Comparison of Serial, Shared Memory and Distributed Memory Approaches for N-Body Direct Approach

Leander S Fernandes

Introduction

Since their inception, n-body simulations have been used to model any system which can be described as multiple point like particles. At the small scale it is applied to molecular dynamics whilst on the larger scale to gravitational problems. The methodology to solve these problems however is the same, requiring the calculation of force exerted by a body whilst applying that force to multiple bodies at every time step. As a result, this problem is of order $O(N^2)$ having a quadratic scaling [1].

It is paramount to find ways of mitigating the effects of this scaling by implementing modern computational techniques and performant code. This project aims to demonstrate the effects of these modern techniques by comparing shared memory (threaded/parallelised) and distributed memory methods of a gravitational n-body direct-approach problem. Comparisons will be made against the serial implementation with a discussion of alternative methods to speeding up the code.

Methodology

Solving the Physics Problem

Direct Approach The direct approach attempts to solve the n-body problem by approximating each planetary body as a point like particle. The state of the system is first initialised by creating an 'n' number of bodies each with an associated mass, position and velocity . During each time step, the acceleration applied on each body is calculated using:

$$a_i = -G \sum_{j \neq i} m_j \frac{r_i - r_j}{|r_i - r_j|^3} \tag{1}$$

The new positions are found using previous velocities with the new velocities found by using the calculated accelerations. The equations used are:

$$x_{t+1} = x_t + v_t t \qquad v_{t+1} = v_t + a_t t \tag{2}$$

This is a simple first order *Runge-Kutta* (RK) method (Euler method) and is known to be flawed at large time steps due to poor energy conservation. As a result, simulations of stable systems can exhibit uncharacteristic behaviours given enough time. Alternative time stepping algorithms include Leapfrog integration (velocity Verlet); where new positions and velocities are calculated out of step, and fourth order RK method (RK4). Velocity Verlet and RK4 offer significant improvements to accuracy with the associated error on the the order of $O(h^2)$ and $O(h^4)$ respectively, where h is the time step. Comparatively, the Euler method has an error of order $O(h^2)$.

Alternative Approaches Other approaches have been used to solve gravitational problems and offer remarkable scaling. A recursive treecode (Barnes-Hut) offers scaling of $O(NLogN)$ with a more complex Fast Multipole Method providing $O(N)$ scaling [2, 3]. Both alternative methods drastically improve scaling however are complex to implement and require elaborate parallelisation in stark contrast to the simplicity of a direct approach.

Implementation

Serial Serial code is one which runs on a single core using a single thread. It is the simplest of the three implementations but can be used as a starting point to produce shared memory and distributed code. The physics problem was first decomposed into stages that could be implemented functionally. The simulation required to be initialised with mass, positions and velocities. Arrays of these quantities were created and populated using random values for a given problem size n. Mass was a one dimensional array whilst position and velocity were three dimensional arrays, working in Cartesian coordinates. A nested loop is then used to simulate the problem. The first loop iterates over time for a specified number of time steps whilst the second loop iterates over each body in the n-body problem. The acceleration a body feels due to all other bodies is calculated and returned as an array mimicking a vector. This vector is then immediately used to calculated the new positions and velocities which are then adjusted in a global array. This process is done for all bodies in the system at each time step.

Several parameters can be set during initialisation: number of bodies n, number of iterations and size of time step. As a result, different problems can be setup.

A separate function is used to initialise our Solar System to test the code on a known stable system.

A feature of the direct approach method that needs to be considered, is it's ability to kick bodies far from the initialisation positions. This is primarily due to the fact that the gravitational bodies are assumed to be point masses. Given sufficient enough time, a single small mass body may reach a point around a larger mass where it acquires a large enough acceleration giving the smaller body it's required escape velocity. To mitigate this effect, a softening factor ϵ is added to the denominator and is defined as $\epsilon >> 0$ and $r >> 0$. The quantity needs to be small enough to not affect long range interactions but large enough to have the desired effect over short range. This gives the force equation a minimum radius; therefore a maximum acceleration a body can feel. Adjusting the softening factor can enable the simulation to coalesce rather than disassociate[4].

Shared Memory `Multiprocessing` is a Python package that enables processor based parallelism. It leverages the the shared memory architecture employed by the *Central Processing Unit* (CPU). Volatile RAM and three layers of cache can be shared between multiple cores on the same CPU. This enables cores to access the same bits of information. Code is initially executed on one thread before splitting across multiple threads to share the workload. After executing their work, threads recombine to enable another cycle of this process.

Natively, Python runs on a single core using a single thread, due to a *global interpreter lock* (GIL). `Multiprocessing` side-steps the GIL by launching multiple subprocesses at specific intersections of the code. Subprocesses attempt to then access available threads, execute synchronous or asynchronous sections of code and finally recombine. Within the code, this is achieved using `Pool()` and `Array()` classes. With a pool of 'threads' enabled, the method `starmap()` is called to evenly distribute sections of an iterable between threads. The pool of threads is then closed and joined after executing all tasks using `close()` and `join()` methods.

An interesting consequence of this method means static or dynamic scheduling cannot easily be implemented since arrays are divided initially into equal chunks for each 'thread'. An additional quirk of this procedure means more processes than threads available can be launched as `Multiprocessing` is not directly requesting threads by turning off the GIL but instead launching subprocesses. These two effects can compound to slow-down a shared memory approach. By launching more subprocesses than available threads, some subprocesses may be backlogged in the thread queue, effectively doubling the number of clock cycles for a given task.

Distributed The distributed memory approach supersedes the restrictions set by a single CPU by employing multiple CPUs to execute code. This could be between, many cores within a CPU, many CPUs on a single node, or multiple CPUs across multiple nodes on a cluster. The same instance of code is run on multiple cores, akin to the Same Instruction Multiple Data 'SiMD' parallelisation used by *graphical processing units* (GPU) making this system highly scalable. As this methodology can employ multiple nodes, a distributed memory setup is required since nodes cannot see or access each other's working memory; unlike a GPU. Therefore, code needs to be sophisticated enough to send and receive data between 'workers' across a motherboard or cluster.

The standard *Message Passing Interface* (MPI) was used to achieve this in Python, enabled by the `MPI4Py` module. When using `mpiexec -n X` in the terminal, an `X` number of workers are spawned; each on different cores. `MPI.COMM_WORLD` first allows all launched instances of the program to see each other with each instance being assigned a unique identifier 'rank' using `Get_rank()`. By convention the master node assumes the zeroth rank whilst other ranks are labeled as workers. The master node first initialises the system and sends each worker its mass array, along with an array of indexes. Distributing the indexes is handled by a collective communicator `scatter()`, which sends each worker a unique set of indexes. The indexes a worker is assigned will be the only bodies within the simulation it will work on. At the start of the time step, the master node sends each worker a copy of the positions and velocity array. Both these arrays and the aforementioned mass is sent through the broadcast collective communicator, `bcast()`. Each worker, and the master node iterate through their assigned bodies and populate a buffer. The buffer is then sent back to the master node using the collective, `gather()`. Fig 1 demonstrates how these method works for data across multiple nodes. The positions and velocities are updated at the master node and then broadcast to all workers at the next iteration, repeating the cycle.

All communicators used within the distributed code are collective methods. *Point-to-point* (p2p) communicators could have be used but would have been significantly less efficient. Functions to mimic `bcast()` and `gather()` could have been built using `send()` and `recv()`. To be general, an iterator would be needed dramatically slowing down the time stepping loop. Implying tree structure achieved by `bcast` is far more efficient than structure built by a custom method. These two p2p methods are also blocking, capable of stopping all processes until the data is received. Through using collectives, the program can escape the slowdown from p2p blocking, in favour of a more parallel blocking achieved with collective communicators.

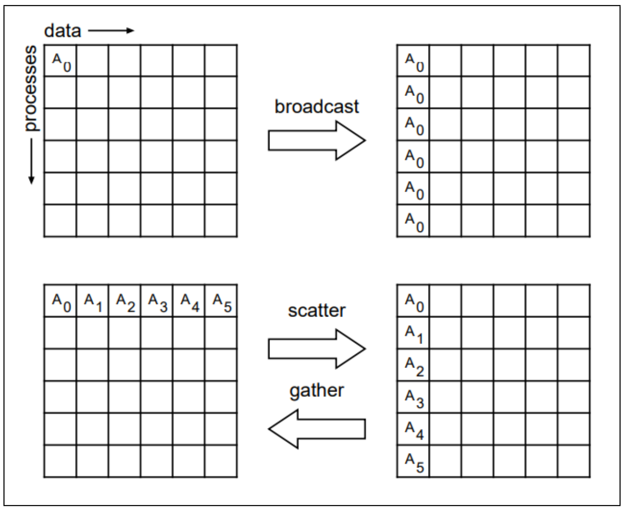

Figure 1: Graphic showing how each of the collective communicative methods spread and share data for a program with six processes. Taken from [5]

An interesting consequence of using `gather()` is not needing to order the incoming data. The arrays being sent to the zeroth node always arrive in rank order requiring no additional manipulation to order. This order is the same as which `scatter()` sent indexes via. Additionally, to balance the load across all workers, `np.array_split()` was used. This function broke the indexes array into even chunks meaning from the outset, each worker would be assigned the same number of tasks. Minimal time is then wasted at the `gather()` blocking method.

Validating the Physics

Graphics were created to verify the program was solving the physics problem at hand. During simulation, body positions could be saved as a `Numpy .npy` file. The file could be loaded and plotted using `Pyplot`. Animations could then be made of the system to easily verify the simulation. Links to full videos can be found in the Appendix Fig 11 and Fig 12. Shown in Fig 10 is a still image from one of these animations demonstrating the simulation does put planets in our solar system on near circular orbits. However this is not perfect; as previously mentioned, the Euler method is energy inefficient and can be seen by the inner planets orbits being abnormal.

To concretely understand how energy was changing within the system, an energies `.npy` file was created to hold data about kinetic and potential energies of all planets. When summed across all bodies, total energy (Hamiltonian) can be plotted. A stable system would see the total energy remain constant with its kinetic and potential components oscillating out of phase. Oscillations of the components here are in phase resulting in the Hamiltonian to vary over time. All quantities also

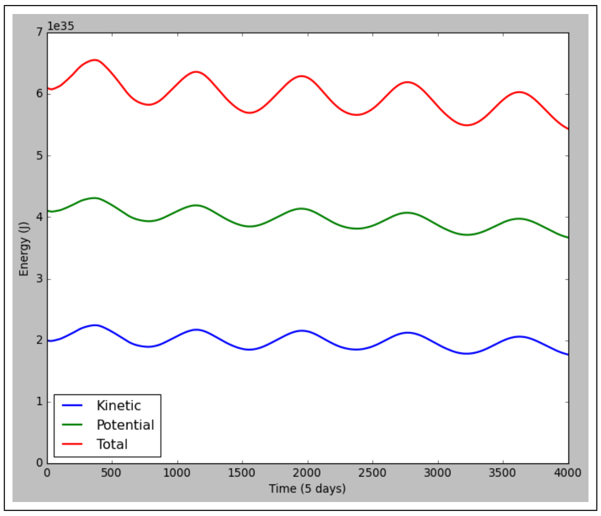

Figure 2: Plot showing how kinetic, potential and total energy of the system shown in Appendix Fig 10 changes over time. Kinetic and potential energies oscillating in phase resulting in total energy to oscillate. An underlying negative gradient can be seen for all quantities.

show an underlying negative gradient, tending to zero. This validates the notion that the methodology's time-stepping technique does not conserve energy.

Results

Results were gathered by taking Serial, `multiprocessing` and `MPI` jobs on BlueCrystal4 (BC4). The `teach_cpu` partition was used allowing access to 20 nodes if needed. Each node has two 14 core 2.4 GHz Intel E5-2680 v4 (Broadwell) CPUs, spread across two sockets. To compare against BC4, results were taken on a personal laptop with a 2.6 GHz Intel i7-10750H (Comet Lake) CPU boosted to 4.8 GHz when taking data. Timings were taken using the `perf_counter()` module from Python's `time` package and `MPI.Wtime` method from `MPI4Py` when dealing with distributed code. Three repeats were taken and the mean used as the final value. The standard deviation was taken to be the error which was then propagated for derivative dependent variables following standard error propagation in quadrature. Initialisation time of systems were in the order $10^{-5}s$ which would be significant for small execution time programs. Five iterations was decided to be suitable, as at this number noise and initialisation time minimally affect timings whilst being small enough to not require large amounts of compute time.

Where appropriate, speed-up and efficiency were calculated for the range of threads/workers. Speed-up indicates how the `multiprocessing` and `MPI` programs scale for different levels of parallelisation whilst efficiency indicates the contribution of the cores to the problem.

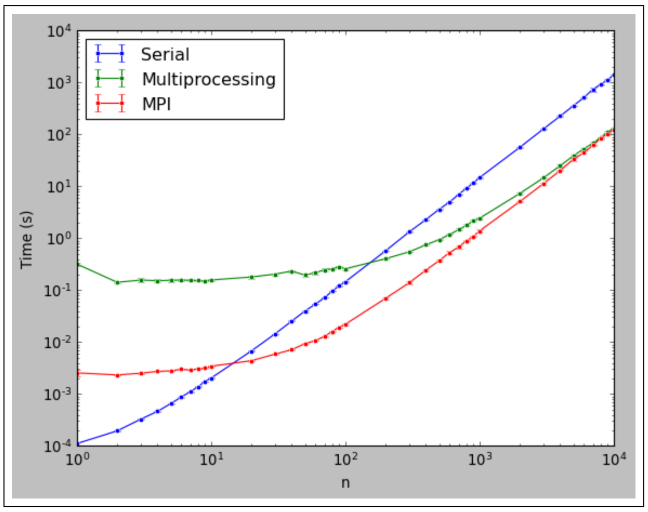

Figure 3: Line plot with error bars showing how timing scales with problem size for serial, multiprocessing and MPI programs. Both scales are log10 to better compare the methods. Multiprocessing used 12 threads whilst MPI used 12 workers.

Speed-up (η) and efficiency (ϵ) is therefore defined as:

$$\eta = \frac{T_1}{T_N} \qquad \epsilon = \frac{\eta}{N} \tag{3}$$

where T_N indicates the time T taken for N number of processes.

Serial vs Multiprocessing vs MPI

Serial, `multiprocessing` and `MPI` are compared at different problem sizes in Fig 3 ranging from 1 body to 10,000 bodies. It can be seen that the serial implementation is quickest for small n due to no communication overheads unlike the shared and distributed techniques with `multiprocessing` being the slowest. Compared to serial, `MPI` becomes the faster method at $n = 20$ with `multiprocessing` overtaking at $n = 200$. As n continues to increase, both parallel techniques merge at $n \approx 10000$. It can be seen that all methods converge to a line with gradient = 2; expected from an n^2 problem. `Multiprocessing` and `MPI` methods provide an order of magnitude speed-up against the basic serial code at a large enough problem size. Notably, `MPI` achieves this by $n \approx 300$.

Scaling

Fig 5 demonstrates how scaling changes with problem size. It can be seen that `mulitprocessing` can achieve a higher speed-up when dealing with larger problem sizes. For $n = 500$ bodies, speed-up peaks at $\eta \approx 5$ but for the larger problem size of $n = 1000$, speed-up reaches $\eta \approx 7$ before stalling and reducing. This occurs at the 14 thread mark which is a key point for BC4. At this

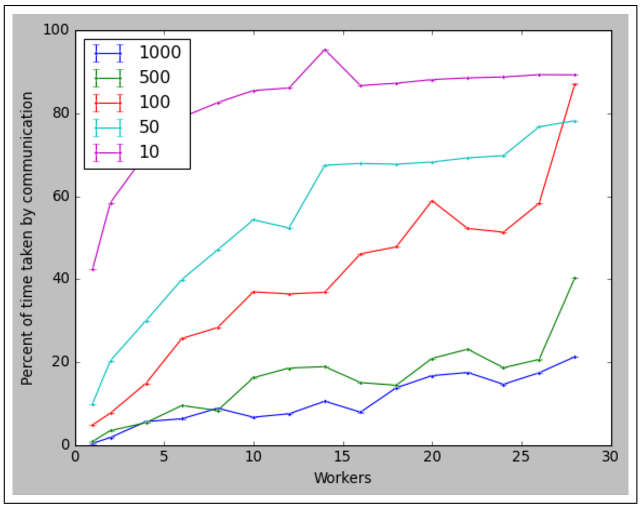

Figure 4: Line plot showing percentage of execution time spent communicating for a range of problem sizes ranging from 1 to 28 workers. Data was collected on a single BC4 node.

point, the subprocesses launched have populated an enter CPU after which, they begin populating the second CPU on the same node. This requires extra data to travel across the system bus, adding latency to the system and program. The efficiency for both system falls dramatically with $n = 500$ having a steeper decline. The efficiency error for low threads is much larger than that if high threads, an artefact of low thread timings having high standard deviations. The fuller graph Fig 5 includes smaller problem sizes but shows immediate slow down; suggesting at low problem sizes, opening and closing subprocesses is a bottleneck for `multiprocessing`.

`MPI` scaling is shown on the reduced Fig 6 demonstrating remarkable scaling when compared to `multiprocessing`. Data for this was collected by reserving 5 nodes and incrementing the number of workers per node. For both $n = 500$ and 1000, the speed-up is near identical reaching $\eta \approx 24$. The efficiency remains above 65% with both problem sizes tracking together. By looking at the fuller scaling plots, we can see smaller problem sizes exhibit a different behaviour. These problems reach a peak speed-up before slowing down. It can also be seen their efficiency falls sharply in comparison to large n. The possible cause for this is the high communication time between nodes for small problem sizes. Communication cannot be sped up, so creates a ceiling for how fast small n `MPI` problems can run. Large n problems are dominated by computation so can show incredible speed-up.

MPI Communication vs Computation

It is important to understand how communication time affects different problem sizes. The absolute communication time cannot be plotted since increasing problem size

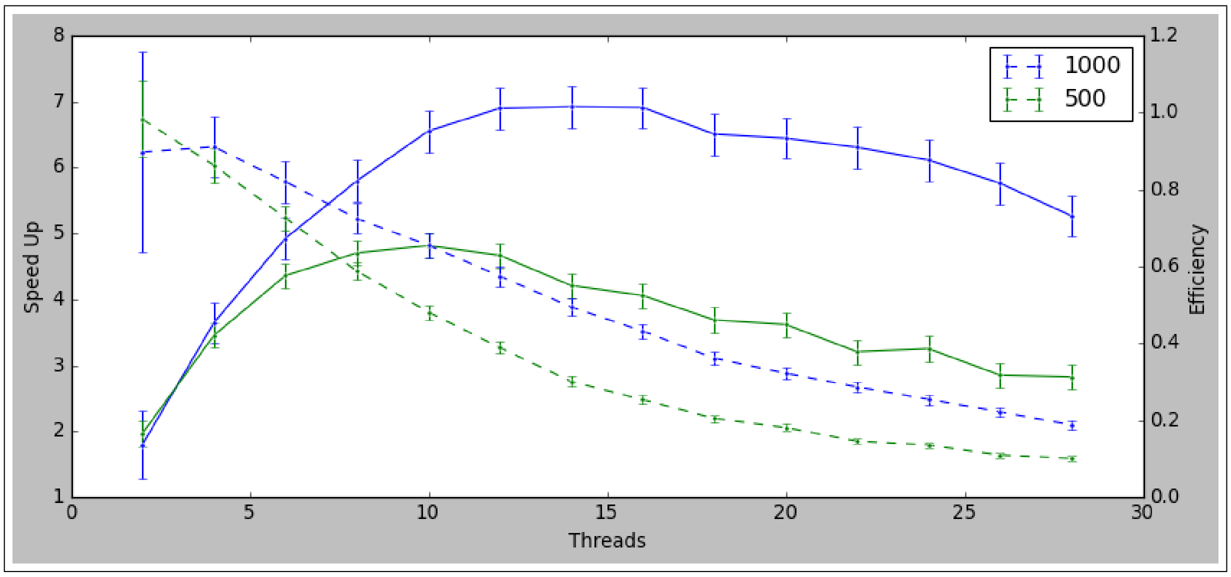

Figure 5: [Shared Memory] Reduced version of Appendix Fig 8. Line plot showing speed up η and efficiency ϵ due to number of threads for a n =100 and 1000 for multiprocessing method. The solid line represents speed-up whilst dashed line represents efficiency. All data was collected on a single BC4 node.

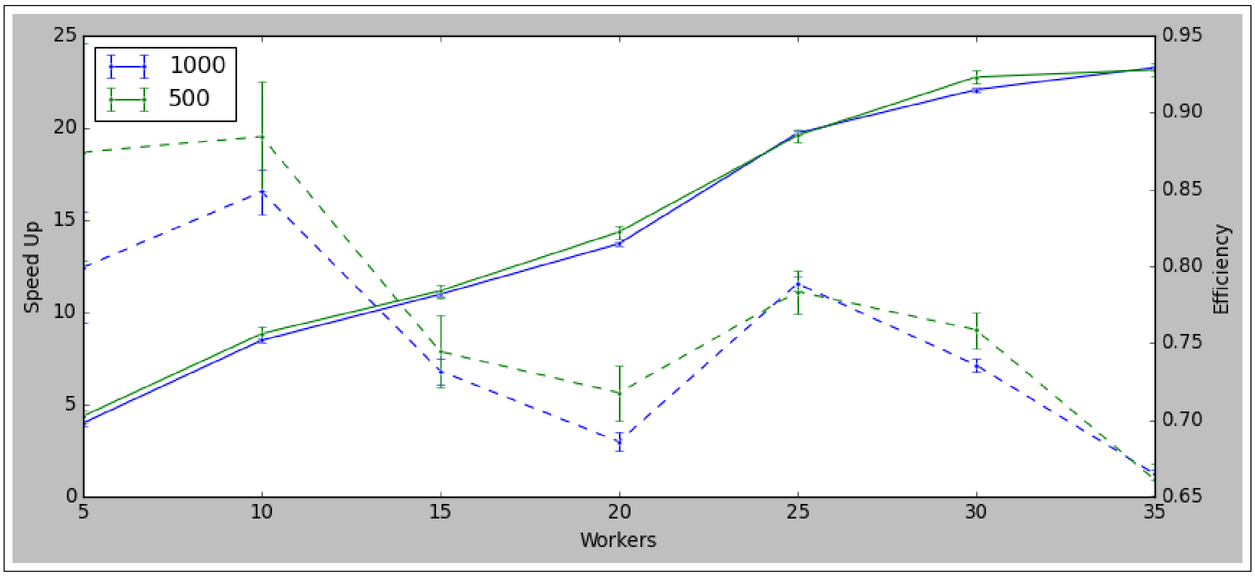

Figure 6: [Distributed Memory] Reduced version of Appendix Fig 9. Line plot showing speed-up η and efficiency ϵ due to number of workers for a n =100 and 1000 for the MPI method. The solid line represents speed-up whilst dashed line represents efficiency. All data was collected on 5 BC4 nodes.

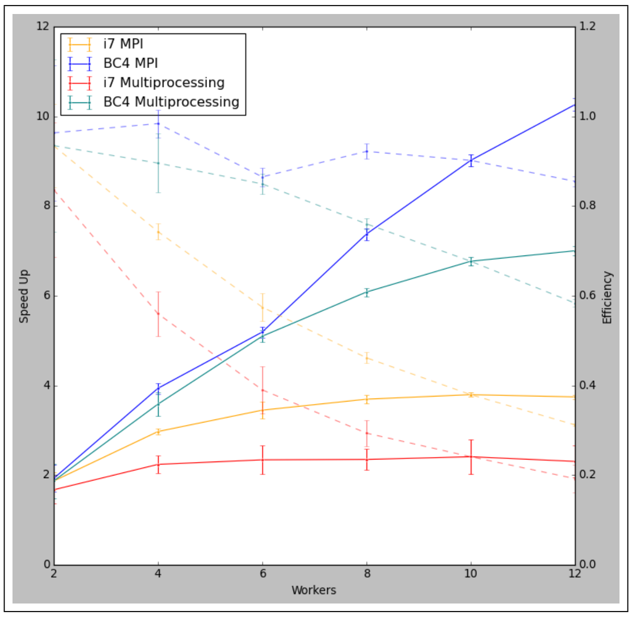

Figure 7: Line plot comparing BC4 and an Intel i7 for a problem size $n = 1000$. Solid line shows speed-up whilst dashed line shows efficiency.

will always cause their communication time to increase. Instead the percentage of the time spent communicating needs to be plot, as shown in Fig 4. This plot reveals how communication disproportionately affects small problem sizes. It can be seen that for `MPI` programs, communication time increases as more workers are added; this is understandable. However the graph also exemplifies the large amounts of time spent communicating by small n simulations: $n = 10$ and $n = 50$. Both these simulations spend upwards of 70% of their time communicating. Conversely, $n = 500$ and $n = 1000$ spent less than 20% up until 26 workers. It is for this reason that serial implementation is faster than `MPI` at small simulation sizes.

Commercial vs Enterprise Setup

Simple bash scripts for `multiprocessing` and `MPI` were run on a single node on BC4 and on a personal computer. Data was taken upto a maximum of 12 threads/workers (the maximum real and hyper-threads available on the i7). We can see on Fig 7 that the enterprise chips available on BC4 are significantly better for speeding up the shared and distributed memory methods. BC4 shows continual speed-up all the way to 12 threads/workers whilst the i7 staggers after reaching 6 threads/workers. At this point all real threads on the CPU are being used with a one-to-one correspondence to cores. After this, virtual threads attempt to aid with processing and executing the program. At the 12 worker stage, the i7 is using 6 real threads, 6 hyper-threads, and 6 cores. At the same stage, BC4 is using 12 real threads, and 12 cores. This then enables BC4 to be very efficient when compared to the i7.

Discussion

It was noticed that results were inconsistent when running shared memory jobs on BC4. Data would be noisy especially for simulations that had fast execution times and would need to be discarded. The cause of this is most likely other jobs executing memory requests at the same time as simulation data is required. It is also a possibility that our simulations began before the previous job was cleared. These hitches are inevitable on a workload manager like SLURM. To mitigate these effects/remove them entirely, an additional keyword can be added to the job scripts. By using `SBATCH --exclusive`, an entire node is requested and reserved solely for the simulation; no other jobs can run until our simulations have finished. This provided much cleaner results however not ideal when working on a shared computing cluster system.

Surprisingly, my implementation showed that `MPI` implementation is faster than `multiprocessing`. This differs from what might be seen in another language like C or alternative implementation for shared memory via Cython and `OpenMP`. This is is probably caused by the method `multiprocessing` employs which is spawning, closing and joining subprocesses. The overheads involved in this process must be causing a slowdown. This method is similar to that of `MPI` but processes are launched each iteration as oppose to just the start like in distributed. To understand this further, finding a method where subprocesses are launched before entering the time-stepping loop could result in `multiprocessing` being faster than `MPI` as less time spent spawning.

Further investigations could be done to fully understand how `MPI` scales. Fig 6 stops at 35 workers, however since `MPI` is highly scalable it could be increased up to 560 workers on the `teach_cpu` partition. Doing this should show high speed-ups but capping at a particular point.

To speed-up code across all the implementations, vectorisation could be used to calculate acceleration across all bodies. Vecotrisation applies mathematical operation to whole arrays instead of individual elements negating the use of a `for` loop in the current `get_acceleration_v2()` function. It is a superior technique however was omitted in this project since it leverages `Numpy` and its ability to pull all available threads on the system whilst also running in C. Alternatively, the loop could be parallelised but this could become convoluted very quickly.

A final attempt at speeding-up code would be in the form of compiling it using the Cython compiler. The compiled code can then be executed using a C compiler, addressing the overheads that may come with numerical loops in Python [6]. An alternative to this is using the `jit` 'Just In Time' decorator from the Python package `Numba`. It compiles function on their first run and stores them in cache. This means subsequent request of that function are readily available as low level code, making their execution time faster. There do exist caveats with this method, mainly being the code required needs to be simplistic and only use `Numpy` functions [7]. Attempts to use other libraries could slow the entire program down.

For completeness, alternative methods to solve the physics problem could be attempted. The RK4 and Verlet methods would provide better energy conservation undoubtedly giving a constant total energy for Fig 2. A result of the energy slowly decreasing is that given enough time, it will near 0 joules. This result for a gravitational system would mean bodies have dispersed to infinity with 0 velocity. The softening factor was added to mitigate this however the 'repelling' effect still occurs. To eliminate this fully, a collision detection method could be added to combine bodies at given radii. This would cause masses to stay involved within the larger system as oppose to being ejected.

References

1. Trenti, M. & Hut, P. Gravitational N-body Simulations. *arXiv.* eprint: `0806.3950` (June 2008).
2. Petersen, H. G. Accuracy and efficiency of the particle mesh Ewald method. *J. Chem. Phys.* **103,** 3668–3679. ISSN: 0021-9606 (Sept. 1995).
3. *The Barnes-Hut Tree Algorithm and its highly scalable parallel implementation PEPC - JuSER* Dec. 2022. `https://juser.fz-juelich.de/record/150809/files`.
4. Das, H., Deb, S. & Baruah, A. Optimal Softening for Gravitational Force Calculations in N-body Dynamics. *Astrophys. J.* **911,** 83. ISSN: 0004-637X (Apr. 2021).
5. *MPI: a message-passing interface standard : version 3.1 | WorldCat.org* Dec. 2022. `https://www.worldcat.org/title/931556155`.
6. Behnel, S., Bradshaw, R., Citro, C., *et al.* Cython: The Best of Both Worlds. *Computing in Science and Engineering* **13,** 31–39 (Mar. 2011).
7. Lam, S. K., Pitrou, A. & Seibert, S. in *LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* 1–6 (Association for Computing Machinery, New York, NY, USA, Nov. 2015). ISBN: 978-1-45034005-2.

Appendix

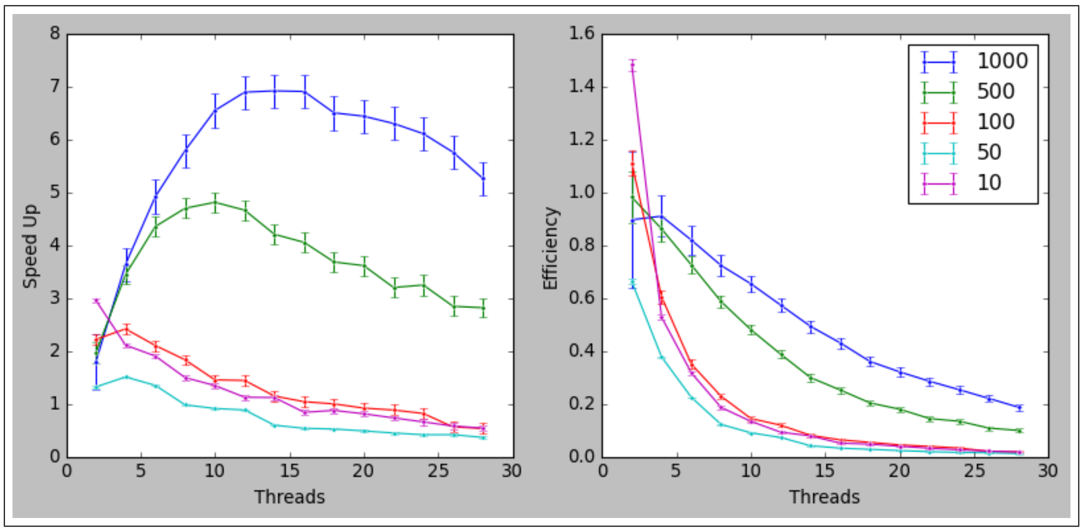

Figure 8: Full version of Fig 5 for multiprocessing.[Left] Plot showing speed up η due to number of threads for a five different problem sizes. [Right] Plot showing efficiency ϵ for the same data across multiple threads.

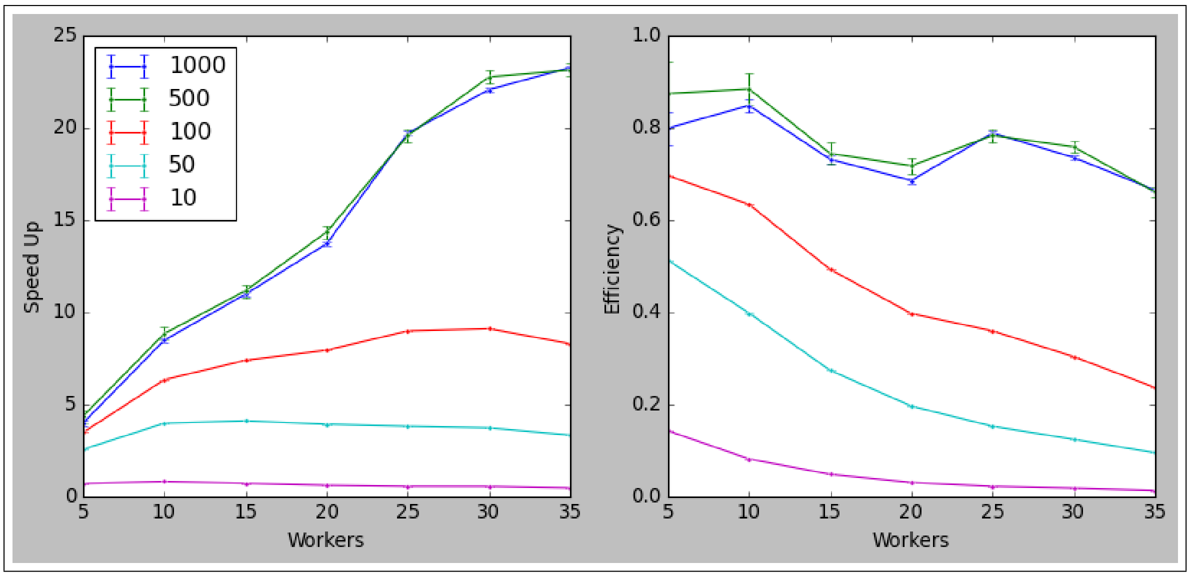

Figure 9: Full version of Fig 6.[Left] Plot showing speed up η due to number of workers for a five different problem sizes. [Right] Plot showing efficiency ϵ for the same data across multiple workers. Data was taken by reserving 5 nodes on BC4.

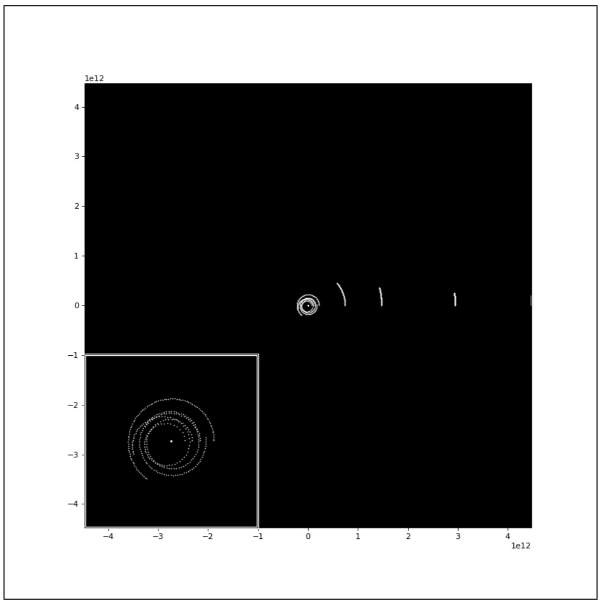

Figure 10: A still frame image taken from a simulation of our solar system for all eight planets and the Sun. Dots indicate a body's previous position showing circular/elliptical motion. The smaller plot focuses on the inner planets.

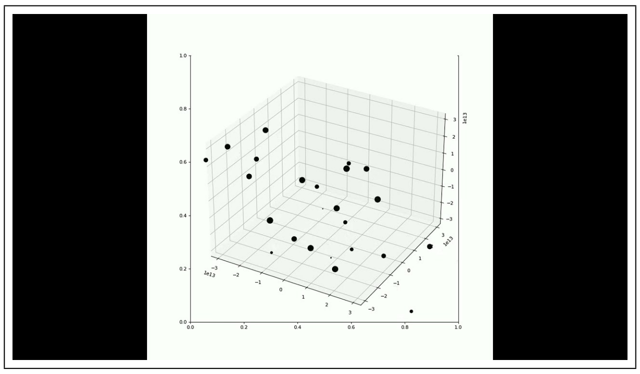

Figure 11: Repelling Bodies Video: Demonstrates how bodies can be ejected from the simulation with the current softening factor. Spheres are sized according to mass with colour pulsing due to an artefact. The final moments of the video shows interactions work as expected with a few bodies influencing each others' paths. [https://www.youtube.com/watch?v=1UCENWC88TU]

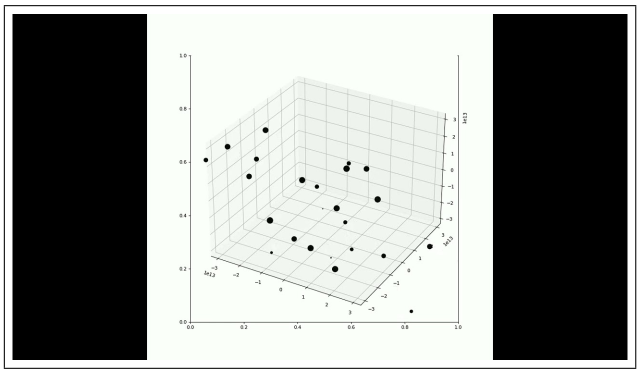

Figure 12: Solar System Video: Full video from which Fig 10 is pulled from. Demonstrates the physics is being solved as intended as bodies are on circular/elliptical orbits. [https://www.youtube.com/watch?v=aPwqSJPeoCo]