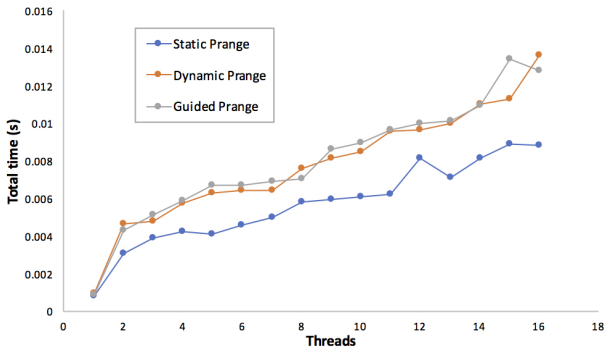

FIG. 18. 3 Particles: The effect of increasing the number of threads beyond the number of particles being modelled can be seen to increase total run time in a linear fashion. This is as expected as beyond the number of particles each thread adds a discrete overhead which does not contribute to computation.

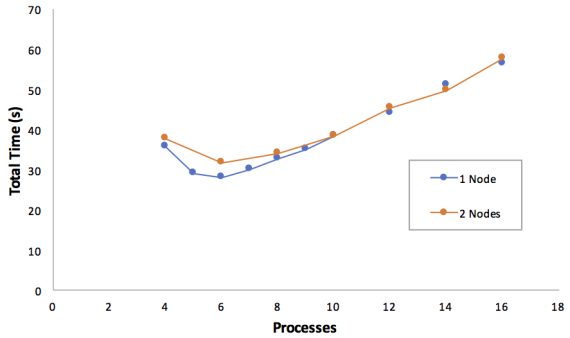

FIG. 20. MPI: The time improvements for a simulation of 100 particles over a range of processes. Using 1 and 2 nodes to compare the communication costs between processes on different nodes. For 100 particles the most efficient number of processes is 6. This is dependant on the scale of simulation. The optimum number of processes will be at the point where communication costs balance with splitting the computation across parallel processes. As expected using 2 nodes is generally slower due to increased communication costs.

DISCUSSION

The best general program, i.e. the program that could be applied to any simulation, turned out to be the static schedule, thread based parallel program. Despite a heavy workload imbalance on the latter threads, static scheduling prevailed as iterations are split between threads at compile time reducing run time overheads. With optimisation for the number of particles being simulated (Figure 19) the dynamic schedule program could out perform static scheduling however there are development overheads which have to be considered.

For 100 particles the comparative program times can be seen in Figure 21. OpenMP's thread based parallelism is by far the most efficient program. Serial Python is naturally the slowest due to data type interpretation overheads. Cython see's significant speed ups due to the compiling of the program into the much faster binary form. The OpenMP program reaps the benefits of Cython's speed ups and thread based parallelism and is thus the fastest code. The MPI parallel program is significantly slower than OpenMP. However, MPI is written using Python which means data type interpretation will slow down the code run time. MPI parallel code is approximately $2\times$ faster than the pure Python program for 100 particles and OpenMP is $3\times$ faster than the Cython program. Thus similar factors of speed up are seen between languages when parallelising. For larger simulations using the OpenMP program speed ups of $5\times$ were achieved between 1 and 16 threads (as seen in Figure 15). Equally large simulations were not possible to run on the MPI program due to the time constraints when using BlueCrystal although it is suspected a similar increase in the factor of speed up will be observed. Examples of the physical simulations can be seen in the appendix.

Program	Time (Seconds)
Python	55.5
Cython	0.23
OpenMP	0.08
MPI	28.1

FIG. 21. Time taken to run each program over 1000 time steps for 100 particles. Optimum times given for OpenMP (16 Threads - Dynamic) and MPI (6 Processes - 1 Node).

Due to time constraints only one MPI program was written as detailed in the MPI method section. Owing to the flexibility of MPI there are many more potential routes to parallelising using Master-Worker architectures. The present program has Worker processes sitting idle once they have completed their section of the nxn array. Future programs could look to spread the workload in computing the total force and updating the positions with available workers. This may incur a cost in communication however for large simulations this may prove a worthwhile improvement. Another method could be to spread the workload required of each worker dynamically to achieve a more even distribution. This could be done by giving each worker a block from the top left and bottom right of the nxn array and working toward the centre until matrix has been computed. Again there may be detrimental communication overheads however it is yet to be tested whether this method could outperform the one submitted with this report.

CONCLUSION

The benefits of parallelising are clear to be seen. Splitting large iterative computations across multiple processes can significantly improve program speed and allows for larger simulations to be run at reasonable times. As the work on simulations grow increasingly more computationally complex and processor power begins to peak the demand to parallelise code will grow. High performance computers such as BlueCrystal which have 10's of nodes each with 16 processors can greatly benefit researchers needing to run simulations which are unfeasible on stand alone machines [11].

The final data collected using BlueCrystal for this report showed that parallelising the direct method gravitational n-body problem was a success. Significant speed ups were achieved using Cython and OpenMP which were particularly easy to parallelise. The simple use of `prange` and Cythonising the Python code resulted in great speed increases. MPI was more complicated to use as it required more thought as to how to utilise parallel processes. Comparing the run times of OpenMP to MPI is somewhat unfair as MPI does not enjoy the speed ups of compiling in Cython. However, similar factors of speed ups due to parallelising were seen between the languages. The recommendation to take from this report is that if it is possible to Cythonise your program to use OpenMP this will achieve the greatest absolute run time improvements. If on the other hand it is not possible to Cythonise the program, similar factors of speed up between serial and parallel can be achieved using MPI.

ATTACHED FILES

1. `Python_Nbody.py`
2. `Nbody_Python_In_Cython.pyx`
3. `Nbody_cythonised.pyx`
4. `Nbody.pyx`
5. `mpi_Nbody.py`
6. `setup.py`
7. `run_Nbody.py`

APPENDIX

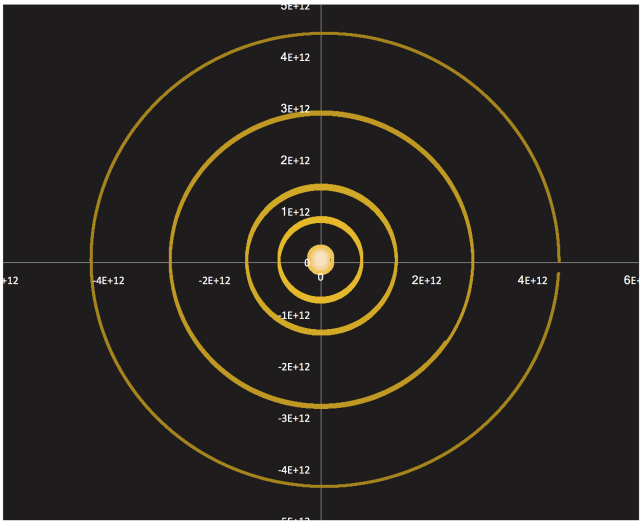

FIG. 22. Stable simulation of the solar system in the x-y plane. Simulated using the Cython program with the Sun centred at the middle and the planets orbiting in a circle around. Evidence that the program produces physical systems.

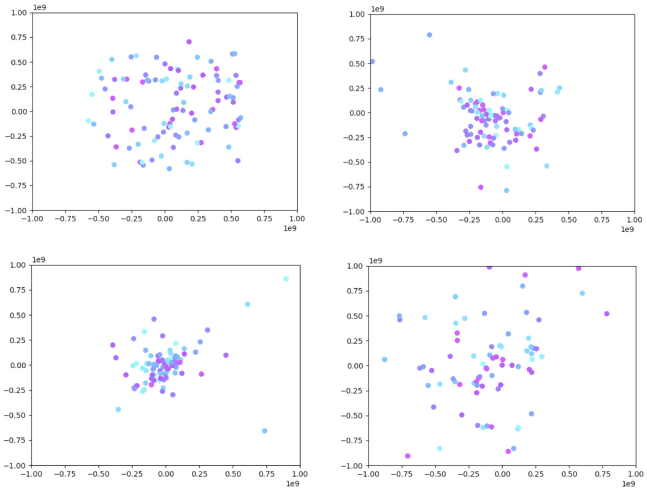

FIG. 23. Matplotlib optimised OpenMP parallel program. Starting from top left working across in rows shows the evolution of a 100 particle simulation. The masses range from 10^{23} kg (light blue) to 10^{24} kg (purple).

BIBLIOGRAPHY

[1] Present, I., 2000. Cramming more components onto integrated circuits. *Readings in computer architecture*, 56.

[2] David House quoted: CNET. (2018). Moore's Law to roll on for another decade. [online] Available at: https://www.cnet.com/news/moores-law-to-roll-on-for-another-decade/ [Accessed 7 Feb. 2018]

[3] Max Roser and Hannah Ritchie (2018) - "Technological Progress". Published online at OurWorldInData.org. Retrieved from: 'https://ourworldindata.org/technological-progress' [Online Resource]

[4] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S. and Smith, K., 2011. Cython: The best of both worlds. *Computing in Science & Engineering,* 13(2), pp.31-39.

[5] Nelson, D., Pillepich, A., Springel, V., Weinberger, R., Hernquist, L., Pakmor, R., Genel, S., Torrey, P., Vogelsberger, M., Kauffmann, G. and Marinacci, F., 2017. First results from the IllustrisTNG simulations: the galaxy color bimodality. *Monthly Notices of the Royal Astronomical Society.*

[6] Lutz, M., 1996. Programming python (Vol. 8). *O'Reilly*.

[7] Smith, K.W., 2015. *Cython: A Guide for Python Programmers*. O'Reilly Media, Inc.

[8] Trenti, M. and Hut, P., 2008. Gravitational N-body simulations. arXiv preprint arXiv:0806.3950.

[9] Aarseth, S.J. and Aarseth, S.J., 2003. *Gravitational N-body simulations: tools and algorithms.* Cambridge University Press.

[10] Nyland, L., Harris, M. and Prins, J., 2007. Fast n-body simulation with cuda. *GPU gems*, 3(1), pp.677-696.

[11] Acrc.bris.ac.uk. (2018). performance - ACRC, University of Bristol,. [online] Available at: https://www.acrc.bris.ac.uk/acrc/performance.htm [Accessed 7 Feb. 2018].