Creating a parallel direct gravity simulator

Gareth D. Price
Department of Physics, University of Bristol.
(Dated: December 7, 2023)

The N-body problem solved with a direct approach is an ideal use case of parallel computing, the pairwise calculations involved in simulating the system allows for massive speedups from increasing thread count. The speed increase of OpenMP vs Serial code has been evaluated for a gravity simulator capable of accurately modelling the solar system. A shared memory method was explored and a distributed memory method discussed.

INTRODUCTION

The N-body problem, the fundamental challenge in celestial mechanics, emerged when Isaac Newton's early predictions of planetary orbits proved inadequate in accounting for the complex gravitational interactions within the solar system. Newton's law of gravitation, which approximates the force between two bodies, lays the foundation for predicting these interactions. However, solving the N-body problem analytically becomes impractical or impossible as the number of celestial bodies increases, leading to a fundamental computational challenge. Many N body systems do not have analytical solutions. Newtons law of gravitation:

$$F = G\frac{m_1 m_2}{r^2}$$

where

- F is the force between the masses;
- G is the Newtonian constant of gravitation
- m_1 is the first mass;
- m_2 is the second mass;
- r is the distance between the centers of the masses.

In this context, the N-body problem involves predicting the motions of multiple celestial bodies under the influence of gravitational forces. The computational complexity of simulating such a system is inherently $\mathcal{O}(n^2)$ because the force between every pair of bodies must be calculated [1], resulting in a significant computational burden as the number of bodies (n) grows. The need for efficient solutions to the N-body problem has driven the exploration of parallel computing approaches to enhance simulation speed.

Then invention of computing allowed for the easy numerical analysis of N-body problems, the most intuitive of these methods is the direct method (also referred to as particle-particle methods). Direct methods numerically integrate the differential equations of motion, which requires a careful approach such as sometimes adding a softening factor to prevent the force tending to infinity at small distances. This softening factor is required as with finite step sizes a small enough separation of masses will result in near infinite acceleration.

The direct method of simulating an N-body problem is inherently and unavoidably a $\mathcal{O}(n^2)$ problem. This is because the force between every pair of bodies must be calculated and the number of pairs of objects for N objects can be calculated as N(N-1)/2.

This means that there is only 2 ways to reduce the time complexity of a direct N-body simulation: Making the serial algorithm more efficient, or running the algorithm in parallel

This paper investigates the application of parallel computing, specifically using the OpenMP API, to address the challenges posed by the N-body problem. The goal is to parallelize the direct method. The study explores the speedup achieved through parallelization and discusses considerations for optimizing the algorithm, paving the way for further scalability and performance improvements.

To achieve decent speedup from parallelisation, the algorithm needed to be highly efficient in serial. This principle can be demonstrated with Amdahl's law, where the maximum potential speedup of an algorithm on a parallel computing platform follows the equation [2, 3]:

$$S_{\text{latency}}(s) = \frac{1}{1 - p + \frac{p}{s}} = \frac{s}{s + p(1 - s)}$$

where

- $S_{latency}$ is the potential speedup of the execution of the whole task;
- s is the speedup in latency of the execution of the parallelizable part of the task;
- p is the percentage of the execution time of the whole task concerning the parallelizable part of the task before parallelization.

Therefore, if I refine the algorithm to shift nearly 100% of the time complexity towards the pairwise force calculation, the potential speedup could reach upwards of 100 times. However, considering that BlueCrystal is equipped with only 28 threads per node, the maximum achievable speedup using OpenMP would be capped at less than 28.

The N-body problem involves two main components: an NxN matrix multiplication that computes the cumulative effect of pairwise gravitational interactions among celestial bodies, and a straightforward arithmetic algorithm for numerical integration.

For these reasons, I chose to implement the entirety of the code for the simulation in the relatively low-level C programming language. By streamlining the algorithm in C to a theoretical minimum number of calculations, I ensure that the overwhelming contributor to the time complexity of the simulation lies in the NxN matrix multiplication, a process inherently capable of being parallelised [4].

The way the summation of the pairwise attraction is calculated in my code follows the following steps:

- Iteration is carried out over the array of N bodies
- For each body, iteration over the list of other bodies (excluding itself) to calculate the force between them
- Summation of these forces

This straightforward process suggests that the algorithm could theoretically be parallelized with N threads. Each thread could be assigned to handle the calculations for a specific body, allowing for simultaneous processing of all bodies and a speedup of N.

Of course for large N this would therefore mean needing to use more than 1 processor, necessitating MPI. I primarily focused on OpenMP to demonstrate the speedup of parallel processing.

GRAPHICAL VISUALISATION

Whilst the calculations of the N-body systems were done using C code, the graphs and visualisations were created using matplotlib in Python. The C code exports a variety of txt files with important information for the Python code to import. Position arrays were stored as whitespace delimited 2d arrays which were reshaped into 3d NumPy arrays upon loading into Python. Animations were generated and can be customised simply within the Python file. Due to the pdf nature of the report animations were not the main focus therefore only an animation of the 83 body solar system model was generated.

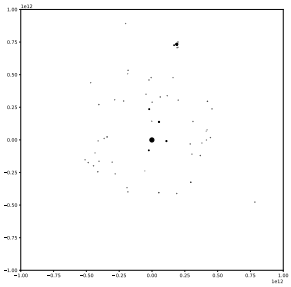

FIG. 1. 2D view of the current state of the Solar system evolved 365 days (May need to zoom in to see clearly)

These graphs and the animation could be used to visually determine the accuracy of the system, any large errors would be noticeable as the orbits of the solar system are highly stable. It also allows one to determine long-term accuracy as planetary harmonics are known so planetary alignments should occur extremely predictably far into the future [5]. There is significant research showing that planetary systems exhibit so much chaos that an N-body simulation is merely calculating a possible state for the system to be in [6] this is interesting because it limits the practical application of N-body simulations for far future speculation.

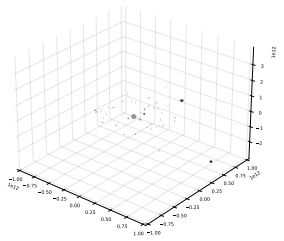

FIG. 2. 3D view of the current state of the Solar system evolved 365 days (May need to zoom in to see clearly)

DATASETS

I used two datasets in this project, one for creating the initial simulator and determining its accuracy, this is a dataset of 83 objects in the solar system including every planet, the major moons, dwarf planets, and asteroids. I created the Solar System dataset myself by retrieving position and velocity data from NASA's JPL Horizons API.

This data allowed me to model 100 years of the solar system and evaluate if any errors occurred and determine if the simulation handled a stable planetary system. When these initial conditions were evolved for 100 years total energy remained constant and earths orbital period, perihelion, and aphelion remained within real world margins

The other dataset used was a 1024 body system generated using NEMO (a toolkit for stellar dynamics) according to a Plummer model, I downloaded this dataset from the University of Maryland N-Body Data Archive [7]. This dataset was only used to test the speed of the simulator on large values of N, as the generated nature of the data makes it uninteresting to model.

PRINCIPLES OF PARALLEL PROCESSING

A serial programme contains a list of instruction for a processor to sequentially calculate. This is inherently slow as many instructions have no reason they need to happen sequentially. For instance in the kick, drift, kick algorithm in a serial

programme every particles position is updated in sequence, despite the fact that the result of the kick step does not depend on any other particle.

Parallel processing approaches this issue by utilising multiple processes, therefore more than one instruction can be calculated at once. Therefore particle A can update its position at the exact same time as particle B. The issue is though that those two processes need to access the same memory.

There are two ways to handle that issue, either the two processes can share a memory that they can both access, this is convenient but scales poorly therefore there is a limit on the practical number of processors in a shared memory computer, this is how OpenMP works. The other way to handle the issue is for the processes to have their own memory but transfer data between themselves, although in practice there is almost always a processor that holds the ”global” memory, which is the memory that is required to be coherent in the computer in order to run the programme properly.

A modern CPU is a single chip with multiple microprocessors (cores), allowing each core to handle separate instructions simultaneously. The memory structure of a typical multi-core processor includes dedicated fast memory for each core (L1 and L2 cache) and a shared, somewhat slower memory pool (L3 cache) for inter-core data sharing. Additionally, multiple CPUs can be connected via a low-latency network to distribute heavy computational workloads. When vast arrays of CPUs are connected it is referred to as high-performance-computing.

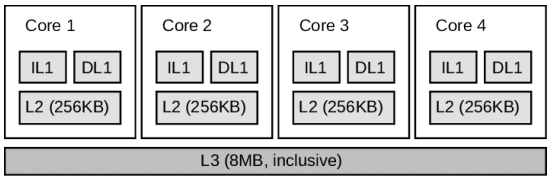

FIG. 3. Simplified modern CPU structure

SERIAL OPTIMISATION

I initially created a serial programme using the NumPy module in Python but it was very slow. I discovered a few tricks to optimising the serial parts of the code, this is important to reduce the non-parallelisable time complexity.

I chose to create the simulator in C due to the nature of the programme and the need to utilise parallel processing. C is faster for tasks like matrix multiplication and large loops compared to Python. This is because C is a compiled language with less overhead, making it more efficient for direct, low-level operations. Python, being an interpreted language, introduces extra layers of complexity and tends to be slower, especially in scenarios involving repetitive tasks like loops and matrix operations. So, for heavy computational work, C is a more suitable and quicker choice.

GCC (the compiler I used) has an optimisation flag -O3 that can be used to auto vectorise loops, using this flag the simulation is made roughly 3.64 times faster. This however does reduce the speedup of the OpenMP code, namely because it reduced the percentage of the time complexity that comes from parallelisable code vs serial code.

SHARED MEMORY

In shared memory systems the processors have fast access to a shared single view of the data and communication between processors is extremely fast. Modern CPUs tend to cache memory as even the shared memory is relatively slow when repeatedly accessing it. The nature of CPU’s caching memory causes two problems:

Access time degradation: Contention arises when multiple processors attempt to access a common memory location. Accessing neighboring memory locations can lead to false sharing. Due to this the scalability of shared memory computers is limited, often ten or fewer processors.

Lack of data coherence: If one processors cache is updated with information that needs to be updated for other processors this causes a problem where the programme must halt to share this information

IMPLEMENTATION OF OPENMP

When I implemented OpenMP on my code it resulted in a tremendous result. I was planning to run my code on the BlueCrystal Phase 4 compute cluster but I was unable to gain an opportunity to run my code due to limited time. Instead I ran all computation on my Ryzen 5 3600 CPU which has 6 Cores each at a base clock of 3.6 GHz.

Using 6 threads with OpenMP and auto vectorisation, a 1024 body simulation can be evolved for 1000 steps in just 3.997 seconds, the same problem in my original Python code (which utilised vectorisation through NumPy but was otherwise very inefficient) takes 95.77 seconds, 23.96 times longer. Something that can be clearly seen by this graph is the almost

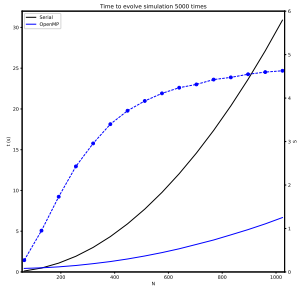

FIG. 4. Time taken against Number of bodies for Serial and OpenMP code

exactly $\mathcal{O}(n^2)$ relation between N and time complexity. In reality when adjusted for N you can see that there is an increase which is$\mathcal{O}(n)$ as well, this additional factor of the scalability of the programme also shows a significant decrease in the OpenMP code.

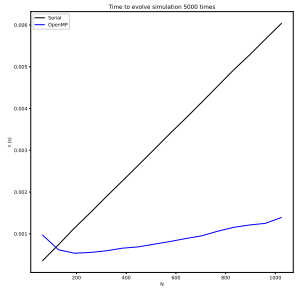

FIG. 5. Time taken normalised for N

The serial C code took 22.487 seconds meaning the utilisation of 6 threads vs 1 is a similar order of magnitude speedup (5.5 vs 4.26) as changing from Python to C.

The speedup of OpenMP was roughly equal to 90% of the thread count for high values of N, this is very promising as according to Amdahls law, a low number of processors such as 6 should see diminishing returns quickly unless the programme is highly parallelisable (e.g. 80+%). Unfortunately to determine the exact percentage of the computational complexity that is inherently serial we would need far more cores. Also due to the nature of the $\mathcal{O}(n^2)$ matrix multiplication and the parallelisable nature of matrix multiplication the simulation is likely nearly 100% parallelisable for large N.

In order to most effectively speed up the programme using OpenMP, any global (outside of the individual threads cache access) memory access must be limited. One area where I massively increased the utilisation of multiple threads was in the acceleration calculation.

In the serial function for calculating the new accelerations of each body it is most logical to simply add each bodies contribution to total acceleration as they are calculated

This can be thought of in these steps:

- Set Body A's acceleration to 0
- Loop through all other N bodies J=0 -¿ J=N
- For Body J calculate the acceleration in a 2 body system it would cause Body A to experience
- Add this to Body A's acceleration (stored in shared memory)
- Move on to the next Body

However in parallel loops this would result in the global variable (Body A's acceleration) needing to be repeatedly accessed by threads, whereas it is much faster in OpenMP to simply have each thread calculate a separate Bodies affect on A and then sum these values after the loop has completed.

This can be thought of in these steps:

- Create temporary variable acceleration, locally stored in the thread looping through A
- Loop through all other N bodies J=0 -¿ J=N
- For Body J calculate the acceleration in a 2 body system it would cause Body A to experience
- Add this to acceleration (stored in processor cache)
- Once loop is complete, access global memory 1 time to assign Body A's acceleration = acceleration

This change in the way variables were assigned resulted in a change from a 2x speedup to a 5.5x speedup for large N, massively increasing the effective parallelisation.

When these temporary acceleration variables were implemented it increased the speed of the OpenMP code drastically:

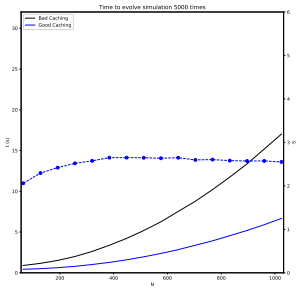

FIG. 6. Speed gain of OpenMP code with improved caching

The following is a graph of the time taken to compute 1000 evolutions of the simulation using different number of threads. This graph shows a close to exponential relation between the two variables which suggests an almost ideal relationship where a doubling of thread count halves time complexity. One way to visualise the relationship between compute time and thread count is to normalise the data. Time taken to compute 1000 evolutions was multiplied by thread count and then divided by the time taken in serial to find the relative total time spent computing by all processors. This graph appears to be linear which suggests we could draw a line of best fit and create a formula for compute time (for 1024 bodies). This would allow the calculation of the maximum viable thread count, above which the extra bloat from sharing memory would negate the parallel benefits of the code.

For extremely small values of N we would actually expect the serial code to run fastest, this is because it can exclusively utilise the faster L1 and L2 caches and does not have to initialise multiprocessing like OpenMP does. However in reality these tiny values of N (reasonably less than 10) are pointless

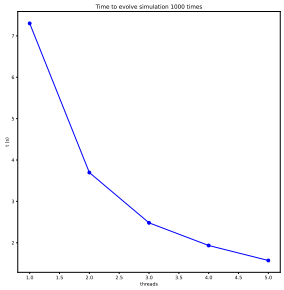

FIG. 7. Time taken to evolve 1024 body problem 1000 steps by thread count

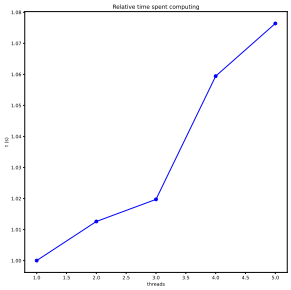

FIG. 8. Relative total time spent computing by thread count

to run unless using an incredibly small step size and a very accurate symplectic method of integration otherwise the loss of accuracy from disregarding so many bodies is not worth the gain in compute speed.

IMAGINING AN MPI IMPLEMENTATION

As MPI is designed around distributed memory systems there was little point in implementing it on my CPU, as without access to BlueCrystal I would be unable to reach the number of threads and CPUs required for MPI to be faster than Serial. However this does not prevent speculation about how MPI could be used to vastly speed up the simulation.

by splitting the list of particles into Y groups (Y is thread count) and each processor would be fed the masses and locations of every particle in the system but as they do not change on this step this data does not need to be kept coherent. The process would then sum up the forces on every particle in its group and update the global memory afterwards. This was each process has

In MPI a standard way to split up the processors is to have a master process and all others be slave processes, this way the force calculations could be run where the master process distributes a share of the force calculations to each slave process which then return the values. A logical way to split yp the force calculations would be by splitting the list of particles into Y groups (Y is number of processes) and each process would be fed the masses and locations of every particle in the system, as they do not change on this step this data does not need to be kept coherent. The process would then sum up the forces on every particle in its group and update the global memory afterwards. This way each process makes the fewest data transfers possible, the limiting factor to the speedup of MPI code.

Due to the global dataset of position, velocity, and acceleration vectors the N-body problem is not well suited for MPI, however due to how parallelisable the force calculations are, for large N MPI will be faster due to its ability to utilise multiple CPUs.

CONCLUSION

Whilst a large performance increase was obtained with OpenMP, the inability of this API to handle distributed memory well renders it ineffective for huge N-body simulations.

For models such as the solar system model with 83 bodies that is show in 1 OpenMP is by far the best choice and showed vast improvements of more intuitive methods such as NumPy in Python (22x faster). If OpenMP could be run with the 28 cores of a BlueCrystal node then it would likely approach 100x faster than the Python programme I started with.

I believe the ideal solution to the N-body problem would be a mixture of OpenMP and MPI, on a system with a large number of processes on shared memory such as BlueCrystal with its 28 cores per node the calculations could be split between 3 or 4 nodes and then OpenMP code run on those chunks to obtain a speedup of close to 100x the serial C code.

REFERENCES

1. Spurzem, R. Direct N-body simulations. *Journal of Computational and Applied Mathematics* **109,** 407–432. ISSN: 0377-0427 (1999).
2. Gustafson, J. L. in *Encyclopedia of Parallel Computing* (ed Padua, D.) 53–60 (Springer US, Boston, MA, 2011). ISBN: 978-0-387-09766-4.
3. Amdahl, G. M. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities* in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Association for Computing Machinery, Atlantic City, New Jersey, 1967), 483–485. ISBN: 9781450378956.
4. Chawan, P., K.Bhagat, S., Limbole, R., Sangale, A. & Sawant, S. Parallel Algorithms for Matrix Multiplication. *International Journal of Networking Parallel Computing* **1** (Nov. 2012).
5. Malhotra, R., Holman, M. & Ito, T. Chaos and stability of the solar system. *Proceedings of the National Academy of Sciences* **98,** 12342–12343. eprint: `https://www.pnas.org/doi/pdf/10.1073/pnas.231384098` (2001).

6. Hussain, N. & Tamayo, D. Fundamental limits from chaos on instability time predictions in compact planetary systems. *Monthly Notices of the Royal Astronomical Society* **491,** 5258–5267. ISSN: 0035-8711 (Dec. 2019).

7. https://carma.astro.umd.edu/nemo/archive/.