Parallel Gravitational N-Body Simulations: The Direct Method

Geri Nicka
School of Physics, University of Bristol.
(Dated: February 7, 2022)

1. INTRODUCTION

Modern processors have multiple cores, enabling computers to run code faster. This creates the possibility of running particular calculations in an algorithm over multiple parallel cores, creating a significant speedup in its run time [1]. Parallel code is much faster at performing calculations than serial code, where calculations are performed by a single core. It is thus crucial to understand the structure of a processor in order to develop highly optimised code. Every core has its own limited amount of cache memory, the fastest type of memory, which may only be accessed by that core itself. Moving on to a higher dimension, every core in a processor is connected to a shared network of cache memory, where data may be shared between the cores in that network. At the highest dimension, clusters are formed, where individual processors are linked to each other, establishing rapid communication between them. This communication is defined by low latency and high bandwidth.

Parallel code is optimised to spread calculations over several cores, taking advantage of their computational power. While certain processors within a parallel code may operate independently, in some calculations, it is frequently required that cores share information between them to allow other cores to operate relying on that information [2]. In such cases, when a core is expecting information from another core, it will remain idle until it receives that information, reducing performance. Thus, it is essential to balance the load between cores and predict which sections of a calculation require information from other cores to achieve maximal optimisation. In parallel code, communication is a major overhead and any time spent without performing calculations translates to low efficiency.

Even so, there is a limitation to the potential speed up of a parallel code even if there is maximal optimisation. In any code, there will always be sections which cannot be turned into parallel code, acting as a bottleneck [3]. Ultimately, the speed up achieved with parallel code will be limited by the fraction of code which cannot be made parallel, according to Amdahl's Law [4].

In this work, parallel N-body interactions are simulated implementing the direct method. Several research areas in Physics and not only, require expensive computational simulations which may be significantly sped up with parallel code optimisation. For instance, astrophysicists utilise parallel code to model astrophysical processes where the number of particles involved is in the order of tens of thousands [5]. Such physical modelling would be painstakingly slow with serial code simulations, rendering relevant research impossible. The direct method of simulating the gravitational N-body problem in parallel reduces computational complexity, leading to a scaling of N^2, where N is the number of objects. Executing certain processes of the simulation in parallel distributes the total workload across multiple cores, radically improving performance. As a result, simulations of extremely large and more complex systems are made possible.

2. COMPUTATIONAL METHODS

The present N-body simulation of gravitational interactions treats masses as point-like and determines their initial positions and velocities in three dimensions. A separate, independent script generates these initial conditions at the start of each simulation. At each separate time step, the acceleration, velocity and position of each mass are calculated in parallel across all available cores. The simulation models Newton's Law of Gravity [6],

$$\ddot{\mathbf{r}}_i = -G \sum_{j=1, j\neq i}^{N} \frac{m_j \left(\mathbf{r}_i - \mathbf{r}_j\right)}{\left|\mathbf{r}_i - \mathbf{r}_j\right|^3}, \tag{1}$$

which is simply the equation of motion of interacting masses in three dimensions. In the expression above, $\mathbf{r}_j$ is the position vector of mass m_j, G is the gravitational constant and $\mathbf{r}_i$ the position of a mass with index i. In the case of planetary orbits, units are scaled for simplicity. Thus, the gravitational constant G is set to one and the acceleration of the particle $\ddot{\mathbf{r}}_i$ is treated as the force per unit mass. Eq. 1 results in a set of 3N second order differential equations, which may be solved when expressed as a system of 6N first order differential equations [7]. There are several numerical methods that lead to the solution of such a system. In this work, the Leapfrog second order method is a adapted; a synthetic integrator which leads to better energy conservation than other methods like the Euler integration [8]. In the leapfrog method, integration depends on the calculation of position at the midpoint, following the procedure below:

$$\mathbf{r}_{1/2} = \mathbf{r}_0 + \frac{h}{2}\mathbf{v}_0, \tag{2}$$

$$\mathbf{v}_1 = \mathbf{v}_0 + h\mathbf{a}\left(\mathbf{r}_{1/2}\right), \tag{3}$$

$$\mathbf{r}_1 = \mathbf{r}_{1/2} + \frac{h}{2}\mathbf{v}_1, \tag{4}$$

where $\mathbf{r}$, $\mathbf{v}$ and $\mathbf{a}$ are the position, velocity and acceleration of the masses respectively. Indices 0, 1/2 and 1 correspond to the start, mid-point and end of an interval over a certain time step of size h. From the Newtonian gravitational potential, proportional to $1/r$, it is known that a singularity arises when two masses are in very close proximity. In such a situation, gravitational potential energy is converted into kinetic energy, accelerating the masses dramatically. As a result the masses gain a velocity much greater than their escape velocity, repelling each other violently. This overshoot may be fixed by introducing a softening factor ϵ [9] greater than zero in the denominator of Eq. 1. giving

$$\ddot{\mathbf{r}}_i = -G \sum_{j=1, j\neq i}^{N} \frac{m_j \left(\mathbf{r}_i - \mathbf{r}_j\right)}{\left(\left|\mathbf{r}_i - \mathbf{r}_j\right|^2 + \epsilon^2\right)^{3/2}}, \tag{5}$$

where all variables are the same as in Eq. 1. It is crucial that the softening factor is small enough so that it does not impact the dynamics of the system when masses are at a large distance from each other. At the same time, it must be ensured that the softening factor is large enough to prohibit nonphysical effects in the simulation. In the case of simple systems like planetary orbits, where close encounters do not occur, the softening factor may be omitted.

2.1 MPI - Distributed Memory

In computer programming, distributed memory describes a network of pairs of cores and memory. In such a network, memory may only be accessed by the core it is paired to (Fig. 1). A script being run on such a pair is called a process or task. The Message Passing Interface (MPI) [10] is a library of functions which allows processes on different core-memory pairs to communicate with other pairs by sending and receiving messages. However, as mentioned above, memory may only be accessed by a core paired to it, forcing cores to communicate with each other. In the case of excessive communication and poor optimisation, the performance and efficiency of a parallel algorithm is decreased. In the present implementation, two major areas were considered to minimise the adverse effects of communication on efficiency. It is important to note that the initiation of communication between cores, referred to as latency, takes practically no time. First, the most efficient way of establishing communication was chosen. More explicitly, the **Bcast()**, **Gather()** and **Allgatherv()** instances were used instead of the standard '**send**' and '**receive**' commands. These were set to operate in a tree structure to send a message, with **Allgatherv()** being used at every step where processes needed to exchange the positions that they calculated. Secondly, the format of the exchanged information was considered. The adaptation of multidimensional arrays is definitely extremely expensive in terms of communication, being orders of magnitude slower than one-dimensional arrays. Arrays containing the positions of the mass along the x,y and z directions were collected and stored in a 1D array instead of three, eradicating excess communication between the cores. This reduced the number of cache misses where expected data from a core is out of the cache. Upon reception, the 1D array containing all positions across all dimensions is split into three arrays, out of which, every core picks the positions required to calculate the acceleration of a mass. After all accelerations are calculated, the arrays are combined into one array accessible by all cores, once again minimising communication. All cores are tasked with the computation including the root core itself. Nonetheless, only the root core is allowed to generate the initial conditions of the system, store the positions at each step and save the data upon termination of the simulation.

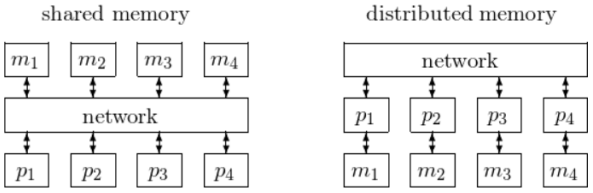

FIG. 1. In shared memory, memory is accessible to every core. On the other hand, in distributed memory, every core has its own memory, accessible to other cores via communication.

2.2 OpenMP - Shared Memory

Unlike distributed memory, shared memory refers to a network where all cores are connected to a universal memory, with every core having access to it (Fig. 1). Algorithms running in a shared memory network are referred to as threads. The benefit of shared memory is that no communication between cores is required as opposed to distributed memory, eliminating communication overhead. There are however, disadvantages. First, only one core may write on an element of memory at a time, leading cores to 'compete' in an effort to write on an element of memory first. Secondly, as every core has access to the shared memory, the data written in a given element may become hard to predict unless there is some sort of coordination. It must be ensured that cores write to elements of memory which are not correlated and if they need to apply changes to the same element they must do so in a successive manner rather than simultaneously. Ultimately, the overhead arising in shared memory is attributed to cores being queued to apply changes in memory.

Being and API, OpenMP [11] requires a C or C++ compiler to become executable. As such, the present work uses the Python programming language to create the simulation and then adapts it in Cython, providing a translation into C or C++ source code. The most important element in the code is the use of **prange** in special 'for loops', whose iterations are partitioned across different cores. These cores have to work independently so the efficiency is determined from the load

balance between the cores. For the OpenMP version it is important to consider Newton's third Law,

$$F_{i,j} = -F_{j,i} \tag{6}$$

where $F_{i,j}$ is the force acted upon mass j from mass i. Consequently, the matrix of accelerations by pairs of masses is anti-symmetric and the computation of its upper or lower half diagonally is adequate. The other half can then be filled with the corresponding opposite values, reducing the amount of calculations by half. Several implementations of this anti-symmetric matrix were explored as far as load balancing is concerned.

Initially, symmetry was not exploited. This resulted in twice as many calculations compared to the symmetric implementation but since every line of the matrix represents the same number of accelerations, the load was balanced fairly between the cores. The advantage of this method is that the positions on each line may be summed immediately after computation, constructing the total acceleration for every mass.

Next, a first version of the symmetric implementation was explored. Here, it was important to sum over all accelerations of a particle on a line carefully, as only half of the matrix is calculated. This means the majority of the lines will contain accelerations computed by different cores. For that reason, half of the matrix must be computed first and after completion the other anti-symmetric half can be added to complete the accelerations matrix. In this symmetric implementation, load balancing is not guaranteed. The initial implementation of symmetry ensures a good load balancing by computing positions (i, j) if $i+j$ is even in the upper diagonal half or odd in the lower diagonal half. Two other variations of the symmetric implementation may be explored. One is the declaration of a temporary matrix in the main function of the algorithm and not in the acceleration computation function. The other is testing the implementation of scheduling and chunk size.

3. PHYSICAL INTERPRETATION AND GRAPHICAL RESULTS

In the present work, the Leapfrog method is implemented to solve the first order differential equations arising from Eq. 5 and simulate a system of randomly distributed masses whose velocities are low and random. The masses eventually drift towards the center of mass of the system, demonstrating the effect of masses in close proximity on the conservation of energy. Ultimately, the benefit of the Leapfrog method and the softening factor are discussed. Interesting topics to explore are the manner in which energy is conserved, the impact of the time step size and the value of the softening factor.

A graphical interpretation of the obtained results from the simulations was generated to validate their physicality. Fig. 2 is a snapshot of the simulation demonstrating the behavior expected theoretically. Masses collapse towards the center of mass of the system, orbiting away from it in elliptical orbits. Several of the masses are trapped in the gravitational potential well and keep orbiting at the center of the system. Another method of validating the physical relevance of the simulation was to check if the energy of the system was conserved. Fig. 2 shows that the energy of the system is conserved relatively well at first but as masses approach each other, their velocities and accelerations overshoot, imposing limits to the Leapfrog integration method. These overshoots manifest themselves in Fig. 2 as narrow spikes in the energy plots.

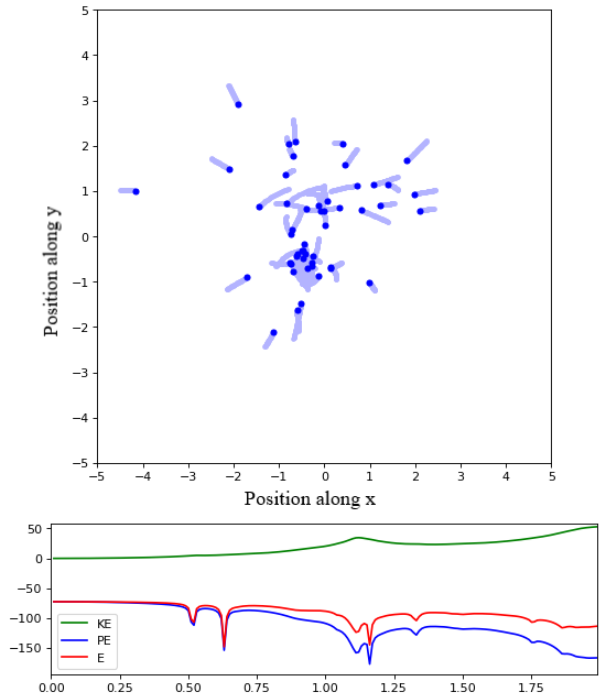

FIG. 2. (**TOP**):Masses move towards the center of mass of the system and orbit away from it when they get too close. Some of the masses keep orbiting around the center of mass. (**BOTTOM**): Plots of kinetic, potential and total energy of the system versus time. Spikes in energies result from the discussed limitations of the Leapfrog method, where the distance between masses is very small.

MPI results

Automated submission scripts for the simulations were created with the aim to obtain a large number of data for analysis. These submission scripts were run on Blue Crystal Phase 4 (BC4) for systems ranging from 500 to 9000 masses. Each time, 2 up to 28 cores were tasked for each number of masses, obtaining a vast amount of data. The availability of such data permitted extensive analysis of the performance of the simulations and allowed the prediction of their behavior at later times using appropriate fitting. First, computation time versus number of cores was plotted, fitted with an inverse power law $a + cx^{-n}$, where x is the number of cores and a, c, n the fit parameters (Fig. 3). Secondly, computation time versus number of objects was plotted, fitted with a power law cx^n, where

x is the number of objects and c, n the fit parameters (Fig. 4). It is established that the complexity of N-body problem simulations scales with N^2, where N is the number of objects. Lastly, a plot of number of cores versus number of objects was also created for a fixed computation time (Fig. 5).

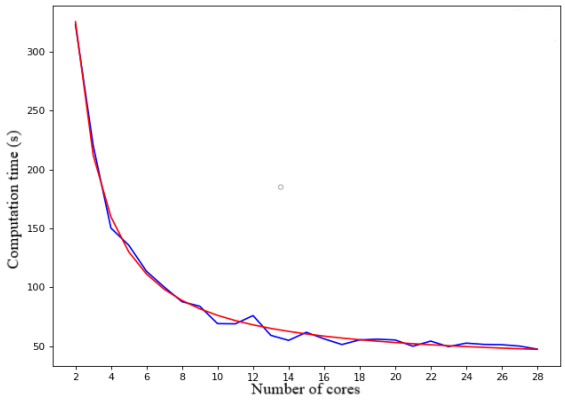

FIG. 3. Computation time versus number of cores for 9000 objects over 200 time steps fitted with an inverse power law $a + cx^{-n}$ (Data in blue and fit in red). Fit parameters are $a \approx 36$, $c \approx 678$ and $n = 1.2$.

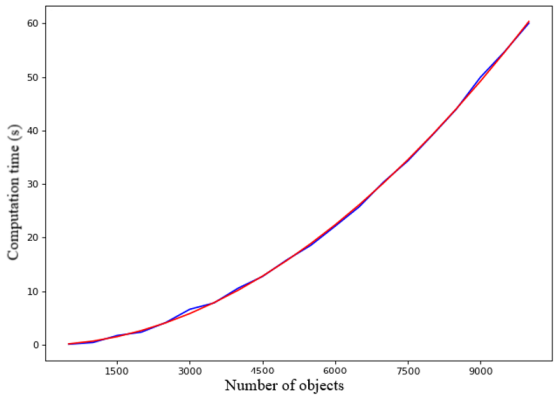

FIG. 4. Computation time versus number of objects on 27 cores over 200 time steps on BC4 fitted with a power law cx^n (Data in blue and fit in red). Fit parameters are $c \approx 10^{-6}$ and $n = 1.94$. The fit validates the expected scaling of the problem with to the power 2.

Fig. 5 demonstrates that the number of cores and number of objects are related in a chosen fixed computation time. It is seen that this relation steepens for a number of cores larger than 15. This can be explained by the architecture of the BC4 nodes. Each node is composed of two units, each of which has 14 cores. Thus beyond 15 cores, these two units need to communicate, creating significant overhead in performance. The anomalous behavior in Fig. 5 clearly demonstrates that.

This work also aimed to explore the effects of communication in simulations where the number of objects was very small to show that algorithms are worth running in parallel only for large and complex systems. However, after the BC4 update, it has become increasingly difficult to obtain reproducible results due to technical issues with the job scheduler. In the future, such testing would yield interesting results, hopefully confirming the suitability of parallel algorithms for simulations of large systems.

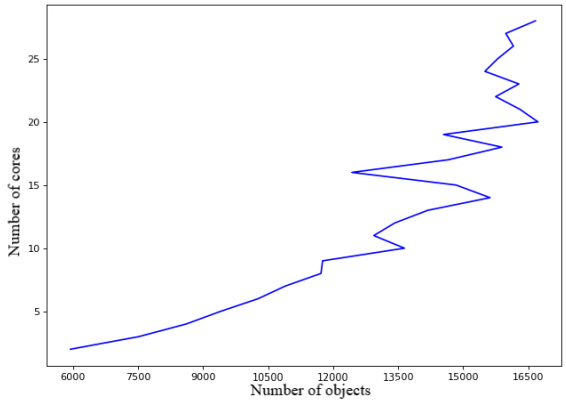

FIG. 5. Number of cores versus number of objects for a fixed computation time of 150 seconds over 200 time steps. Predicts how many cores are required to simulate a system with a given number masses in exactly 150 seconds. Beyond 15 cores, communication overhead becomes significant.

OpenMP results

The same analysis for the MPI method was also performed for the OpenMP method, confirming the scaling of the N-body problem once more.

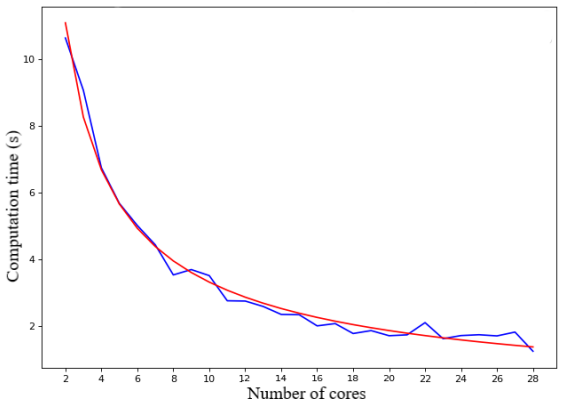

FIG. 6. Computation time versus number of cores for 3000 objects over 200 time steps fitted with an inverse power law $a + cx^{-n}$ (Data in blue and fit in red). Fit parameters are $a \approx -0.5$, $c \approx 18$ and $n = 0.7$.

Analysing data from Fig. 6 reveals that OpenMP simulations are significantly faster than MPI simulations, with OpenMP being approximately 2.7 times faster on 2 cores and 5 times faster on 28 cores. Here it is important to note that data

collection for OpenMP was performed after the BC4 update and as mentioned above, the job scheduler tended to meddle with the data due to technical issues. As such direct comparison by inspection between Fig. 3 and Fig. 6 was not possible. Nevertheless, the goodness of the fits allowed this comparison though predictive modelling, knowing that the problem scales with N^2, where N is the number of objects. The performance comparison between MPI and OpenMP agrees with previous attempts in the work of others [11].

Moving back to the exploitation of symmetry in OpenMP, the methods described earlier were tested on Dell G3 with 6 cores for 2000 objects. The odd/even method was approximately 5 times slower than the non-symmetric method. This was attributed to the fact that matrix elements which were not computed need to be updated, resulting in decreased performance. More explicitly, several cache misses occured. Defining the matrix in the main function instead of the acceleration calculator function provided substantial performance improvement but still remained 3 times slower than the non-symmetric method. However, when using a static, imbalanced schedule, performance was moderately increased, although it was still around 2 times slower than the non-symmetric version. In an attempt to explore the dynamic schedule with default chunk size, computational time was about 1.6 times slower than the non-symmetric version.

Unfortunately, it was impossible to explore the optimal chunk size for dynamic scheduling due to the highly variable results BC4 produced and the failure of multiple job submissions after the latest update.

5. CONCLUSION

The benefits of parallel programming have become clear in this work. Distributing expensive iterative computations across multiple cores can dramatically enhance performance, allowing larger simulations to be completed at viable times. As modern research heavily relies on computationally expensive simulations, the demand for parallel algorithms has increased.

The simulations and analysis in the present work demonstrate that parallel algorithms of the gravitational N-body problem using the direct method were successful, confirming the scaling of the problem with N^2, where N is the number of objects. The most significant speed ups were achieved with OpenMP, beating MPI by 5 times at maximum utilisation of the 28 cores on BC4. MPI required a more complicated procedure to achieve maximal optimisation than OpenMP and even so, it did not manage to beat it in performance. This is partly due to communication overheads and Amdahl's Law but also because MPI does not enjoy the speedups of compiling in Cython, as opposed to OpenMP. It has been shown that even though OpenMP is relatively easier to implement than MPI, it achieves the greatest absolute performance. On the other hand, if compiling a Python program into Cython is not possible, similar speedups between serial and parallel code may be achieved using MPI.

[1] Langdon, G. (1990). Book review: Highly Parallel Computing by George Almasi and Allan Gotlieb (Benjamin/Cummings, 1989). ACM SIGARCH Computer Architecture News, 18(4), 90. doi: 10.1145/121973.773543.

[2] Tutorials — HPC @ LLNL. (2022). Retrieved 7 February 2022, from https://hpc.llnl.gov/documentation/tutorials.

[3] Hill, M., Marty, M. (2008). Amdahl's Law in the Multicore Era. Computer, 41(7), 33-38. doi: 10.1109/mc.2008.209.

[4] Rodgers, D. (1985). Improvements in multiprocessor system design. ACM SIGARCH Computer Architecture News, 13(3), 225-231. doi: 10.1145/327070.327215.

[5] Hamada, T., Nitadori, K. (2010). 190 TFlops Astrophysical N-body Simulation on a Cluster of GPUs. 2010 ACM/IEEE International Conference For High Performance Computing, Networking, Storage And Analysis. doi: 10.1109/sc.2010.1.

[6] Category:Newton's law of universal gravitation - Wikimedia Commons. (2022). Retrieved 7 February 2022, from https://commons.wikimedia.org/wiki/Category:Newton

[7] Dennis G. Zill (15 March 2012). A First Course in Differential Equations with Modeling Applications. Cengage Learning. ISBN 978-1-285-40110-2.

[8] C. K. Birdsall and A. B. Langdon, Plasma Physics via Computer Simulations, McGraw-Hill Book Company, 1985, p. 56.

[9] MPI. (1993). Proceedings Of The 1993 ACM/IEEE Conference On Supercomputing - Supercomputing '93. doi: 10.1145/169627.169855.

[10] "About the OpenMP ARB and". OpenMP.org. 2013-07-11. Archived from the original on 2013-08-09. Retrieved 2013-08-14.

[11] Rocchetti, N., Nesmachnow, S., Tancredi, G. (2019). Comparison of Tree Based Strategies for Parallel Simulation of Self-gravity in Agglomerates. Communications In Computer And Information Science, 141-156. doi: 10.1007/978-3-030-16205-$4_1$1.