Assignment 5

High Performance Programming

Author:

Fredrik Gustafsson

Sara Hedar

Uppsala

February 27, 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 of the code is important. To reduce the computation time compared to a naive $O(n^2)$ 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.

Theory

The equations governing the attracting force between objects follows Newton's law of gravity:

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

where G is the gravitational constant, 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 avoid this we modify equation 1 into:

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

where N is the total number of object.

To update the velocity and position of an object i 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 and $\boldsymbol{x}_i^n$ is position of object i at time step n.

Algorithm

The naive way to do the n-body simulation is to simply loop over each object i and calculate the force from each object j, $i \neq j$, with equ1-5. Such approach however would have a time complexity of $O(n^2)$, a better algorithm for this problem would be the Barnes-Hut algorithm. The Barnes-Hut, algorithm works by sorting the objects in a quadtree. This then allows us to, if the distance from our object is sufficiently large, replace several nodes with one node with a centre of mass, and total mass. Both the centre of mass and the total mass of a node gets updated when the node is split or an object is added further down in the tree quadtree. To calculate if the distance is sufficiently large we compare a value θ to θ_{max} where θ_{max} is chosen and θ is calculated with the following expression:

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

However when using the Barnes-Hut one should keep in mind that it uses approximations which leads to errors in the resulting positions of the objects.

Parallelisation

To increase performance while still maintaining a sufficiently low power consumption parallel computing can be utilised. Parallel computing is "simultaneous use of multiple compute 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).

The Solution

The initial condition to the system of interstellar 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.

To keep track of the interstellar objects a dynamically allocated array of the composite data structure called `struct` was used. Each `struct` constrained the position data and the mass of the object. To implement the Barnes-Hut algorithm a quadtree was constructed in each time step. Each tree node contained a reference to it's four children, 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 form the centre of the region to the boundary, a flag determining if the node is a leaf node. If an object is located in the region represented by the quad node then the node will also have a reference to the corresponding `struct`.

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

Performance and Discussion

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 the code the execution time was measured five times using the `time` command for each set of parameters and then the fastest time was recorded.

Complexity of the Algorithm

To investigate the complexity of the Barnes-Hut algorithm a value for θ_{max} was chosen such that the error (maximum difference in position) did not exceed 0.001. This value was found to be 0.25, and the error as a function of θ is plotted in figure 1

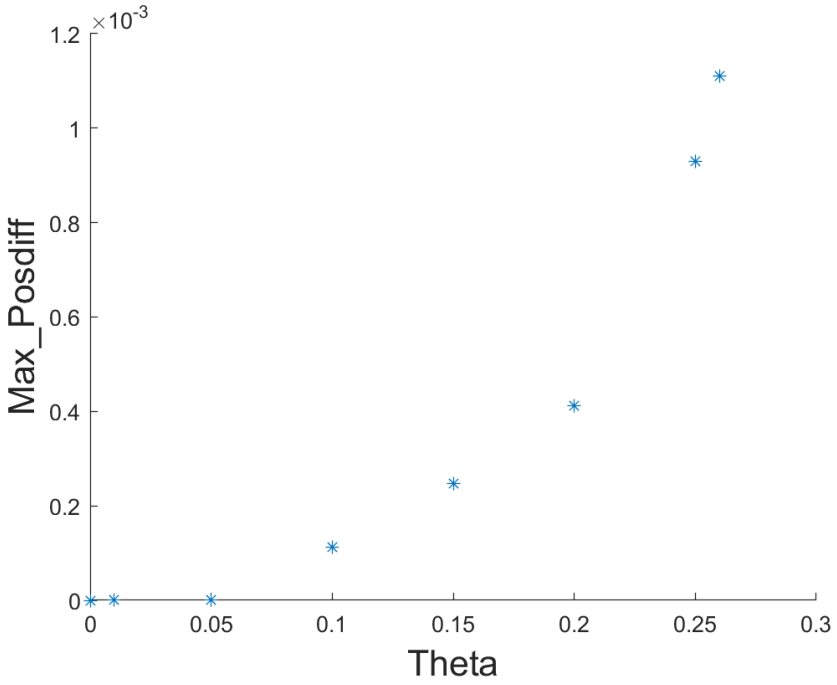
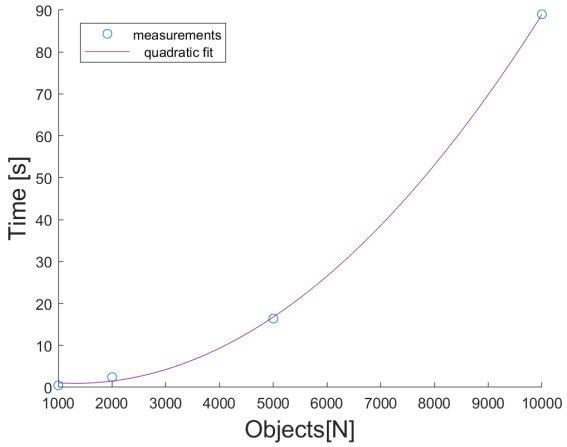
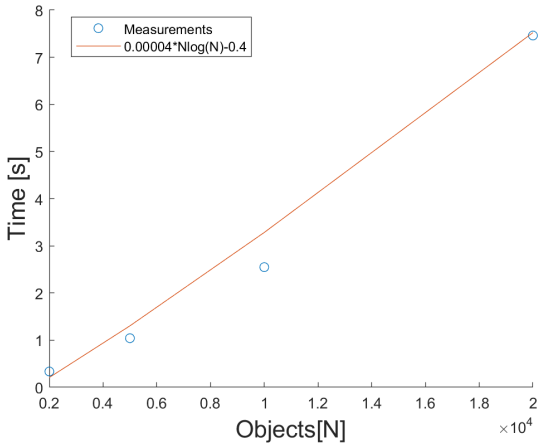


Figure 1: Maximal difference in position for different value of θ. The simulation was performed over 200 time steps for $N = 2000$ and with 1 thread, after the code was optimised.

One could try to find a more exact value of θ, however doing this optimisation would only yield a marginally better time performance. The execution time for $\theta_{max} = 0$ and $\theta_{max} = 0.25$ together with quadratic fit and a reference function respectively is shown in figure 2a and 2b below.



(a) $\theta = 0$.



(b) $\theta = 0.25$.

Figure 2: Plots of user time as a function of the number of simulated objects for $\theta_{max} = 0$ and $\theta_{max} = 0.25$. In the figure for $\theta_{max} = 0$ a quadratic fit is added. In the figure for $\theta_{max} = 0.25$ the function $f(N) = 0.00004 * N * \log(N) - 0.4$ is added as a reference. The simulation was performed over 50 time steps and 4 threads, after the code was optimised.

Serial Optimisation of the Code

After determining θ_{max}, serial optimisation was performed on the code using compiler optimisation flags. The results of the optimisation is presented in the tables bellow. The following parameters were used: $N = 2000$, steps $= 100$ and $dt = 10^{-5}$.

Table 1: The modifications done to the different models and the error before and after modifying

	-O0	-O1	-O2	-O3	-Ofast	-funroll-loops	-march=native	-mtune=native
Real	9.998	8.971	8.959	8.926	1.642	9.986	9.951	10.011
User	9.938	8.938	8.875	8.859	1.609	9.922	9.922	9.969
Sys	0.000	0.000	0.031	0.000	0.016	0.000	0.016	0.000 height

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

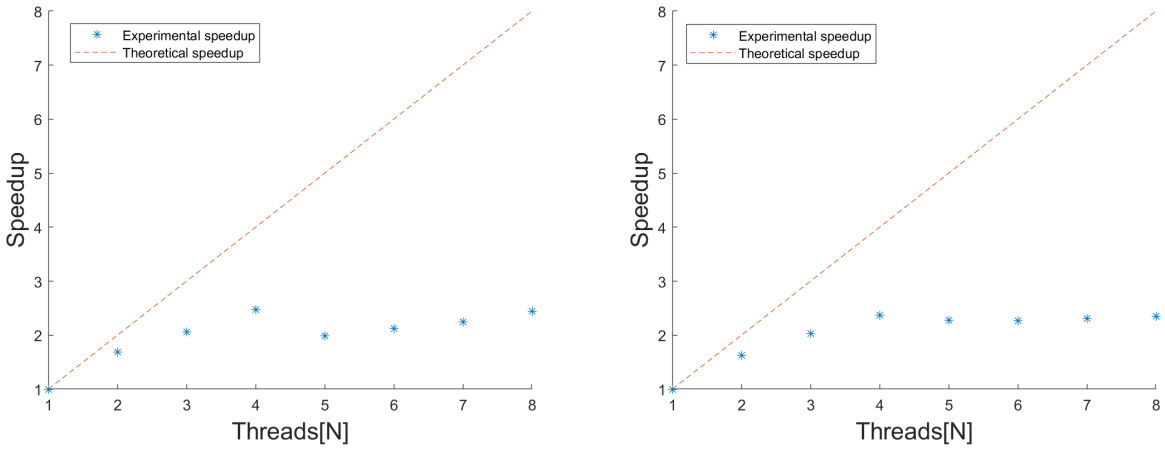
The conclusion of the serial optimisation was that the combination of the compiler flags `Ofast` and `-march=native` gave the best performance.

We kept the problem with padding of `structs` in mind when designing the `structs` used in our implementation. That is storing data types that take up more storage before the data types that takes up less space. We also tried to consistently store data on stack when possible.

Parallel Implementation

To increase performance the algorithm implemented was paralleled using pthreads. The part of the program which takes practically all of the execution time is the calculation of the acceleration, which is then used for updating the velocities of the particles. Since the acceleration acting on a certain particle at a specific time step is independent of the acceleration acting on the other particles these computations can be computed in parallel. The update of the position could also have been made parallel, though since these calculation are fast compared to the acceleration calculation, even after implementing parallelisation, the performance gain would probably be negligible.

The execution time for a simulation with $N = 10000, 20000$, with $\Delta t = 1e-5$, $\theta_{max} = 0.25$ and number of steps $= 50$ was recorded for different number of threads. In figure 3 the speedup compared to using 1 thread with the current implementation is plotted.



(a) Speedup for system with N = 10000.

(b) Speedup for system with N = 20000.

Figure 3: Speedup for different number of threads when $N = 10000, 20000$, with $\Delta t = 1e-5$, $\theta_{max} = 0.25$ and number of steps $= 50$.

As seen in the figure above the highest speedup, 2.4, was achieved for 4 threads. This is probably due to the fact that the computations was performed on a computer with two physical cores and two threads per physical core, that is four in total.

Finally we used the valgrind tool was used to confirm that no memory leaks were possible, that all dynamically allocated memory had been freed.

Changes Compared to Previous Assignments

Compared to the previous assignment we tried to increase data locality by only storing the necessary data in the quadtree. The data regarding the brightness was stored on stack and not used during the simulation. This required a change in the way we read the input data and write the final simulation result, which with the new implementation takes longer time. Though this should be compensated by the performance boost from the increase in data locality.

Discussion

The shortest execution time was achieved when the the `-Ofast`, `-march=native` compiler flags where used. The option `-march=native` optimises the code for the computer architecture where the program is compiled. Hence it may lead to reduced performance on other computer architectures.

When using the `-Ofast` flag the compiler accepts larger rounding errors, which should be kept in mind. To investigate the effect on the accuracy of the simulation when using the `-Ofast` flag the result was compared with the output from the same simulation but compiled with the `-O3` flag instead. We tried this for both $\theta = 0$ and $\theta = 0.25$ over 200 time step and no differences were detected.

One thing which might have improved our performance is to reduce the number of if statements and thereby decrease branch miss prediction. However we can not see how such a change could be implemented since a recursive function utilises if statements to determine if we have reached a base case or not. If statements are also utilised in order to determine in which quadrant to add new nodes for which two if statements are required.

A clear improvement of the execution time was achieved compared to the algorithms implemented in the previous assignments. With the quadratic algorithm from assignment 3 the simulation for 5000 objects over 100 time steps the execution time was 8.995s. With the Barnes-Hut algorithm, assignment 4, with $\theta = 0.25$ the execution time was 5.584s.

Using more threads improved our execution time as seen in figure 3. However as one can see the speedup is not following the theoretical speedup (speedup = number of threads). Instead it appears as if the speedup achieved a maximum value of around 2.4 for both systems when using four threads. For our new code the execution time for 5000 objects and 100 time steps became 2.030 which can be compared with the time for the code from assignment 3 and 4 above. Using one thread gave us a result of 5.233s which is a bit faster than the results from assignment 4. This decrease in time when using one thread is probably be due to the increase in data locality of the new implementation, as previously mentioned.

Distribution of Roles

The workload was equally shared between the group members, Sara and Fredrik.

References

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