High Performance Programming
Assignment 3

Linus Lindell, **Samuel Medlock**, **Markus Norling**

February 2023

UPPSALA
UNIVERSITET

Contents

1 Introduction

1.1 Problem

The problem to be solved in this assignment is to calculate the position and velocity for a variable number N particles that is subject to Newton's law of gravity in two dimensions. The calculations should be done for a number of time steps, with a step size of 10^{-5} seconds and an algorithm scaling with $\mathcal{O}(N^2)$. In order to do this, code was written in the C-programming language.

1.2 Formulae

For particle i, the force exerted on this particle by all other particles can be described as:

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

Where G is the gravitational constant, ϵ_0 is a smoothing operation set to 10^{-3}, m_i and m_j are the masses of the particles and $\mathbf{r}_{ij}$ is the vector that describes the difference in position between the two particles. Since the acceleration, position and velocity of each particle are of interest, and not the force itself, the following equations were instead implemented with the appropriate time step:

$$a_i^n = \frac{F_i^n}{m_i} \tag{2}$$

$$u_i^{n+1} = u_i^n + \Delta t a_i^n \tag{3}$$

$$x_i^{n+1} = u_i^n + \Delta t u_i^{n+1} \tag{4}$$

2 Solution

2.1 Data structures

To store the data for each particles a `struct` called `planet_data_t` was created:

```
typedef struct{
    double pos_x;
    double pos_y;
    double mass;
    double vel_x;
    double vel_y;
    double brightness;
}planet_data_t;
```

This made it easy to allocate memory for the data and also to read the input data from the chosen input file.

Two arrays of pointers were created and memory for datatype `planet_data_t` was allocated for each element:

```
planet_data_t* arr[N];
planet_data_t* arrtemp[N];
for(int i = 0; i<N; i++){
    arr[i]=(planet_data_t *)malloc(sizeof(planet_data_t));
    arrtemp[i]=(planet_data_t *)malloc(sizeof(planet_data_t));
}
```

The input file was read by opening the file with the function `fopen()` in binary read mode. The layout of the file was utilized, and the data were read in groups of six using `fread()`, assigning each datapoint to the corresponding member of the `struct`:

```
FILE *stream;
stream = fopen(path_to_file, "rb");

//reading input data to the array
for(int i =0; i<N; i++){
    fread(&(arr[i]->pos_x), sizeof(double), 1, stream);
    fread(&(arr[i]->pos_y), sizeof(double), 1, stream);
    fread(&(arr[i]->mass), sizeof(double), 1, stream);
    fread(&(arr[i]->vel_x), sizeof(double), 1, stream);
    fread(&(arr[i]->vel_y), sizeof(double), 1, stream);
    fread(&(arr[i]->brightness), sizeof(double), 1, stream);
}
fclose(stream);
```

2.2 Algorithm

The problem was solved with a simple $\mathcal{O}(N^2)$-algorithm consisting of two `for`-loops. The decision to perform the calculations separately for the x- and y-directions were made instead of the usage of vectors. After declaring all variables a loop over each particle i, and another nested loop over each particle j were created. The second loop is used to implement the summation in (1). After the second loop finishes the new acceleration, velocity and position is calculated for particle i. The new data were then stored in a temporary storage until all calculations for each particle was done on a time step. The ordinary array was then updated and this was repeated for every time step.

Initially the inner `for`-loop contained an `if`-statement to check if $i == j$ (this is refered to as the "first version"), but in order to optimize the code the inner `for`-loop was changed so that the `if`-statement could be avoided. The second version instead has the inner `for`-loop split into two `for`-loops, one that is `for(j=0; j<i; j++)` and a second loop that is `for(j=i+1; j<N; j++)`. with this modification the `if`-statement is avoided and the code could potentially become faster. This also has the potential of allows for better loop-unrolling, which is examined in 3.

3 Performance and discussion

When timing the code with different compiler optimization flags, the input file "ellipse_N_02000.gal" was used for 200 time steps. All times presented in this report is for runs of the whole program, and were made by using the '`time ./galsim ...`'-command. All timed runs were made on the university's "Vitsippa"-linux system, which has an AMD Opteron 6282SE CPU and the GCC version 11.3.0 compiler.

3.1 First Version

The first version of the code, where the inner `for`-loop contains an `if`-statement, was timed after being compiled with a number of different compiler optimizations, as seen below in table 1.

Table 1: Timings for different compiler optimizations on the first version of the code.

Optimization	Time
Unoptim.	0m34,084s
-O1	0m11,674s
-O2	0m8,916s
-O3	0m9,021s
-Ofast	0m8,902s
-Os	0m18,098s

3.2 Second Version

The second version of the code, where the `for`-loop had been split into two, was timed with the same compiler optimization flags as the first version, and then also compiled combined with the `-funroll-all-loops`-flag. The result of this is seen below in table 2 and 3.

Table 2: Timings for different compiler optimizations on the second version of the code.

Optimization	Time
Unoptim.	0m33,370s
-O1	0m12,153s
-O2	0m8,900s
-O3	0m9,268s
-Ofast	0m8,893s
-Os	0m13,958s

Table 3: Timings for different compiler optimizations on the second version of the code.

Optimization	Time
-funroll-all-loops.	0m33,420s
-O1 -funroll-all-loops	0m12,332s
-O2 -funroll-all-loops	0m8,896s
-O3 -funroll-all-loops	0m8,904s
-Ofast -funroll-all-loops	0m8,499s
-Os -funroll-all-loops	0m14,554s

3.3 Timing for different N's

In order to examine how the computational time depends on N (the number of particles), the second version that had been compiled with the `-Ofast -funroll-all-loops`-flags where ran on input files with different N's. The input files "ellipse_N_x.gal were ran for 200 time steps where "x" is the different N's that were tested. N's from 10 up to 10 000 were tested and the results are plotted below in 1.

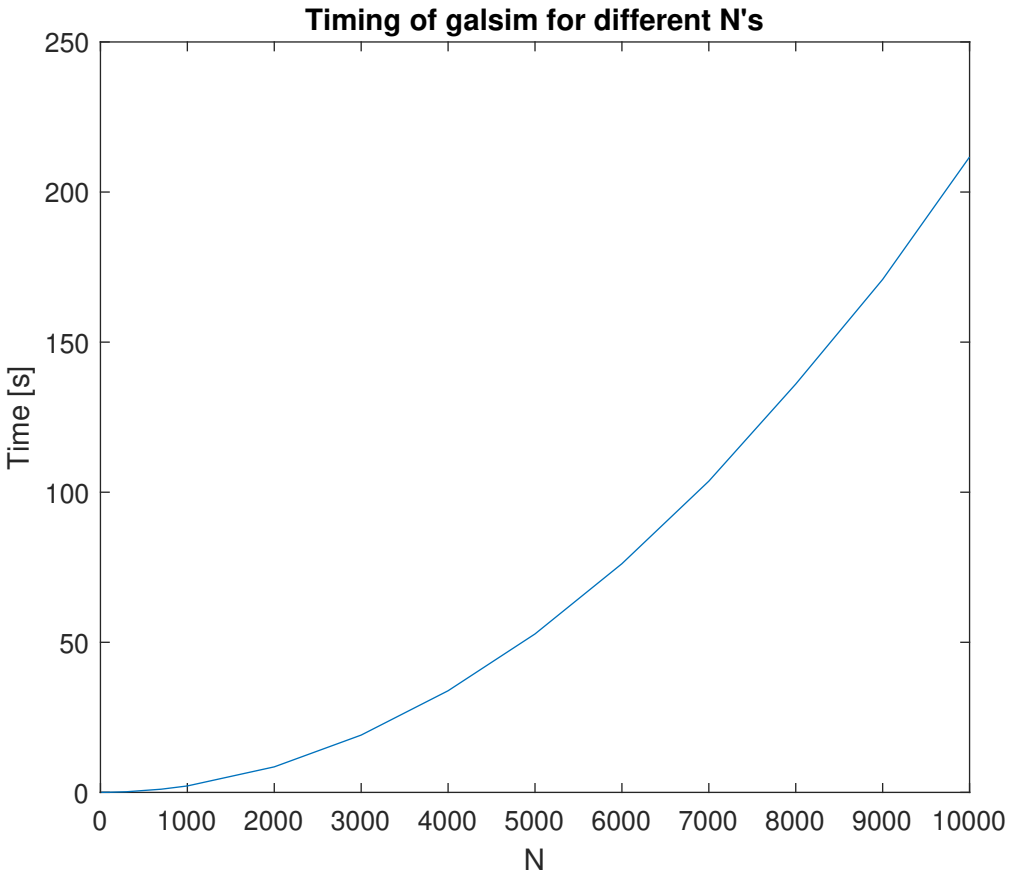

Figure 1: Time taken to run the galsim-program for input file with different N's. 200 time steps were computed in each run.

3.4 Discussion

Instead of creating an array of pointers pointing to N structs, an array of 6*N doubles could have been implemented. Every seventh element in the array would be a new particle and the order of the data would be known. This might have been an easier solution than the current implementation. However it would not have been more memory efficient as every member of our struct was a double and therefore no padding had to be used for memory allocation.

The graph [1] in 3.3 confirms that the complexity of the algorithm grows with $\mathcal{O}(N^2)$, as was the goal of the project. The fastest execution time was accomplished using the compilation flags "`-Ofast -funroll-all-loops`" on the second version of the code, where the for-loop was split in two. The split was done in order to get rid of the `if`-statement check in each iteration, which saved 0.7 seconds circa compared to the first version. There was also an assumption made that this would allow the compiler to unroll the loops more efficiently, however the time difference was smaller than anticipated.

4 Parallelization

Once the problem had been solved with serial code, the code was modified to run in parallel with Pthreads and then with OpenMP. The parallelized versions of the code was then timed for different number of threads, and these where then compared to the fastest time that was achieved with the serial code (second version compiled with `-Ofast -funroll-all-loops`), in order to get a speedup factor for the different number of threads. The computations were ran on the same system as the one used for the serial code, which has 32 hardware threads.

4.1 PThreads

The version of the code that uses PThreads take advantage of an intermediate function, "`thread_func()`" thats sits "inbetween" the `for`-loop that loops over the timesteps and the `for`-loop that computes the forces on each particle. This is done since the function called from `pthread_create()` only can be sent one argument, while we want to send multiple arguments to the function with the force calculating loop. This could be done by sending a struct, but we've choosen the intermediate function instead, which takes a thread-ID as input argument, calculates start and end-indices for the calculation-loop, and then calls the caluclation-function with these as well as the other needed arguments.

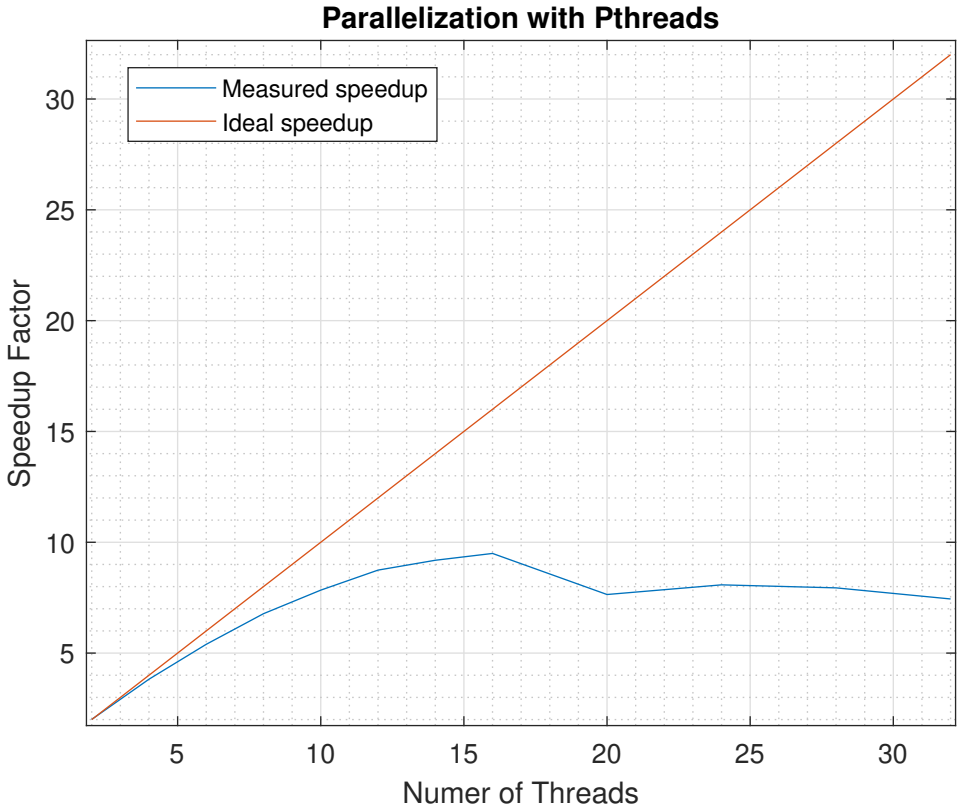

Figure 2: Speedup factor with different number of threads using PThreads and "ellipse_N_02000.gal" as input file with 200 timesteps, plotted along with the ideal speedup factor.

4.2 OpenMP

The parallelization with OpenMP is much more straight forward to implement than the one with PThreads, it's essentially done with only one line of code, placed right before the `for`-loop that caluclates all forces. This line specifies which variables that are private to each thread, so that each thread can use variable throughout it's calculations, without other threads changing their value in the meantime.

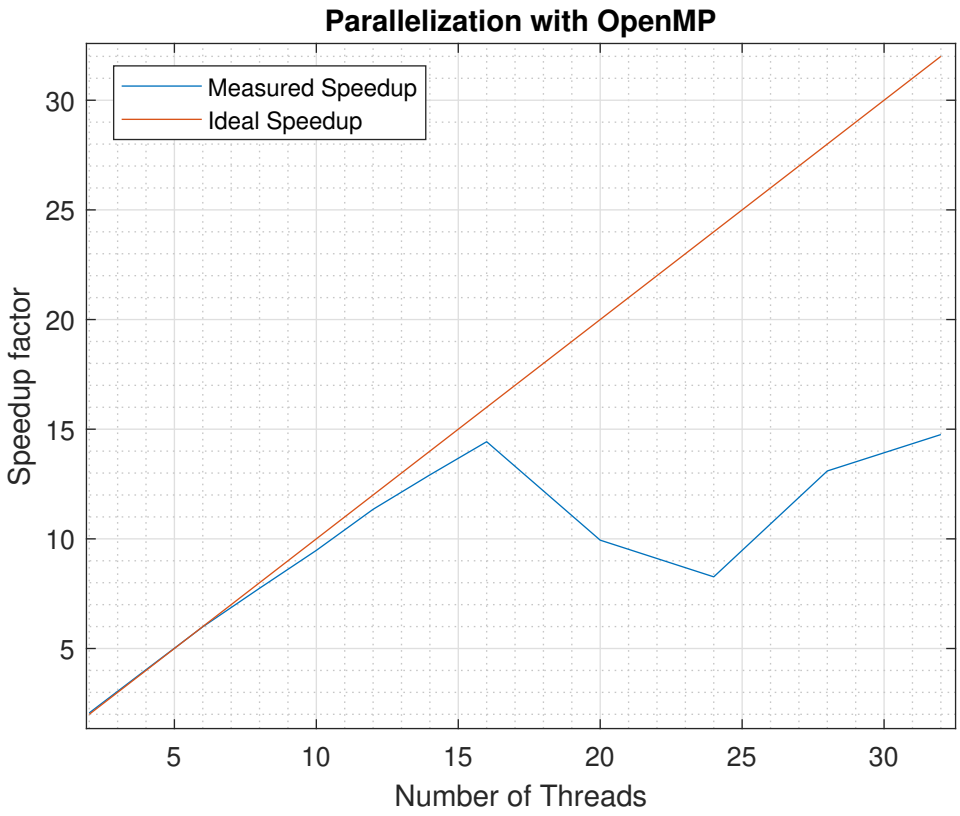

Figure 3: Speedup factor with different number of threads using OpenMP and "ellipse_N_02000.gal" as input file with 200 timesteps, plotted along with the idel speedup factor.

4.2.1 Parallelization for larger problems

For the OpenMP-parallelization, different numbers of threads were also tested for input files with larger N, to see how the measured speedup compares to the ideal speedup when the problem becomes larger. The results of this can be seen in the figures below.

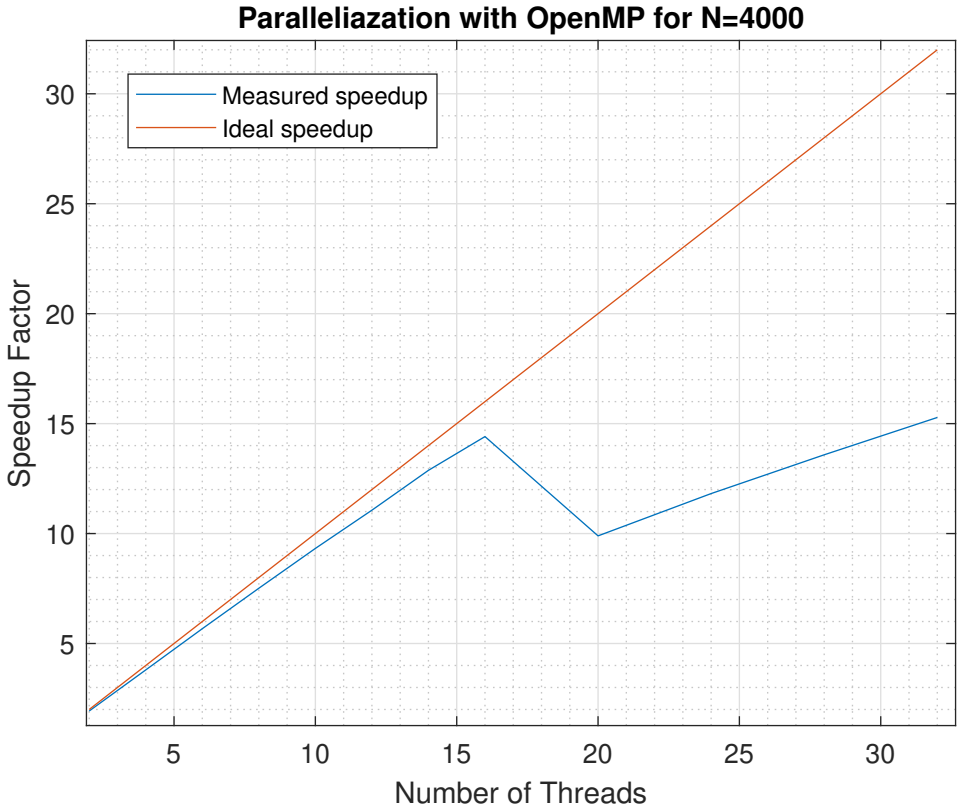

Figure 4: Speedup factor with different number of threads using OpenMP and "ellipse_N_04000.gal" as input file with 200 timesteps, plotted along with the idel speedup factor.

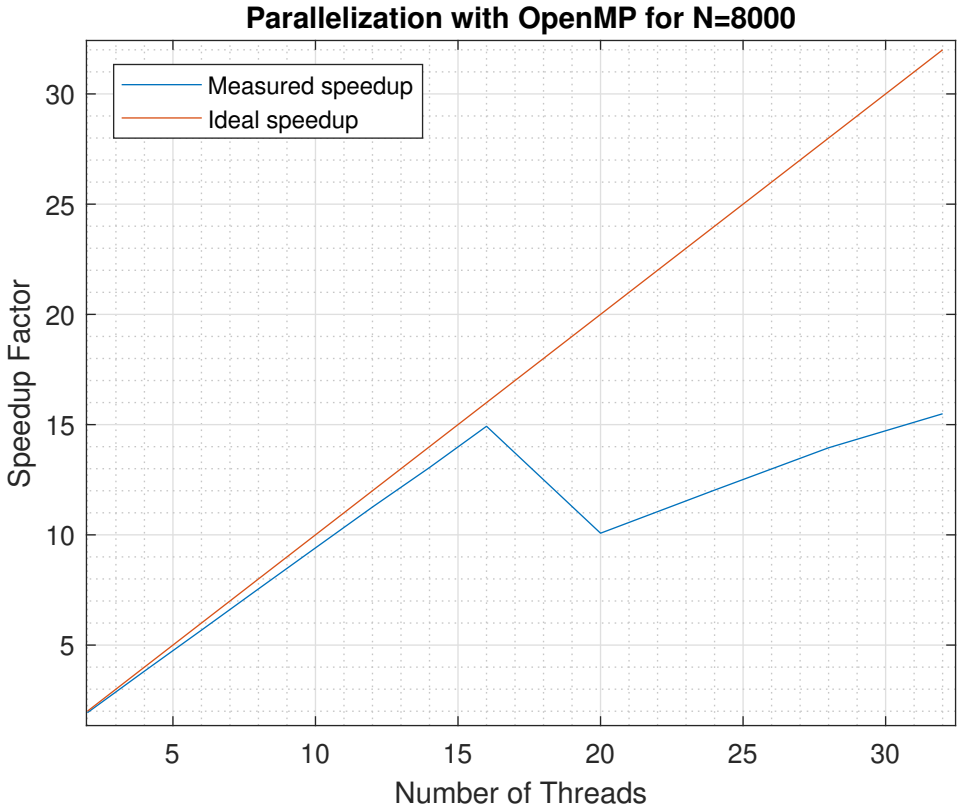

Figure 5: Speedup factor with different number of threads using OpenMP and "ellipse_N_08000.gal" as input file with 200 timesteps, plotted along with the idel speedup factor.

4.3 Parallelization discussion

As seen in figure 2 and 3, both parallelizations increasingly improve the speed of the code up until 16 threads, whereafter the perfomance decreases from the parallelization with 16 threads, at least initially. A clear difference is that the parallelization done with OpenMP follows the ideal speedup much closer then the one that was done with PThreads. This most likely happens because of the extra steps that are used in the PThreads-parallelization, where a intermediate function is used.

When the OpenMP-parallelization was applied to larger problems we see that up until 16 threads, the speedup is closer to the ideal speedup for N=8000 than for the smaller problems where N was 2000 and 4000. This is in line with what would be expected, that we can get closer to ideal speedup when the problem is larger.