Uppsala Universitet

High performance programming
1TD062

Assignment 3 - Galaxy simulation

Agelii, Carl.
Forsberg, Emil.
Lindström, Viktor.

March 7, 2024

Introduction to the problem

The main force exerted between galaxies is the gravitational force. Newton's law of gravity states that in two dimensions, the gravitational force exerted on body i from j is given by the following equation:

$$F_{ij} = -\frac{Gm_i m_j}{r_{ij}^3}\mathbf{r}_{ij} \tag{1}$$

Here G is the gravitational constant, m_i and m_j are the masses for body i and j respectively, $r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ is the distance between the bodies, and $\mathbf{r}_{ij} = (x_i - x_j)\hat{e}_x + (y_i - y_j)\hat{e}_y$. To deal with instability problems we modify Newton's law of gravity to include Plummer spheres:

$$F_{ij} = -\frac{Gm_i m_j}{(r_{ij} + \epsilon)^3}\mathbf{r}_{ij} \tag{2}$$

This is an N-body problem, the total gravitational force (exerted by all other bodies) for each body will be calculated to update the velocity and position over time. This can be done with, for example, the symplectic Euler method:

$$a_n^n = \frac{F_i^n}{m_i}$$

$$u_i^{n+1} = u_i^n + \Delta t a_i^n$$
$$x_i^{n+1} = x_i^n + \Delta t u_i^{n+1}$$

For this simulation the following values were used: $\epsilon_0 = 10^{-3}$, $G = 100/N$, and $\Delta t = 10^{-5}$.

Solution

Serial implementation

It was chosen to use a struct to store the position, velocity, mass, and brightness of each particle. The code reads all the data from the input file and stores it in N particle structures. This is a very efficient way of storing the current values for these parameters, another way would be to store everything in a vector, since it could fit in the cache if small enough. The code then sends all of these particles as input in the step function. The step function calculates all the forces and updates the velocity and position accordingly for all particles. Memory is allocated to the variables Fx and Fy with malloc to store the forces and freed when the simulation is done, and it is no longer needed to avoid memory leaks. Step is called nstep amount of times as a loop in the main function. It is also possible to split the step function into smaller functions, for example, one function that calculates the force, and one that steps and uses symplectic Euler. But this introduces more function calls and thus might be slower.

Parallel implementation

Parallelizing the code comes with some drawbacks. In the serial implementation, the inner loop for j calculates the force on both i and j, which significantly reduces the computation. In a parallel implementation, this would lead to several threads trying to change the same value. In our implementation, we parallelized the calculation of the force on particle i in each time step, meaning particle j includes all particles.

Pthreads

Pthreads can use one pointer address as input to the pthread_create function. Each thread needed a lot of input, for example which particles to calculate the force on, positions and so on. This lead to a lot of global variables being used. Also, we created a struct with a pointer to the particle struct and a start and stop value. This may not be ideal because of cache size, but we could not find a way that performed better. The start and stop values indicated which particles were supposed to be calculated by each thread. Since the number of particles is not always divisible by the number of threads, a remainder had to be distributed among the threads. However, for large N, this should not lead to poor load balancing.

OpenMP

When implementing the OpenMP approach all that was needed was to initialize the parallelisation with a single line of code where we made the number of threads an input argument. The heaviest work load lies in the force computation on each particle. Hence, that is what was decided to parallelize using the line: #pragma omp parallel for num_threads(n), where n was the mentioned input argument.

Discussion of performance

Serial Optimization

Optimization techniques were used regularly throughout the development of this project. This means that in table 1, "None" optimization actually means some optimization. For example, when calculating the force in each time step, two nested loops are needed. The value $m_i \cdot G$ is constant in the inner loop, so the value is calculated once in the outer one. Actually, producing constant values was done regularly. ϵ_0 was defined at the top of the program. Other constants, such as Δt and G, depended on the input of the user. These were set as constant in main.
Some optimizations were done in hindsight. The final best time can be seen under "manual optimization" in table 1. When calculating the force over all the particles, it was deemed smarter to use two separate loops over j to i and $i+1$ to N in the inner loop so an if statement could be avoided. (Since it should be summed when $i \neq j$). When using this scheme, the Arrhenius machine completed the $N = 3000$ file after 33 seconds. However, we came up with a strategy to just use one loop instead, going from $j = i+1$ to $j < N$ in the inner loop. This produced the same result but completed it in 19.7 seconds instead, a pretty significant improvement. Another thing that could be optimized was the initialization of the force vectors in each time step. Firstly, this was done with a separate loop. Afterwards, it was tried to do it in the inner loop so the separate loop could be

disposed of. This produced the correct result, but was ultimately slower. This is likely because it tried to reach the value of the vector N^2 times, instead of N times as with the separate loop. It was deemed impossible to do this N times without producing a separate loop. Another possible optimization was to allocate the temporary force arrays on the stack instead of the heap in the step function. However, this did virtually nothing. This is strange, since static memory is generally faster than dynamic.

As previously mentioned the simulation was ran on Arrhenius, which has an Intel(R) Xeon(R) CPU E5520 @2.27GHz with gcc version 11.4.0.

The simulation-time was measured using the linux command "time" in the terminal, with the results for the different optimization methods being presented in table 1 below. Here, 3000 particles were simulated over 100 time-steps.

Table 1: Measured time for simulations with different optimisations, N=3000, nstep=100

Optimisation method	Time for simulation [s]
None	33.0
Manual optimisation	19.7
Manual optimisation with -O3	10.5
Manual optimisation with -O2	10.5
Manual optimisation with -Ofast	10.5

As seen in table 1 above, our original script (referred to as None) took 32.9 seconds to finish.
After reviewing the script, after it's completion, and making the optimization-changes mentioned above, the simulation time was reduced to 19.7 seconds.
Some optimization flags (-O3, -O2, -Ofast) were also implemented on the manually optimized version. this was done by using the flag "-O3", "-O2", and "-Ofast" respectively in the terminal. The resulting time using the respective flags shows a clear decrease in time, but there seems to be no difference between the three optimization-flags.

Lastly, the time complexity of the program was studied. Eight different simulations were done with different N and 100 timesteps. The results can be seen in figure 1.

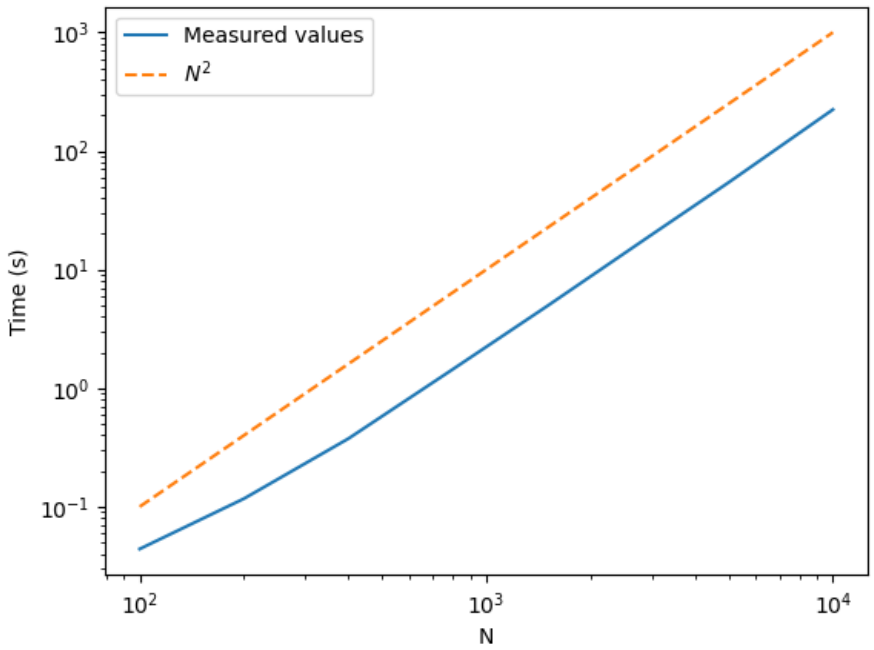

Figure 1: The measured and expected times in a loglog-plot.

When applying linear regression to the logarithmic values, the slope was ≈ 1.9, which is pretty close to 2. Table 2 shows the values explicitly.

Table 2: Values of N and t

N	$t(s)$
100	0.044
200	0.117
400	0.375
800	1.44
1500	4.93
3000	19.69
5000	54.22
10000	223.5

Here it clearly shows that the time complexity grows as $\mathcal{O}(N^2)$.

Parallel Optimization

After finishing the serial optimization, the parallel optimization was begun. Two parallelisation techniques where used and their results are discussed below. The computer Vitsippa was used for this project, which has an AMD Opteron(tm) Processor 6282 SE with gcc version 11.4.0. Note that it's not the same computer as was used for the serial optimization, which might affect the result of the comparison of the different speedups. All simulations where done with the same values as when doing the serial optimization above. Before the parallel optimization was begun, the first fix that was implemented was to reverse one of the serial optimizations made previously (the one where we changed from two for-loops to one). Without this fix, a race condition would occur.

Figures 2, 3, 4 and 5 below shows the speedup for different amounts of threads used with OpenMP and Pthreads. The simularions used 500 particles for figures 4 and 2 and 5000 particles for figures 5 and 3 particles.

Pthreads

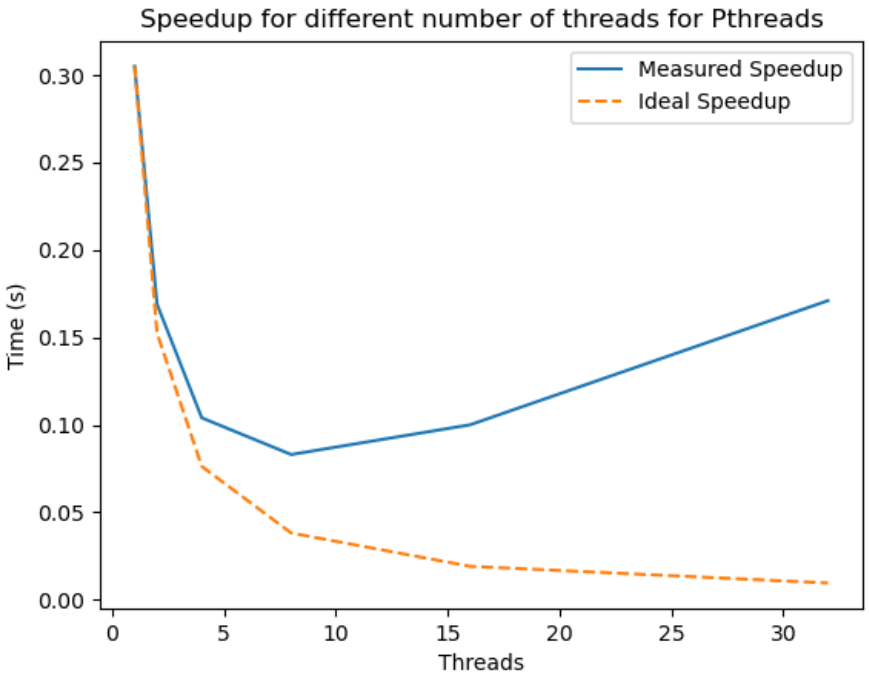

Figure 2: Plot of speedup for different number of threads using Pthreads

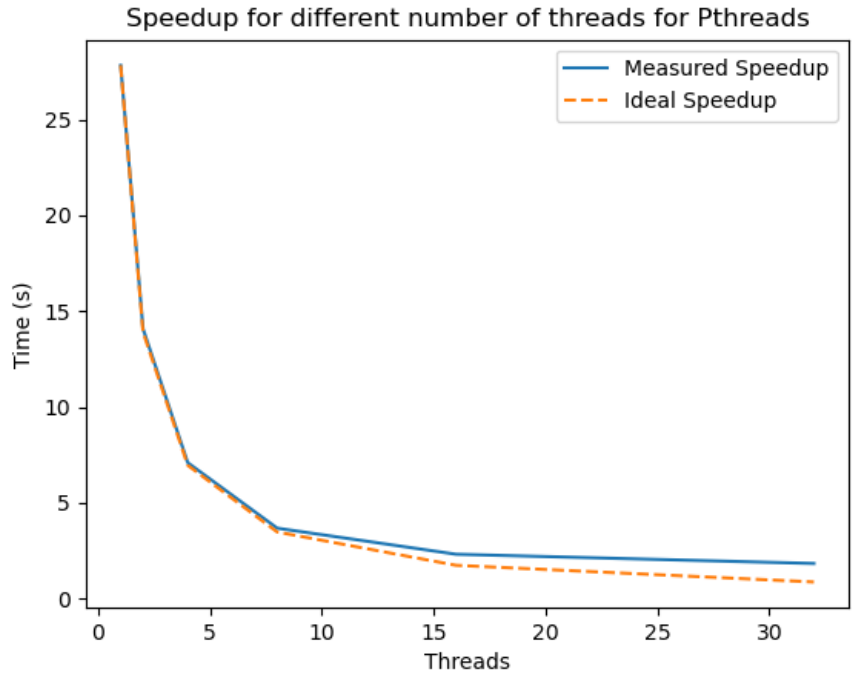

Figure 3: Plot of speedup for different number of threads using Pthreads

OpenMP

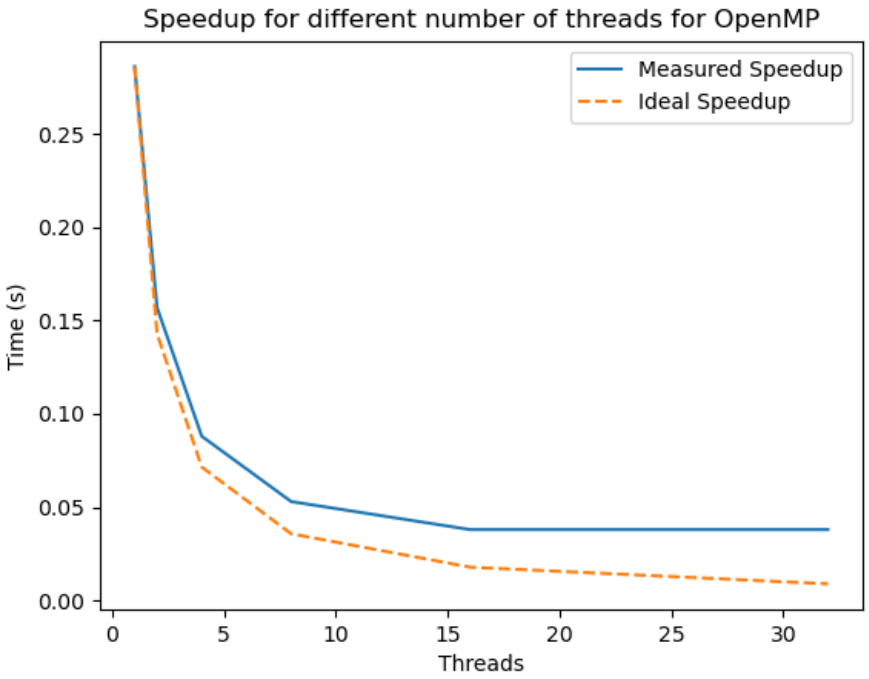

Figure 4: Plot of speedup for different number of threads using OpenMP for 500 particles.

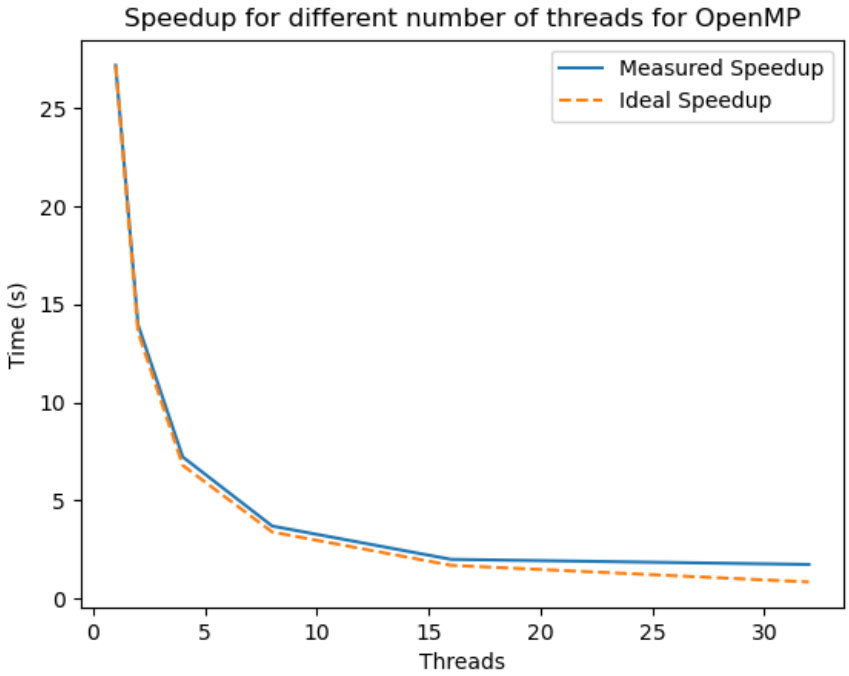

Figure 5: Plot of speedup for different number of threads using OpenMP for 5000 particles.

As can bee seen when comparing our implementations of Pthreads and OpenMP, Pthreads didn't perform as well as OpenMP for low numbers of particles as the measured speedup for Pthreads diverges faster from the ideal speedup compared to OpenMP (compare figures 2 and 4). However, for a large number of particles they performed about the same (see figures 3 and 5).