High Performance Programming – Assignment 4
Group 9

Claude Carlsson
Edvin Germer

March 2, 2023

UPPSALA
UNIVERSITET

Contents

1 The problem **3**
1.1 Introduction . . . 3
1.2 Materials . . . 3
1.3 Equations . . . 3

2 The Solution **4**
2.1 Data structures . . . 4
2.2 Algorithm . . . 4

3 Performance and discussion **5**
3.1 Computer specifications . . . 5
3.2 Profiling and memory leakage . . . 5
3.3 Optimization Theory . . . 5
3.3.1 Serial Optimization . . . 5
3.3.2 Parallel Optimization . . . 5
3.4 Optimizing our code . . . 6
3.4.1 Serial Optimization . . . 6
3.4.2 Parallel Optimization with Pthreads . . . 7
3.4.3 Parallel Optimization with OpenMP . . . 9
3.5 Measurements . . . 9
3.5.1 Serial Optimization . . . 10
3.5.2 Parallel optimization with pthreads . . . 11
3.5.3 Parallel optimization with OpenMP . . . 13

1 The problem

1.1 Introduction

In this assignment, we created and both serially and parallelly optimized a program, galsim.c, that simulates motions of n particles inside a galaxy. To simulate the galaxy, we need to solve the N-body problem. We did so in two dimensions using a set of governing equations and using the symplectic Euler method. Each particle in the galaxy has a set of starting attributes: position, velocity, mass, and brightness. This data was provided beforehand and was used as input to our program. Our program is also able to visualize the evolution of the galaxy by using the X Window System (X11).

1.2 Materials

Given in this assignment is:

- A set of binary files containing the configuration and properties of the particles that will be the input to our model.
- An additional set of binary files containing the configuration and properties of the particles in the galaxy after some time steps. This will be used as a referential output to compare our model.
- A graphical routine and instructions to implement a visual representation of the simulation.
- A comparison script for checking that our model produces the correct result.
- A shellscript for checking that the assignment is in the correct format.

1.3 Equations

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

$$\begin{aligned} \mathbf{r}_{ij} &= (x_i - x_j)\mathbf{e}_x + (y_i - y_j)\mathbf{e}_y, \\ r_{ij}^2 &= (x_i - x_j)^2 + (y_i - y_j)^2, \end{aligned} \tag{2}$$

$$\begin{aligned} \mathbf{a}_i^n &= \frac{\mathbf{F}_i^n}{m_i} \\ \mathbf{u}_i^{n+1} &= \mathbf{u}_i^n + \Delta t \mathbf{a}_i^n \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \Delta t \mathbf{u}_i^{n+1} \end{aligned} \tag{3}$$

Equation 1 is the major governing equation for this problem and is used to iterate over all particles in our system and calculated the individual forces between them. These forces can then be used as seen in equation 3 according to symplectic euler in order to update the positions and velocities of all particles in our system for each timestep. Equation 2 defines how to calculate the vectors used in the force equation as well as the relative distance between each particle.

2 The Solution

2.1 Data structures

The input data used for this problem is read from a binary file which contains 6 doubles worth of data for each particle at time zero. These six doubles are:

- Coordinates for x and y
- Velocities for x and y
- Mass of each particle
- Brightness of each particle

This input data is iterated over and stored in our galsim program in the form of a struct array, with one double for each particle attribute.

We want to update all particles at once for each time step and so we have a secondary struct array just containing the two doubles for the x and y coordinates which is used for temporary storing the new positions until all particles have been iterated over. After that the result can be copied over the the original array of structs.

The reason for using structs is to keep the code readable and organized, one alternative could be to only use stand-alone variables, however, since this code is divided into functions it can become messy to pass each variable to each function. One drawback for using structs could be that when doing operations on the elements they are not so close in memory and thus makes the performance worse. It is also easier if we choose to parallelize since we already have an organized code by using structs and also have the code divided into functions.

2.2 Algorithm

The "brute force" algorithm with for calculating the evolution of the particles, with $\mathcal{O}(n^2)$ time complexity, is as follows when graphics is not enabled:

Algorithm 1 Solving the galaxy simulation problem

```
Read the input and get the starting values
for x ← 1 to N_steps do
    for i ← 1 to N_particle do
        for j ← 1 to N_particle do
            Calculate the distances between particle i and j
        end for
    end for
    for i ← 1 to N_particle do
        for j ← 1 to N_particle excluding i = j do
            Calculate and sum the force contributions between particle i and j
        end for
        Calculate the force of particle i
        Calculate the acceleration of particle i
        Calculate and update the velocity of particle i
        Calculate and store the position of particle i
    end for
    for i ← 1 to N_particle do
        Update the position of particle i
    end for
end for
```

In this algorithm, $N_{particle}$ is the amount of particles and N_{steps} is the amount of steps.

3 Performance and discussion

3.1 Computer specifications

System: UU Linux Host "Barany"
Processor: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
Operating System: Ubuntu 22.04 x86_64
Compiler version: gcc version 11.3.0 (Ubuntu 11.3.0-1ubuntu1 22.04)

3.2 Profiling and memory leakage

To get an idea of which part of the code takes a long time to run, we can use the utility tool gprof to do a profiling. Also, to understand the utilization of cache we can use the tool valgrind. Both of these tools gives an output that tell us how the program performs.

Valgrind can also check for memory leaks, memory leaks are important to address since it can cause the program to consume unnecessarily amount of memory. With valgrind, when you use it for memory leaks, you get an output that shows you which blocks of data are not freed.

3.3 Optimization Theory

3.3.1 Serial Optimization

In our process to optimize the program serially we looked at three different kind of overarching techniques which will be presented briefly: reducing instructions, optimizing memory usage, and instruction-level parallelism (ILP). To achieve these optimizations, we used compiler optimization flags, but also modified the code.

In general, the easiest optimization tool that gives good results is the flag "-O", there are many variations of this flag, but to simplify, there are three different levels to this flag, "-O1", "-O2" and "-O3". The first level tries to reduce the code size and executing time, the second level tries to optimize without making the executable bigger, since size can affect the instruction cache negatively. The third level optimizes even more, this can lead to increased compilation time and bigger executable size. [1] [2] This flag optimizes across the three different overarching techniques mentioned earlier.

There are several ways to reduce the instructions of a program, except the optimization flag, one can look at strategies such as: removing loop invariants, function inlining, make boolean evaluation faster by using boolean short circuit, strength reduction by using cheaper operations, and avoiding denormalized floating-point values. [2]

Optimizing the memory usage in a program includes strategies such as: improving the utilization of the cache memory by looking at data locality or loop blocking, memory allocation by only allocating the memory the program needs, and using keywords such as "restrict" or "const" in the code. [3] Whenever optimizing for memory usage, it is important to check for memory leakage and to apply the optimizations in the right way. Otherwise, it may cause the program to not run as expected or even slower.

Even though we only optimize the program serially, we use something that is called instruction-level parallelism, this aims to optimize in a way that allows the computer to perform multiple instructions at the same step. Some related strategies and concept to ILP are: loop unrolling which lowers the amount of iterations in a loop, avoiding branches, branch prediction, and auto vectorization. [4]

3.3.2 Parallel Optimization

Parallelization is the process of creating and managing threads in order to run a program in parallel, which allows the program to access the full computational power of the computer. Often, there are some parts of the code must be run serially and other parts the code can be optimized to run on

multiple threads in parallel [5]. This means that the first task when trying to parallelize a program is to identify which segments of the program that are parallelizable, i.e. which tasks that can be broken down into independent subtasks. Once the parallelizable parts of the code have been identified the actual parallelization process depend on which programming language and what libraries are used, but generally you want to split the work as evenly as possible on all threads used. For this assignment we will use our serially optimized code, written in C, and then use Pthreads and OpenMP to parallelize the program.

By using the gprof tool to analyze the serial code, we can identify that the majority of the runtime of the program (over 90%) is taken up by computing the distance matrix (in the *get_dist* function) and by computing the updated positions for each new particle (in the *update_particle* function).

The calculation of the distance matrix is a perfectly parallelizable task since all elements in the matrix can be calculated independently by iterating over all the particles. The process of updating each particles position for each timestep is also a perfectly parallelizable task since the update of one particle does not depend on the update of any other particle.

3.4 Optimizing our code

3.4.1 Serial Optimization

Based on the optimization theory discussed in section 3.3.1 we tried to improve the efficiency of our program. This was an iterative process and the results can be seen in table 1.

Our original solution, is the first time we were able to run the program with no resulting error when our calculations and the reference output provided in the assignment files. This version has no deliberately implemented optimization methods and no compilation flags and the resulting performance is thus quite poor.

The V2 version has one small but very impactful difference. When calculating r^3 we originally used the *pow*() function, in this version we do the calculation manually by replacing $pow(r, 3)$ with $r * r * r$ which resulted in a large performance increase.

The V3 version has another small improvement. We noticed that when calculating the force we first multiply by m_i and for calculating the acceleration we then divide by m_i. Since we are really only interested in the acceleration it does we can simply skip the multiplication and division with the same number m_i. This can be confirmed by plugging equation 1 into equation 3.

The V4 version uses a matrix lookup table for calculating the distances between points. This is possible because for each timestep we are first calculating all the new coordinates serially but wait to actually move the particles to their new locations until all new coordinates have been calculated. This means that during each iteration of the time, all the distances between points are constant. Originally we calculated all distances for each particle, but this is not efficient since the distance between particles i and j is the same as the distance between particles j and i. The matrix is constructed in the following way where each row (particles i) correspond to the distance the all other columns (particles j), diagonal elements is the distance between each points and itself and $r_{i,j} = r_{j,i}$. This means that we only have to calculate the upper triangle of the matrix and then copying the elements to the lower third, effectively reducing the calls to the expensive *sqrt* function by half. A visual representation can be seen in matrix 4.

$$\begin{bmatrix} 0 & r_{0,1} & r_{0,2} & r_{0,3} \\ r_{1,0} & 0 & r_{1,2} & r_{1,3} \\ r_{2,0} & r_{2,1} & 0 & r_{2,3} \\ r_{3,0} & r_{3,1} & r_{3,2} & 0 \end{bmatrix} \tag{4}$$

To further improve upon the speed and efficiency of our program we added the *const* keyword before all constant variables in our code. For example variables like: The correctional term e_0, the gravity constant, the timestep and so on. We also added the restrict keyword before all pointers in the

functions, letting the compiler know that these pointers can be accessed and written to with no interference from other processes. We also added loop unrolling in the *update_particle* function to try and further improve on the efficiency. We used an unrolling factor of 4, meaning that 4 iterations over the particles is handled by every step of the for loop.

By adding some compilation flags we tried to further improve on the program. We used $-ffast-math$ and $-march=native$ together with the $-O1, -O2$ or $-O3$ flags which resulted in a huge performance increase.

3.4.2 Parallel Optimization with Pthreads

In order to parallelize the code with Pthreads it is required to do some manual restructuring of the code, to utilize the Pthreads library properly.

One of the major differences in the code structure for the parallel Pthread solution, compared to the serial one, is that the two functions, *get_dist* and *update_position*, which originally took several inputs of various data types, now have to be rewritten into taking only one void pointer as an input, and the resulting computation can not be returned in the usual C fashion. To deal with this technicality we construct two new structs, one for *get_dist* and one for *update_position*), and these structs define the input/output that each thread has to work with. We define an array of these structs and when calling the functions we pass a pointer to the corresponding struct and cast it to void. Once in the function, this void pointer is then cast back into a struct pointer and "unpacked" where each element from the struct is stored as a local variable in each thread. When the computations are done, the results are written into the struct that each thread has a pointer to and the function then exits. The extra steps for passing variables to and form the functions come at a cost of increased computational power and time, but this extra time is small in comparison to the speed up that comes from the multiple threads.

To implement the Pthread optimization, both the function and the calling of the functions had the be altered. The two algorithms described below (algorithm 2 and 3) aim to provide a simplified overview of how the calling of the functions have changed.

Algorithm 2 Simplified Original Serial Simulation Loop

```
for x ← 0 to N_steps do
    Calculate the distance matrix
    for i ← 0 to N_particle do
        for j ← 0 to N_particle do
            Calculate new positions in a temporary structs
        end for
    end for
    for j ← 0 to N_particle do
        Transfer result from temporary struct to official particle structs.
    end for
end for
```

Algorithm 3 Pthread Simulation Loop

for $x \leftarrow 0$ to N_{steps} **do**
  for $t \leftarrow 0$ to $nThreads$ **do**
    Calculate $\frac{N_{particle}}{nThreads}$ rows of the distance matrix
  end for
  for $t \leftarrow 0$ to $nThreads$ **do**
    Calculate new positions for $\frac{N_{particle}}{nThreads}$ particles in a temporary array
  end for
  for $j \leftarrow 0$ to $N_{particle}$ **do**
  end for
end for

To parallelize the *get_dist* function, we are iterating over the number of threads and for each thread we are calling the function once. For each function call we passing the thread index, pointer to the particles structs and a pointer to the dynamically allocated matrix for storing the results, as a void pointer. Once in the function the arguments are unpacked and the calculations can begin. There are a few possible options for how the values for the matrix can be calculated. The most straightforward way would be to split the matrix into segments (one for each thread) and then allowing each thread to calculate the distances for those indices. The naive approach by letting the first thread calculate the first couple of rows, the second thread the next couple of rows and so on is not an effective solution. The reason for that is because we are only calculating the upper triangle values of the matrix, and than mirroring those element to the lower triangle (leaving the diagonal untouched). For every row we have fewer and fewer elements, resulting in each next thread would have less and less work, which would result in unbalanced thread utilization. The more balanced approach, which we chose to implement, is to simply split the calculations where each thread is responsible for alternating rows. The thread index is used to determine which set of rows that that thread should work on. All threads are writing to the same distance matrix but because each thread works on different indices, there is not conflict and a mutex lock and unlock is not necessary.

The parallelization of the *update_particle* function is implemented in a similar but slightly different way than the *get_dist* function. Also, the iteration over the first particle index has been moved to the actual function for a cleaner main loop and ease of parallelization. Just like for the *get_dist* function, for each time step, we are now iterating over the total number of threads. For each thread we are calling the *update_particle* function with the same arguments as in the serial version (the particles struct, the temporary struct for the positions and the distance matrix) but now also with a start and an end index, which represent which segment of the total number of particles each thread should update. Similarly to the *get_dist* function the arguments are passed to the function as void pointers and then cast back inside the function and unpacked to local variables. Then the updated positions are calculated in the same way as in the serial code. Also for this function, all the threads are writing to the same array of temporary position structs, but to different indices so there is once again no need for mutex lock and unlock, which gives an effective parallelization.

3.4.3 Parallel Optimization with OpenMP

Algorithm 4 OpenMP Simulation Loop

```
for x ← 0 to N_steps do
    #pragma omp parallel for schedule(dynamic, nthreads*10)
    Calculate the distance matrix
    #pragma omp parallel for schedule(dynamic, nthreads*10)
    for i ← 0 to N_particle do
        for j ← 0 to N_particle do
            Calculate new positions in a temporary structs
        end for
    end for
    #pragma omp parallel for
    for j ← 0 to N_particle do
        Transfer result from temporary struct to official particle structs.
    end for
end for
```

When optimizing was done using OpenMP, we parallelized three things; the calculation of the distance matrix, the calculations for the particles, and the transfer of the positions to the original struct. The directives from the OpenMP API that we used were: "parallel for" and "parallel for schedule(dynamic, nthreads*10)". Virtually, the only change in the code is to add these lines before the for-loop, this shows the stark contrast between pthreads and OpenMP, that, while pthreads is a low-level API, OpenMP is a high-level API.

The directive "parallel for" parallelize the for-loop, the other directive "schedule" allows us to control how the iterations are divided between the threads. For OpenMP, the three common types of scheduling are: static, dynamic and guided. Static makes the iterations divided into chunks of a fixed size and each thread gets a interval, dynamic makes so that each thread executes a chunk and request a new chunk, guided is a hybrid approach of the two. [6] When just using "parallel for", then static is going to be used since it is the default.

When using the directive "schedule" one can also define the chunk size, we choose the chunk size to be ten times the number of threads. This gave us the best performance, and it has to do with the program creating less of an overhead in the task scheduling since it is using larger chunk size. The idle times of the threads are also lowered by using dynamic, since if we had just divided the load, the later parts of the iterations might take longer time, which leads to idling for the earlier threads. The last loop has only "parallel for" since it made no difference in the choices for the directive "schedule". So we left it to its default, that is, static.

3.5 Measurements

All timing were made using the *time* command in the terminal when calling the functions. The reported times are the *real* times which gives an accuracy of three decimal points. In the parallel optimization sections, we define ideal as scaling perfectly with the number of threads. For example, if a program takes 10 seconds to run single-threaded, it should take 5 seconds when using 2 threads and 1 second when using 10 threads.

3.5.1 Serial Optimization

Version	N=100	N=500	N=1000
Original Solution	0,213	5,155	20,603
Algorithm V2	0,095	2,251	8,862
Algorithm V3	0,093	2,227	8,849
Algorithm V4	0,090	2,081	8,374
Added Const keyword	0,091	2,083	8,366
Added Restrict keyword	0,089	2,081	8,363
Added Loop Unrolling	0,087	2,010	8,060
-ffast-math -march=native	0,086	2,000	8,077
-O1	0,042	0,843	3,399
-O2	0,042	0,844	3,388
-O3	0,040	0,799	3,226

Table 1: The measurements are 10-run averages in seconds and for 200 timesteps of size 10e-5

Best timing
For N=3000 and 100 timesteps our best timing was: 18.814s

Time Complexity

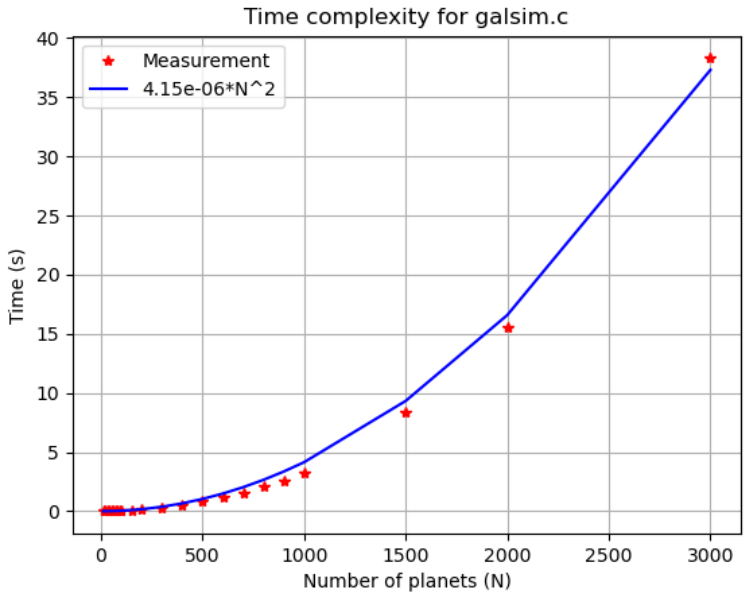

Figure 1: The time complexity for the galaxy simulation as a function of the number of planets N with 200 time steps and dt=10e-5

The expected time complexity for this brute force approach for the N-body problem is expected to have time complexity $O(N^2)$. By running our program and taking timing measurements we can confirm that this is the case. In figure 1 we can see that if we try to fit a function to our observed data on the form $f(N) = C * N^2$ we get the function $f(N) = 4.15e\text{-}0.6 * N^2$ which nicely connects our data points and confirms the theory.

3.5.2 Parallel optimization with pthreads

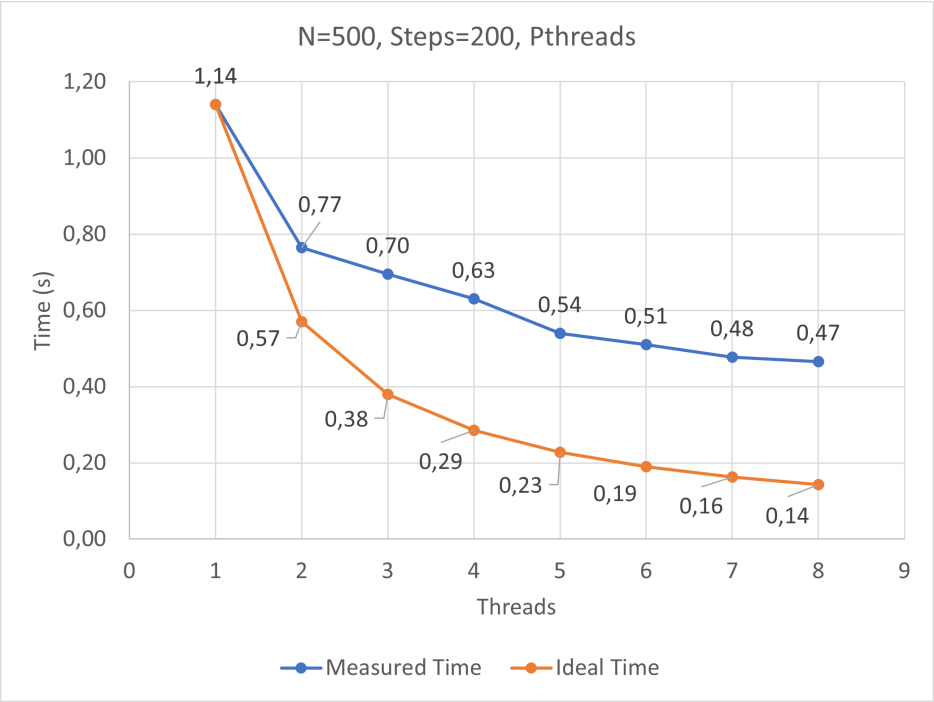

Figure 2: The average time of ten runs with step size = 200 and particles (N) = 500 when parallelization is done using pthreads

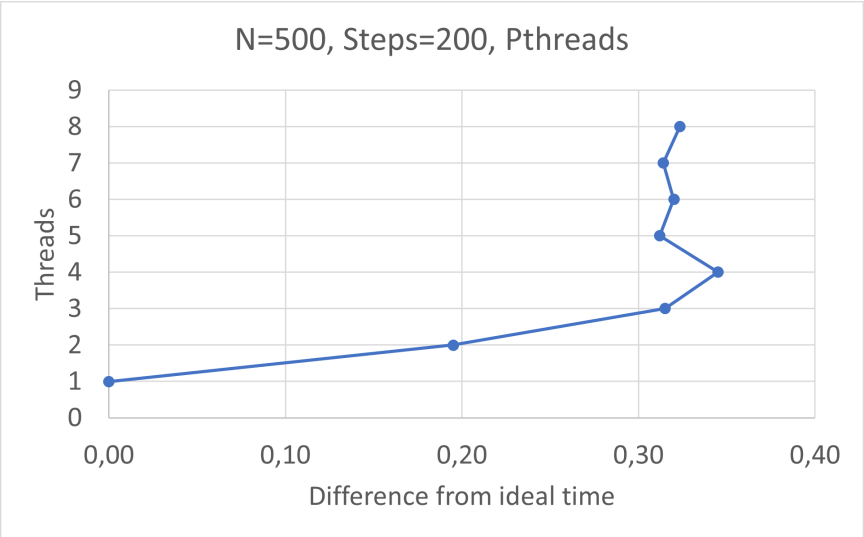

Figure 3: The average time difference between measured and ideal of ten runs with step size = 200 and particles (N) = 500 when parallelization is done using pthreads

From figure 2 we can see that we did not achieve optimal multi-threading, however, the time did decrease for each additional thread. The largest jump was from one thread to two threads, however, for each additional thread up to about five threads, the gap widens between the measured time and the ideal time, this can be seen in figure 3. The diminishing of return of adding additional threads is very notable in this case. The ideal reduction from one thread to eight is a time reduction of 87.5%, but we only got approximately a time reduction of 58.8%.

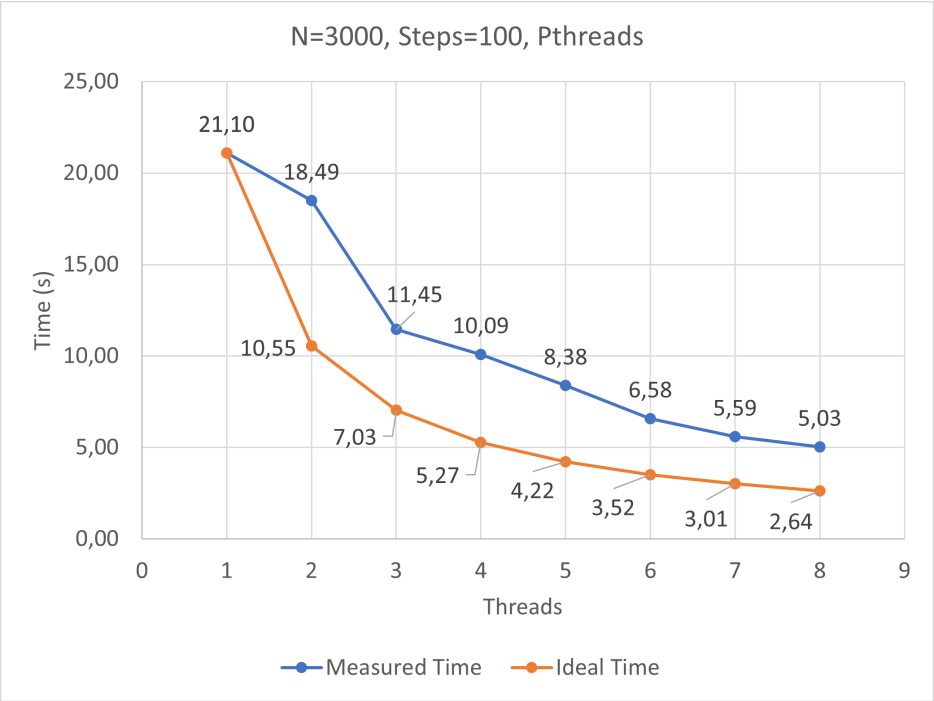

Figure 4: The average time of ten runs with step size = 100 and particles (N) = 3000 when parallelization is done using pthreads

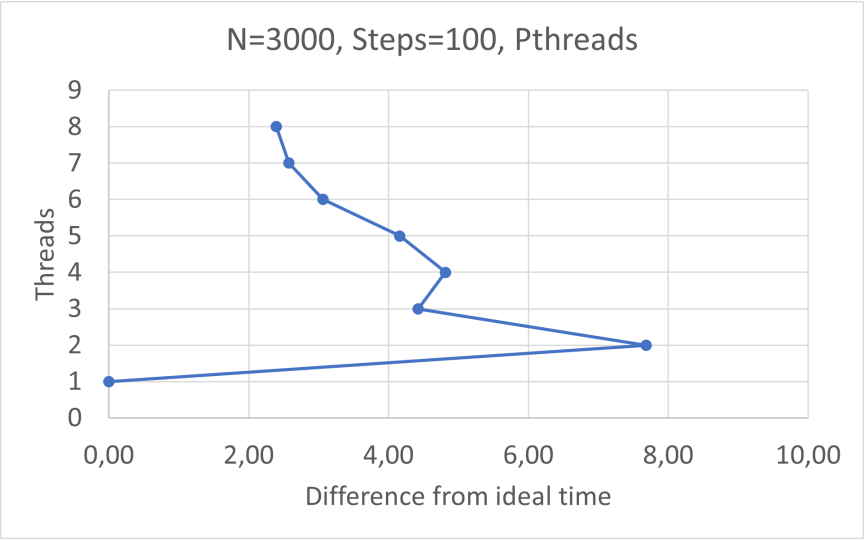

Figure 5: The average time difference between measured and ideal of ten runs with step size = 100 and particles (N) = 3000 when parallelization is done using pthreads

From figure 4 we can see that we achieved fairly good multi-threading, and the largest jump was between two and three threads, it is unclear why two threads performed so poorly compared to three threads, but it could have to do with overhead. Compared to figure 3 where we got diminishing returns, we now get increasing returns, which can be seen in figure 5. There is a fundamental difference in this setup since we now have more particles and fewer steps, this can affect performance in many ways, one of those ways could be the overhead of the program. The ideal reduction from one thread to eight is a time reduction of 87.5%, but we only got approximately a time reduction of 76.2%.

3.5.3 Parallel optimization with OpenMP

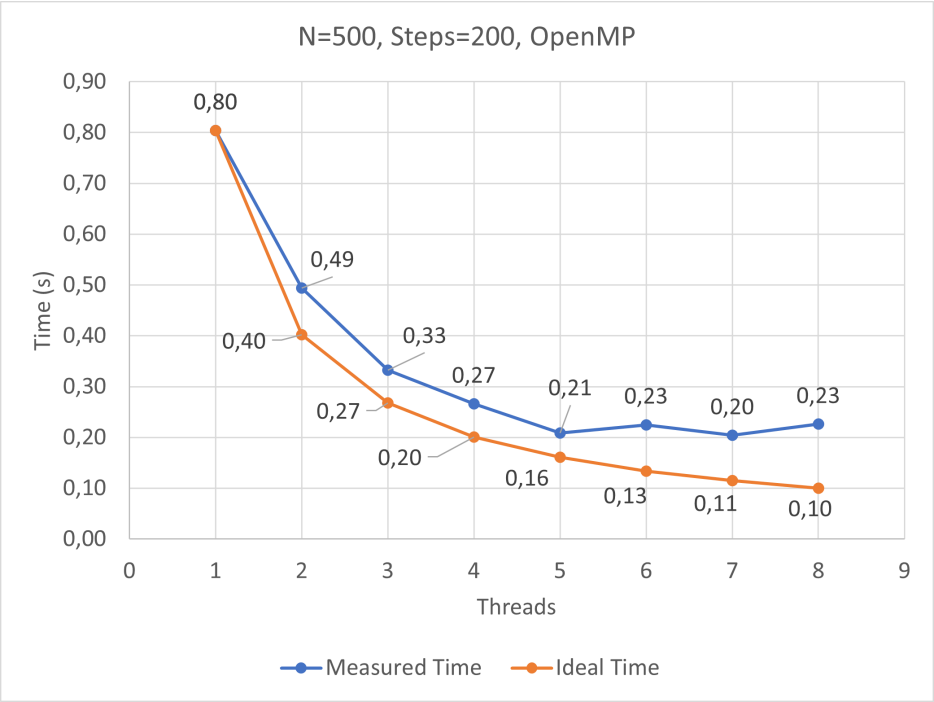

Figure 6: The average time of ten runs with step size = 200 and particles (N) = 500 when parallelization is done using OpenMP

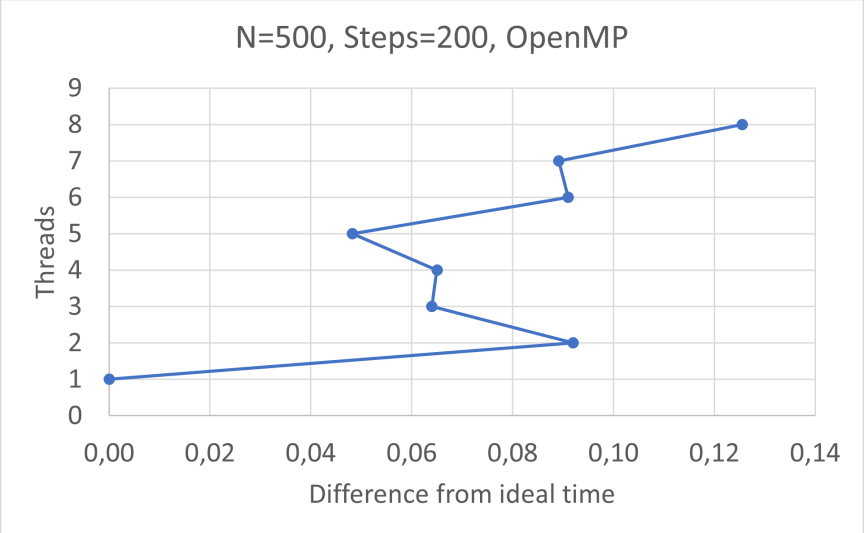

Figure 7: The average time difference between measured and ideal of ten runs with step size = 200 and particles (N) = 500 when parallelization is done using OpenMP

From figure 6 we can see that we achieved fairly good multi-threading up to five threads. After five threads, it flat-lined and our performance got either worse or stayed the same, this is illustrated in figure 7. The largest jump is from one thread to two. Once again, it seems that when we have fewer particles and more steps we achieve diminishing returns. The ideal reduction from one thread to eight is a time reduction of 87.5%, but we only got approximately a time reduction of 71.2%.

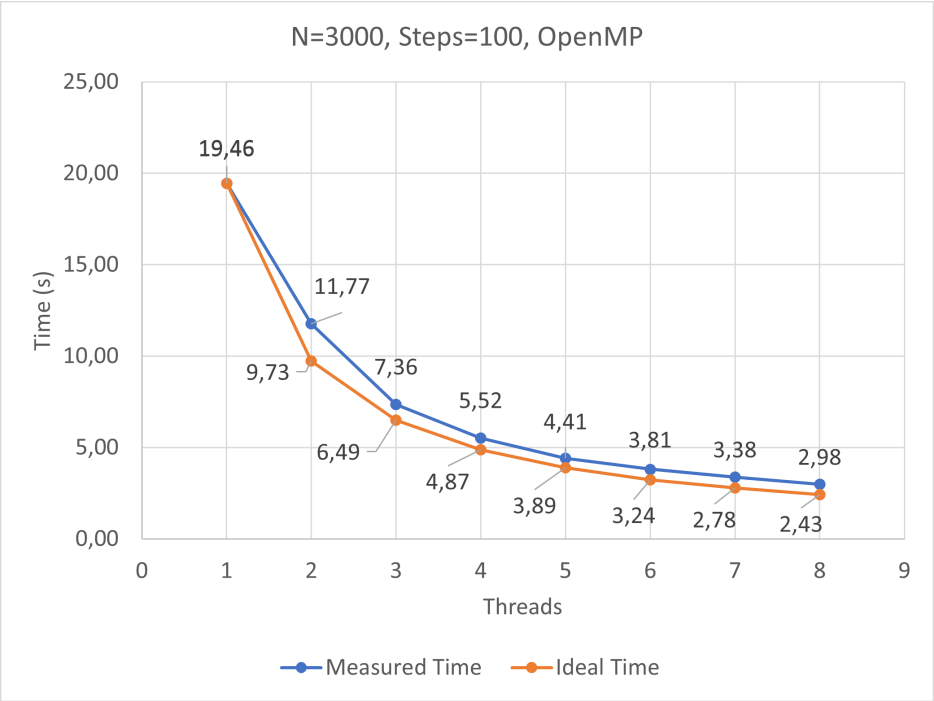

Figure 8: The average time of ten runs with step size = 100 and particles (N) = 3000 when parallelization is done using OpenMP

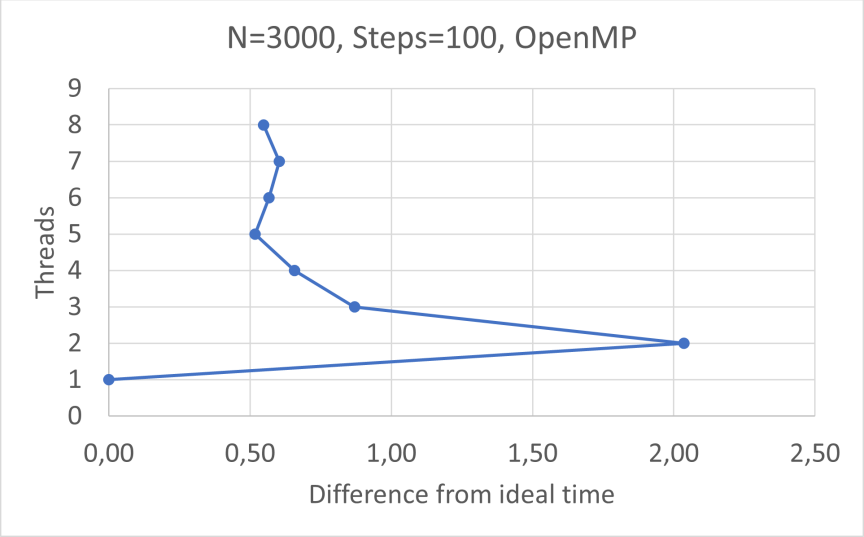

Figure 9: The average time difference between measured and ideal of ten runs with step size = 100 and particles (N) = 3000 when parallelization is done using OpenMP

From figure 8 we can see that we achieved excellent multi-threading. Furthermore, can see that from figure 9 the gap follows the same pattern as it did using the same arguments for the program but using pthreads, shown in figure 5. However, there are subtle differences, when using pthreads the gap between measured and ideal decreased for virtually every thread, while for OpenMP, the gap stays constant after five threads. The ideal reduction from one thread to eight threads is a time reduction of 87.5%, but we only got approximately a time reduction of 84.7%. This is very impressive since it only required us to add three lines to the program, excluding importing libraries and logic for assigning the number of threads.

References

[1] GCC optimize options. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed on February 16, 2023.

[2] High performance programming uppsala university spring 2022 lab 5: Serial optimization part 1, reducing instructions. Course material, 2022.

[3] High performance programming uppsala university spring 2022 lab 6: Serial optimization part 2, memory usage. Course material, 2022.

[4] High performance programming uppsala university spring 2022 lab 7: Serial optimization part 3, instruction-level parallelism (ilp). Course material, 2022.

[5] High performance programming uppsala university spring 2022 lecture 5: Pthreads. Course material, 2022.

[6] OpenMP Architecture Review Board. Openmp 5.2 api syntax reference guide, 2020.