# A Parallel Approach to simulating Gravitational N-Body simulations

Harry Jordan
*Department of Physics, University of Bristol*

(Dated: February 7, 2022)

Directly calculating N-body simulations is computationally expensive, therefore it is often practical to use parallel techniques in order to fully utilize the given hardware available. This report aims to compare an OpenMP parallel approach with different serial counterparts. Efficiency data for the OpenMP simulation is included, along with comparison to a potential MPI method.

## INTRODUCTION

Parallel computation is a type of computing where multiple processes or calculations are carried out simultaneously [1]. Parallel computing has long been used for high performance computing (HPC) however due to the physical restraints preventing frequency scaling [2] (higher frequency processors), parallel computing has become the dominant architecture as it allows for higher performance as well as lower heat and power consumption. This has mainly been achieved through the use of multi-core processors [3]. To better understand parallel processing, it is useful analyse the structure of a multi-core processor. A multi-core processor (as the name implies) is processor that contains multiple cores, with each core having access to a small amount of fast memory known as cache memory. However this memory can only be accessed by that core, so in order to perform parallel computation, the cores need to be able to send and receive data between each other. In a multi-core processor, this is achieved using another layer of cache that each core can access. Another potential approach can be through a computing cluster. A cluster is a collection of individual processors that are linked in such a way that they can quickly communicate with each other. Both of these approaches achieve a similar result, by dividing the workload and spreading them to different cores the computation time can be significantly reduced.

Certain tasks can be programmed in parallel as they have no data dependency, that is to say that at no point do any of the tasks have any influence on the outcomes of the other tasks. These are so called ”Embarrassingly parallel” problems. However for many problems, there are data dependencies between different tasks [4]. The communication between different tasks is a major source of overheads, therefore it is important to analyse what parts of the calculation need to communicate and which parts do not. There are also ultimately limits on the amount of potential performance gains that parallel programming can provide. If there are parts of the program that cannot be coded in parallel, then they will ultimately become a bottle neck. This can be described from Amdahl’s law:

$$T_p = T_1 \left( F_s + \frac{F_p}{p} \right) \tag{1}$$

Where $F_s$ and $F_p]$ are the fractions of sequential and parallel code, $p$ is the number of processors, and $T_1$ is the current execution of time of the program.

## N-BODY SIMULATION

The simulation begins by generating random positions for $n$ number of particles, with the velocity of each particle being set to zero. These parameters are made into two separate arrays for position and velocity respectively on the master core. The code then splits into parallel to calculate the forces between each particle pair. The force between each particle is calculated through this equation.

$$F_{ij} = \frac{G m_i m_j \left( \mathbf{q}_j - \mathbf{q}_i \right)}{\left\| \mathbf{q}_j - \mathbf{q}_i \right\|^3} \tag{2}$$

where $F_{ij}$ is the force between the two particles i and j, $G$ is the gravitational constant, $m_i$ and $m_j$ is the mass of particles i and j respectively (for the cloud of particles, $G$ and $m$ is set to 1) and $q_i$ and $q_j$ is the position of the two particles. The program calculates the difference between the individual particles and all other particles. This difference is split into a x, y and z component. These position components then calculate there respective force compoents. The force is then used to calculate the velocity and position through these equations of motion.

$$v'_x = v_x + \frac{F_x}{M} t \tag{3}$$

$$x' = x + v_x t + \frac{1}{2} \frac{F_x}{M} dt^2 \tag{4}$$

where $v'_x$ and $x'$ is the new velocity component in the $x$ direction and $x$ coordinate respectively with $dt$ being the time step value defined inside the function. These steps are run in a loop up to a defined time value ($t$) therefore a smaller value of $dt$ will lead to a more accurate simulation but will require more computation to complete.

One problem with this N-body simulation is that the particles are collision less therefore there is a need to limit the spatial resolution of the gravitational force. This is because the Newtonian potential has a $1/r$ singularity [5], therefore at where particles get closer that that that would normall cuase a collision if they had a defined size, the force of acceleration for a given time step would cause the particle to 'jump' to a new position. This means that at very small distances, particles can effectively repel each other. To prevent this, an new term is added to the force equation $\eta$ which is known as the softening parameter. This value is situation dependent, $\eta$ must be small enough to not effect interactions over large distances but large enough to stop the aforementioned singularity. The optimum value for $\eta$ can be approximated [6] as being a factor 1.5-2 smaller than the average distance between particles at the densest regions. However when Rodionov and  tried to implement a varying softening parameter in code, they found such approaches had limited value as it leads to increases with potential energy over time.

### Shared memory: OpenMP

To parallel program using shared memory, a widely used technology is OpenMP. OpenMP is an Application programming interface that allows programs to access the shared memory cache between cores. This is the major advantage of OpenMP as there is no need to specify communication as all processors have the same shared memory. There is still some overhead due to communication, as cores can only have access to part of the information at once, but this overhead is much lower than for a distributed memory system.

Instead of communication, the major source of overheads for the shared memory approach is load distribution. OpenMP code works by starting with a single master thread that splits into multiple parallel threads. These parallel threads compute a section of the calculation and come back together into serial when each section is complete. In an ideal case, the amount of work done on each section is exactly the same, therefore each core finishes at exactly the same time. However if the load is unbalanced, cores will need to wait for the other cores to finish in order to stay synchronised. However, OpenMP makes it possible to dynamically set the work load for each core. This will increase performance as different cores will have to wait less time for all cores to complete there tasks. Comparing this to MPI, it may seem possible to produce the same effect on a distributed memory system however as stated previously, communication is a big overhead so the same approach could not be implemented without losing performance overall.

There are also practical limitations for shared memory systems, high bandwidth, low latency cross bars(shared memory between cores) is expensive [7] and cache-coherancy is hard to maintain at scale.

### Distributed memory: MPI

For this program, MPI has not been used however it is worth discussing for the sake of comparison to OpenMP. OpenMP can only work for when the cores can access a shared memory cache. This is fine for a multi-processor but for a computing cluster this is not possible. MPI or Message Passing interface is a message passing standard that allows for nodes on a cluster to communicate and run code in parallel. This is the major advantage of MPI compared OpenMP as it is scalable for potentially much more cores than can be allowed for a multi-core processor. For example, BlueCrystal 4 has 525 nodes each with 28 cores. OpenMP can only access 28 processes whereas MPI can potentially access 525.

The disadvantage of MPI is that sharing data between cores has to be directly specified unlike OpenMP which is automated. This also means that load balancing has to be preplanned. Another disadvantage to using MPI is the overhead generated through communication between the cores. The actual communication event itself is not responsible for the actual overhead but the sending and waiting for data to arrive between cores wastes time, leading to slower performance.

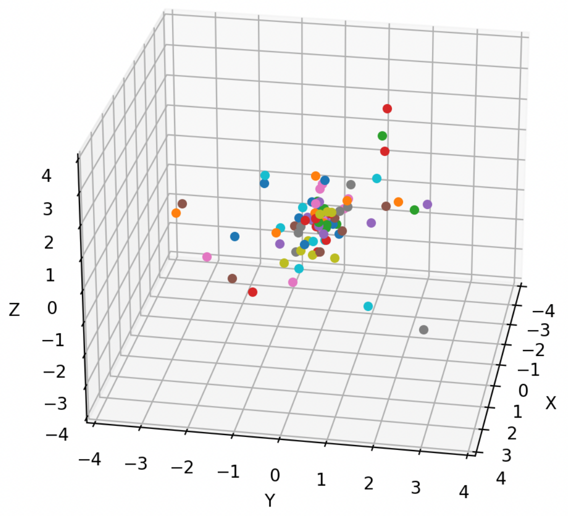

FIG. 1. A single frame from the 3D animation of 100 particles simulation. Each particle in this simulation has equal mass and starts with a random position and is stationary. Axes are arbitrary length units

## COMPARING SIMULATIONS

### Graphics

Calculations for the simulations have been coded in python, which proved useful as it allowed the simulation and graphics to be produced seamlessly through matplotlib. To generate the animations, the position of each timesetp of the simulation was saved to a 3d-array and then using the matplotlib.animation module, each position is iterated though to create the animation. Figure 1 displays a single frame from the 3D animation.

## Serial Approaches

As stated previously, the calculations for the simulations were coded in python. Python 3 is a great general purpose language however it is not as fast as other languages, namely C++ or Fortran. As well as this, python does not immediately have access to OpenMP as it is designed to work for C++. Therefore even for a single core, there is still a large amount of room for increased performance.

### *Numpy*

The first major problem for python in terms of performance is that it is an interpreted language and not a compiled one. This is advantageous in some aspects but because interpreted languages need translated before they are run, this increases total execution time. This means that control flow statements(mainly for loops) are significantly slower. In the original python code, a nested for loop is used to iterate through all particles which leads to a large performance drop.

However, because the code uses arrays to store variables, the nested for loop can be changed into a single for loop using Numpy vectorisation. Vectorisation in this instance refers to SIMD (single instruction, multiple data) action which means that a single instruction performs the same operation on multiple operands in parallel, allowing a for loop to be removed.

### *Numba*

Despite the improvement from Numpy, the program cannot compete with a similar program written in C++ and it is still not possible to use OpenMP to utilize parallel processing. To fix this, it is possible to convert python code to C++ using Cython, a super set of the python language that give C like performance. However, Cython requires a good understanding of C++ to use effectively which undermines its use. An alternative to cython is Numba which allows compilation into machine code with little intervention. As well as this, numba allows access to OpenMP through the use of the prange function.

A comparison of these three serial approaches can be seen in figure 2, where the code was timed using the time.time() function in python. For the serial program, the simulations were run on a Apple M1 chip processor running at 3.2 GHz. The errors in each plot are from the standard deviation in the mean result from 3 runs. These differences in run time are caused from background processes running on the same cores which will slow the simulation.

For low $n$, the strictly python version runs slightly faster than the numpy version. This is because in general, numpy functions are more computationally expensive per function call however numpy arrays are very efficient, therefore despite the functions being more computationally expensive, the speed of the arrays compensates for it.

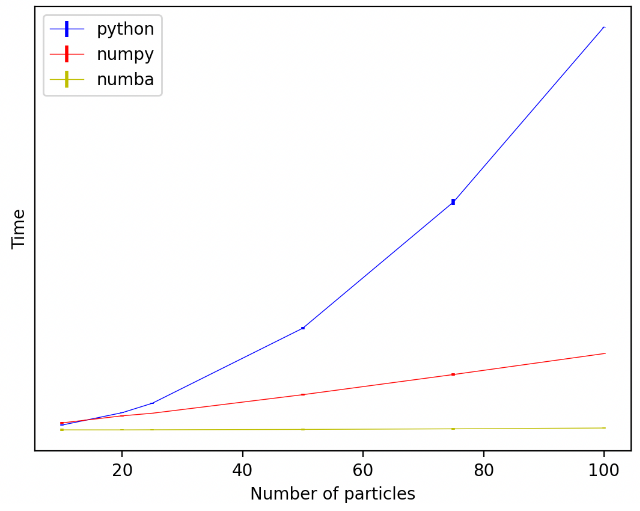

FIG. 2. Comparing the time taken to complete the simulation calculation for a range of $n$. The python time data shows an $n^2$ relationship which is what is expected from a $O(n^2)$ calculation.

From the data it is obvious that the performance of the Numba approach is superior. The speedup for 100 particles from the python approach is a 230x speedup.

## OpenMP

As with the serial code, the simulations in this section were calculated on a Apple M1 chip running at 3.2Ghz, with the results of which being seen in figure 3. For parallel simulations, it is more useful to refer to the efficiency of the parallel program, $\eta$, which is calculated from this equation:

$$\eta = \frac{T_1}{T_N} \tag{5}$$

where $T_N$ is the time taken to run with N cores and $T_1$ is the time take to run with one core. This therefore gives a comparison between the serial and parallel approach regardless of calculation.

This graph comparing a high core count and a low core count shows what is already known with shared memory approaches. For a very small number of particles (n=10) the serial calculation is faster than the 2 core calculation. This is even more apparent with the 8 core calculation as this cant beat serial approach even up to 400 particles. This shows how the communication overhead for can be more than the time saved running calculations. However this for n=8 the time difference is abnormally high. This could most likely be due to not using the omp_get_wtime() which could effect the time results. This function was not able to be used as Numba no longer has support for OpenMP functions . However there would still be a time difference between the high and low number of cores. This is due to the individual cache memory having a lower latency than the shared memory cache for

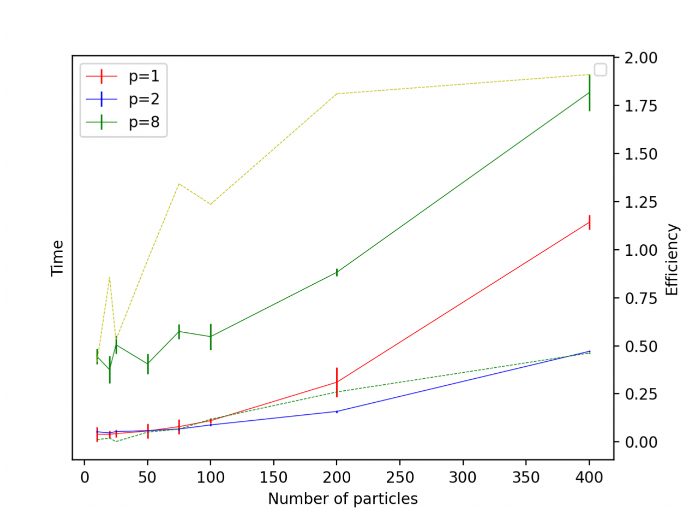

FIG. 3. Comparing a OpenMP parallel process running in 1,2 and 8 thread modes for $n$ number of particles. The solid lines represent the plot of the execution time and the dashed lines represent the efficiency.

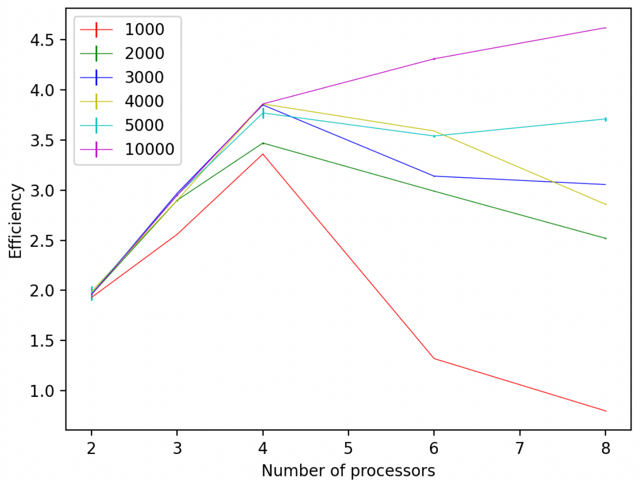

FIG. 4. A graph that shows efficiency against number of cores for different values of $n$

all processors. Despite having not been tested, it is generally agreeable that an OpenMP approach would have greater performance than an MPI approach. This is again due to the overhead of communication in trying to perform parallel calculations being greater than any time saved, which effect MPI greater as its communication has more overheads than OpenMP.

## Scaling

Due to problems in running tasks on BlueCrystal phase 4, it is not possible to show the effects of large numbers of processors however from the data already gathered it is possible to still see a general overview of what is expected. Figure 4 demonstrates this over a range of 8 cores.

This diagram shows that depending on the number of particles, there is an optimum number of cores for calculating it. As the number of processors increases, the efficiency of all $n$ does increase as of course the more processors you have, the more the workload can be divided and therefore the performance of the task will be faster. At a certain number of processors, the efficiency reaches a peak value and for all except n=1000, this peak generates a general plateau of efficiency, all though generally in the downward direction. If the number of processors keeps increasing, there will be a point where the extra communication necessary between cores outweighs the work done by the new cores, leading to the efficiency to decrease. However the decrease in efficiency over the number of processors happens much faster than expected, especially for OpenMP with its lower latency. This again could be due to the use of the improper timing function. This effect would also be more apparent for MPI as the overheads are much greater.

## Discussion

The results from OpenMP were not as expected, as the low latency memory should allow the number for the number processors and efficiency be directly proportional if the code can be written 100 percent in parallel. It is difficult to ascertain what exactly is happening for the numba parallel approach as no longer allows for any OpenMP functions to be imported in. This also prevents any form of tuning to be made to the OpenMP process so its not possible to vary the load balancing. OpenMP allows for different kinds of load balancing to reduce overheads. Static scheduling allows for a direct level of load balancing similar to MPI where the load distributed to the cores is planned in advance. However as stated before this can lead to cores waiting for other cores to finish there calculations leading to a synchronisation barrier that decreases performance. An alternative approach would be to use dynamic scheduling where the thread scheduling is done by the operating system based on a scheduling algorithm. Numba uses dynamic scheduling which should give it a performance increase over static scheduling but there is no way to manually set a scheduling mode so it isn't possible to vary the chunk size to see what size causes the maximum efficiency. Cython and C++ support openmp in its entirety and allow for the scheduling and chunk size to be set manually. An improvement to the force calculation could be improved. The algorithm could be rewritten in order to store the forces spatially instead of particle by particle. This would allow the algorithm to skip half of the calculations by applying newtons third law as the force applied on one particle by another is equal and opposite for the other particle. This would allow the performance of the algorithm to double in speed. Going beyond the scope of this problem, there are also non-direct algorithms that reduce the mathematical complexity such as the Barnes-hut algorithm [8]which scale with $O(n \log n)$ instead of $O(n^2)$.

## CONCLUSION

The efficiency for the OpenMP approach appears to act strangely where only tasks with large $n$ are efficient with large $p$. The causes for this might be due to inaccurate timing causing difference but could be from an unseen flaw in the program itself. Overall, the parallel implementation does still lead to increased efficiency if certain conditions are met.

[1] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., USA, 1989.

[2] S.V. Adve et al. Parallel computing research at illinois: The upcrc agenda. 2008.

[3] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, mar 1983.

[5] Joshua E Barnes. Gravitational softening as a smoothing operation. *Monthly Notices of the Royal Astronomical Society*, 425(2):1104–1120, 2012.

[6] Sergey Rodionov and . Optimal choice of the softening length and time-step in n-body simulations. *Astronomy Reports*, 49:470–476, 06 2005.

[7] Dimitrios Serpanos and Tilman Wolf. Chapter 4 - interconnects and switching fabrics. In Dimitrios Serpanos and Tilman Wolf, editors, *Architecture of Network Systems*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 35–61. Morgan Kaufmann, Boston, 2011.

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