Assignment 6

High Performance Programming

Author:

Fredrik Gustafsson

Sara Hedar

Uppsala

March 1, 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 execution 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 the application programming interface OpenMP.

Theory

The attractive force between objects follows Newton's law of gravity. The force acting an 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 avoid this we modify equation 1 into:

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

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

A naive way to perform a n-body simulation is to simply loop over each object i and calculate the force from each object j, $i \neq j$, with equation 1-5. This approach however would have a time complexity of $O(n^2)$. An algorithm with a better complexity for this problem is the Barnes-Hut algorithm with time complexity $On \log n$. In the Barnes-Hut algorithm the objects to simulate is stored in a quadtree (when simulating in two dimensions). When the distance from out object is sufficiently large we approximate serveral objects as a single body with their combined mass and position of the center of mass. using their combined mass and center of mass. Both the combined mass and center of mass is stored in the tree-structure is and is updated when new objects are added further down in the tree. 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 approximations are made, which leads to errors in the simulation.

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". OpenMP is a application programming interface (API) which consists of a set of compiler directives and functions in the languages C, C++ and FORTRAN. 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 for the parallelisation. The threads in OpenMP have access to global data while also having private data on stack.

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 from 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.

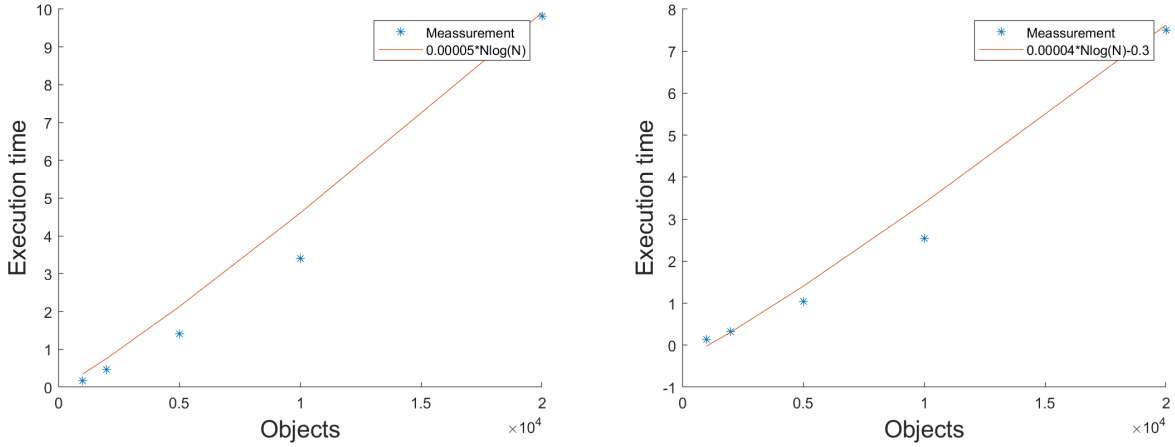
Performance and Discussion

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 lowest 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. 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 two and four threads and a reference function is presented in figure 1a and 1b below.



(a) Two threads.

(b) Four threads.

Figure 1: Plots of user time as a function of the number of simulated objects for number of threads = 2 and number of threads = 4. In the figure for number of threads = 2 a the function $f(N) = 0.00005 * N * \log(N)$ is added. In the figure for number of threads = 4 the function $f(N) = 0.00004 * N * \log(N) - 0.3$ 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 $dt = 10^{-5}$. The code was compiled without the `-mopen` compiler flag during the serial optimisation process.

Table 1: Real, User and Sys time as reported by the time command for different optimisation flags.

	-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

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 and Optimisation

To increase performance the algorithm implemented was parallelised using OpenMP. 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. Since OpenMP synchronises the threads after each "pragma omp for" block the position updates could also easily be made parallel by keeping it inside the main parallel block. In table 2 below the results from using different schedules as well as the effects of parallelising part of the copying of read and written data is presented.

Table 2: Real, User and Sys time as reported by the time command for parallel optimisation only changing configuration if the time improves.

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

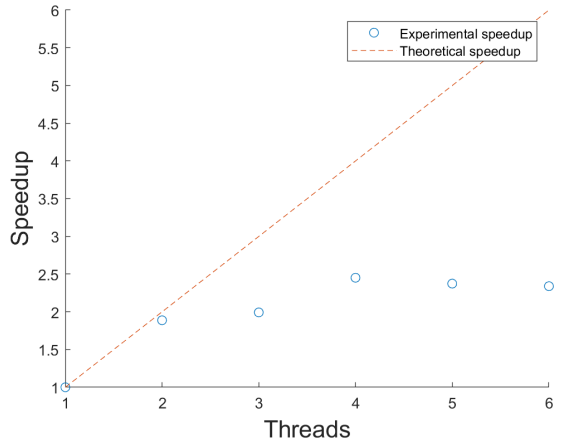
Different schedules for parallelisation of position update	No schedule	Static,1	Static,2	static,4	Dynamic	Guided	Auto
Real	5.237	5.261	5.243	5.263	5.266	5.324	5.259
User	20.688	20.859	20.781	20.906	20.859	21.000	20.891
Sys	0.047	0.016	0.031	0.016	0.047	0.000	0.000

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

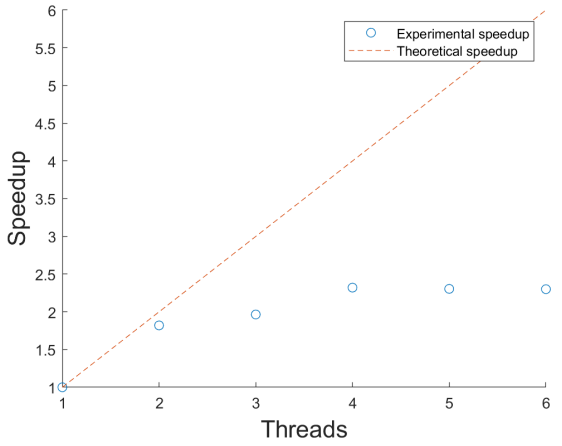
Different schedules for parallelisation of copying writing data	No schedule	Static,1	Static,2	static,4	Dynamic	Guided	Auto
Real	5.334	5.322	5.277	5.259	5.303	5.323	5.312
User	21.172	21.125	20.938	20.922	21.094	21.125	21.094
Sys	0.000	0.016	0.031	0.000	0.016	0.016	0.000

Based on the results from table2 above the best achieved time comes from not parallelising the copying of data part, while using the default schedule for the parallel calculation of acceleration and update of the position.

With the mentioned configuration 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 2 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 2: 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 threads in total.

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 in the case that the executable is simply transferred and not recompiled.

When using the `-Ofast` flag the compiler accepts larger truncation 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 serial algorithms implemented before. 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 2. However as one can see the speedup does not follow the theoretical speedup (speedup = number of threads). Instead it appears as if the speedup achieved a maximum value of around 2.45 for N = 10000 and 2.32 for N = 20000, both when using four

threads. For our new code the execution time for 5000 objects and 100 time steps became 2.063 which can be compared with the time for the code from assignment 3 and 4 above.

Comparing the speedup between OpenMP and Pthreads shows a small difference 2.45 compared with 2.47 for N = 10000 and 2.32 compared to 2.36 for N = 20000. In general Pthreads should be able to give better performance than OpenMP, since it is a lower level tool which provides more control. Though we only got a slight improvement in performance when using Pthreads and we consider OpenMP to be easier to implement. When choosing implementation method one should consider both performance and time needed for implementing the method.

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).