# Simulating Large Gravitational Systems Through Parallel and Approximate Techniques; The Barnes-Hut Method

*University of Bristol, James Paget*

$27^{th}$ November 2024

## 1 Abstract

Numerical study of physical systems has become increasingly important as greater computing power is made available through individual and multi-layered components, allowing more complex systems to be researched. In order to take advantage of this however parallel computing is required, of which 3 methods will be considered here. To explore the efficiency of these approaches a classical and Barnes-Hut approximation of gravitational motion will be simulated using both commercial and supercomputer hardware.

## 2 Introduction

### 2.1 Preface

For this project two computers will be used, the first of which is a *HP Pavilion* $14-dv2504sa$ which has a 64-bit $12^{th}$ *Gen Intel Core* $i5-1235U$ CPU with 10 cores (2 performance, 8 efficiency) with a base clock speed of $1.3GHz$, and turbo clock speed of $4.4GHz$ (changes depending on workload)[3] . The other computer is the *'Blue Crystal Phase 4'* supercomputer, which consists of approximately 580 nodes[4] , each containing 2 14-core *Intel* $E5-2680v4$ CPUs with a clock speed of $2.4GHz$[5] . Access to the supercomputer is performed through a remote login from another computer, wherein job batches can be queued via the *'slurm'*[6] queueing manager. These computers will be referred to as *'HomePc'* and *'BC4'* respectively for the rest of the report.

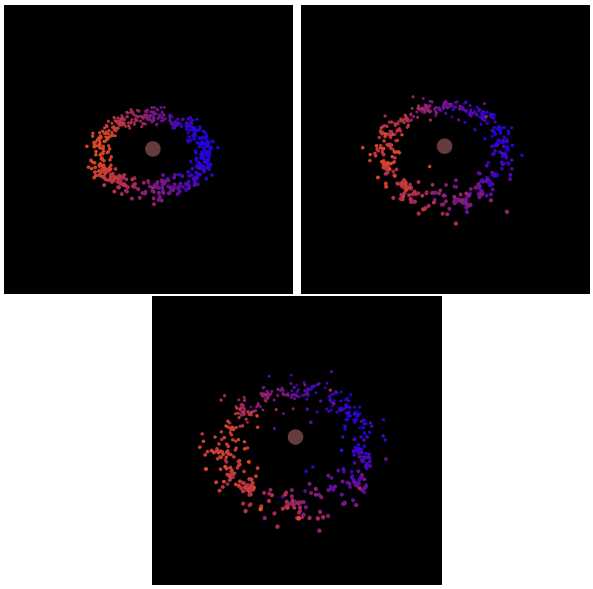


Figure 1: The evolution of a system of 500 particles initialised in a disc arrangement. Figures recorded at frames 0, 150 and 300, where some particles have began completing a full revolution after 300 steps. Stability of the ring pattern is dominated through the mass of the large central particle, annulus width and initial particle velocity. Colour here represents the velocity direction relative to the view perspective (interpreted as a red or blue-shift).

### 2.2 Simulation

In reality, gravitational forces act between all bodies with mass in the Universe according to the established equation (1) , however this approach will quickly leads to unfeasible computation times when scaled to larger systems, due to the $N^2$ relationship it follows ($N$ bodies, each with $N-1$ interactions). As gravitational force follows an inverse square law, it appears reasonable to assume that (relatively) close particles will dominate particle motion, and the remaining particles will have lower impact. This being said, distant but massive particles will have significant influence, and so these contributions must be included as well. The *Barnes-Hut* method, first published in 1986[8] , addresses these assumptions to allow a system to have forces calculated with order $Nlog(N)$. This approach utilises tree-based (referred to as an *Octree*) calculations and data storage to optimise particle calculations, and has been applied in extreme cases as described here[12] .

## 3 Theory

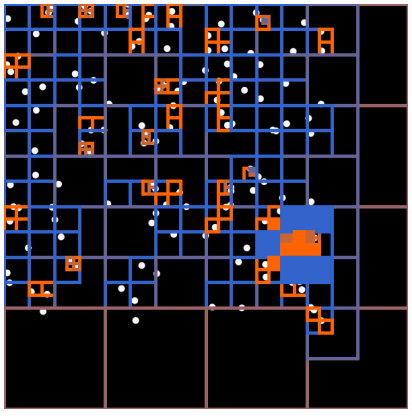


Figure 2: Structure of the Quadtree, with colour representing Quad depth in the tree. The filled-in regions highlight the Quads located through a nearby search algorithm within some circular radius around a point. The remaining points outside this range (non-solid) have their total mass and centre of mass collected during this process.

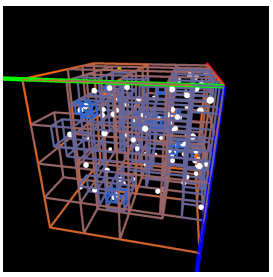


Figure 3: The generalised 3D case of a generated Octree. The colour of the Oct bounds represent the depth of the Oct in the tree. External Octs containing a particle are visualised (as well as internal Octs), whereas empty Octs are not displayed.

### 3.1 Octrees

Octrees are a recursive data storage structure that allow data to be linked through this recursion. Essentially the tree structure is made from underlying *Oct* class objects that can store other Octs (child Octs) as well as an arbitrary data object (particles in this case with mass, position and velocity). Starting from a single Oct, data can be placed into this structure, where it will either directly be stored or cause a split into 8 child Octs if data is already present, wherein the 2 data points will now be placed again at the split location. When this procedure is followed for $N$ particles a tree is formed with either a single or 'Null' piece of data in these final *'external'* Octs, and a chain of *'internal'* Octs leading to each. As data is placed this way, the particle parameters can be stored and summed in previous Octs. This structure and held information allows Octs near to a point to be found as well as all the mass excluded from this to be found simply as well, allowing the Barnes-Hut method to take advantage of this fast information collection at the smaller cost of the formation of this tree[8] .

$$\boldsymbol{F_{12}} = \frac{Gm_1m_2}{r_{12}^2}\hat{\boldsymbol{r}}_{\boldsymbol{12}} \tag{1}$$

### 3.2 Shared Memory

Parallel programming allows the tasks a program performs to be shared amongst other processes that occur

simultaneously. One approach to a program dealing with data in a parallel system is to have all memory of the program be shared between *'threads'* (a simultaneous computation of a section of code), used by the *OpenMP* (Open Multi-Processing) API which facilitates multi-processing. Regimes like this offer convenience as all threads being run have access to the same pool of information, meaning each can individually pull and push to this dataset, however without proper management this can ultimately lead to *race conditions* where threads simultaneously operate on a memory address, resulting in invalid values. Hence programs must be written to specifically avoid racing, as well as queuing slow downs too (queued threads will not race, but will lower performance). Thread management also incurs overhead time losses too, therefore sufficiently small operations will actual result in a net loss of computational time. Hardware limitations (such as slow data buses between memory blocks or smaller inter-CPU cache sizes) will make shared memory approaches more enticing, however, as distributed memory would prove slower for this specific computer's architecture.

### 3.3 Distributed Memory

This regime runs multiple parallel tasks (or *'ranks'*) on separate, local blocks of memory meaning information must be explicitly transferred between them if needed. *MPI* (Message Passing Interface)[10] is one implementation of this which manages communication between launched ranks. Due to this locality, race conditions are less problematic (but still possible) however queuing can occur more easily when combining data (managed through careful usage of blocking and non-blocking communication types). However, as mentioned, fast data buses are required to connect these separate memory blocks to capitalise on the parallel performance gain, and there are also overheads incurred from rank management as well as, crucially, data being moved between ranks.

$$SpeedUp = S_N = \frac{Time_N}{Time_0} = \frac{T_N}{T_0} \tag{2}$$

### 3.4 Hybrid Systems

The premise of distributed and shared memory are not mutually-exclusive, and so systems using both regimes are often used for further parallelism in a program, leading to further speed-ups (2) if implemented correctly. Often the approach is for several shared memory systems to perform their separate calculations, which are then brought together through distributed memory methods. More program complexity occurs because of this (as race conditions for both systems must be carefully managed), however the performance improvement and flexibility in which processes are parallelised can be hugely beneficial.

## 4 Method

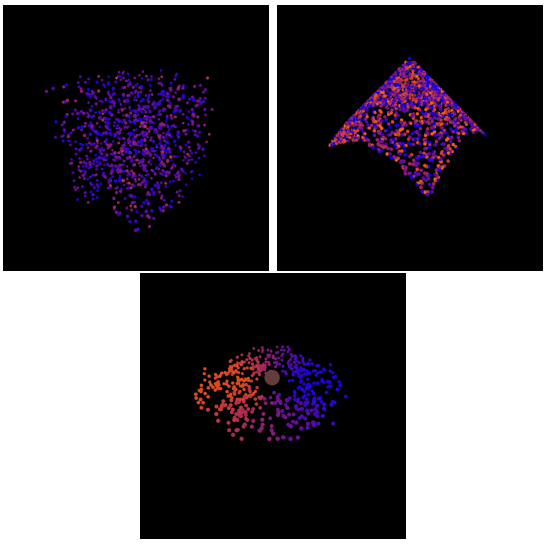


Figure 4: Shows the various allowed particle generation arrangements, which can switched between and configured if required. The cubic bounds of the tree used here are $2.5 * 10^9 m$ side lengths, $L$, where particles are removed if not placed into the tree within this region.

$$\frac{d\dot{\boldsymbol{x}}_1}{dt} = \frac{d\boldsymbol{v}_1}{dt} = \frac{\boldsymbol{F}_{12}}{m_1} = \frac{Gm_2}{r_{12}^3}\boldsymbol{r_{12}} \tag{3}$$

$$dx_1 = v_1 dt \tag{4}$$

## 4.1 Overview

The program is run in an OpenMP and MPI core Python run file, utilising compiled Cythonised methods. These latter files contain parallel methods for gravitational interaction calculations between a given target particle and a list of others (particles specified in a flat 1D form, listing its mass, position and velocity components 5 ), returning time-step particle parameters (according to Euler stepping (3) (4) with a chosen $dt$, but left generalised for other methods if required). The core Python files generate and traverse the Octree for particles and their interactions[11] , which are then calculated in parallel. Barnes-Hut interaction grouping is referred to as the $'ReducedTree'$ method within the program, as well as a $'PureLinear'$ and $'LinearTree'$ method referring to interaction groups being found through a direct 1D list of particles and a tree-based method, respectively (used to compare against the Barnes-Hut method).

## 4.2 OpenMP specifics

Cythonised OpenMP code implemented here uses the $'prange()'$ function to calculate interaction sets in parallel, receiving a memoryview of particles in a 1D format with the first 7 elements specifying the target particle, and the remaining elements specifying particles acting on this target. Forces are found and stored in a temporary 2D memoryview (each thread accesses and sums in reduction to a personal index), and these forces are summed through Numpy vectorisation afterwards, returning the updated target parameters. *noGIL* (Global Interpreter Lock)[9] is used throughout the looping section of this code block as is required by *prange*, but also offers improved performance (equally improved with *cython.boundscheck(False)* calls).

## 4.3 MPI specifics

For MPI's implementation, the same format of interaction data was given as used in OpenMP, however MPI required manual implementation of a static schedule, performed using Numpy slicing operations in a MASTER rank (chosen as the 0th) which then sent only the relevant data chunks to each WORKER rank (for rank numbers $n > 1$, meaning at least 2 ranks are required for the MPI code to function). Once received, each WORKER calculates their interacting forces in sequence, followed by a blocking *gather* [10] (which implements a more time efficient algorithm to collect the data from each rank then simply sending all data back to MASTER manually) called by all ranks to synchronise and gather forces found into the MASTER, which then returns a resultant updated particle. Similarly here this data must be kept separate from the Python's main dataset as each parallel operation interacts with this full set, and is synchronised at the end of each complete simulation time-step through an MPI *comm.bcast()*[10] call.

Additionally, MPI also implements a *main_modified()* method where each rank generates and searches the tree simultaneously, and only performs sequential interaction calculations every $n$ particles found (where $n$ is the number of WORKER ranks). This means work is distributed in a static format amongst ranks, and then gathered into the MASTER rank when completed, to then return a fully updated particle set (note the order ranks return particles is not important). This approach favours the Barnes-Hut method as interactions are kept smaller (less sequential time loss) and particle numbers are kept high (now searched in parallel, hence more efficient).

## 4.4 Hybrid specifics

A hybrid implementation of both methods (within *main_mixed()*) is also considered here as, unlike in MPI's situation, OpenMP could not be used to search the Octree in a parallel as it fundamentally involves Python Oct class objects, disallowed in OpenMP's noGIL regime. Therefore here the tree is searched using MPI, and the interactions calculated in OpenMP.

This allows systems with both many particles and many interactions to be considered and theoretically achieve large speed-ups.

# 5 Results

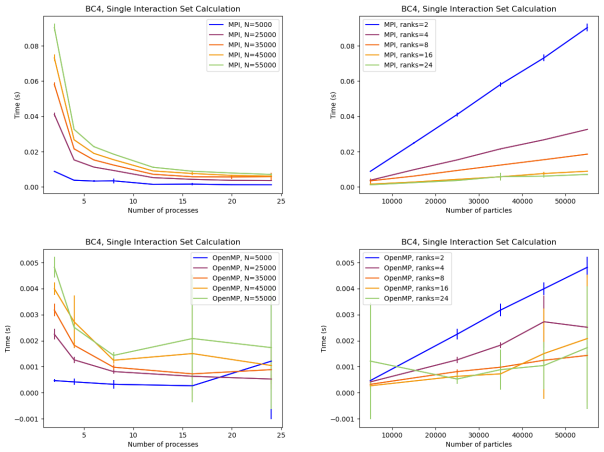



Figure 5: Timing data collected from BC4 off just the parallel process for both OpenMP and MPI, calculating just a single set of interactions of varying length. Results are averaged over 5 readings of $N$ randomly generated particles.

Data was collected for the speeds of just the parallel code sections, recorded directly from with the function using *'MPI.Wtime()'* and *'openmp.omp_get_wtime()'* respectively.

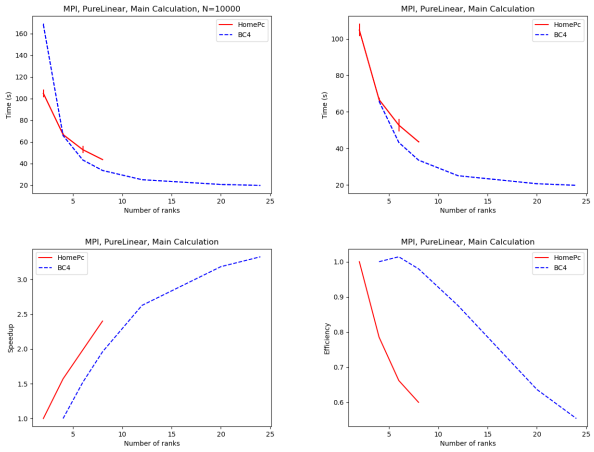
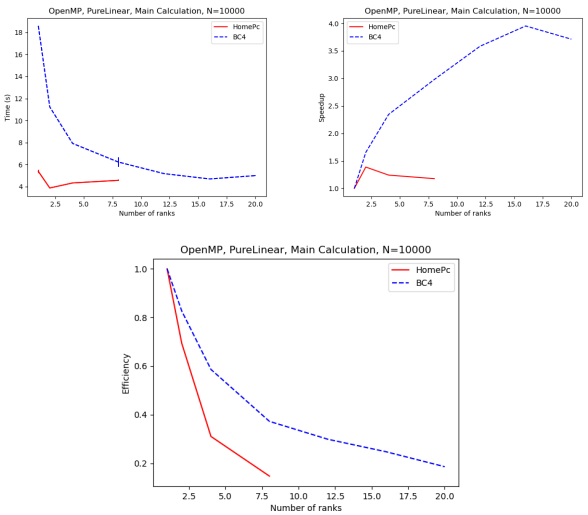



Figure 6: Data for a pure-linear system (all particles interact with all others) found on BC4 and HomePc for a large dataset, comparing OpenMP and MPI. This version of calculation did not search the tree for particle interactions, instead pulling the full list directly from the initial generation of the particles. This shows the times for a single time-step calculation (only 1 frame considered as each frame, in a random distribution, should be very similar hence this saves time and further time-steps would only produce similar results). 3 samples are averaged for each displayed data point.

A full frame of the simulation was timed using the *cProfile* library [2] when considering the *main()* function. This will serve to compare parallel scaling properties with the Barnes-Hut method later, and as further evidence that each parallel technique is working as intended in a more simplified case.



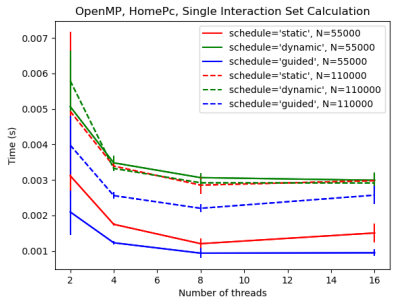




Figure 7: Data recorded on the HomePc, with 3 data points averaged for each rank considered, and a standard deviation error found from this. The schedules were tested for various thread numbers.

Experimentation with performance from different schedules is considered (for OpenMP's *prange()*). 'Static' scheduling breaks the problem set into chunks of constant size and assigns them to each thread. 'Dynamic' scheduling assigns a follow-up job to a thread once it finishes its previous, balancing workload as oppose to work number. 'Guided' scheduling assigns groups of jobs to each thread as it finishes its prior group, with the size of the group tapering to a chunk of 1 job at the end[9] .

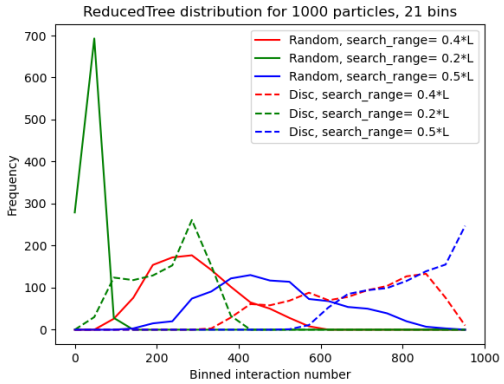



Figure 8: Data for the distribution of particle interactions (across all particles) in a Barnes-Hut calculation with varying search ranges, which define the circular radius considered by the Octree when efficiently looking for Octs, and then particles, to interact with. These calculations were performed on the HomePc for a single time-step (not averaged). L defines the the full width of the system (of the root Oct), taken as $2.5 * 10^9 m$ here.

Another tuning factor, explored in figure 8, considers varied Barnes-Hut search ranges, so applicability to different system sizes and parallel approaches can have their performance optimised.

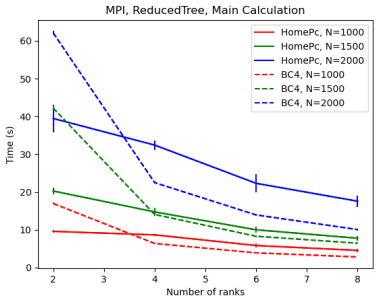



Figure 9: Data collected on both HomePc and BC4 for Barnes-Hut method, with a search range of 0.4*L, where L=$2.5 * 10^9 m$. This utilised the *main_modified* approach where the tree search is parallelised and the interaction calculation is linear.

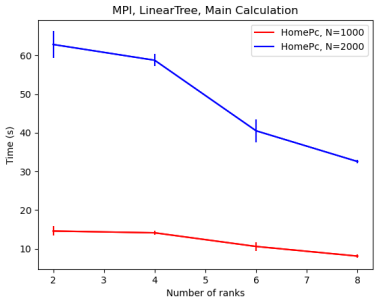



Figure 10: Similar approach to figure 9 , however each particle interacts with all others, not using the Barnes-Hut method. Slight performance improvement is shown for varying rank numbers, and is shown to scale worse with smaller system sizes.

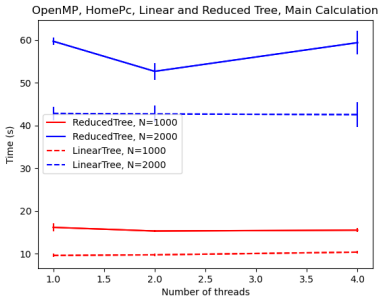



Figure 11: OpenMP data for both interaction calculations involving tree based searching. A search range of 0.4*L, where L=$2.5 * 10^9 m$, with the *main_modified* method is used here with an average of 3 samples per data point, with error bars shown.

Data for the tree-based interaction grouping methods are now compared in overall times and scalability with processes. Interestingly here figure 11 shows how the Barnes-Hut methods can be slower than linear interactions for smaller system sizes due to overheads of the nearby Oct search algorithm.

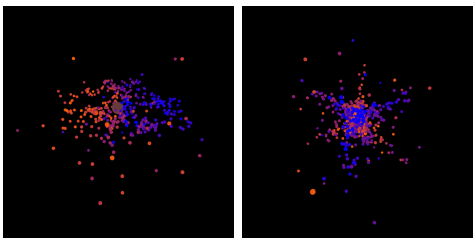


Figure 12: Evolved particle states when time-step or mass is too large ($dt = 10$, hence $dt = 0.1$ used now). 1000 particles simulated.

$$particles = [p_1, p_2, p_3, ..., p_N]$$
$$p_N =< m, x, y, z, u, v, w > \quad (5)$$

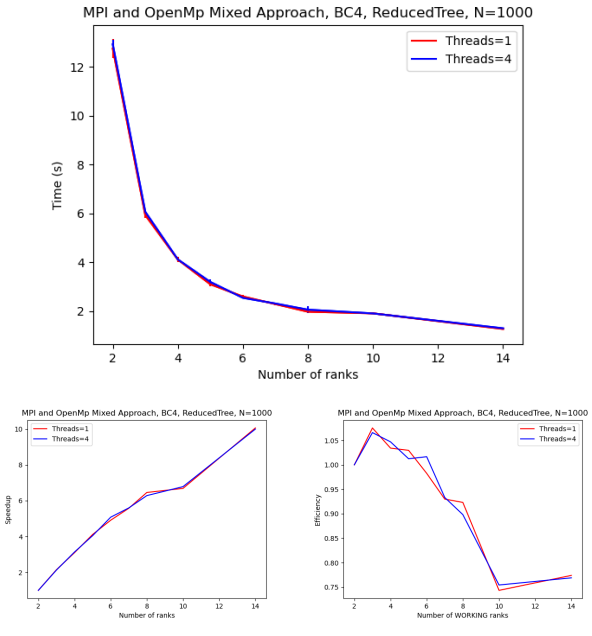



Figure 13: Data acquired from BC4 running the Hybrid approach of the main Barnes-Hut simulation calculation, timed for a single time-step, where 1000 particles were generated in a randomised 'disc' arrangement. A search range of 0.2*L was used, and 3 data points were collected an averaged, with error bars on the time figure.

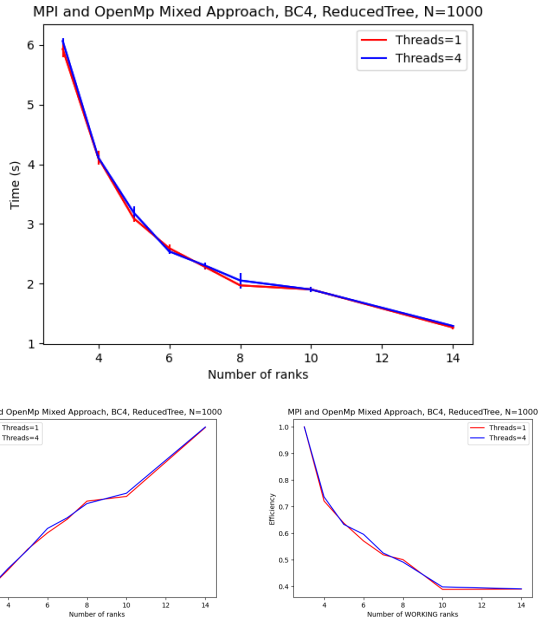



Figure 14: Data acquired from BC4 running the Hybrid approach of the main Barnes-Hut simulation calculation, as in figure 13 , except the data for 2 ranks is removed (outlier behaviour).

This data required 2 BC4 nodes to be generated when considering the 4 thread cases, as a maximum of 48 processes were running for the final data points.

# 6 Analysis

## 6.1 General

With the single interaction set calculation, figure 5 , it is observed that OpenMP performs its calculations much faster (roughly 5 to 10 times) than MPI. Currently MPI equally distributes particles to calculate in an interaction to all ranks equally based on the number of particles, not when interactions are finished to balance time spent by each rank. Hence it is possible for a slower rank to bottle-neck the others within the *comm.gather* step before contributions are totalled. OpenMP uses its inbuilt dynamic scheduling system however, and so would not suffer these losses. Both approaches show good time scaling with processes up to 10 cores, wherein OpenMP starts to show very high standard deviation and weaker scaling, but still strong

raw times.

$$Efficiency = E_N = \frac{SpeedUp_N}{Processes_N} = \frac{S_N}{P_N} \quad (6)$$

Major points to note for figure 6 is the unusual discrepancy in computation time between the HomePc and BC4 for OpenMP, and the difference in time between the two methods. The latter point is a result of the slower calculation times seen in figure 5 , however the scaling seen with processes is acceptable (but not exceptionally strong) as a result of a large portion of the program still being sequentially dominated by a slow Pythonic processes (retrieving interactions, building the tree, etc.), hence the value can only at best tend to this sequential time (as processes tend to infinity)[7] , a limitation seen in further figures too. The efficiency (6) of the MPI calculation is actually shown to be greater than the OpenMP (for each process number considered), and so despite being physically slower, it does scale better in parallel. As for the disparity between the HomePc and BC4 in OpenMP, it could be related to the amount of cache memory available in both systems, resulting in the HomePc being able to quickly access this large continuous array of interactions faster each particle than BC4.

Figures 9 , 10 and 11 demonstrate the effectiveness of the modified method for getting MPI speed-ups against the seemingly non-scaling approach used by OpenMP. Here, OpenMP parallelises only the interaction calculations due to the Pythonic class structure of the tree, so it cannot be considered in parallel in this regime[9] (this would not be true if the Octree was Cythonised, however this was not possible to to time restrictions), and so as seen in the single calculation figures 5 the time of this sequential Python tree searching far outweighs the extremely fast OpenMP interaction calculation. Hence the sequential time dominates to such a degree that only minute (far out scaled by the error involved) changes in time are present. This fact is exacerbated by the small system sizes considered, however the effect would have persisted significantly even for larger systems. The MPI results appear to show relatively good speed-up for the process numbers considered.

The hybrid figures 13 and 14 showcase some important remarks about the program so far. First and foremost, no significant different between the 1 and 4 thread cases can be seen, which in theory should have provided another sizeable increase in speed-up for the system. However, this result is not unexpected in this case as we have observed from figure 5 that the raw speeds of the OpenMP were fast, and we have already argued in figure 11 that the cost of searching the tree in Python, even when parallelised with the MPI inclusion, is too slow to see noticeable improvement from this addition. MPI parallel tree searching reduces previous sequential overheads, however very large systems would be required to see noticeable improvement from increasing threads. Critically these figures do show that for the full set of raw results we get some very good efficiencies for this method (when considering the 1 thread case) which would have in part been due to the slight anomalous data for the 2 rank system observed (breaking Amdahl's law [7] in which going from 2 to 3 ranks, or equally 1 to 2 WORKER ranks, lead to a larger than double speed-up which in reality was likely due to an uncharacteristically slow rank 2 run time). Even when omitting this anomalous 2 rank data point, we still see decent speed-ups, specifically a roughly 4.5 times speed-up when going from going from 2 to 13 WORKER ranks (hence considering efficiency of workers, not including the MASTER rank in this efficiency), gives approximately a 0.692 WORKER efficiency at 13 WORKERS, which is a strong result (note that the graph shows the full efficiency relative to the original 3 rank starting data, hence an efficiency of 1.0 at 3 ranks). This shows the parallel technique works well in the implementation, but other slow processes overall give slower performance over the full time-step.

### 6.2 Refinements

With the scheduling figure 7 it is important to note that the HomePc only has 10 cores, which is shown as the time taken tends to some value for later cores, and the efficiency cores do not appear to perform considerable work after 8 cores (before tailing-off occurs). Here the scheduling is shown to make a considerable impact on total time (with dynamic scheduling performing particularly well, and so was used in general for prior calculations), which becomes less significant the more particles were present.

Considering figure 8 allows different approaches to be optimised for the Barnes-Hut method. For example, a Hybrid approach (MPI tree searching over all particles, OpenMP calculation of interactions within each) would benefit (in terms of potential speed-ups) from a distribution weighted more towards a large number of interactions (search ranges closer to 1.0*L). For the *'main_modified'* approach described previously, having fewer interactions (closer to 0.0*L) would favour this, however obviously too few interactions would not accurately capture true gravitational effects, and too high would neglect the purpose of the Barnes-Hut method (reduces to PureLinear method at 1.0*L).

## 7 Improvements

An obvious improvement would be the Cythonisation of the Octree, leading to faster overall calculations, and so better practical application of the Barnes-Hut method. Doing this would also allow OpenMP to search through the tree in parallel, offering another optional avenue for parallelisation that also improves Barnes-Hut efficiency (larger data set, the particle number, to be given to each thread as oppose to the smaller interaction set). This means the hybrid program could also have the roles of OpenMP and MPI switched and compared to see which performs better, as well as seeing performance gain from both threads and ranks in general too.

The scheduling figure 7 showed that OpenMP had improved performance through dynamic scheduling for some of the mid to smaller sized system, and so implementing this procedure into MPI could provide some performance boost. This would also suit the Barnes-Hut as each particle may have a wide range of numbers of particles they interact with (figure 8 ), and so dynamic methods would better spread workload between ranks and so reduce wait times.

The performance of MPI communication may also have been improved through further use the *broadcast* and *gather* functionality, as oppose to *send* and *receive* communication, despite this sending extra unnecessary information. This is due to greater optimisation of these functions in their data collection methods across several ranks, favouring larger setups.

Further implementations of shared and distributed memory could also have been considered, such as *Numba*, to see how well this compared to OpenMP in terms of shared memory optimisation, allowing the simulation performance to be improved and further parallelism experimented with (2 key aims of this report). On this note, GPU testing would have allowed a slightly different philosophy of parallel processing to be explored, where vastly more threads are used for a series of smaller jobs (using of the order of at least 'a few 100 threads'[1] in several thread blocks). Due to large thread numbers used here GPUs have much greater performance than CPUs, but have their own difficulties in terms of programming complexity and structure.

## 8 Conclusion

The data collected here has shown both OpenMP and MPI to provide speed improvements to the purely linear and tree based techniques investigated, only being limited by slow Python execution times but otherwise each demonstrating good efficiencies, especially in the final Hybrid approach which although was not able to make effective use of its second level of a parallelism

due to these Pythonic limitations, still achieved substantial efficiencies and so effective parallel implementation. If tested for even larger systems this would further help improve efficiency through higher thread workload, and so dramatically improve the performance of the simulation compared to traditional sequential methods, and so this parallelisation has proven successful. With further Cythonised implementation of the Octree generation and traversal then it is almost certain that much shorter raw times would be observed, and so the efficiency of the parallel methods observed would persist and lead to faster computation time. When compared to linear interaction methods, in figures 13 , 9 and 5 we see this Barnes-Hut method implementation scales slightly better with system size, and so proves more suitable to explore large systems of particles.

## References


[1] Gpu performance background user's guide. *NVIDIA Corporation*, $1^{st}Feb$.2023.

[2] Python 3.13.0; the python profilers. *Python Software Foundation*, 2021-2024.

[3] Intel core™ i5-1235u processor. *Intel Corporation*, 2022.

[4] Acrc hpc system information. *University of Bristol, High Performance Computing*, 2024.

[5] Bluecrystal 4 and bluepebble technical specifications. *University of Bristol, High Performance Computing*, 2024.

[6] Slurm workload manager documentation. *SchedMD*, 2024.

[7] Jason D. Bakos. *Embedded Systems ARM Programming and Optimization.* Elsevier Inc., 1026.

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

[9] Stefan Behnel. Cython; using parallelism. *Cython3.1.0a1 Users Guide*, 2024.

[10] Lisandro Dalcin. mpi4py.mpi.comm. *MPI for Python Documentation*, 2024.

[11] Tom Ventimiglia  Kevin Wayne. The barnes-hut algorithm. 2011.

[12] Mathias Winkel, Robert Speck, Helge Hübner, Lukas Arnold, Rolf Krause, and Paul Gibbon. A massively parallel, multi-disciplinary barnes–hut tree code for extreme-scale n-body simulations. *Computer physics communications*, 183(4):880–889, 2012.