Uppsala Universitet

High Performance Programming

1TD062

Assignment 3

By:
Jacob K. Malmenstedt Ture
Hassler

March 3, 2023

Contents

1 Introduction

The N-body problem is a classic simulation task, where the gravitational forces from N different objects are calculated for each step, and from there the path of the objects can be simulated.

To compensate for the numerical instability when the radius goes to zero a factor ϵ_0 was added, giving the so called Plummer spheres. The acceleration and force on each object is given by eq. 1.

$$\boldsymbol{a_i} m_i = \boldsymbol{F_i} = -G m_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{ij} + \epsilon_0)^3} \boldsymbol{r_{ij}} \tag{1}$$

2 Solution

2.1 Serial Code Solution

During the development of the solution, several different iterations and methods were tried, both different implementations and slightly different algorithms.

The data that needed to be saved for each particle was the position, mass, velocity and brightness, furthermore all values were given in doubles. This was implemented using the following struct:

```
struct Particle
{
    double x;
    double y;
    double mass;
    double vx;
    double vy;
    double brightness;
};
```

To keep track of all the particles, an array was created that pointed to each individual particle's struct.

As the performance timings is run without graphics the code checks if the user wants to display graphics or not. If the user wants graphics the code runs a copy of the normal code with the difference that it also updates the graphics in each timestep. The reason we have a entire copy of the code for the graphics part of the code instead of for example just calling an update-function is that when the code is run in "performance-mode" unnecessary calls to a separate update-function might slow down the program rather than just having the entire code written out. This might increase the size of the file but since we do not care about size and only time it increases performance slightly.

The actual part of the program that is used to update positions of all particles in the timed non graphics part of the code uses a rather simple algorithm (in pseudocode):

```
for all timesteps
    for all particles j
        set current particle to p1
        for all particles k=0, k<j
            calc. acceleration on p1 and save in a sum
        for all particles k=j+1, k<N
            calc. acceleration on p1 and save in sum
        update velocity using calculated acceleration
        reset acceleration sum to 0
    for all particles j=0, j<N
        update pos using velocity
```

Our implementation of this algorithm in code can be seen below:

```
for (int i = 0; i < nsteps; i++) // for all timesteps
{
    for (int j = 0; j < N; j++) // for all particles update acc and vel
    {

        p1 = particles[j]; // current particle
        for (k = 0; k < j; k++) // calculations of acc for all particles before current particle
        {

            p2 = particles[k];
            rij = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
            acc_k = p2.mass / ((rij + e0) * (rij + e0) * (rij + e0));
            acc_x += acc_k * (p1.x - p2.x);
            acc_y += acc_k * (p1.y - p2.y);
        }

        for (k = j + 1; k < N; k++) // calc for all particles after current particle
        {
            p2 = particles[k];
            rij = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
            acc_k = p2.mass / ((rij + e0) * (rij + e0) * (rij + e0));
            acc_x += acc_k * (p1.x - p2.x);
            acc_y += acc_k * (p1.y - p2.y);
        }
        acc_x *= -G;
        acc_y *= -G;

        // update velocity
        particles[j].vx += acc_x * delta_t;
        particles[j].vy += acc_y * delta_t;

        // reset acceleration
        acc_x = 0;
        acc_y = 0;
    }
```

```
36
37      for (int j = 0; j < N; j++) // for all particles update pos
38      {
39          // update position
40          particles[j].x += particles[j].vx * delta_t;
41          particles[j].y += particles[j].vy * delta_t;
42      }
43  }
```

One alternative algorithm that could theoretically be an improvement on our algorithm is the following:

```
for all timesteps
    for all particles j
        set current particle to p1
        for particles k=j+1, k<N
            set particle[j] to p2
            calc. force between p1 and p2
            add to acceleration sum of p1
            add to acceleration sum of p2
        update velocity using calculated acceleration
    for all particles j=0, j<N
        update pos using velocity
        reset acceleration sum to 0
```

This algorithm would give an improvement over the first one because we know that the force between two particles are equal and opposite and therefore we can compute the acceleration of particle 1 with respect to particle 2 at the same time that we calculate the acceleration of particle 2 with respect to particle 1. In this implementation we would need to store the acceleration sums for all particles but we can drastically decrease the number of computations needed as we basically get two accelerations computed for the price of one.

We implemented this second algorithm in code but found that it was actually a bit slower than our original algorithm. We were a bit confused by this as we thought that the decrease in the number of calculations would result in a lower run-time. One explanation we thought of was that the storage of the additional values for acceleration for each particle would mean that the particle would take up more space in memory and therefore when data is read into cache we would not read in as much data that can be used in the following calculations and thus increase our cache-misses. Since the computer would then have to go "further" to get data for the next calculation this would cause our code to run slower and even decrease performance to such a level that even the decrease in calculations would not make up for it.

After our code has calculated and updated the last position we write the data to an output file and free the memory allocation for all particles.

A large part of our optimization was made to minimize the time in the innermost for-loop, since this is the one where the program will spend most of it's time.

The innermost loops are divided into two separate loops to avoid having an if-statement inside a single loop that checks that we do not calculate the acceleration with respect to the

current particle. The current implementation first calculates the acceleration for all particles before the current and then for all particles after and thus does not need an if statement.

We chose to compute the values for r_{ij} and store it in a variable as this uses an expensive square-root calculation that we wanted to avoid doing multiple times. In this way we save one square-root calculation everytime the innermost loop executes. We also chose to save a variable called acc_k which is a common constant for both the acceleration in y- and x-directions. This saves us from having to recompute the division and cube a second time. We observed that the multiplication with $-G * m_i$ in eq. 1 is applied to all components of the acceleration sum so we decided to move this out of the inner loop and apply it afterwards and therefore saving some computations inside the inner loop. Also the m_i cancels out in our implementation and we could ignore it completely.

2.2 Parallelized Code Solution

The serial code from section 2.1 was parallelized using two different frameworks, first using Pthreads and then using OpenMP.

All timings were performed on a M2 Macbook Air with 8 cores using the clang compiler, unless stated otherwise.

2.2.1 Pthreads code solution

For the pthreads solution we tried a couple different iterations. Due to the counter intuitive results of getting a slower serial code when calculating the forces only once for each pair of particles, our innermost for loop had the advantage of being fully parallelizable. Since each thread are only updating their own particles (during the calculation loop), there is no risk of overwriting changes made by another thread and we don't have any synchronisation slowdowns.

After the innermost calculation for loop, the threads need to synchronize before updating the position, to avoid using a updated value of the position while still using the position for the calculations.

Our first most naive (and outright stupid) parallelization was creating a separate thread for each N. For comparison our serial code took $1.784s$ compared to $21.525s$ afterwards. This is much much slower due to the overhead required for creating the 500000 threads. Both measurements were on the $N = 5000$ version with 100 steps. This version of the code ran the particle update code in serial.

Our next version split the particle calculations among a given amount of threads, with each thread receiving a given number of particles. However the threads were still recreated for each timestep, and the position update was performed in serial.

We then tried to parallelize the particle position update as well by adding a barrier inside each thread. This allowed us to move the code for updating the position to be performed inside each thread.

Our fourth and fastest version created the threads only once, reducing the thread overhead, and instead used a barrier to synchronize both between calculating the forces, and updating the position. For the code performed inside each thread, see below:

```
void *calc_forces(void *arg)
{
    /* Calc forces for one specific particle */
    struct Pthread_data *input = (struct Pthread_data *)arg;

    int lb = input->lowerB;
    int ub = input->upperB;

    double acc_x = 0, acc_y = 0, acc_k; // acceleration
    double rij;                         // distance between particles
    struct Particle p1, p2;             // particles

    for (int i = 0; i < nsteps; i++)
    {
        for (int j = lb; j < ub; j++) // calculations for all particles between lb and ub
        {
            p1 = particles[j]; // current particle

            for (int k = 0; k < N; k++) // calculations of acc for all particles acting on current particle
            {
                p2 = particles[k];
                rij = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
                acc_k = p2.mass / ((rij + e0) * (rij + e0) * (rij + e0));
                acc_x += acc_k * (p1.x - p2.x);
                acc_y += acc_k * (p1.y - p2.y);
            }
            acc_x *= -G;
            acc_y *= -G;

            // update velocity
            particles[j].vx += acc_x * delta_t;
            particles[j].vy += acc_y * delta_t;

            // reset acceleration
            acc_x = 0;
            acc_y = 0;
        }
        // Wait for all threads to reach the barrier to sync before updating pos
        barrier_inner();

        for (int j = lb; j < ub; j++) // update pos
        {
            // update position
            particles[j].x += particles[j].vx * delta_t;
            particles[j].y += particles[j].vy * delta_t;
        }
```

```
47          barrier_outer(); // Wait for all threads to reach the barrier to sync
    between timesteps
48      }
49      return NULL;
50  }
```

The barrier used a mutex lock, to synchronize the threads both between updating the positions, and between each time step. The code for this was borrowed from the lab exercise 9 task 1, and can be seen below:

```
1  void barrier_inner(){
2      int mystate;
3      pthread_mutex_lock(&lock);
4      mystate = state;
5      waiting++;
6      if (waiting == NUM_THREADS)
7      {
8          waiting = 0;
9          state = 1 - mystate;
10         pthread_cond_broadcast(&mysignal);
11     }
12     while (mystate == state)
13     {
14         pthread_cond_wait(&mysignal, &lock);
15     }
16     pthread_mutex_unlock(&lock);
17 }
```

2.2.2 OpenMP code solution

Due to limited support of OpenMP with Clang compiler, all the following timings were performed with a M2 Macbook Air, but using GCC-12 as the compiler.

The OpenMP parallelization was rather straightforward. Due to the parallelizable inner loop, we simply added the command

```
1  #pragma omp parallel for private(p1, p2, rij, acc_k, acc_x, acc_y, k) schedule
       (dynamic)
```

in front of the loop that calculates all accelerations for all particles since these are 'embarrassingly parallel' computations. In this command we also specify which variables that needs to be private since the threads by default shares all variables except the loop counter. We still need all threads to wait before continuing to update all positions and the 'parallel for' command has an implied barrier at the end of the for loop but to make sure that the barrier is applied we decided to add an explicit barrier between these parts. This is also to ensure that the same threads continue with the next part of the code and we do not destroy and create threads unnecessarily in between these loops.

After all threads have reached the barrier we call

```
1  #pragma omp parallel for schedule(static)
```

to update all positions of all particles since this is also an operation which is easily parallelized. We do not need to assign any private variables here as all the threads update the positions in different parts of the struct and should never conflict with each other.

We tested different kinds of schedules for both parallel loops and in theory the static schedule should improve data locality while dynamic/guided should decrease downtime for the threads. We concluded that there was not a huge difference but the dynamic schedule for the first loop and static schedule for the second loop seemed to be a little faster than the others. We also tested different 'chunk'-sizes and did not find a huge difference for different values here either but for the dynamic schedule in the first loop 'chunk'-size 4 seemed to give marginally better results than other values.

3 Performance and discussion

3.1 Serial code performance and discussion

We chose to optimize the performance of our code towards our personal hardware, in this case a M2 Macbook Air. All timings were on this machine unless stated otherwise and were performed on the $ellipse\_N\_03000.gal$ file with 100 steps and 0.00001 timestep. The compiler used was Apple clang version 14.0.0, but when comparing with gcc-12 we saw no noticeable difference, using the -O3 -ffast-math flags as default. All timing were done using the built in unix time function, measuring the total wall time for running the entire file.

We tried two different approaches for saving the data for each particle, In the first iteration of the code, the first part read the data and placed it in three arrays; one for the position and mass of each particle, one for the velocity and one for the brightness. The reason we placed the data in these three arrays were because these data values are needed in different frequency in the calculation and we do not want to load in unnecessary data to the cache. The thought was that this takes up space that might otherwise be filled with data that we need in the calculations. Particularly the innermost loop that calculates the total acceleration of a particle with respect to all other particles uses only data about the position and mass and we therefore thought that having less unused data in the cache might speed up this part of the code. However, due to the large size of the list we instead got a lot of cache misses.

We then tried changing to having the data saved in a struct instead. This turned out to be slightly faster and significantly reduced the amount of L1 cache misses. For comparison, for one specific run the time was reduced from 0.804s to 0.675s. We chose to continue with this implementation. When checking with valgrind on the linux computers, we achieved a 1.6% L1 cache miss rate and a very few last level cache misses.

One interesting thing we noticed when changing between the array implementation and the struct implementation, was that it became faster on our computers but slightly slower on the school linux computers. This could perhaps be because of the different ratio between the memory and cpu speed of the school computers compared to ours, but we could not make any conclusions.

We also tried different versions of reordering the innermost for-loop in an attempt to further

increase the performance. One thing we did, to avoid any if statements in the innermost loop was split it into two loops, one for all particles before the current and one for all particles after. This was to avoid having to check if the current iteration through the loop was about to calculate the effect of gravity on itself. For comparison, the code using an if statement was around 0.5s slower. We also tried minimizing the amount of operations needed whenever possible, especially with respect to division and the sqrt call, since they need significantly more flops. Instead we chose to save these in temporary variables. Much of this had very minor effects on our measured time, which we believe could be due to the compiler already performing several of these changes when using the -O3 flag.

We tried several different compiler optimization flags. Our final version of the code took, for example 5.903s to run without any optimization flags, 1.014s with -O2 and 0.659s with -O3 -ffast-math. We ended up using the -ffast-math even though it can increase the numerical error, since the measured error when compared to the reference output data, was still very low.

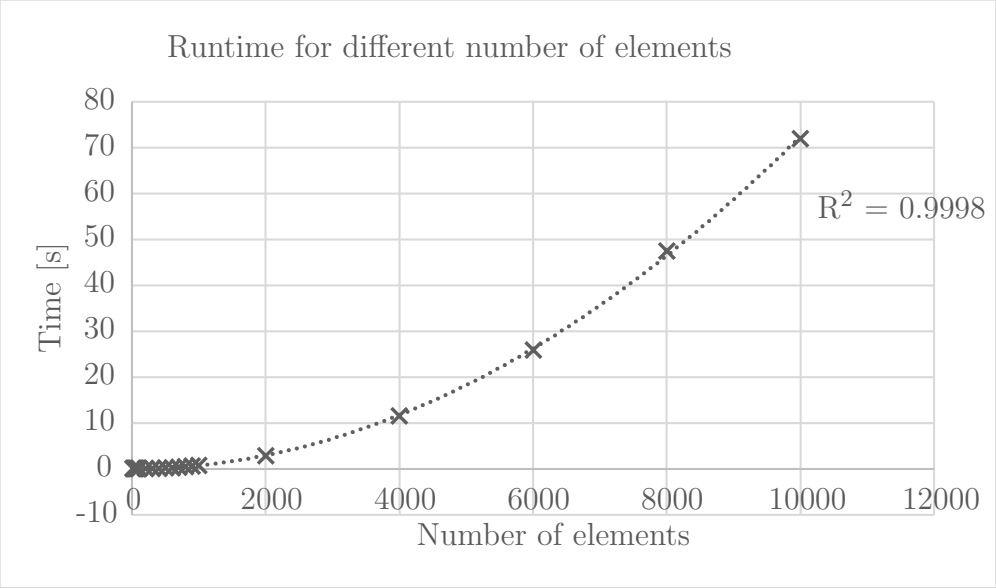

Figure 1: Plot of the different runtimes for a different number of elements

The time was measured for different N using the $ellipse\_N\_10000.gal$ file, but only loading up to a given amount of particles. Each program was run for 1000 steps with 0.00001 timestep. The time clearly follows the $O(N^2)$ time complexity we expect from this straightforward implementation.

3.2 Parallelized code performance and discussion

For the parallelized versions of the code, we saw significant speedups compared to the serial version. All time measurements were performed using a M2 Macbook Air with 8 cores, however due to compatabillity issues two separate compilers were used, both Clang and GCC-12, with slightly varying speeds.

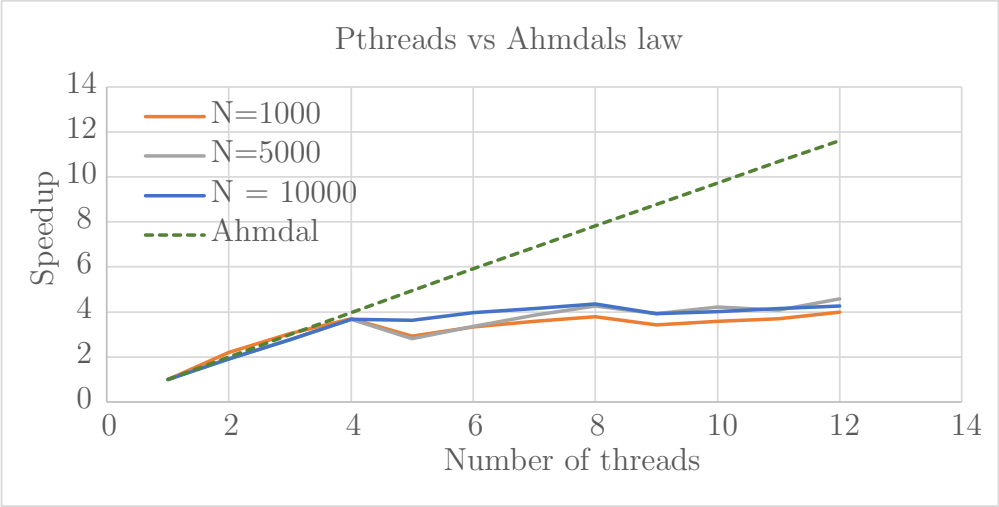

Figure 2: Plot of the speedup of the Pthreads version of the code as a function of the number of threads used compared to the theoretical speedup calculated using Ahmdal's law. Using the GCC-12 compiler.

3.2.1 Pthreads version discussion

Our goal when paralellizing with pthreads was to perform as much of the calculations in parallel as possible, and the reduce the number of threads created to reduce overhead.

Changing from creating the threads each loop, to just creating them once was marginally better, reducing the time from around $0.45s$ to $0.43s$. One thing we did as well was to try to divide the workload as evenly as possible, dividing the remainder of the particles among the threads.

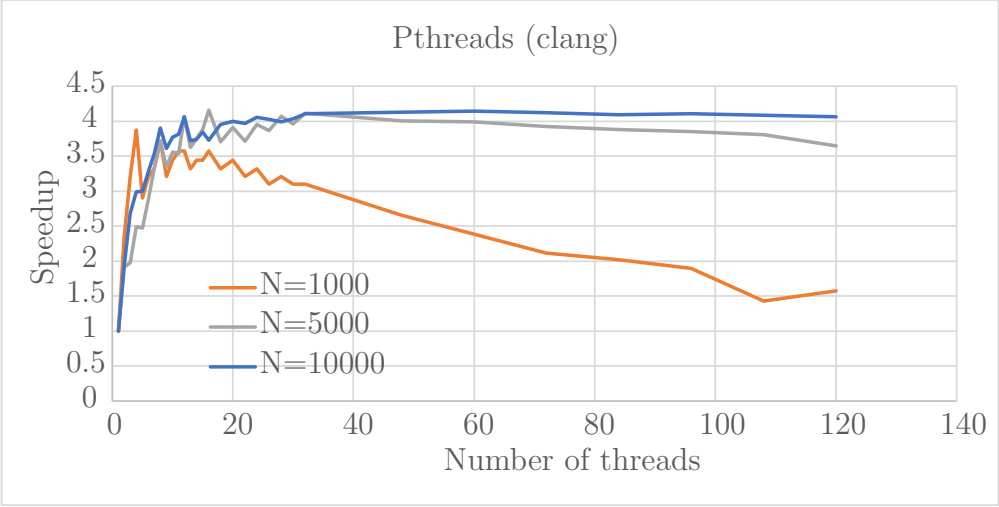

Figure 3: Plot of the speedup of the Pthreads version of the code as a function of the number of threads used. Using the Clang compiler

The clang version of the code was consistently faster, measuring in at around $0.436s$ compared

to around 0.645s for the GCC-12 compiler. This could be due to either clang performing some optimizations related to our decision to store the particles as structs, or that it is better suited for the M2 chip, since Apple is optimizing Clang specifically for their own hardware.

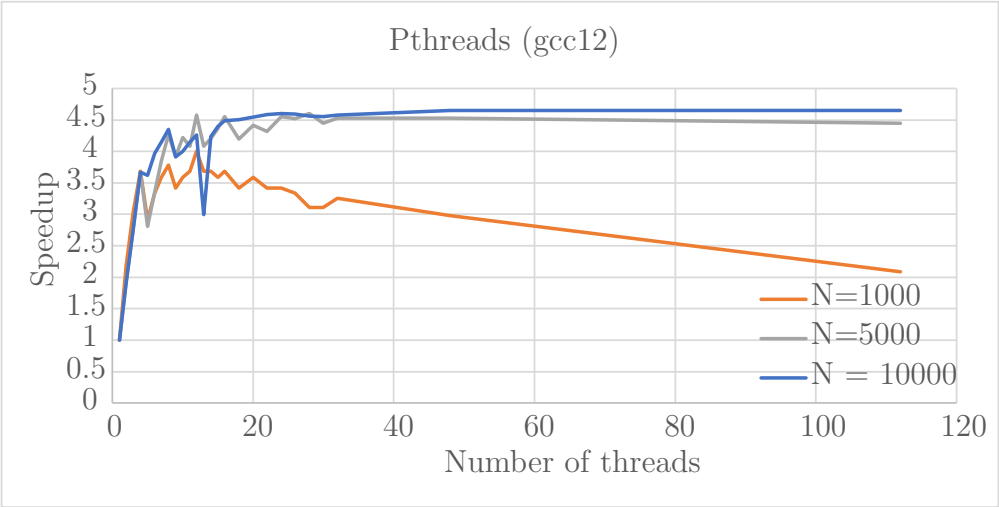

Figure 4: Plot of the speedup of the Pthreads version of the code as a function of the number of threads used. Using the GCC compiler

When looking at the speedup plots, see figure 3 and 4 we can see that the speedup flattens out after 8 threads, corresponding to the number of physical cores on the computer. The total speedup is lower than the ideal number 8, due to both the overhead in creating the threads, and the time spent waiting for synchronization between the threads. We achieved speedups of around 4 for both versions of the code. The gcc-12 version had a larger speedup, however this could easily be due to variance in the first initial run.

However we received our fastest time with more than 8 threads. Since some of the cores avaiable will always be busy with various other tasks on the computer, such as displaying the windows and my programming IDE, the number 8 might not necessarily be the fastest number of threads. What happens when we choose more threads than cores is that the operating system will divide the work between each thread, in turn effectively performing the load balancing between threads for us. We achieved our fastest time of 0.421s when using 16 threads.

For smaller problems, such as $N = 1000$ we did not see the same speedup, especially when the total number of threads was large. If the total runtime is short, the additional overhead to create the threads will become a not take more time than we gained from the parallelization.

We also did an attempt to measure how much time was spent in serial and parallel parts of the code, and plotted this compared to Ahmdal's law, see figure 2. However since the time spent in serial was relatively small, it should look similar to the ideal speedup for a embarrassingly parallel problem. One weird thing we noticed is that the speedup appears to flatten out after 4 threads instead of the expected 8. While 8 is still faster we expected a larger speedup. One possible explanation of this is due to the M2 chip having 4 performance

(p) cores and 4 efficency (e) cores. It could be possible most of the speedup is achieved when running on the p cores, and the additional e cores only contribute slightly. This could also explain why using a larger number of threads compared to cores was faster, due to the uneven performance of each core. The same could possibly explain why schedule dynamic was also faster in for the OpenMP version of the code. See section 3.2.2.

Another reason for the speedup being slower than the ideal could be due to worse cache performance compared to the serial code, due to false sharing. The D1 cache miss rate was 29.1%, compared to 1.6% for the serial code. See section 3.2.2.

3.2.2 OpenMP version discussion

The speedup achieved in the OpenMP version is in large part comparable to the Pthreads version of the code with the OpenMP version giving slightly better speedup. As can be seen in figure 5 the speedup for $N = 1000$ was considerably higher than for the rest of the runs with higher number of particles. We believe that this is due to the first run, using only one thread, was slower than normal because of it being a and thus the speedup calculated from this time when increasing the number of threads gives a false speedup increase.

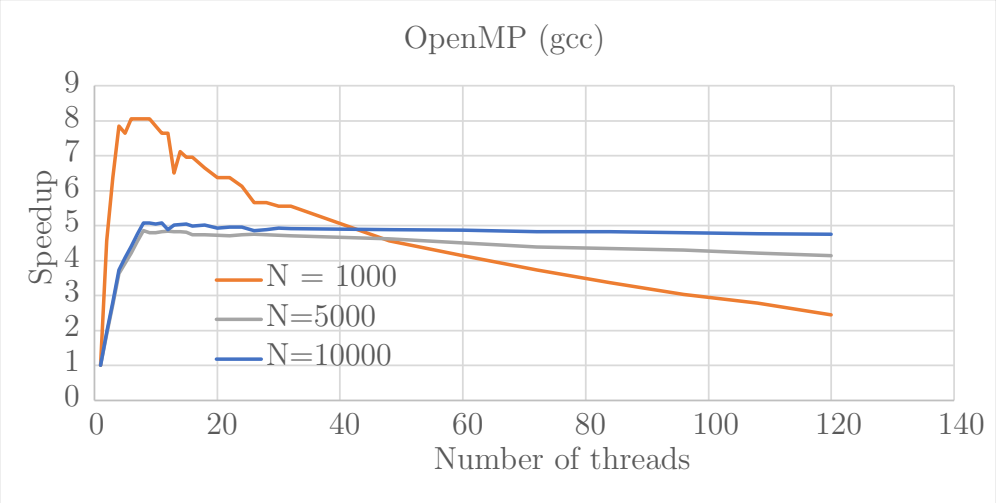

Figure 5: Plot of the speedup of the OpenMP version of the code as a function of the number of threads used. This version was compiled with gcc.

The fastest time achieved with OpenMP was $0.607s$ for $N = 5000$ when using dynamic schedule and chunk size 4 for the first loop and static schedule for the second loop. This is comparable with the GCC-12 compiled version of the Pthreads code but still slower than the Clang version. When changing the schedules and chunk size the run times did not change much but still noticeable.

Although the OpenMP version was not faster than the Pthreads version it gave a similar performance while being much easier to implement and decreases the risk for small errors which is more likely in the Pthreads version.

One reason the OpenMP version is not achieving the ideal speedup is due to additional cache misses, in a phenomena called false sharing. The D1 miss rate was 1.6% for the serial code version, while for the OpenMP version it was 25.0%. This could be due to the different threads updating the particles and therefore invalidating the entire cache line for all the other threads, so next time any of the threads tries read their data on the same cache line they will get a cache miss.