Final Report Assignment 3-6

High Performance Programming

Author:

Fredrik Gustafsson

Sara Hedar

Uppsala

March 19, 2019

The Problem

Introduction

When simulating a large number of objects, such as a n-body system of planets and stars, the choice of algorithm and optimisation techniques is essential to increase performance. To reduce the execution time compared to a naive $\mathrm{O}(n^2)$ algorithm, the Barnes-Hut algorithm was implemented to simulate the n-body system. To further improve performance the code was optimised using serial optimisation and part of the code was computed in parallel using Pthreads and the API OpenMP.

Theory

The attractive force between objects obeys Newton's law of gravity. The force acting on object i can thus be written as:

$$\boldsymbol{F}_i = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{r_{i,j}^2} \hat{\boldsymbol{r}}_{ij}, \tag{1}$$

where G is the gravitational constant, N is the total number of objects in the system, m the mass and $r_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2$. However, there is a built in instability when $r_{ij} << 1$. To handle this we modify equation 1 into:

$$\boldsymbol{F}_i = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{i,j}^2 + \epsilon_0)^3}, \hat{\boldsymbol{r}}_{ij}. \tag{2}$$

To update the velocity and position of an object i from time n to time $n+1$ the following time scheme was used:

$$\boldsymbol{a}_i = \frac{\boldsymbol{F}_i}{m_i}, \tag{3}$$

$$\boldsymbol{u}_i^{n+1} = \boldsymbol{u}_i^n + \Delta t \boldsymbol{a}_i^n, \tag{4}$$

$$\boldsymbol{x}_i^{n+1} = \boldsymbol{x}_i^n + \Delta t \boldsymbol{u}_i^{n+1}, \tag{5}$$

in which $\boldsymbol{u}_i^n$ is the velocity of object i at time step n, $\boldsymbol{x}_i^n$ is position of object i at time step n and Δt is the difference in time between two time steps.

Algorithm

A naive way to perform a n-body simulation is to simply loop over each object i and compute the force acting on it from each object j, using equation 2. The problem with this implementation is that the algorithm in question would have a time complexity of $\mathrm{O}(n^2)$. So when the number of objects to simulate is increased with a factor of 2, the execution time of the program is expected to increased by a factor of 4.

To reduce the execution time for large simulations the Barnes-Hut algorithm, with a complexity of $\mathrm{O}(n \log n)$, when implemented correctly, can be used. In the Barnes-Hut algorithm the objects to simulate is stored in a quadtree (when simulating in 2D space). When the distance from i is sufficiently large we approximate several objects as a single body with their combined mass and position of the centre of mass. Both the combined mass and the sum of each objects position multiplied with its mass (here after called centre of mass) of the objects further down the tree is stored in each tree-node. Both values are then updated when a new object is added further down the tree.

To determine if a group of objects can be approximated as a single body a value θ is calculated for each region and compared with a chosen parameter $\theta_{max} \in [0, 1]$. The θ value of a region is defined as:

$$\theta = \frac{\text{width of current box containing particles}}{\text{distance from particle to centre of box}}. \tag{6}$$

When $\theta \leq \theta_{max}$ the region is treated as an equivalent mass. Compared to the naive algorithm this implementation of the Barnes-Hut algorithm is equivalent, when it comes to complexity, for $\theta_{max} = 0$. When θ_{max} is increased larger approximations are allowed and the complexity goes towards $\mathrm{O}(n \log n)$. However, when using the Barnes-Hut algorithm one should keep in mind that the performance gain comes at the cost of lowered accuracy of the simulation.

Parallelisation

To increase performance while still maintaining a sufficiently low power consumption parallel computing can be utilised. Parallel computing can be defined as "simultaneous use of multiple computer resources to solve a computational problem" [1]. A thread can be defined as an "independent stream of instructions that can be scheduled to run as such by the operating system".

POSIX (Portable Operating System Interface for UNIX) threads or Pthreads is a portable standard for threaded programming in C. Pthreads are used on shared memory computers where all threads have access to global data. Phreads takes a low level approach to threading and is considered easier to program compared to local name space models using message passing (MPI).

OpenMP uses the shared address space model and is based on threads. Compared to Pthreads, OpenMP is a higher level approach as it uses compiler directives. The threads in OpenMP have access to global data while also having access to private data on stack.

The Solution

The initial condition to the system of stellar objects was provided as an binary file containing floating point values of type `double`. For each object 6 values was supplied; position in the two dimensions, velocity in the two dimensions, mass of the object and brightness of the object. Both mass and brightness was considered constant during the simulations while both the positions and velocities were updated. When the simulations was completed the results were to be written to a binary file on the same format as the input data (initial conditions provided).

The problem was divided into three parts; loading the initial conditions from a binary file, performing the simulation and storing the final solution into a binary file. The problem was solved in one single .c file, with multiple subroutines. The number of objects to simulate, the input file name, the number of time steps, the size of the time step, the option to use graphics and the number of threads was given as input parameters to the program. The graphics option was not implemented.

In the quadratic implementation of the simulation a statically allocated array was used to store the brightness and dynamically allocated array was used to store the position, mass and velocity. Both the position and velocities were stored using the composite data structure `struct`, which in this case consisted on two double values. A pointer to the position, mass and velocity was sent into the subroutine which advances the system one time step. This subroutine was called the same number of times as set in the input parameters to the program. The brightness data was not sent to the time step function to improve data locality.

In the Barnes-Hut implementation a dynamically allocated array of `struct` was used to keep track of all of the stellar objects. Each `struct` contained the position and mass data of a specific object. To implement the algorithm a quadtree was constructed in each time step of the simulation. A node in the quadtree contained a pointer to its four children nodes, the total mass of the particles further down in the tree, the centre of mass of the particles further down in the tree, the coordinates of the centre of the region, the distance from the centre of the region to the boundary and a flag determining if the node is a leaf node. If only one object is located in the region represented by the quad node then the node will also have a reference to the `struct` representing that object.

A dynamically allocated array was also used in the Barnes-Hut implementation to store the velocities of each object. A reference to the stored velocities was set as an input argument to the force calculation subroutine, where it was also updated. After the velocities of all particles were computed the positions were also updated according to equation 5. Finally the quadtree was deleted before starting the next iteration of the time loop.

Performance

The computations was performed on Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz, Unbuntu 18.04.1 LTS on Windows 10. The compiler used was GCC 7.3.0.

When optimising and testing the code the execution time was measured five times using the `time` command for each set of parameters and then the run with lowest user time was recorded.

Complexity of the Algorithms

To measure the complexity of the two algorithms, naive and Barnes-Hut, measurements was taken in which steps $= 50$ $\Delta t = 10^{-5}$ while N was varied. However, to measure the complexity of the final Barnes-Hut algorithm a θ_{max} value had to be determined for which the error in position was under 0.001. To determine a suitable value θ_{max} was increased from 0.02 until the threshold was exceeded. The result is shown below in figure 1 and θ_{max} was determined to be 0.25. The results from the complexity investigation of the two implementations is presented in figure 2.

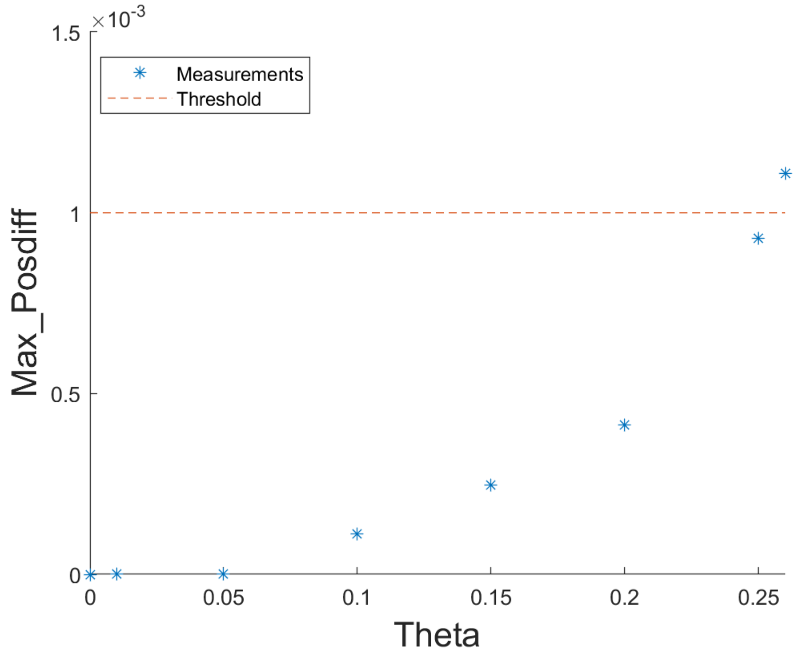


Figure 1: Maximal difference in position as a function of θ_{max}, with steps = 50, $\Delta t = 10^{-5}$ and $N = 2000$.

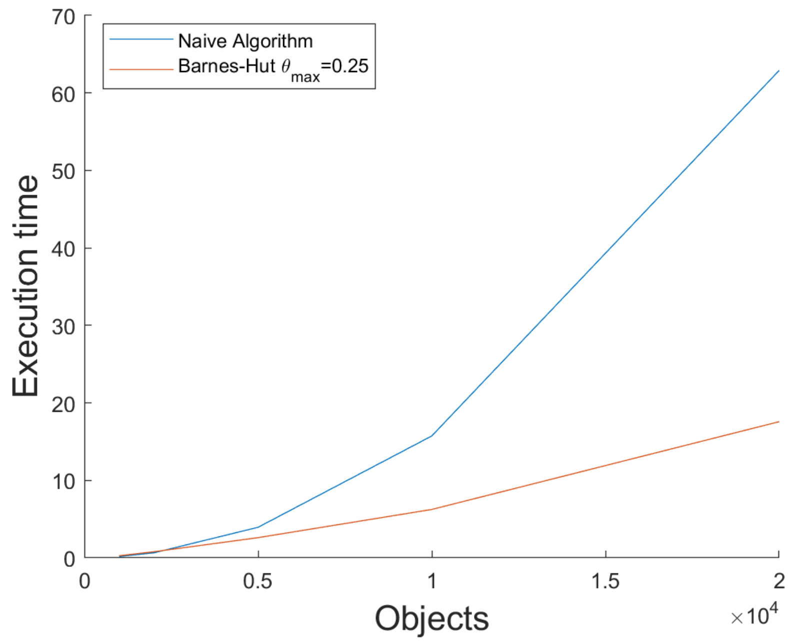


Figure 2: Time complexity for serial naive and serial Barnes-Hut algorithm, steps = 50, $\theta_{max} = 0.25$ and $\Delta t = 10^{-5}$.

Serial Optimisation of the Code

Serial optimisation was performed on all implementations of the simulation to increase performance. The main tool used was compiler optimisation flags. We also tried to improve performance by keeping data locality in mind. An example of this is not storing information which is not needed in the force calculations in the `structs`, such as the brightness, and designing the `structs` in way that eliminates unnecessary padding.

In all four implementations the same combination of compiler flags gave the best result, `Ofast` and `march=native`, see table 2-5 in appendix. The GCC flag `Ofast` turns on maximum compiler optimisation [2] and allows disregard of strict standard compliance, which may reduce accuracy [3]. The `march=native` option tells the compiler that it should produce code adjusted to the computer architecture it is compiled on. Hence this flag might result in increased performance on a certain computer architecture but reduce it on other architectures. To test if the `Ofast` flag reduced the accuracy notably the simulation was performed over 200 time steps with $\Delta t = 10^{-5}$, for both the quadratic and the Barnes-Hut implementation ($\theta_{max} = 0.25$). The result of the simulation was then compared to a simulation with the same parameters, but compiled without the `Ofast` flag. As no difference in the result was detected we drew the conclusion that the accuracy with the `Ofast` flag is sufficient for this specific problem.

Parallel Implementation and Optimisation

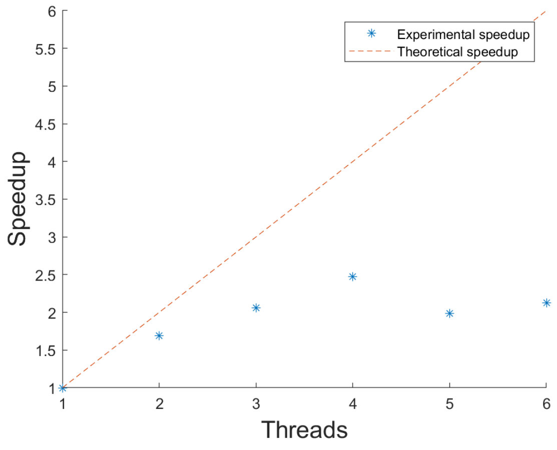
To increase performance part of the code was parallelised using Pthreads and OpenMP respectively. To get the highest speedup the part of the code where most of the execution time is spent, the force calculations, was first parallelised. Since the force acting on a certain object at a specific time step is independent of the acceleration acting on the other objects this computation can be made i parallel. When using OpenMP parallisation of other parts of the code was also explored.

Pthreads

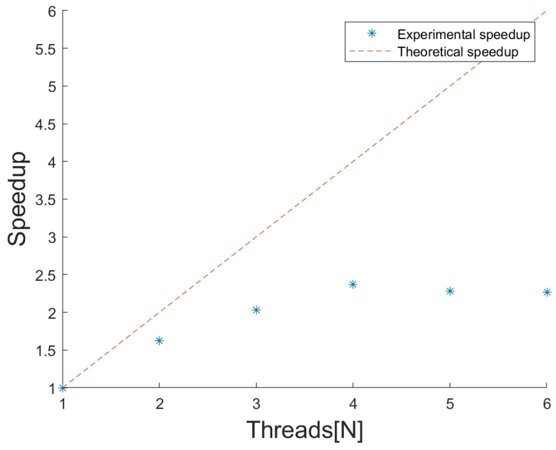
When implementing the Pthread parallelisation a `struct` for the input arguments required to calculate the force and update the velocities was defined. An auxiliary function which accepted a void pointer argument was written along with a function to calculate indices to partition the data for each thread. Threads were created and joined in each time step.

After Pthreads were implemented serial optimisation was again performed with different compiler flags, see table 4 in the appendix. As before the `Ofast` and `march=native` flags gave the best performance.

Then the performance for different number of threads was measured. The resulting speedup is presented in figure 3. The highest speedup was achieved with 4 threads for both $N = 10000$ and $N = 20000$ being 2.47 and 2.36 respectively.



(a) Speedup with $N = 10000$.



(b) Speedup with $N = 20000$.

Figure 3: Speedup for $N = 10000$ and $N = 20000$, with steps $= 50$, $\theta_{max} = 0.25$ and $\Delta t = 10^{-5}$.

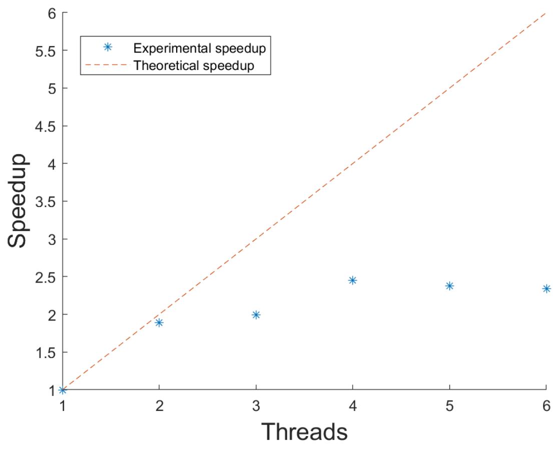
OpenMP

In the implementation with OpenMP parallelisation of the force calculation, position update, sorting of data after reading the initial condition and prior to writing the final result was explored. First parallelisation was added to the force calculations and different setting for schedule was explored. This was done to investigate how synchronisation, data dependencies and load balance affect the performance. Since the addition of parallelisation improved the performance see table 6 in the appendix this was kept. The best performance was received with the default settings for schedule. With these settings the lowest execution time achieved was 5.237s.

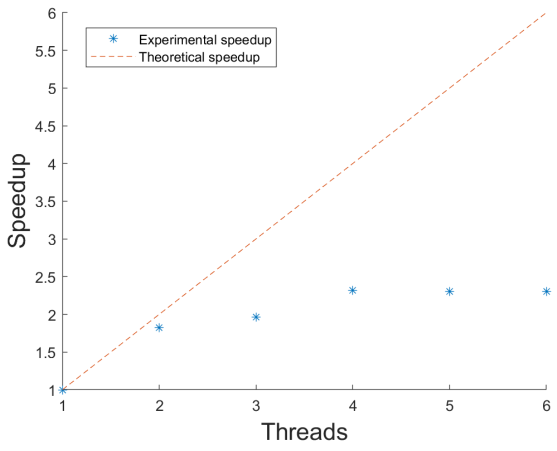
The next step was to add parallelisation to the position update loop. Different settings for schedule were also explored in this case and the default setting gave the best performance, see table 7 in the appendix. Parallelisation was then added to the loops used for sorting the initial condition and the final results of the simulation. Neither additions of parallelisation, for any setting tested, improved the performance and hence were not included in the final version of the program. The results form the test can be found in table 8 and table 9 in the appendix.

Our last attempt at improving performance was to move the creation of the threads from within the time loop to outside of the time loop. This was tested to possibly reduce the overhead of creating threads. The lowest execution time achieved was 5.250s and thus slightly higher than previous attempts. Thus this configuration was discarded.

Hence the final version of this implementation of the simulation included parallelisation of the force calculation and the position update. The default setting of `for` in OpenMP was used in both cases and the treads were created in each time step. With this configuration the speedup for different number of threads, compared to 1 thread, for a simulation with $N = 1000$ and $N = 2000$, $\Delta t = 10^{-5}$ and $\theta_{max} = 0.25$ is plotted in figure 4 bellow. The highest speedup achieved was 2.45 for 4 threads when $N = 10000$ and 2.32 for 4 threads when $N = 20000$.



(a) Speedup when $N = 10000$.



(b) Speedup when $N = 20000$.

Figure 4: Speedup for $N = 10000$ and $N = 20000$.

Summary of Optimisation

As we can see from table 1 the best serial execution time was achieved with the Barnes-Hut algorithm. When adding parallelisation the best time was achieved with Pthreads running on 4 threads. The best speedup was also achieved with Pthreads, 2.47 for Pthreads compared to 2.45 for OpenMP. Pthreads with 4 threads also exhibits a slightly better scaling behaviour with N, complexity, as shown in figure 5.

Table 1: Best user time for each implementation with parameters steps $= 100$, $N = 5000, \Delta t = 10^{-5}, \theta_{max} = 0.25$ for Barnes Hut and 4 threads with parallelisation.

	Naive	Serial Barnes-Hut	Pthread	OpenMP
Real	8.995	5.584	2.030	2.063

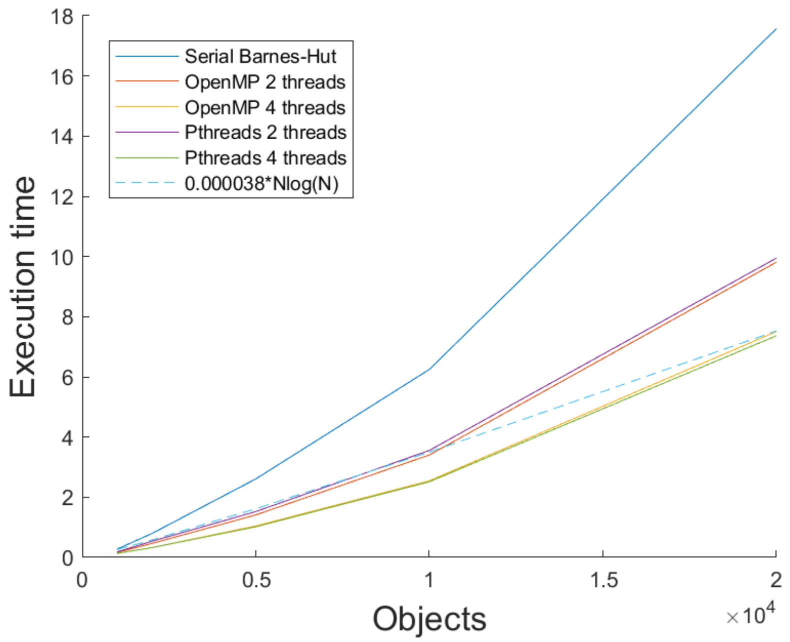


Figure 5: Time complexity for OpenMP and Pthread parallelised code with steps = 50, $\theta_{max} = 0.25$ and $\Delta t = 10^{-5}$.

Discussion

As figure 2 shows, our serial implementation of the Barnes-Hut algorithm scales better with N. Hence this implementation will have a lower execution time when simulating large number of objects. But for a small number of objects to simulate the naive implementation could still be preferred. As previously mentioned, one should be careful when using the Barnes-Hut algorithm as the approximations made in it results in lower accuracy. To increase performance one could increase θ_{max} and thus approximate larger clusters of objects as a single body. But this comes at the expense of the accuracy of the calculation, a trade off one should have in mind when choosing θ_{max}.

The same set of compiler optimisation flags gave the best performance in all of the implementations; `Ofast` and `march=native`. The `Ofast` option consistently gave a significantly increased performance compared to the `O3` flag. The `Ofast` flags allows larger approximations in calculations. In this simulations this difference is obviously not significant for accuracy, as we tested this as previously motioned, but for the execution time. The `Ofast` will also produce larger executable and prolong compiler time. The size of the executable is 19kB and the compiler time is bellow 1s. When compiling the final program with the `Os` compiler flag instead, which optimises for size, the executable is 18kB. As the difference in size of the executable and the compiler time is so low we believe that the use of the `Ofast` flag is motivated. In the naive implementation we tried to manually on-roll the position and velocity update loop, which gave no increase in performance. Because of this and the fact that the `Ofast` option includes loop unrollment performed by the compiler manual loop unrollment was not included in the final version of the program.

The `march=native` adjusts the code produced to the computer architecture it is compiled on. Hence it may reduce performance on other computer architectures; reduced portability. As the application scope of this program is fairly limited we assume that the person running this program also have the knowledge to compile the program. Hence we believe that it is justifiable to use the `march=native` flag to improve performance.

The best speedup was achieved with Pthreads was 2.47 and OpenMP was 2.45, both using four threads. The explanation why four threads gave the best performance is probably due to the fact that the program was run on a CPU with two physical cores with two threads on each, four threads in total. When using four threads the synchronisation for switching between threads should be minimised, while still utilising all of the computer cores.

The theoretical speedup of four cores is 4, but we only manage to get a speedup of around 2.5. This difference could be partially be explained by the fact that not the entire program can be executed in parallel, e.g. creation of the tree. Another explanation could be the increased overhead form creating and joining threads. In both implementations this is done in each time step. When using OpenMP we tried to only create the threads once in the simulation, but no performance gain was achieved. Hence out conclusion is that affect on performance is relatively small. Another problem with using multiple threads is the synchronisation time needed due to serial parts of the code. Synchronisation between threads is needed after the force calculations. If the objects to simulate is well distributed on the entire space then the Barnes-Hut algorithm should allow an even amount of approximation, hence resulting in a even load balance. In the Pthread implementation we distribute the objects to simulate evenly over the threads, hence assuming a even distribution of objects in the space. In the OpenMP implementation we tried different schedules for the load balance, but it gave no performance gain compared to the default settings. As the Pthread and OpenMP implementations gave similar performance we believe that the objects to simulate are probably evenly distributed, both in the initial conditions and on average in the simulation.

When comparing the performance of the two different parallel implementations one can conclude Pthreads gave a slightly higher speedup and flatter complexity plot, see figure 5. Pthreads is a lower level method and provides more control compared to OpenMP. Hence if implemented correctly Ptherads should be able to outperform OpenMP. An advantage of the higher level model OpenMP is that it is simple to implement as it utilises compiler directives. When choosing implementation method on should consider both performance and time needed to implement the parallelisation. As the performance gain of using Pthreads in this program is only slight one could argue that OpenMP is preferred in this applicaton.

Conclusion

The best performance achieved was with the Barnes-Hut algorithm, compiler flags `Ofast` and `march=native` and using Pthreads with four threads. The speedup compared to the naive algorithm for a simulation with 100 time steps of size 10^{-5} for 5000 objects was 4.4310.

References

[1] J. Rantakokko. 2019. *High Performance Programming, Lecture 9-10 - Pthreads.*

[2] A. Kruchinina. 2019. *High Performance Programming, Lecture 1 - Intro.*

[3] A. Kruchinina. 2019. *High Performance Programming, Lecture 4 - Instructions.*

Appendix

Tables

Table 2: Real, User and Sys time as reported by the time command for different optimisation flags for naive algorithm, using N = 2000 $\Delta t = 10^{-5}$, steps = 50.

	-O0	-O1	-O2	-O3	-Ofast
Real	24.060	22.498	22.223	22.273	0.774
User	24.016	22.484	22.172	22.234	0.766
Sys	0.000	0.000	0.016	0.016	0.000

	-Ofast -funroll-loops	-Ofast, -march=native	-Ofast, -march=native, -funroll-loops	-Ofast, -mtune=native	-Ofast, -mtune=native, -funroll-loops
Real	2.823	0.647	0.650	0.777	2.823
User	2.797	0.625	0.641	0.766	2.797
Sys	0.000	0.016	0.000	0.000	0.000

Table 3: Real, User and Sys time as reported by the time command for different optimisation flags for serial Barnes-Hut using N = 2000 $\Delta t = 10^{-5}$, steps = 100, $\theta_{max} = 0.25$.

	-O0	-O1	-O2	-O3	-Ofast
Real	3.844	8.935	8.956	8.929	1.643
User	1.984	8.922	8.922	8.906	1.625
Sys	1.844	0.016	0.000	0.000	0.000

	-Ofast -funroll-loops	-Ofast, -march=native	-Ofast, -march=native, -funroll-loops	-Ofast, -mtune=native	-Ofast, -mtune=native, -funroll-loops
Real	1.645	1.541	1.616	1.642	1.641
User	1.625	1.531	1.594	1.625	1.625
Sys	0.000	0.000	0.016	0.000	0.000

Table 4: Real, User and Sys time as reported by the time command for different optimisation flags using Pthread using N = 2000 $\Delta t = 10^{-5}$, $steps = 100$, $\theta_{max} = 0.25$.

	-O0	-O1	-O2	-O3	-Ofast
Real	9.998	8.971	8.959	8.926	1.642
User	9.938	8.938	8.875	8.859	1.609
Sys	0.000	0.000	0.031	0.000	0.016

	-Ofast -funroll-loops	-Ofast, -march=native	-Ofast, -march=native, -funroll-loops	-Ofast, -mtune=native	-Ofast, -mtune=native, -funroll-loops
Real	1.685	1.538	1.620	1.652	1.645
User	1.656	1.484	1.578	1.609	1.609
Sys	0.000	0.016	0.016	0.016	0.016

Table 5: Real, User and Sys time as reported by the time command for different optimisation flags using OpenMP using N = 2000 $\Delta t = 10^{-5}$, $steps = 100$, $\theta_{max} = 0.25$.

	-O0	-O1	-O2	-O3	-Ofast
Real	9.982	8.935	8.956	8.929	1.643
User	9.938	8.922	8.922	8.906	1.625
Sys	0.016	0.016	0.000	0.000	0.000

	-Ofast -funroll-loops	-Ofast, -march=native	-Ofast, -march=native, -funroll-loops	-Ofast, -mtune=native	-Ofast, -mtune=native, -funroll-loops
Real	1.645	1.541	1.616	1.642	1.641
User	1.625	1.531	1.594	1.625	1.625
Sys	0.000	0.000	0.016	0.000	0.000

For table 6 -9 the static schedule was also tested with chunks of size 8 and 16 however since they didn't achieve a better result they have been left out for readability.

Table 6: Real, User and Sys time as reported by the time command for OpenMP with different schedules when using parallelisation for calculation of acceleration

Different schedules for parallelisation of acceleration calculation	No schedule	Static,1	Static,2	Static,4	Dynamic	Guided	Auto
Real	5.252	5.310	5.405	5.294	5.398	5.296	5.303
User	20.859	21.031	21.266	21.031	21.125	20.984	20.969
Sys	0.016	0.000	0.016	0.031	0.16	0.00	0.016

Table 7: Real, User and Sys time as reported by the time command for OpenMP with different schedules when using parallelisation for updating position

Different schedules for parallelisation of position update	No schedule	Static,1	Static,2	Static,4	Dynamic	Guided	Auto
Real	5.237	5.261	5.243	5.263	5.266	5.324	5.259
User	20.688	20.859	20.781	20.906	20.859	21.000	20.891
Sys	0.047	0.016	0.031	0.016	0.047	0.000	0.000

Table 8: Real, User and Sys time as reported by the time command for OpenMP with different schedules when using parallelisation for copying read data

Different schedules for parallelisation of copying read data	No schedule	Static,1	Static,2	Static,4	Dynamic	Guided	Auto
Real	5.288	5.258	5.314	5.287	5.311	5.317	5.288
User	21.016	20.875	21.094	21.016	21.078	21.094	21.000
Sys	0.000	0.000	0.031	0.000	0.016	0.031	0.016

Table 9: Real, User and Sys time as reported by the time command for OpenMP with different schedules when using parallelisation for copying writing data

Different schedules for parallelisation of copying writing data	No schedule	Static,1	Static,2	Static,4	Dynamic	Guided	Auto
Real	5.334	5.322	5.277	5.259	5.303	5.323	5.312
User	21.172	21.125	20.938	20.922	21.094	21.125	21.094
Sys	0.000	0.016	0.031	0.000	0.016	0.016	0.000