HPP Assignment 4

Dan Friedman, Elsa Rosenblad, Sam Varahram

March 7, 2025

1 Introduction

In this assignment, a code that calculates the evolution of N particles in a gravitational simulation will be implemented in C using the symplectic Euler time integration model. First, a serial program will be written and optimized, after which parallelization will be introduced using both Pthreads and OpenMP.

2 Implementation

Each particle variable is stored in an array. The simulation itself consists of two loops implementing the governing equations: one computes the forces, and the other updates the particles using the time integration model. Both functions are executed at each time step.

2.1 Computing Gravitational Forces

The function `compute_forces` calculates the gravitational forces between every pair of particles. The force $\mathbf{F}_i$ exerted by particle i on particle the other $N-1$ particles is given by:

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

where:

- G is the gravitational constant,
- m_i and m_j are the masses of particles i and j,
- $\mathbf{r}_{ij} = \mathbf{p}_j - \mathbf{p}_i$ is the displacement vector,
- $r_{ij} = \|\mathbf{r}_{ij}\|$ is the distance between the particles.

Algorithm **??** details the implementation.

Algorithm 1 Compute Forces

0: **procedure** COMPUTE_FORCES(pos_x, pos_y, $mass$, $force\_x$, $force\_y$, $num\_particles$, G)
0: **for** $i \leftarrow 0$ to $num\_particles - 1$ **do**
0: $force\_x[i] \leftarrow 0.0$
0: $force\_y[i] \leftarrow 0.0$
0: **end for**
0: **for** $i \leftarrow 0$ to $num\_particles - 1$ **do**
0: $pos\_xi \leftarrow pos\_x[i]$
0: $pos\_yi \leftarrow pos\_y[i]$
0: $mass\_i \leftarrow mass[i]$
0: $total\_force\_x \leftarrow 0.0$
0: $total\_force\_y \leftarrow 0.0$
0: **for** $j \leftarrow i + 1$ to $num\_particles - 1$ **do**
0: $dx \leftarrow pos\_x[j] - pos\_xi$
0: $dy \leftarrow pos\_y[j] - pos\_yi$
0: $r \leftarrow \sqrt{dx * dx + dy * dy} + EPSILON$
0: $r3\_inv \leftarrow 1.0/(r * r * r)$
0: $force\_magnitude \leftarrow G * mass\_i * mass[j] * r3\_inv$
0: $force\_xij \leftarrow force\_magnitude * dx$
0: $force\_yij \leftarrow force\_magnitude * dy$
0: $total\_force\_x \leftarrow total\_force\_x + force\_xij$
0: $total\_force\_y \leftarrow total\_force\_y + force\_yij$
0: $force\_x[j] \leftarrow force\_x[j] - force\_xij$
0: $force\_y[j] \leftarrow force\_y[j] - force\_yij$
0: **end for**
0: $force\_x[i] \leftarrow force\_x[i] + total\_force\_x$
0: $force\_y[i] \leftarrow force\_y[i] + total\_force\_y$
0: **end for**
0: **end procedure**=0

2.2 Updating Particle Positions and Velocities

The function `update_particles` updates the velocity and position of each particle according to:

$$\mathbf{a}_i^n = \frac{\mathbf{F}_i^n}{m_i} \tag{2}$$

$$\mathbf{u}_i^{n+1} = \mathbf{u}_i^n + \Delta t \mathbf{a}_i^n, \tag{3}$$

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t \mathbf{u}_i^{n+1}. \tag{4}$$

The implementation is shown below.

Algorithm 2 Update Particles' Positions and Velocities

Require: $pos_x, pos_y, vel_x, vel_y, mass, force_x, force_y, num\_particles$
Ensure: Updated $pos_x, pos_y, vel_x, vel_y$
0: **for** $i = 0$ to $num\_particles - 1$ **do**
0: $ax \leftarrow force_x[i]/mass[i]$
0: $ay \leftarrow force_y[i]/mass[i]$
0: $vel_x[i] \leftarrow vel_x[i] + \Delta T \cdot ax$
0: $vel_y[i] \leftarrow vel_y[i] + \Delta T \cdot ay$
0: $pos_x[i] \leftarrow pos_x[i] + \Delta T \cdot vel_x[i]$
0: $pos_y[i] \leftarrow pos_y[i] + \Delta T \cdot vel_y[i]$
0: **end for**=0

3 Performance and Discussion

3.1 Serial program

For all run times below, O3 optimization has been used with gcc compiler version 14.0.3 on an Apple M2 processor in a 2022 Macbook Pro. The operating system is macOS Ventura 13.4. The recorded times in the tables below are wall times for the simulation itself, i.e. the two functions described in the previous section. The time for reading the input and writing the results is not measured since it takes a negligible amount of time in comparison to the simulation. Running Operformance analyser to check where performance is spent shows that `compute_forces` consumes 90 percent of the performance, and that the line that computes $1/r^3$ consumes 36 percent of the performance alone. Running gprof shows that 99 percent of the running time of the program is spent inside the `compute_forces` function. Optimization efforts should clearly be focused there.

One major design choice in this program is choosing between an Array of Structures or Structure of Arrays, i.e. storing the particles in a struct containing the variables or in six individual arrays. When writing this program, for the sake of simplicity in the mind of the writer, a struct was used. Running times

Function	% Time	Cumulative	Self	Calls	Self ms/call	Total ms/call
compute_forces	98.98	32.11	32.11	100	321.10	321.10
_init	0.96	32.42	0.31	-	-	-
update_particles	0.06	32.44	0.02	100	0.20	0.20
get_wall_seconds	0.00	32.44	0.00	2	0.00	0.00
read_input	0.00	32.44	0.00	1	0.00	0.00
write_output	0.00	32.44	0.00	1	0.00	0.00

Table 1: Gprof output when simulating 3000 particles for 100 time steps.

Number of particles	Time steps	Wall time (seconds)
100	200	0.0034
500	200	0.0872
1000	200	0.2784
2000	200	1.0460
5000	200	6.5733
7000	200	12.8505
10000	200	26.7166

Table 2: Simulation time as a function of the number of particles for 200 time steps for the AoS implementation. All times above are the lowest recorded time of five attempts.

with the AoS implementation are shown in table 2. As a possible performance improvement, the code was rewritten to use SoA instead. This should improve performance, mainly due to spatial locality benefits. For example in the function that computes the forces, only the mass and position of each particle is used, meaning that the velocity and brightness does not need to be loaded into the cache as we would have done with an AoS. It should also be more amendable to auto vectorization. Table 3 shows the running times with this implementation. It appears that SoA runs considerably faster than AoS. The big difference is due to that the loops are now vectorized by the compiler.

Number of particles	Time steps	Wall time (seconds)
100	200	0.0019
500	200	0.0353
1000	200	0.1059
2000	200	0.3582
5000	200	2.4530
7000	200	4.5568
10000	200	9.4077

Table 3: Simulation time as a function of the number of particles for 200 time steps for the SoA implementation. All times above are the lowest recorded time of five attempts.

Another area where optimizations can be made is of course inside the al-

gorithms themselves: making sure not to call unnecessary functions such as `sqrt()` or `pwr()`, not dividing in a loop, and using the simplest arithmetic possible[2]. Further it is generally not recommended to declare global variables in serial programs[1, p. 26]. In this application there are however only two doubles (time delta and epsilon) that can reasonably be declared globally, so it will not matter in terms of running time. Lastly, an attempt was made at using `restrict` for the arguments were applicable in the `compute_forces` function, however this slowed down the program by roughly ten percent.

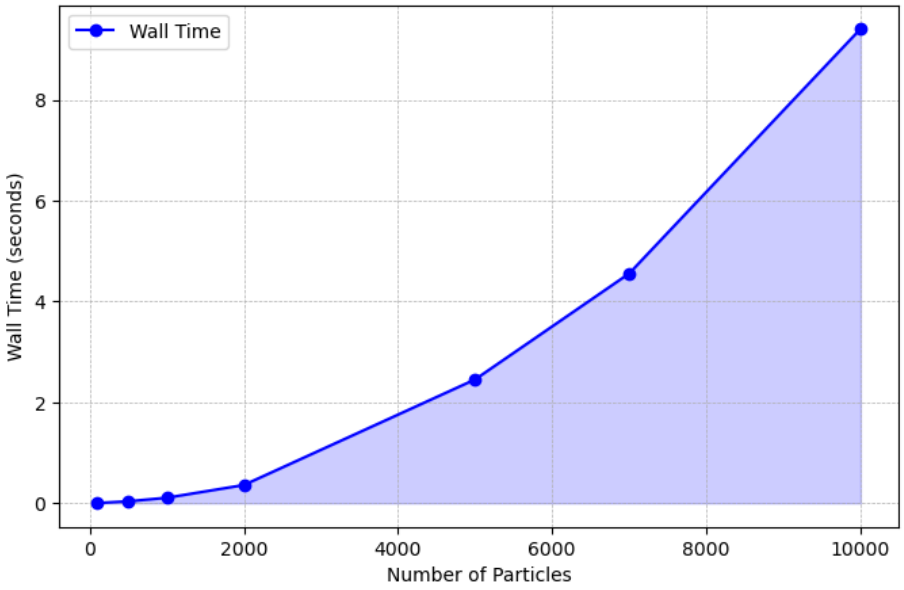

Figure 1: Wall time as a function of the number of particles. This figure is based on the data in table 2. It is clear that the time complexity is $O(N^2)$.

3.2 Parallel program

As seen in table 1, nearly 99 percent of the program's runtime is spent in the `compute forces` function. It is here we will even spend enough time to gain any performance from parallelization. For the other functions we will likely lose performance due to overhead costs.

3.2.1 Pthreads

To introduce parallelization to algorithm 1, the outer loop (over i) will be divided into chunks based on the selected number of threads. However, to pass the particle data to the different threads, it is by the design of Pthreads necessary to use a pointer to a struct. Thus we need to return to AoS instead of SoA that

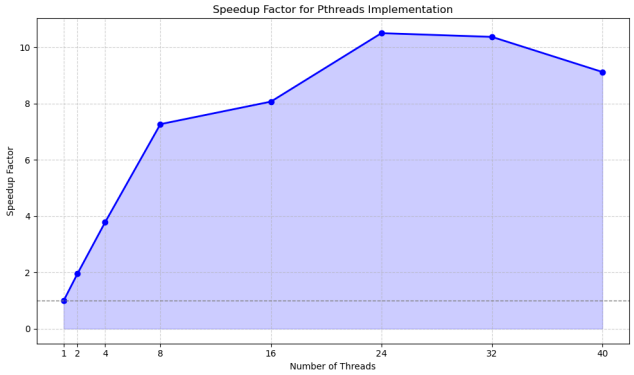

Figure 2: Speedup as a function of number of cores for the Pthreads implementation.

was introduced as a serial optimization. The big challenge here is to maintain the use of Newton's third law in the compute forces function. All threads will need to update the force arrays for i and j. One way is to use mutexes to lock them until the respective loops have finished to avoid a race condition. As discussed later however, this approach is a major bottleneck. Also, when initializing the force vectors, we must wait (using a barrier) until all threads have initialized their vectors to zero before continuing.

To achieve this, the compute forces function can essentially remain the same except for the introduction of mutexes for the force arrays. A new function is needed to partition the outer loop into chunks depending on the number of threads selected by the user. A naive approach is to partition as chunk = num_particles/num_threads. Each thread created will call the compute forces function for its data chunk. Using this partitioning however creates a major load imbalance. To fix the two major issues - load balancing and use of mutexes - we need to give each thread a local force array, and use "round robin" scheduling to fix the load balancing. Once each thread's local array is filled, they they are added to a global force array afterwards. Figure 2 shows the performance with these concepts introduced.

3.2.2 OpenMP

For the OpenMP implementation the main pitfall is still how to handle the race condition for the force arrays. We do not need to manually implement round robin scheduling however, since dynamic scheduling in OpenMP can be used instead for the same effect. There is also no need for a global force array since we can use OpenMP's `reduction` clause. The performance of the OpenMP implementation is shown in table 3.

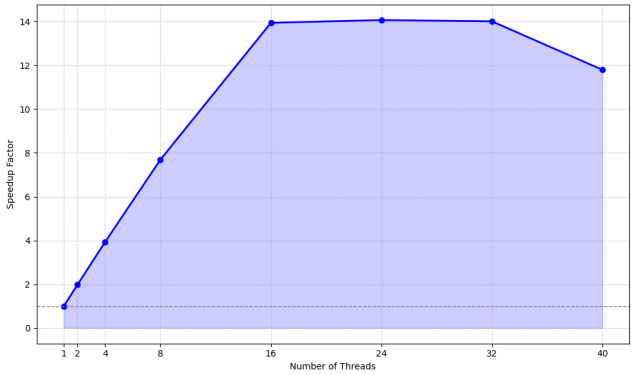

Figure 3: Speedup as a function of number of cores for the OpenMP implementation.

3.2.3 Performance Issues

From figures 2 and 3 we notice that there is essentially no speedup after 16 cores. We believe that there are two main reasons for this. The first is that the part of the program that handles the global variable (the global force array) forces all threads to wait. This is a serial bottleneck that reduces the speedup when the number of cores increases. The second potential reason is NUMA effects due to the dual sockets on the University's Linux servers. However, the speedup prior to that is close to linear as a function of the number of cores which means that the parallelization is actually working quite efficiently.

References

[1] Agner Fog. *Optimizing software in C++ An optimization guide for Windows, Linux, and Mac platforms.* Technical University of Denmark, 2024.

[2] Rantakokko Jarmo. Lecture notes. 2025.