Final Report for Assignments 3-6
High Performance Programming

Olof Björck, Gunnlaugur Geirsson

March 20, 2019

Contents

1 Problem

The N-body problem is a common computer performance benchmark which uses Newtonian mechanics to simulate the interactions and movements of particles in space. It can be implemented numerically using Plummer spheres and symplectic Euler time integration. A straightforward implementation requires $O(N^2)$ computations for N particles but this can be improved by using a tree-type approach, the so-called Barnes-Hut algorithm. In this report, we describe and compare the process of solving the N-body problem in two dimensions, from the single-threaded straightforward implementation to a multi-threaded version of the Barnes-Hut algorithm using OpenMP and Pthreads to parallelize. We investigate the performance and complexity of the different implementations to determine their scalability and performance with the problem size N and the number of threads used.

2 Solution

Our solution algorithm can be generalized into the following steps, regardless of implementation:

- Read input data from the command line;
- Read input data about particles from input file;
- Simulate with some algorithm; and finally
- Write particles' updated data to output file.

2.1 Particle structure

The simulation is done by updating the state of the particles each given timestep, with the timestep size, and the number of timesteps to simulate, given as input. The input variables are all set as constants (keyword `const`) and every particle is represented by separate arrays of information, i.e. position, mass, and velocity. The two straightforward ways to represent the particles in C is either as an array of particle structs, or as a "particles" struct of arrays. We tested both methods for the straightforward $\mathcal{O}(N^2)$ and found no real speedup difference using either approach. When implementing Barnes-Hut, we decided to use a single particles struct of array pointers, as this allowed for greater optimization by removing the need to dynamically allocate a particle struct for each node.

2.2 Quadtree

The Barnes-Hut algorithm includes building a quadtree. In order to do this, we create a quadtree node structure consisting of a pointer to children nodes, the total node mass and center of mass, and the node side length and center position of the node. A subtle optimization to the node structure is that it is not necessary to store four separate pointers to each child, if all four children are contiguously allocated in memory. Only a pointer to the first child is required, which can then be incremented to reach the other children. The tool `valgrind` was of great use when implementing the quadtree, as buggy recursion in the tree would often cause null pointer exceptions, illegal memory reads and memory leaks during development.

With this tree-node structure we also update the compound node mass and center of mass for each node traversed when inserting a new node into the quadtree. That is, we do not need a separate function to calculate the node masses and center of masses after the quadtree had been built. This eliminates the need to traverse the quadtree yet again between construction and simulation. The conditionals when inserting a new particle are also ordered so likely conditions occur first, for example checking if a node has children before checking if it is already occupied (i.e. is a leaf or center node). The quadtree is also designed to never attempt to insert non-allocated nodes, and as such performs no null-pointer checking. This increases performance at the cost of some safety.

To improve performance we also perform no bounds checking when inserting a new node. This removes then need for a conditional when looping through the particles, at the cost of undefined behaviour if particles start leaving the defined playing field. This limits the model at the price of performance, a common optimization occurrence.

2.3 Simulation

When calculating particle interactions, it is apparent that computing the forces between particles is redundant. Instead of calculating forces, we calculate the resulting accelerations of the particles to avoid first multiplying by mass and then dividing by mass later.

The brightness information for the particles is not used in any calculation. We chose to store brightness in a separate array used only for writing to output file.

2.3.1 Straightforward $\mathcal{O}(N^2)$ approach

Comparing every particle requires the use of a nested for-loop which iterates over every particle, for every particle. However, when calculating the resulting forces (or accelerations), the straightforward approach allows for easy use of Newton's third law. Because the force between two particles acts equally on both, we only need to compute each interaction once. This reduces the range of the nested for loop to only include particles which the outer loop has not yet iterated over, effectively halving the number of calculations. It also removes the need to loop split to avoid computing a particle's interaction with itself.

2.3.2 Using a quadtree

The particle forces (and then the particle velocity and position) are calculated by recursively traversing the quadtree, checking if each node traversed satisfies the `theta_max` condition. If the condition is met, then the node is treated as a particle and the acceleration computed, otherwise the tree traversal continues. Due to the tree structure there is unfortunately no simple way to utilize Newton's third law with this approach, unlike the $\mathcal{O}(N^2)$ approach.

2.4 Parallelization of Barnes-Hut

2.4.1 Parallelism in the algorithm

Most of the computation time in each timestep is spent updating the particles, not building the tree (see table 1). It is thus logical to focus any efforts of parallelizing the code on the particle update algorithm.

The computations to update particle positions are entirely parallel for each particle within a single timestep, i.e. the particle positions can be updated without affecting one another, in any order. A positive and very welcome result of the information necessary for the computations being stored in the quadtree is that there is no data dependency between the particles being updated; the computations are thus easily parallelized.

2.4.2 Pthreads

The parallelization using Pthreads is done by splitting the serial updating of particles into parallel tasks of equal workload (i.e. static workload distribution) for each thread, with the last thread making sure that all particles are updated even if the number of particles is not evenly divisible by the number of threads. This approach causes the last thread to work on more particles than the other threads, possibly causing a load imbalance and a negative effect on synchronization. One solution is to perform a second parallelization of the particles that did not fit in the first parallelization. However, this did not lead to any speedup, likely due to the increased overhead cost of a second parallelization, and that the leftover particles are very few unless using many cores, e.g. a maximum of three extra particles when using four cores. A greater hurdle is that some particles are more expensive to compute than others, which also results in load imbalance. The only way to improve this would be to write a more thorough, dynamic workload distribution.

2.4.3 OpenMP

The parallelization using OpenMP is done by splitting the serial updating of particles using OpenMP precompiler pragmas. This ensures that the code works regardless of if OpenMP is used when compiling or not. The OpenMP parallelization otherwise works the exact same way as with Pthreads, with the exception of dynamic work scheduling per thread. As mentioned, some particles are very clustered and fast to

compute with Barnes-Hut, while others are far away and thus much more expensive. With static scheduling this causes load unbalance, but by using some form of dynamic scheduling it becomes easier to achieve a balanced workload. This becomes is more true the more threads are scheduled, and thus we may see greater speedup with more threads than CPU cores, until the thread overhead starts to dominate.

As it is simple to parallelize small loops with OpenMP, it is tempting to parallelize every trivially parallel operation in the code. However one must always consider the thread overhead. For example, parallelizing nested loops in recursive functions, which occurs when creating or traversing the tree, results in the creation and destruction of an exponential number of threads based on the recursion depth and thus a decrease in performance compared to a serial approach.

3 Performance and discussion

The Barnes-Hut algorithm is not trivial to parallelize into separate tasks; hence, we worked together most of the time when writing and debugging the code, and when writing the report. There was no clear division of work.

The program was compiled and run on an Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz processor with 2 multi-threaded cores (4 threads) in macOS Mojave version 10.14.3 (18D109). The compiler used was Apple LLVM version 10.0.0 (clang-1000.11.45.5). This will be referred to as the 2-core computer. We also investigated the speedup using several threads on the university server Vitsippa which uses an AMD Opteron(tm) Processor 6282 SE with 16 multi-threaded cores (32 threads) The compiler used was GCC 4.4.7 20120313 (Red Hat 4.4.7-23). This will be referred to as the 16-core computer. All timing results presented are the real-time (wall time) runtimes of the program, timed with the built-in timing command `time` in the terminal (bash).

3.1 Optimization techniques attempted

We used some general serial and parallel optimization techniques throughout the code:

- Strength reduction: for example, changing the theta condition to multiply `theta_max` by the particle/node distance instead of dividing the node size by it provided a noticeable speedup of about 5%;
- Evaluating cheap conditionals before evaluating more expensive ones: for example, evaluating whether a quadtree node has children before checking `theta_max` during simulation;
- Tried to do as little work in general as possible: for example, computing the compound node masses and centers of mass while building the tree, instead of traversing it again;
- Tried to improve branch prediction by doing as little work as possible inside loops specifically, and declaring loop conditions to be known at compile time whenever possible. Most of these are done at the cost of safety, such as ignoring null-pointer checking when building the quadtree;
- Usage of keywords: Declaring `const` and `__restrict` was done whenever possible. Keyword `__attribute__((pure))` was utilized for one function, used for finding the correct quadrant to place a new node in, but it did not seem appropriate for most other functions. Keyword `inline` has been tested for multiple functions (barring recursive ones), but seems to offer no speedup in any case - the compile; recddis oneit so whetnhw to irline onen tor
- Padding and packing: Variables in structs were ordered largest first, so as to minimize padding. Use of keyword `__attribute__((packed))` decreased performance in all cases and so was not used. Keeping structs as small as possible proved important for good performance. For example storing the node children contiguously, so the node only requires a single pointer instead of four, reduces its size by 24 bytes and noticeably increases speedup.
- Load balance: The Pthreads implementation statically allocates workloads per thread. This is improved by the OpenMP implementation, which can dynamically schedule work distribution. We found the best performance with `type=auto` (same as `type=dynamic`).

The code was compiled with flag `-O3` (`-O3` performed better than any other `-O` flag) and using `-march=native`. In addition, the following compiler flags and general optimizations were used when compiling and throughout the code; due to all of them increasing speedup, even if just slightly.

- `-ffunroll-loops`: It is often necessary to loop through all four children of a quadtree node, which are unrolled with this compiler flag instead of manually. The same is true for any loops which make or join Pthreads. Due to the program relying more on recursion than loops, there was no major speedup when using compiler unrolling, nor any obvious cases of manual unrolling in the code (besides the trivial cases mentioned).
- `-ffast-math`: Using this flag provided more speedup, at next to no increase in error.
- `-ftree-vectorize`: Included in -O3. Due to the inherently recursive nature of the code, there are not many operations which are simple to vectorize. Only the function call to insert a node is vectorized according to -fopt-info-vec. Either way, it appears to give a slight speedup. Explicit vectorization was not done in the code.

3.2 Best results

The best time found for 5000 particles (from input file `ellipse_N_05000.gal`) with 100 timesteps, step size 0.00001, and `theta_max` = 0.12 was 1.733 seconds. This timing was achieved with the Barnes-Hut implementation parallelized using OpenMP.

3.3 Space complexity

The space complexity of Barnes-Hut compared to the $O(N^2)$ method is shown in figure 1. The Barnes-Hut clearly has a much better performance for large N, with an order of around $O(N^{1.2})$ for large N. Due to the tree-search method involved and the slightly superlinear performance, it seems reasonable to assume that Barnes-Hut is $O(N \log N)$ for high enough `theta_max`. We used `theta_max` = 0.12 which we found to result in an acceptably low error. A comparison of the Barnes-Hut complexity for different number of threads is shown in figure 2. We can see that the complexity remains the same for any number of threads, but that using 4 to 8 threads inc ebsases eperetter than the single-threaded version.

3.4 Bottlenecks

Using `valgrind cachegrind` it seems that the greatest serial code bottleneck is during simulation, when checking for `theta_max`, with around 20% of branch predictions failing on this condition. There is no clear solution to this however, as the check must be made to decide whether to continue traversing the tree. Unfortunately we are unable to utilize `gprof` to study the bottlenecks further, due to problems generating `gprof` profiling data for the program on both our computers and the university server vitsippa.

3.5 Parallelization

As we could not utilize `gprof` to profile the execution time of the program, we have to rely on software timers and use the C function clock() instead. It is apparent that the earlier assumption, regarding the computation time required to create the tree being marginal compared to the time spent updating the particle positions, is indeed true for large N. This can be seen in table 1. Hence our decision to focus on parallelizing the simulation part of the code is justified. This can also be seen by using the tool `valgrind callgrind`, which shows that the majority of all function calls are done by the particle simulation as opposed to the node insertion (around 3 billion function calls against 50 million for a $N = 3000$ ellipse galaxy).

3.5.1 Using 2 cores

The speedup using the 2-core computer by using several threads in a simulation of $N = 10000$ particles is shown in figure 3, and in a simulation of $N = 1000$ particles is shown in figure 4. The corresponding execution times for the speedup plots $N = 10000$ and $N = 1000$ are shown in figures 5 and 6 respectively. It

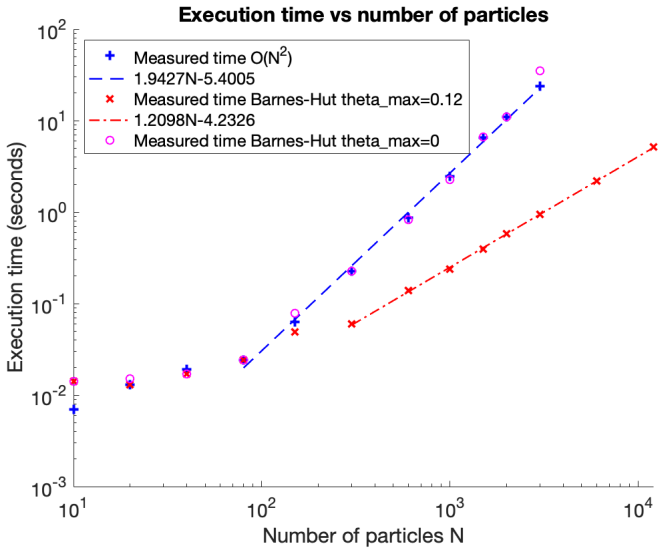


Figure 1: The execution time plotted against number of particles included in the simulation for a naive implementation and a single-thread Barnes-Hut implementation of the N-body problem. The $O(N^2)$ complexity is shown by the straight line $1.9427N - 5.4005$ with slope approximately 2 and the Barnes-Hut complexity is shown by the straight line $1.2098N - 4.2326$. Both algorithms were run with 100 time steps.

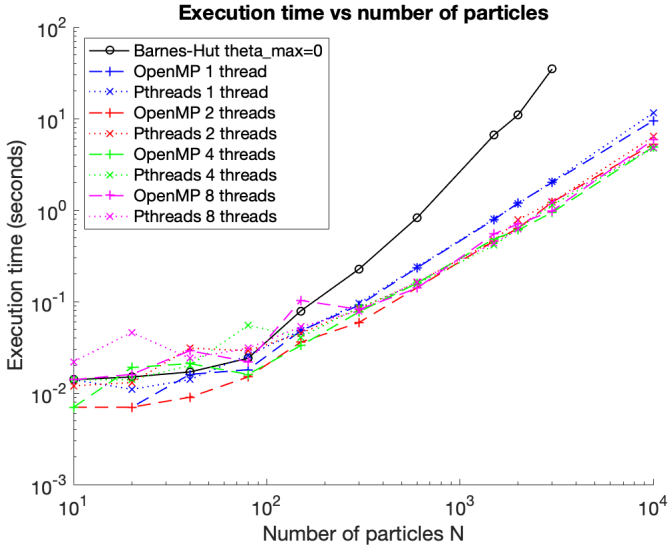
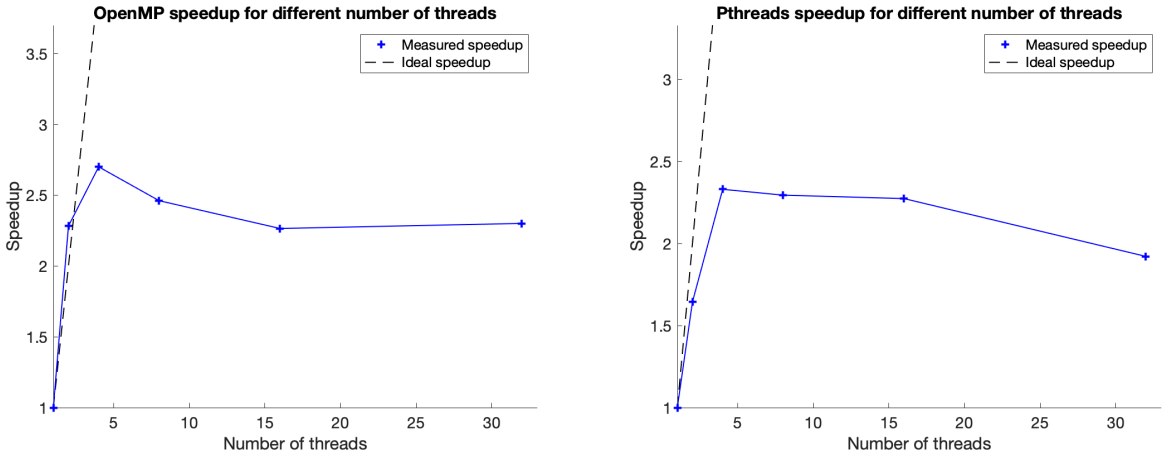


Figure 2: The execution time plotted against number of particles included in the simulation for a naive implementation and the Barnes-Hut implementation of the N-body problem. The Barnes-Hut implementation is multithreaded for various number of threads. Both algorithms were run with 100 time steps.

Table 1: Total serial execution time of different parts of code, simulating an elliptical galaxy of size $N = 10000$ for 100 timesteps, with `theta_max` $= 0.2$

Computation	Time in seconds [s]
Building and freeing quadtrees	0.1832
Simulating particles	12.2821



(a) OpenMP **(b)** Pthreads

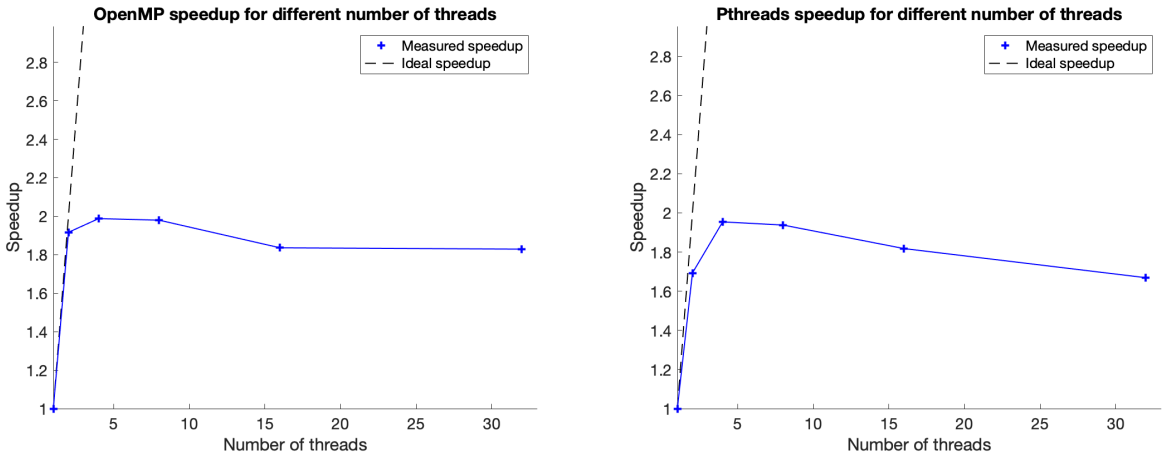
Figure 3: Speedup plotted against the number of threads used in the simulation for a galaxy of size $N = 10000$ on the 4-core computer. Additional parameter values: `n_steps` $= 100$ and `theta_max` $= 0.12$

is obvious that multi-threading gives a significant speedup over entirely serial code, and that the OpenMP implementation slightly outperforms the Pthreads one. In both cases the speedup when using 4 threads is approximately 2 compared to using 1 thread. However, increasing the number of threads beyond 4 does not seem to entail any speedup in any case, but rather decrease performance as the thread overhead increases. This is expected behaviour, as the processor used has 4 threads. The speedup when using several threads in a smaller simulation of $N = 1000$ particles is shown in figure 4. The speedup is not as large as with $N = 10000$ which can be explained by the smaller simulation not making as much use of the threads every iteration, more relative time is spend on the thread overhead.

3.5.2 Using 16 cores

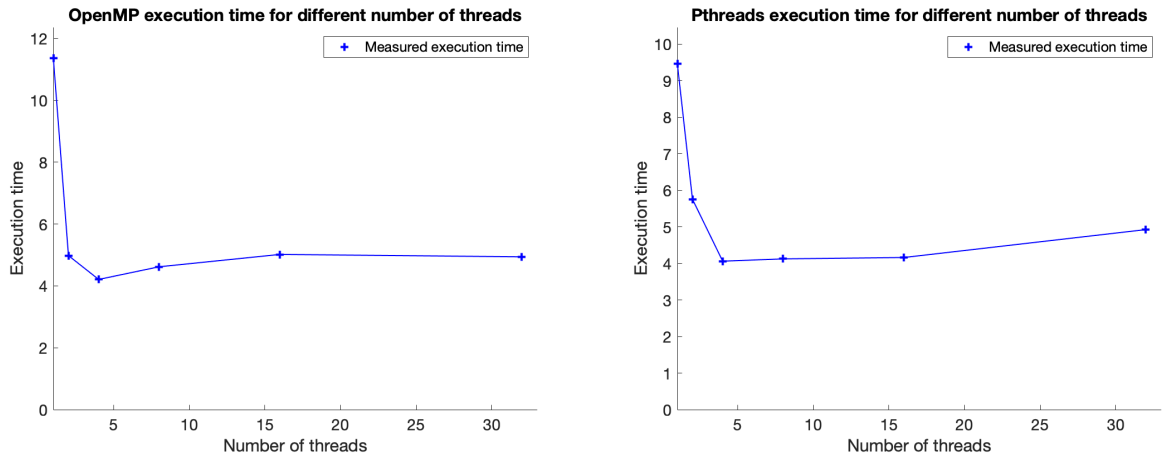
The speedup by using several threads in a simulation of $N = 10000$ particles using the 16-core computer is shown in figure 7, and the execution time is shown in figure 8. From this we can discern that the speedup is the greatest using 128 threads for OpenMP and 64 cores for Pthreads. Similarly to using 2 cores, it is obvious also for 16 cores that multi-threading gives a significant speedup over entirely serial code. The OpenMP implementation is slightly faster with a speedup of approximately 12 than the Pthreads one with a speedup just below 12. It is interesting that using both OpenMP and Pthreads, the greatest speedup was achieved using more than 32 threads, more than the number of threads on the computer processor. The OpenMP case is especially curious, as using as many as 128 threads should in theory only increase thread overhead without improving parallel performance. We believe the explanation for this is the dynamic workload scheduling with OpenMP increasing performance, as theorized in section 2.4.3.

Another possible explanation is that this is an anomaly due to the server diverting processing power to other server users, causing some runs to appear slower than they would truly be if we had all the server time for ourselves. This could be remedied by doing multiple (preferably thousands) of runs and averaging the computation time.



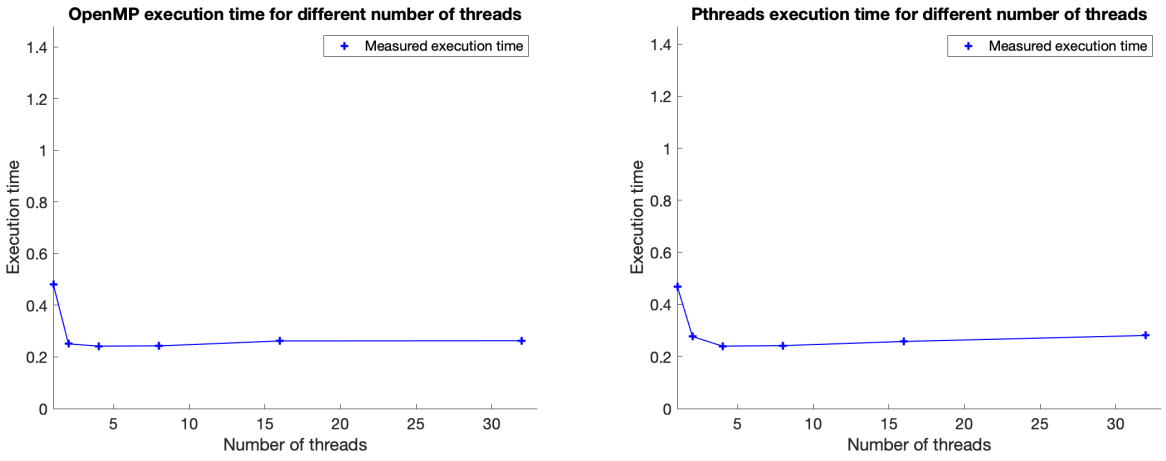
(a) OpenMP **(b)** Pthreads

Figure 4: Speedup plotted against the number of threads used in the simulation for a galaxy of size $N = 1000$ on the 4-core computer. Additional parameter values: `n_steps` $= 100$ and `theta_max` $= 0.12$



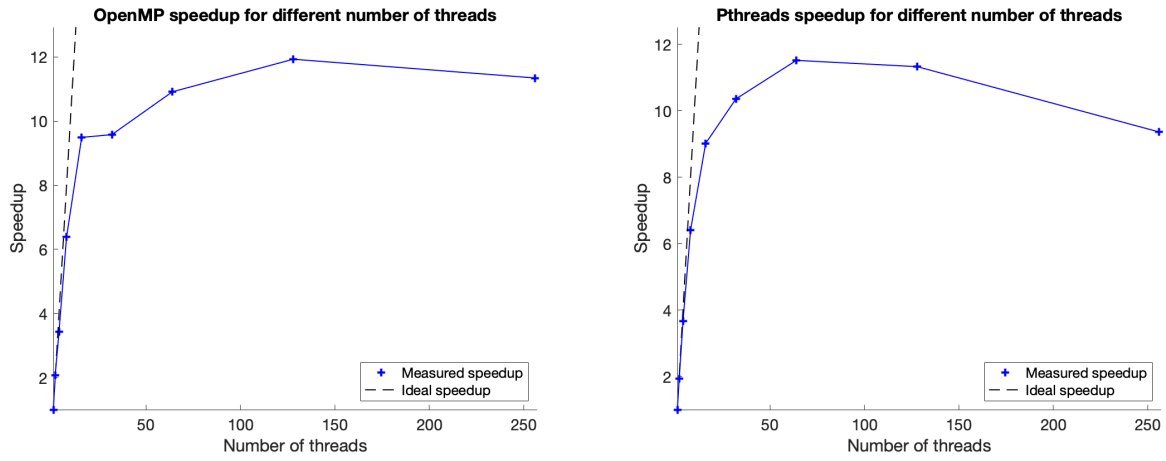
(a) OpenMP **(b)** Pthreads

Figure 5: Execution time against the number of threads used in the simulation for a galaxy of size $N = 10000$ on the 4-core computer. Additional parameter values: `n_steps` $= 100$ and `theta_max` $= 0.12$.



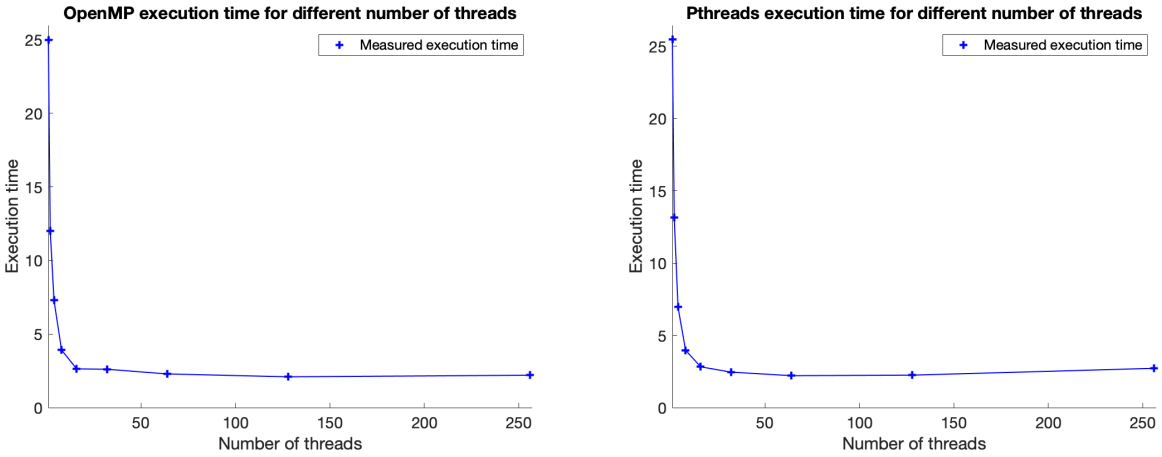
(a) OpenMP **(b)** Pthreads

Figure 6: Execution time plotted against the number of threads used in the simulation for a galaxy of size $N = 1000$ on the 4-core computer. Additional parameter values: `n_steps` $= 100$ and `theta_max` $= 0.12$



(a) OpenMP **(b)** Pthreads

Figure 7: Speedup plotted against the number of threads used in the simulation for a galaxy of size $N = 10000$ on the 16-core computer. Additional parameter values: `n_steps` $= 100$ and `theta_max` $= 0.12$



(a) OpenMP

(b) Pthreads

Figure 8: Execution time against the number of threads used in the simulation for a galaxy of size $N = 10000$ on the 16-core computer. Additional parameter values: `n_steps` $= 100$ and `theta_max` $= 0.12$.