Uppsala University
Department of Information Technology

UPPSALA
UNIVERSITET

The gravitational N-body problem

Jannes van Poppelen
Csongor Horváth
Christos Pilichos

Contents

1 Introduction

Every introductory classical mechanics course touches upon the central force problem, which studies the dynamics of two bodies due to their gravitational interaction. This problem is swiftly converted into a one-body problem due to underlying symmetries, which gives rise to an analytical solution [1]. However, like its quantum mechanical analogue, the many-body problem, no general analytical solution exists to the N-body problem for arbitrary N [2]. Solutions to the N-body problem are therefore approximated using numerical methods, as is done in this work.

Central to the problem is Newton's law of gravitation, which says that for two particles of masses m_i and m_j, respectively, particle i experiences a force by particle j given by

$$\mathbf{F}_{ij} = -G\frac{m_i m_j}{r_{ij}^2}\hat{\mathbf{r}}_{ij}, \tag{1}$$

where $\mathbf{r}_{ij} = (x_i - x_j)\hat{\mathbf{x}} + (y_i - y_j)\hat{\mathbf{y}}$, x_i, y_i are the x and y coordinate of particle i, respectively, and G is the gravitational constant. For N bodies this is taken to be $100/N$. The force on particle i due to the $N-1$ other bodies then generalizes to

$$\mathbf{F}_i = -Gm_i \sum_{j=0,\, j\neq i}^{N-1} \frac{m_j}{r_{ij}^2}\hat{\mathbf{r}}_{ij}. \tag{2}$$

However, this expression is numerically unstable if $r_{ij} \ll 1$. Instead, so-called Plummer spheres are used, which introduce a small parameter ε_0 to avoid divergences caused by the denominator

$$\mathbf{F}_i = -Gm_i \sum_{j=0,\, j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\varepsilon_0)^3}\mathbf{r}_{ij}. \tag{3}$$

Finally, the dynamics are implemented through the symplectic Euler scheme, which computes accelerations, velocities, and positions according to

$$\mathbf{a}_i^n = \frac{\mathbf{F}_i^n}{m_i}, \qquad \mathbf{u}_i^{n+1} = \mathbf{u}_i^n + \Delta t \mathbf{a}_i^n, \qquad \mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t \mathbf{u}_i^{n+1}. \tag{4}$$

This algorithm is first implemented and optimized serially in C. Then, to speed up computations, the algorithm is parallelized using both pthreads and OpenMP. In order to test its performance, several N-body simulations, with various N ranging from 10 to 10000 have been done, where the computational domain is the unit square. The masses of the particles need not necessarily be all the same, as this also makes for more interesting dynamics. Even though the code itself is not very complex, various choices regarding the implementation and optimization will be elaborated on.

2 Solution

Reading and storing data from input files can be done in various ways. The simplest and most straightforward way is to introduce a `struct` of six variables of length `N` and type `struct`, as this makes reading input using `fread` trivial due to the struct being commensurate with the input data of one particle. Moreover, this proves equally useful in writing away particle data using `fwrite`. An alternative way to store the input data is to create six arrays of length N and type `double` and save one variable per particle (x-coordinate, y-coordinate, etc.) in each array. While this does not occupy more memory, it does complicate how the reading and writing of data is done. A benefit to this would be that the data of particles are stored next to each other in the memory.

Time evolution is handled according to the previously described symplectic Euler scheme. In the code, time evolution is handled using a `for` loop. Then, at each time step for each particle, the interactions with the other $N-1$ particles are computed and summed up. This is done using two `for` loops, which yields the expected $\mathcal{O}(N^2)$ complexity. Velocities of particles are updated in the same loop that computes the interactions, which is possible since gravity is a conservative force that is independent of velocities. If there was an attempt to update positions in the same loop that interactions are computed, particles will experience erroneous forces, hence updating positions is done in a separate loop after all velocities are updated. Interactions can be split into two separate loops as well, which prevents computation of the "self-interaction", i.e. gravitational interaction with itself. While this vanishes trivially and need not necessarily be dealt with, explicitly omitting it saves $N \cdot n_{\text{steps}}$ computations, where n_{steps} is the number of iterations of the simulation. For simulations with small n_{steps}, this saves minimal time, however, when n_{steps} is sufficiently big it is expected to save time. One must still ensure that simulations are not too long so that numerical instabilities take over. The double loop structure of the code also makes it easy to recognize which parts are parallelizable.

The main part of the code consists of three nested loops. The loop handling time evolution cannot be parallelized, while the loops governing the computations would benefit enormously from parallelization. In order to parallelize with Pthreads, the body of the code doing the computations is condensed into a thread function, that is supplied to each thread. Additionally, the range $(0, N-1)$ in the outer loop is split into `N_threads` evenly divided intervals, which are supplied to the threads as arguments. This way, each thread does the same amount of work, as load balancing is not a problem for this algorithm. Parallelizing using OpenMP is rather straightforward for this algorithm and requires a `pragma omp parallel for` call to divide the outer loop equally among the number of threads.

Graphics to visualize galaxies are also implemented within the code, however, an alternative route has been chosen rather than the graphics routine that was provided with the assignment. Much simpler and less time-consuming is to pipe the x- and y- coordinates of each particle straight from C to Gnuplot at each timestep [3]. In order to do this, a pipe has to be opened that prints data from

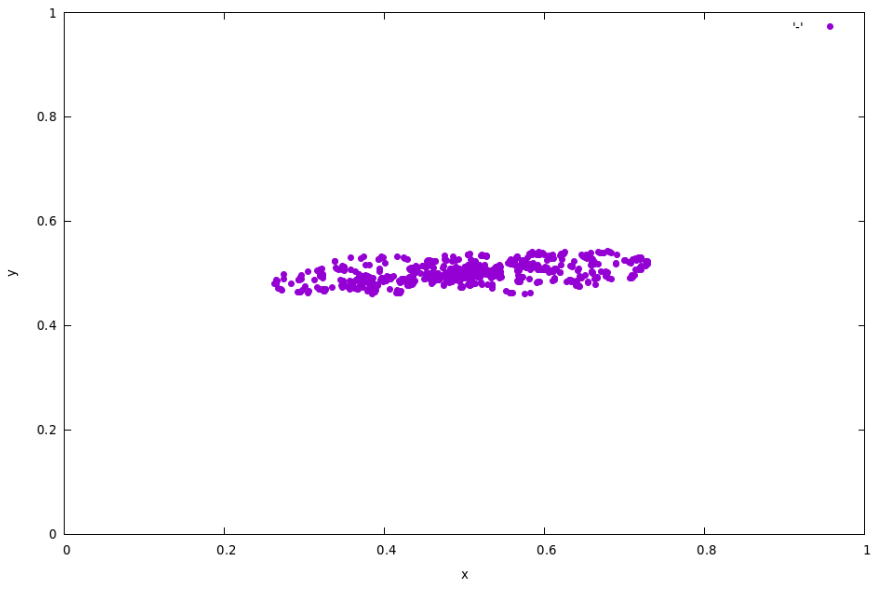


Figure 1: Snapshot of a simulation for $N = 1000$.

the simulation to a separate terminal running Gnuplot. This is done as `FILE * gnuplotPipe = popen("gnuplot -persistent","w")` in the code, where the flag `-persistent` ensures that the window created by Gnuplot does not close after every time step. An example of how such a galaxy looks is provided in figure 1. After every instance that Gnuplot updates the plot, it is important to "flush" the pipe with `fflush(gnuplotPipe)`, which ensures that the positions of all particles in the galaxy are written to Gnuplot before other operations are performed.

3 Serial performance and discussion

After confirming the functionality of the code by comparing various outputs with their corresponding reference outputs, the focus shifted towards enhancing performance by optimizing the serial code. Several methods were explored, with varying degrees of effectiveness, with some yielding significant improvements, while others had negligible impact on the execution time.

Execution times were measured using the `get_wall_seconds` function from any of the HPP labs on a machine with an AMD Ryzen 5 3500 CPU [4]. The source code was compiled with GCC 11 (gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0). Measured times exclude the reading and writing of data so that the focus of individual optimization techniques is solely on the computations done during the time evolution. For reference, simulations were done for 100 time steps using `ellipse_N_03000.gal` as the starting configuration. Time steps were 10^{-5} each. To compare the effects of individual optimization techniques, a naive and unoptimized simulation was timed, which took approximately 82.990 wall seconds. For each optimization technique, a short description and its impact on the code are presented.

Compiler optimization
Appropriate compiler flags can already significantly optimize the code without having to do any manual alterations. Optimizations technique done by the compiler includes function inlining, strength reduction, loop unrolling, and many more. Compiler flags that were considered are presented below. The effect of each compiler flag is presented in table 1.

- `-O3`: The compiler optimizes aggressively, even if might increase the size of the executable code, or take a longer time to compile.
- `-Ofast`: It follows all the `-O3` optimizations, but also leads to possible unsafe optimizations, such as violation of IEEE floating point rules by getting rid of subnormal numbers.
- `-march`: The compiler utilizes the features of the CPU architecture that is used for compiling the code. However, enabling this flag restricts the portability of the executable code.

Method	Execution time (s)	Benefits (%)
`-O3`	22.566	72.80
`-Ofast`	3.346	95.96
`-O3 -march=native`	21.509	74.08
`-Ofast -march=native`	3.075	96.29

Table 1: Effect of compiler flags on the execution time.

By using the `-Ofast` flag , the execution time was reduced by 96.29%, while the `-O3` flag benefitted only 72.80%. Moreover, using `-march=native` reduced the execution time even further. It is worth mentioning that the accuracy after each simulation is checked to ensure that accuracy is not traded off for computational gain. For the remaining optimization techniques, the flags `-Ofast -march=native` are used as they resulted in the greatest performance increase. Note that this already avoids the computation of the self-interaction, as will be elaborated further on.

Strength reduction
One way to reduce the computational load of the code is to choose operations that are computationally the most efficient. To this end, we proceeded with a variety of changes such as the `constant` declaration for the number of timesteps `nsteps`, the step size `dt`, the number of particles `N`, and the gravitational constant `G`. Furthermore, math functions like `pow` were eliminated, and floating-point multiplication was preferred over floating-point division. Differences in the code can be seen in the two code blocks below. The first block is unoptimized, while the second block contains all changes mentioned above.

```
rij= sqrt(pow((particles[i].x-particles[j].x),2) \
+pow((particles[i].y-particles[j].y),2));
C= (-G*(particles[j].m))/(pow((rij+eps0),3));
ax+= C*(particles[i].x-particles[j].x);
ay+= C*(particles[i].y-particles[j].y);
```

```
rij= sqrt((particles[i].x-particles[j].x) \
          *(particles[i].x-particles[j].x) \
          +(particles[i].y-particles[j].y ) \
          *(particles[i].y-particles[j].y));
denom= 1/((rij+0.001)*(rij+0.001)*(rij+0.001));
C= (-G*(particles[j].m)*denom);
ax+= C*(particles[i].x-particles[j].x);
ay+= C*(particles[i].y-particles[j].y);
```

The denominator of the force was also computed explicitly in order to avoid multiple arithmetic operations on the line `C` was computed. It turned out that the optimization techniques that were implemented had marginal, if any, improvements in the execution time. Since the compiler takes care of strength reduction, this is not entirely unexpected. Math functions were avoided when possible. A possible sacrifice of accuracy for a smaller execution time could be achieved by using `sqrtf` instead of `sqrt`. While this significantly improved the performance, it also reduced the accuracy by a tiny amount. Since more accurate results are preferred, usage of `sqrtf` was avoided. Since it is good practice, the `pow` function from the `math` library has been replaced by equivalent ones.

Function Inlining.
Inlining functions removes function calls by pasting the body of a function whenever it is read by the compiler. Together with the `static` keyword to restrict the function only to the same file it is called from, this can speed up code if the function gets called many times. Squaring and cubing of numbers is done many times per time step in the code, which means it might be beneficial to `inline` the `pow2` and `pow3`

functions as presented below.

```
static inline double pow2(double x){
        return x*x;
        }
static inline double pow3(double x){
        return x*x*x;
        }
```

However, once again minimal improvement in the execution time was found, with the fastest time of 3.084 seconds. Due to the rather high variance in the measured times, one should not conclude that inlining functions has a negative effect on the execution time.

Elimination of branches
Several branches can be created by including `if` statements in a code. If a branch is present in a loop, this can significantly increase the execution time. As previously mentioned, a lot of computations can be avoided if self-interactions are not computed. In the code, two different approaches were tried. A first attempt made use of two loops in computing the interactions. The first loop `for(j=0;j<i;i++){...}` computes interactions from `j=0` to `j=i-1` , whereas the second loop `for(j=i+1;j<N;j++){...}` computes interactions from `j=i+1` to `j=N`, so that effectively computations for `i=j` are avoided. For 100 time steps, the time saved is not that much, however, if the number of time steps is increased, the effect is more visible. Alternatively, both loops were merged into one loop `for(j=0;j<N;j++)`, which when followed by `if(j!=i){...}`, also entirely skips computations for `i=j`. Even though this creates a branch for which the condition has to be checked at every iteration, it proved considerably faster than the previous approach.

Loop unrolling
There was also an attempt to manually unroll loops, however, with the `-Ofast` flag, this is already handled by the compiler. Any attempts to do this manually indeed showed no real performance increases.

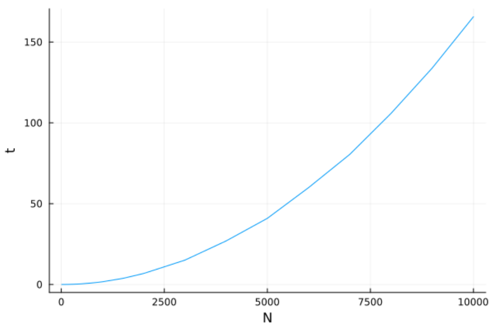
Register Keyword
The `register` keyword passes a message to the compiler that a variable is being used constantly so that it can be put in the register for fast access. Unfortunately, using this for some variables did not allow for noticeable performance increases.

Newton's Third Law
To attempt to further improve on the complexity of the algorithm, we attempted to cut the number of computations by using Newton's third law, which states that $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$. Naively using both for-loops would result in $N(N-1)$ computations, whereas implementing Newton's third law gives $\sum_{i=1}^{N}(N-i) = \frac{N}{2}(N-1)$ computations, which should cut the number of computations in half. However, this requires an additional array of length $\frac{N}{2}(N-1)$ to be defined to store the interactions for later use, which becomes increasingly expensive for large N. Moreover, this array will have to be updated for each time step, which significantly decreases

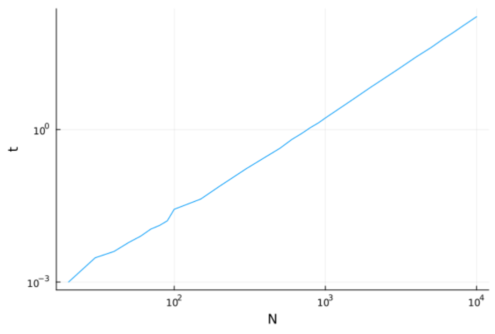
the efficiency of the cod. While being a creative effort to reduce the number of computations, Newton's third law does not end up yielding improvements.

Quadratic complexity

In order to verify the $\mathcal{O}(N^2)$ complexity of the algorithm, simulations have been done from $N = 10$ up to $N = 10000$, for $n_{steps} = 100$. Results are plotted in figure 2. While figure 2a looks like it satisfies the quadratic behaviour, the plot has been reproduced with logarithmic axes in order to verify the behaviour. Performing a linear fit gives a slope of roughly 1.934, which confirms the quadratic complexity of the algorithm. Note that a different machine was used to compute execution times than before, which should be fine since this part only concerns itself with the quadratic complexity.



(a)



(b)

Figure 2: (a), Simulation times, in seconds, as a function of the number of particles. (b), Simulation times, in seconds, as a function of the number of particles, plotted with logarithmic axes.

Conclusion

Many different manual implementations of optimization techniques have been tried, however, most of these were in vain, since the optimization flags of the compiler already handle these. The most noticeable improvement besides compiler flags was avoiding computing the self-interactions at every time step. Ultimately, the fastest execution time turned out to be 3.075 wall seconds on a machine with an AMD Ryzen 5 3500 CPU.

4 Parallel performance and discussion

Code analysis

To begin with, it is necessary to determine which parts of the code should be run in parallel. The code spends most of its simulation time in the computation of the gravitational interaction between the particles. This is handled by two `for` loops, which should then be parallelized. There are other parts of the code that could be written in parallel, such as the reading and writing of the galaxy data, but these are not worth it. Like previously mentioned, the loop handling the time evolution cannot be parallelized as it would mess up computations. The remaining two loops that can be parallelized compute gravitational interactions and update positions, respectively.

To motivate which loops are worth parallelizing, some measurements have been on a device with an Intel Core i3-10110U CPU, running on Ubuntu 22.4.02. Parallelization was handled using OpenMP. The measurements were done using $N = 1000, dt = 10^{-5}$, and $n_{steps} = 2000$, which, for comparison, gave a simulation time of 5.074 seconds using the serial code.

First, the inner loop which handles updating the particle positions was parallelized. This yielded a noticeable increase in the simulation time of 5.494 seconds. It turns out that parallelizing gave too much overhead for it to give any positive yield. The time spent in this loop has been measured explicitly, which turned out to be smaller than 10^{-30} seconds, which is negligible, implying, at least for small N, no improvement is expected from parallelizing the outer loop.

Looking at the nested/double loop, several measurements have been done for a different number of threads to see how they respond to parallelization. Considered are outer loop parallelization, inner loop parallelization, and nested parallelization where both loops are parallelized. The results are shown in table 2. For OpenMP, the performance is best for only outer loop parallelization and is what will be considered for the remainder of the assignment.

`N_threads`	2	4	16
Only outer loop	2.963	2.976	2.901
Only inner loop	3.685	3.992	>> 20
Nested parallelization	3.434	3.161	3.151

Table 2: Simulations times for the nested loop for a various number of threads.

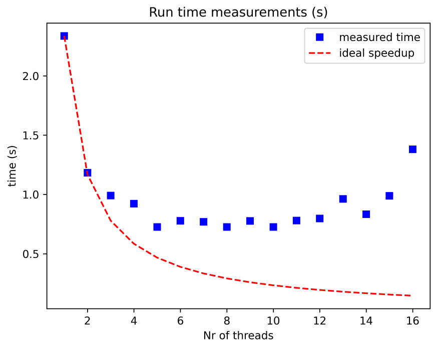
Parallelization using OpenMP

As previously discussed, we only need to parallelize the outer `for` loop. This can be done by adding only one line to the code:

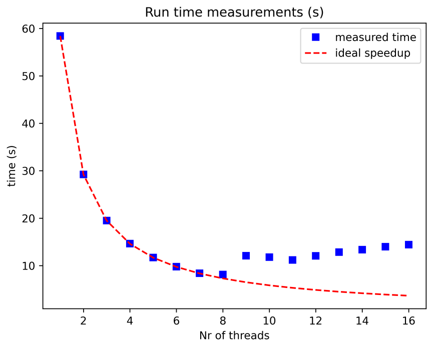
```
...
    //Time evolution
for(double t=0;t<=T;t+=dt){ //Evolve time

    #pragma omp parallel for private(i,j,rij,C) \
        shared(particles) num_threads(N_thread) //newline
    for(i=0;i<N;i++){ //For setting acceleration ...
```

For these measurements only, a variable called `N_threads` has been defined to set the number of threads. Measurements have been done on a machine provided by the university. The server has an Intel Xeon E5520 CPU, which has 8 cores, so a performance increase is expected until at least 8 threads. Results for two different measurements are given in figure 3.



(a)



(b)

Figure 3: (a), Simulation times, in seconds, as a function of the number of threads. Parallelized using OpenMP with $N = 100$ and $n_{steps} = 10000$. (b), Simulation times, in seconds, as a function of the number of threads. Parallelized using OpenMP $N = 5000$ and $n_{steps} = 100$.

As expected, simulation times become significantly less, up to the point when 8 threads are being used. Thereafter, simulation times become worse due to inefficient usage of the resources. The shortest simulation time of 7.776 seconds was reached at 8 threads on the simulation with $N = 5000$. For more than 8 threads, performance decreased most likely due to the long overhead or the shared access of particle information.

Parallelization using Pthreads

Since Pthreads is more low-level than OpenMP for parallelization, we can supply each individual thread with a function it should execute, as well as an argument. As mentioned before, load balancing is not an issue for this algorithm, so lower- and upper bounds for each `for` loop per thread are supplied by the following function, which writes these bounds into a `struct` named `data_t`.

```
    for (int i=0;i<N_threads;i++){
        data[i].LB=(i*N/N_threads+1)*(i!=0);
        data[i].UB=((i+1)*N/N_threads)*(i!=0)+ \
            N/N_threads*(i==0); }
```

The body of the serial code has been rewritten to a function that is supplied to each thread. A major difference with the serial code is the exclusion of the `if(i!=j)` statement, as was discussed in the previous report. This avoids a branch, and while it does some more computations that evaluate to zero anyways, maintaining this structure is better for the compiler. The inner loop had also been parallelized using Pthreads, however, this improved the performance only minimally, and hence has not been excluded from the final code.

```
void* thread_func(void *arg){
data_t* data=(data_t*)arg;
int LB=data->LB;
int UB=data->UB;
for(int i=LB;i<=UB;i++){
    double ax=0,ay=0;
    for(int j=0;j<N;j++){
    double rij=sqrt(pow2((particles[i].x-particles[j].x))+ \
        pow2((particles[i].y-particles[j].y)));
        double C=(-G*(particles[j].m))/(pow3((rij+eps0)));
        ax+=C*(particles[i].x-particles[j].x);
        ay+=C*(particles[i].y-particles[j].y);
        }
        particles[i].ux+=ax*dt;
        particles[i].uy+=ay*dt;}
}
```

Timing measurements have been done on the same machine as for the parallelization using OpenMP. N and n_{steps} are the same as for each OpenMP measurement. The results are presented in figure 4.

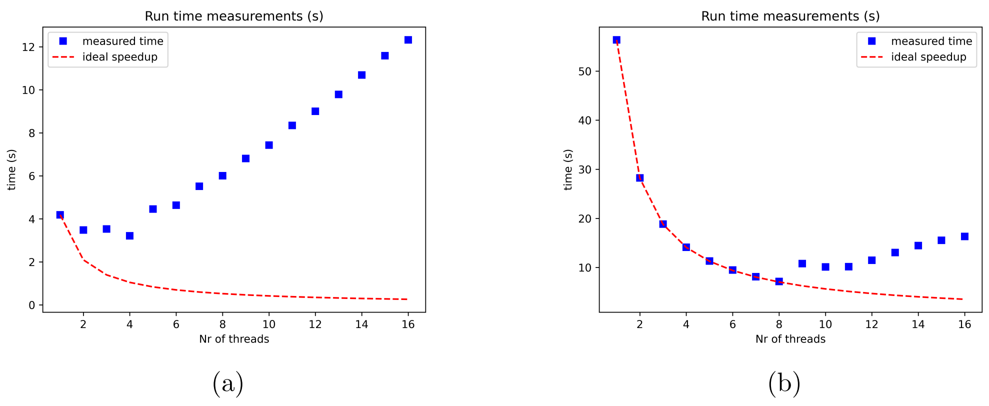
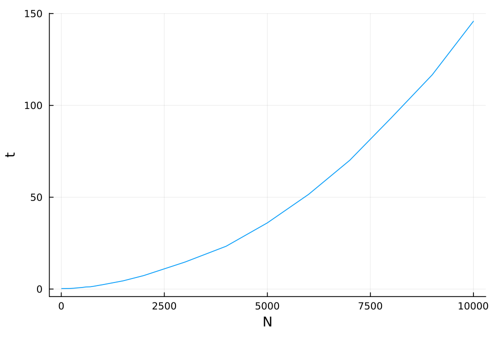


Figure 4: (a), Simulation times, in seconds, as a function of the number of threads. Parallelized using Pthreads with $N = 100$ and $n_{steps} = 10000$. (b), Simulation times, in seconds, as a function of the number of threads. Parallelized using Pthreads $N = 5000$ and $n_{steps} = 100$.

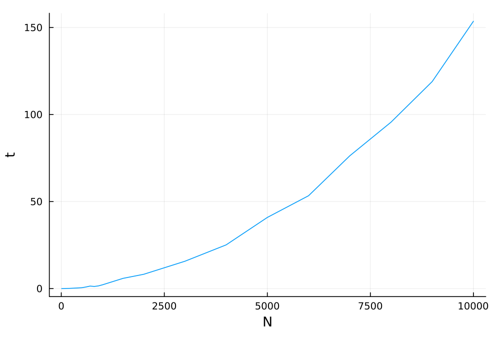
Interestingly, it turns out that measurements for $N = 100$ only slightly benefit from parallelization. A minimum simulation time is obtained for 4 threads, which only gives a slight improvement over the serial code. Apparently, if more than 4 threads are being used, there is too much overhead for the simulations to benefit optimally. For $N = 5000$, results are more aligned with expectations. Simulation times are drastically reduced up to 8 used threads, after which it again turns out that there is too much overhead for the code to benefit from using more threads.

Quadratic complexity

Quadratic complexity is once again verified by doing simulations, on the same machine as previous parallel simulations, from $N = 10$ up to $N = 10000$. However, this time $n_{steps} = 500$ to ensure that the noise in simulation times for small N is negligible. 8 Threads have been used for these simulations, as this turned out to give the optimal performance.



(a)



(b)

Figure 5: (a), Simulation times, in seconds, as a function of the number of particles. Parallelized using POSIX. (b), Simulation times, in seconds, as a function of the number of particles. Parallelized using OpenMP.

Both graphs once again appear quadratic. Performing a linear fit on a plot with logarithmic axes yields slopes of 1.019 and 1.709 for Pthreads and OpenMP, respectively. Parallelization should not change the complexity of the algorithm, which is obtained only using OpenMP. For Pthreads, it shows that the complexity is approximately $\mathcal{O}(N)$. Simulations using Pthreads take significantly longer than OpenMP for small values of N most likely due to the amount of overhead, which is not as noticeable for OpenMP. OpenMP

References

[1] H. Goldstein, C. Poole, and J. Safko, “Classical mechanics,” 2002.

[2] P. Coleman, *Introduction to many-body physics.* Cambridge University Press, 2015.

[3] T. Williams, C. Kelley, and many others, “Gnuplot 5.4.5: an interactive plotting program.” `http://gnuplot.sourceforge.net/`, March 2010.

[4] J. Rantakokko et al., “Lab assignments HPP,” *Available through high performance programming course page on Studium*, 2023.