Version November 27, 2024
Preprint typeset using LaTeX style openjournal v. 09/06/15

PARTICLE MESH SIMULATION

2066040
University of Bristol
Version November 27, 2024

Abstract

This report explores the ability to parallelize the Particle Mesh method for N-body Simulations, finding successful implementation of both MPI and OpenMP and evaluating how each behaves under different simulation limits.

1. INTRODUCTION

Modeling the dynamics of an N body system under Newtonian Gravity using the direct approach, in which each pairwise force is calculated, becomes an increasingly complex problem as the number of particles reaches a large number. We can consider an N-body system, where particle i is the particle on which we are computing the gravitational force. We can write our force equation as the sum over all particles (i, j) where we ignore pairwise forces of the same index. This results in the expressions:

$$F_{ij} = -G\frac{m_i m_j}{|r_i - r_j|^2}\hat{r_{ij}} \tag{1}$$

$$a_i = \sum_{i\neq j}^{N}\frac{m_j}{|\vec{r}_i - \vec{r}_j|^2}\hat{r_{ij}} \tag{2}$$

$$v_{t+1} = a_t * dt \tag{3}$$

Therefore, to compute all values of acceleration for an N-body system, N coupled, non-linear second order differential equations must be solved. Excluding the trivial cases of $N = \{0, 1, 2\}$, there is no analytical method of solving these equations. Each extra unique particle pair results in the algorithm solving for $\frac{N}{2}$ pairs. This results in the complexity of the algorithm increasing with

$$O(\frac{N(N-1)}{2}) \tag{4}$$

Computing the singular force term F_{ij} is of order $O(1)$, meaning that the asymptotic limit of any algorithm by direct approach, and without the utilization of parallel processing, is of $O(N^2)$ as N^2 dominates in this limit. As this is done over every time step, with most accurate simulations taking $dt \to 0$, results in total complexity of $O(N^2T)$, which is exceedingly taxing on hardware as more accurate simulation parameters are used. This problem has been well known for decades, and as such numerous work around have been developed to provide a solution that reduces to problem from a quadratic complexity. This report explores utilizing an indirect approach of generating the forces on each particle, the Particle Mesh method. However, the main speed up comes from the use of multi-core processors and parallel computing.

1.1. *Shared vs Distributed Memory*

A processors ability to compute is parallel is determined by the number of cores available to it. A core acts as a unit that computes independently, following its own set of instructions. Threads are the way cores execute tasks, managed by the operating system. For a multi-core CPU, there are several levels of memory available at each level of the system. Registers are used for immediate calculations within cores, and have little to no latency. Cache's span the three levels, from $\{L_1 \to L_3\}$ where L_1 acts a cache available for each core, L_2 is shared between a couple of cores and L_3 is shared among all cores. In an optimal parallel program, RAM is accessed as little as possible to reduce latency.

When paralleling code, tasks that would otherwise be computed by one core are computed by many, and data is stored locally (physically) to reduce accessing time. In some cases, resources need to be shared between cores, requiring synchronization between them. The two main programs of interest for this approach are OpenMP (Open Multi-Processing) and MPI (Message Passing Interface). OpenMP operates under a shared memory framework. Each core utilized by openMP generates threads within a process. Each thread computes a component of a task, interacting exclusively with $L2/L3$ caches to synchronies with all other threads across all cores. It can be said that openMP follows a fork model, where prongs on threads are generate in parralel with each other to compute tasks, reading from the same persistant memory. MPI on the other hand works with a schema that has isolated memory for each core, accessing the $L1$ caches. For systems that are clustered (not local to the same processor), MPI perfoms exceptionally well. OpenMP is fundamentally limited to a single processor, given the need to access shared memory, but is relatively simple to implement and performance scales directly with the number of cores available in the CPU. On the other hand, MPI can effectively scale infinitely across as many cores are available, as each core acts independently. This however creates a communication overhead if synchronization is required, and data management becomes a key requirement.

2. PARTICLE MESH THEORY

It can be seen clearly that the implementation of parallel code will reduce complexity of the problem to:

$$O(\frac{N^2 t}{p}) \tag{5}$$

where p is the number of cores available. However, it is possible to fundamentally speed this up by re-framing the problem as a potential problem, as done in the Particle Mesh approach. The particle mesh method focuses on using the potential in order to work out the dynamics of each particle. From the Poisson equation, we have:

$$\nabla^2 \phi = 4\pi G \rho \tag{6}$$

where ϕ is the gravitational potential and ρ mass density. The main idea is to calculate the force directly from the potential, as $F = -\Delta\phi$. To work with gravitational potential, space must be transformed into a discrete set of N grids. An overview of the process for computing using the PM method is:

- Calculate the mass density
- Apply a Fourier transform to K space
- Solve the potential calculation in K space
- Apply an inverse Fourier transform back to real space
- Calculate the dynamics of each particle.

This algorithm has a total complexity of $O(N \log N * T)$, a considerable speed up in comparison to the direct approach. However, there is a caveat that this simulation is accurate in the limit of the gird lengths. The main speed up involves computing calculations across grids rather than particles, where $N_g < N_p$, however, the particle mesh method cannot simulate dynamics within girds. This renders it alone as obsolete for near-neighbor interactions, making it more useful for long range systems (where the average particle separation $r_i - r_j >> L_g$) Isele-Holder *et al.* (2012). As such, and by recommendation of Bird *et al.* (2022), Hairer *et al.* (2002), as second script using a second fine potential was created. It works in a smaller resolution than the coarse potential, and is used to mediate small scale interactions.

3. ALGORITHM

3.1. *Initialization*

To begin, an appropriate physical system was chosen to model. The Cosmic Linear Anisotropy Solving System Lesgourgues (2011)(CLASS) provide Boltzmann Code for numerous different cosmic simulation. For this experiment, a matter power spectrum was generated at a redshift $z = 0$. The N-body simulation would evolve the system to a future universe state. This involves around 1.6×10^7 particles. All most all parallelized function used MPI, with the CIC function using MPI and OpenMP. Cython, a c-wrapping python library, is used to compile all code to ensure a fast run time. For quicker analysis, small systems of random particles stored as CSV's are also used.

3.2. *Hilbert Curve Partition*

Grouping MPI nodes with local particles increases the efficiency of potential calculations as lower numerical values are used Bird *et al.* (2022). To group local particles together, a Hilbert curve function is used. Recursively, an N-dimensional space is subdivided into a smaller subspace, meaning that it generates 2^d smaller cubes until the dimensional is reduced to $1D$. This transformation ensures that all adjacent particles in ND remain local in $1D$. Each particles points has a Hilbert Index assigned, where the dimension of d represents the number of bits used to assign this index which encodes its spacial locality. Once particles are grouped by their indices, a linked list is created to map each particle's Hilbert index to a specific region of the simulation space.. As a result, each MPI thread can act independently with local particles and local grid space, this reduces the need for inter-thread communication without compromising accuracy and ensures all calculations can occur within branches. This method is particularly beneficial for the fine potential calculation.

3.3. *Cloud in an Cell*

A cloud in a cell(CIC) method is used to calculate the mass density of each grid. This method interpolate particles properties across neighboring cells. This is done to apply a mesh over the simulation space in order to work with the continuous potential. Optimally, every particle would exist in a single gird, but often with a large number of particles this in not possible. The two main methods available to assign mass-density, are Nearest Gird Point scheme (NGP) and CIC. CIC advances slightly further than NGP by distributing mass across the 8 neighboring cells. This reduce noise in calculations as the mass density gird gathers a greater degree of continuity Birdsall and Fuss (1969). This also improves the eventual potential calculation using the FFT, as it allows for clearer interpolation between particle positions and grid density Brackbill (2016). A hybrid of both MPI and OpenMP was used in the implementation of this program, each MPI core is assign a local region of particles, and openMP assigns threads to compute the particles densities within each cell. The justification for this choice is detailed in the results.

3.4. *Fine Potential*

The fine potential is introduced to compute inter cell dynamics over particularly dense regions. Here the use of the Hilbert Indices and the Linked list are used to compute nearest neighbor potential calculations through an explicit method. In order to achieve this as fast a possible, MPI nodes are re-assigned particles based on their local according to their Hilbert index. Each pairwise calculation of neighboring particles is calculated only, which serves as a suitable approximation given the dominance of potential values when particles are nearby. This potential is computed on 4/5 steps, meaning that approaching the course potential calculation, each grid has the lowest number of particles that it can.

3.5. *Coarse Potential*

To work out the larger system dynamics, we use slab fast Fourier transform to compute the long range potentials. In order to resolve the potential, a convolution is used. A convolution describes a transformation that is a point-wise multiplication in one domain with respect to the other, this is usually a complex and computational expensive process. We begin with approximating the potential using the finite differencing method:

$$\frac{\phi(\vec{r}+1) - 2\phi + \phi(\hat{r}+1)}{(\Delta\vec{r})^2} = 4piG\rho(\vec{r}) \tag{7}$$

We apply the slab FFT, which partition's the 3D potential into grids along the z axis, and uses MPI to compute the potentials across each grid. Applying A Fourier transform to the finite differences equation gives:

$$\frac{(e^{ik\Delta\vec{r}} - 2 + e^{-ik\Delta\vec{r}})}{(\Delta\vec{r})^2}\Phi(\vec{k}) = 4\pi GP(\vec{k}) \tag{8}$$

the LHS of this equation can be re-writen as a constant value (considering just x):

$$K = \frac{-4\sin^2(k\Delta x/2)}{\Delta x^2} \tag{9}$$

meaning that the potential can be solved using:

$$\Phi(\vec{k}) = \frac{-4\pi G}{K^2}P(k) \tag{10}$$

We can get back to the continuous case by taking the limit of K as $\vec{r} \to 0$. We then apply the IFFT to move these values back to real space

3.6. *Dynamics*

For dynamics, the Verlet integration method is used. This method is symplectic, meaning that is conserves volume in phase space for the dynamics of each particle. This means that the Hamiltonian of each particle remains roughly constant, and any error sources $O(\Delta t^2)$ do not accumulate over time, but rather oscillate around the true energy value Hairer *et al.* (2002). This is ideal for a long-range simulation, as this ensures a steady energy value for the system. The Runga-Kutta method was attempted, but stability in the system diverged early in the simulation. The updated position of a particle is given by:

$$X_{new} = 2x - x_{prev} + a_x(\Delta t)^2 \tag{11}$$

where in the case of a finite sized simulation, boundary conditions are used to wrap particles leaving the edges of the simulation back round to the other side. The acceleration is calculated from a helper function

3.7. *Graphics*

Individual time frames and analysis was generated using the python library *mat_plot_lib*. For full video renders for lower particle numbers, the *manim* libary was used.

4. DESIGN CHOICES AND RESULTS

When working on an optimal runtime, it is important to compare different methodologies to achieve parallelism. To begin, each program section was timed using the python library *time* to test for bottlenecks in the code. Over each script. The results for the run time over the first iteration are bellow. Four cores were used with a gird size of 256^3 for 100 steps with $1.6x10^7$ particles:

Process	**Coarse Potential (mean ± std)**	**Coarse and Fine Potential (mean ± std)**
Hilbert Partition	0.306 ± 0.669	0.318 ± 0.701
Cloud In a Cell	0.009 ± 0.003	0.007 ± 0.015
MPI Grouping (Allreduce)	0.005 ± 0.000	0.008 ± 0.001
MPI Synchronization	0.000 ± 0.000	0.000 ± 0.000
Coarse Potential	0.017 ± 0.003	0.025 ± 0.002
Fine Potential	-	0.006 ± 0.011
Update Positions	0.000 ± 0.001	0.001 ± 0.002
MPI Gather	0.001 ± 0.004	0.002 ± 0.006
Step Total	0.078 ± 0.237	0.062 ± 0.261

TABLE 1: Comparison of runtime and standard deviation for functions across Coarse Potential and Coarse and Fine Potential.

In the low simulation limit (or minimal hardware access) immediate overhead is easy to identify. From initial testing, the *cloud_in_a_cell*() function provided the largest overhead. To improve, four approaches were taken to assign mass density in each grid. Separate Cython scripts using MPI, openMP, a hybrid of the two and a serial script were written for the cloud in a cell method. To calculate $O(n)$, we require a fair test, and as such each was allowed 4 cores to run (excluding the serial script). $N = \{1 \times 10^5, 1 \times 10^6, 1 \times 10^7, 1 \times 10^8\}$ particles where generated across a mesh of 1024^3 volume with unit cell sizes. A consistent random seed is used between each program to ensure particle positions are constant. This reducing any conflict in computation time for more "local" distributions, as the run time is reduced for spatially closer particles due to calculation of smaller numerical values. Times were averaged over 3 separate runs on an apple $M2$ chip (up to 8 core, 4 Avalanche and 4 Blizzard). The results are displayed below, where an average is taken due to an increases temperature after consecutive runs possibly leading to increased run time, and background processes running on cores.

N	**Hybrid (mean ± std)**	**Serial (mean ± std)**	**MPI (mean ± std)**	**OpenMP (mean ± std)**
10^5	24.61 ± 0.83	0.36 ± 0.01	18.20 ± 0.34	1.37 ± 0.06
10^6	23.84 ± 0.11	7.81 ± 0.03	23.73 ± 0.07	19.60 ± 0.10
10^7	24.04 ± 0.38	96.59 ± 0.38	95.27 ± 0.21	223.87 ± 4.34
10^8	24.14 ± 0.24	NAN	NAN	NAN

TABLE 2: Average computation time and standard deviation for different values of N and parallelization methods.

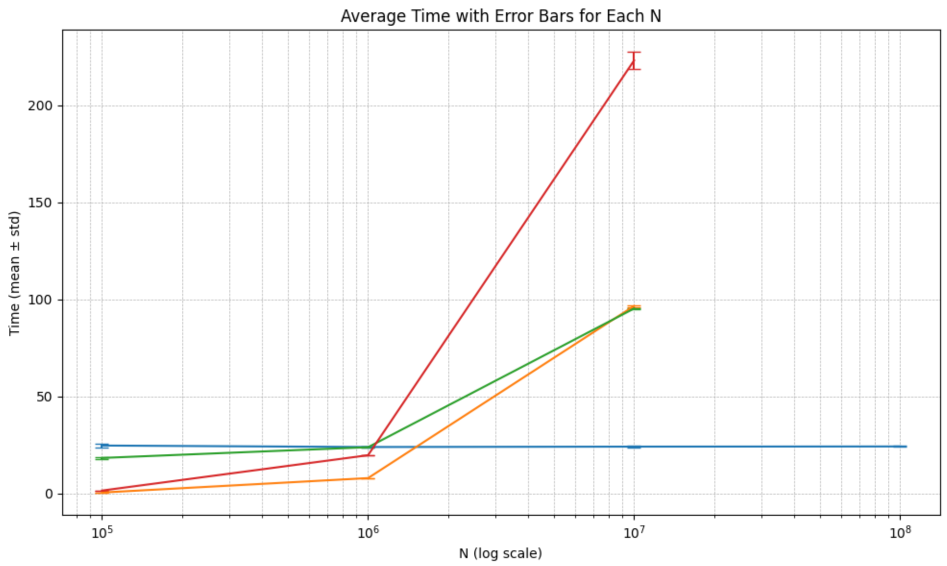


FIG. 1.—: Cloud in a Cell Analysis

Where the NAN values are for run times that exceeded the memory limit of the device. Quite clearly, in a larger simulation limit, using a combination of openMP and MPI best suits these conditions, with an even runtime regardless of particle size. The large overhead in both the Hybrid and MPI method is associated with MPI's bandwidth limits (or latency) with large data communication over separate threads. However, this becomes a necessary step as the number of calculations increase dramatically. OpenMP does not suffer from latency as data is shared, however it will struggle as memory capacity is limited at higher N, meaning that each thread accessing the same data block induces caching issues. This test was done in a low core limit, where greater core number reduces the run time on all processes significantly excluding the serial program.

The function *hilbert_partion*() has a significantly large runtime. To evaluate whether this is an effective function for speeding up process, we evaluate how the combined script performs with increasing N and core number. To do this, one script had the function removed. Below is a plot of performance against core access:

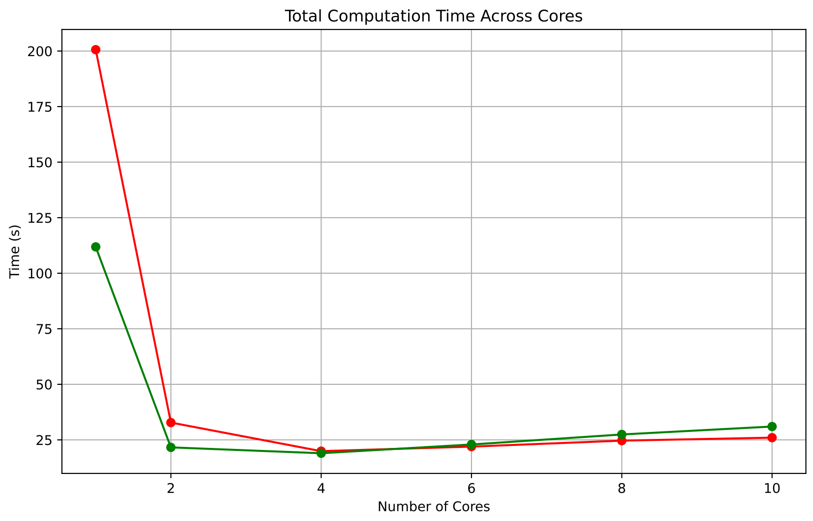


Fig. 2.—: Evaluation of run time with and without the use of Hilbert indexing. Red is with, Green is without

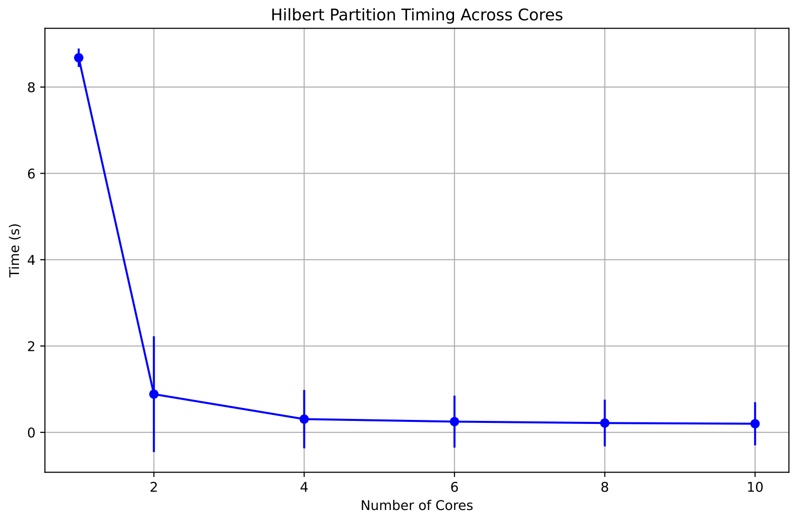
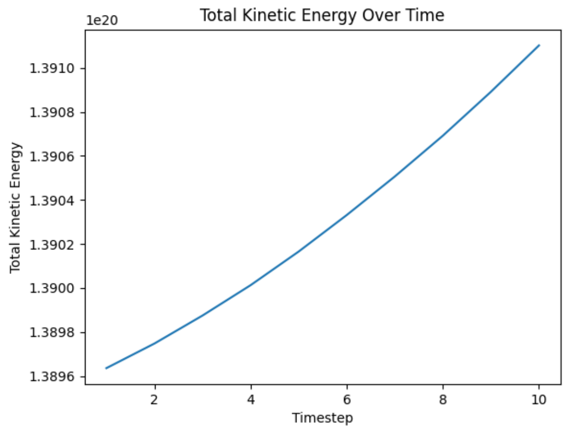


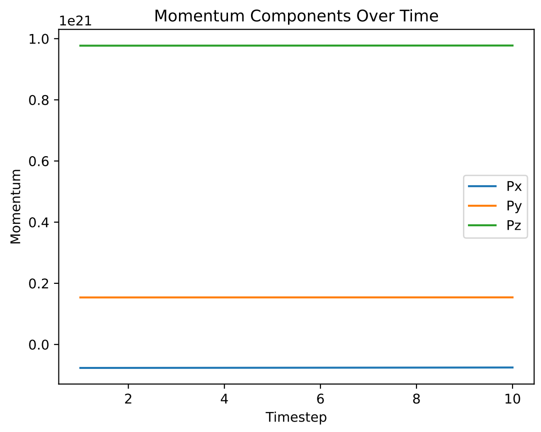
Fig. 3.—: Run time of scripts using Hilbert Function

Evidently, the system is over parallelized in the limits of test, but this is done to ensure that other functions variance with core size is minimized, as this would skew the test. As seen in figure 2, the run time with the re-assignment of MPI particles runs faster than without. Figure 3 shows that increasing the number of cores also decreases the runtime of this function. It was not possible to generate a direct openMP equivalent for partitioning, given the high volume of input data of each time step and its shared memory architecture, and as such was not used. In testing for openMP, race conditions were established frequently due to the modular re-assignment of Hilbert indices, as often the re-distribution of particles across the simulation space resulted in threads collecting too many particles and miss-assigning them, decreasing simulation accuracy.

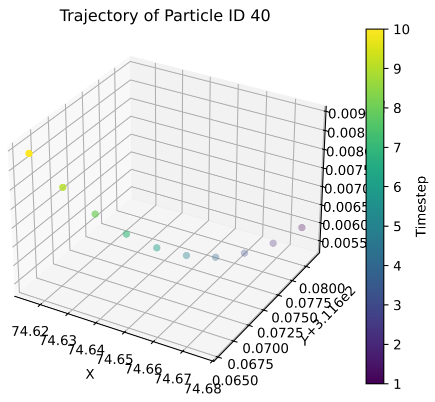
Next we address the accuracy of the simulation. To total energy and momentum are plotted below across a run of $1.6x10^7$ particles on a gird size of 512. The figures show 10 seconds of real time simulated, however this corresponds to 10000 iterations of the simulation as the time step = 0.001. It should be noted that each metric (position and velocity) is scaled relative to the total grid size, meaning that in the plot these values are kept small. Generally, the longest stable runtime corresponded to 10 seconds of real time.



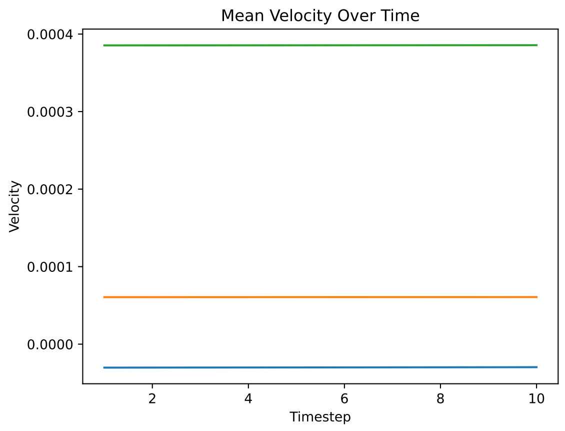
(a) Kinetic Energy Change



(b) Momentum Change



(c) Total Trajectory of a particle



(d) Mean Velocity

Fig. 4.—: Simulation Performance. Figs two and 4 have three lines, green is in the x direction, blue is y and orange is z. In Image 3, the colour gradient of each point represents the point in time it was observed

5. MPI AND OPENMP COMPARISON

In trying to assign an optimal configuration to this program, different configuration limits called for different methods. The two largest time saving methods were outlined above. Each other function scalled in the following way. OpenMP provided a faster run time whilst the simulation remained relatively memory inexpensive, as seen in the figure 1. Generally, the overhead for initialing MPI and the introduced latency in joining and merging data over each time loop is not worth doing for a small particle number and large gird sizes. There is also the fact that the memory storage of each node is not guaranteed to be in the same location for each core (with reference to each individual architecture) creating further delays in data synchronization. However, we also see minimal variation between the serial and openMP approach. Given that a single core can directly access $L1$ cache memory faster than N cores can access grouped $L2/L3$ memory, there was a small margin of simulation parameters that favored using openMP. This makes sense, as openMP can dynamically balanced work load where as MPI suffers from slow communication between cores. Given the need to re-establish a data ledger at the end of each time step, and the scaling of the program, MPI was selected as the primary choice of parallelism for scaling. To ensure efficiency with the frequent partitioning and collection of data from MPI nodes, the $MPI.Comm.Allreduce$ function was used over other broadcasting and collecting functions like $MPI.Bcast$ and $MPI.Gather$. $MPI.Comm.Allreduce$ is a function that specialize in optimizing the number of tasks carried out for distribution and combination processes. Functions such as $MPI.Bcast$ and $MPI.Gather$. are blocking, meaning that certain cores have to pause and wait for catch up from others, which significantly alters the run time when this operation is used frequently. One other issue with MPI is when each core has an intensive memory load, where increasing core number decreases efficiency due to memory requirements. Essentially, the speed up in time step completion due to an increased number of cores is outweighed by the encumbered core systems computing high memory workloads individually. If each core requires more memory than its local L1 cache can supply, then memory is moved into a shared $L3$ cache. This leads to all cores effectively working in a shared memory system without the load balancing available with openMP, meaning drastically increased run times. This was observed in initial testing, where apparent speed ups were in fact slower, as lower core numbers have less contention when reading and writing to $L3$ when forced there by memory requirements compared to high core numbers. This was however mitigated when moving to Blue Crystal simulations. Mac M2 chips have a higher memory capacity in both $l1$ and $l2$ chips, but blue crystals $intelXeone5 - 2680v4$ processors have $L3$ caches of $32mb$, whereas $M2$ chips do not have any $L3$ caches. This meant that with simulation parameters that required parallelizing for a speed up, Blue Crystals shared $L3$ cache allowed enough storage per chip to prevent this bottleneck, as well as possibly computing across other chips (i.e not all nodes in the same CPU). This allowed for proper analysis of MPI at high core workloads as well as defining a tuning parameter. There are 3 main variables that contribute to run time in this PM experiment:

- Particle Number
- Gird Granularity
- Core Number

to define a tuning parameter, we take the ratio of a singular core running the experiment against the the total number of cores being run. Bellow is a graph of tuning parameter for each process.

$$\eta = \frac{T_1}{T_n} \tag{12}$$

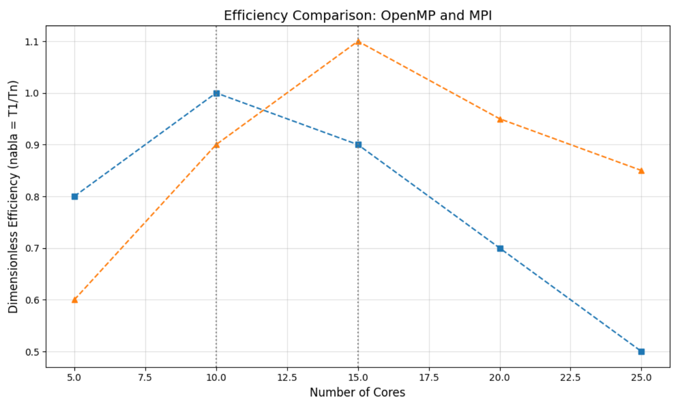


Fig. 5.—: OpenMP vs MPI efficiency vs core number normalized. yellow is MPI, blue is OpenMP

6. CONCLUSION

It can be seen from $Fig5$ that MPI is most efficient with a higher core value, where as openMP is more efficient with fewer cores. This is alined with the prediction that MPI should work better as core number increases, and indicates a successful implementation of parallelized code. As a point of improvement, the duration of stable simulations was short, around 1000 iterations. The addition of kinetic energy dampener was introduced to maintain stability, however this revealed a principle problem with the code. In the limit of parallel processing (i.e, a very high particle number and grid density) simulation dynamics would break down before the main advantages of parallel processing could be achieved. However, the pm method relies on large separations in particles, which is hard to introduce with a large number of particles. In the low limit, dynamics would last indefinitely, but wouldn't require parallel processing. As such, to complete this experiment energy was consistently drawn from the system. To improve this simulation, a careful consideration of optimizing run time vs memory demand would be evaluated more, as excessive memory need becomes a tangential problem to parallelism, requiring a carefully optimized approach. A new approach would be to consider a method to ensure short range potentials did not lead to excessively high energies.

REFERENCES

R. E. Isele-Holder, W. Mitchell, and A. E. Ismail, The Journal of Chemical Physics **137**, 174107 (2012).

S. Bird, Y. Ni, T. Di Matteo, R. Croft, Y. Feng, and N. Chen, Monthly Notices of the Royal Astronomical Society **512**, 3703 (2022).

E. Hairer, C. Lubich, and G. Wanner, *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations*, Springer series in computational mathematics No. 31 (Springer, Berlin New York, 2002).

J. Lesgourgues, "The Cosmic Linear Anisotropy Solving System (CLASS) I: Overview," (2011), arXiv:1104.2932 [astro-ph].

C. K. Birdsall and D. Fuss, Methods in Computational Physics **9**, 141 (1969).

J. U. Brackbill, Journal of Computational Physics **317**, 405 (2016), arXiv:1510.08741 [physics]