HPP Group Assignment

Olof Björck, Gunnlaugur Geirsson

February 13, 2019

Assignment 3

The problem

The evolution of a number of particles in space can be simulated using Newtonian mechanics. This can be implemented numerically using Plummer spheres and symplectic Euler time integration. A straightforward implementation requires $O(N^2)$ computations for N particles. In this report, we implement a straightforward simulation of the evolution of a number of particles in space in two dimensions. The initial state of the particles will be read from an input file and after the simulation, the updated state of the particles will be written to an output file.

The solution

Our solution algorithm is rather simple and can be divided in the following steps: Read input data from the command line; Read particles input data from input file; Allocate particles in memory; Simulate the particles' movements; Write particles updated data to output file; and finally Free particles.

The simulation is done by updating the state of the particles by a timestep several times, with the timestep and the number of timesteps to simulate given as input. The input variables are set as constants and every particle is represented by a struct containing the state of the particle.

Representing a particle in C using a struct data type is logical and straightforward to implement. This particle struct can contain all necessary information about a particle needed for the simulation such as coordinates, velocity, and mass. There are two distinct ways to allocate all particles; either as an array of particle structs or as a single particles struct composed of arrays. We implemented the former, an array of particle structs, as we deemed this to be faster than a particles struct with arrays due to cache optimization. We will need several pieces of information about a particle during runtime and an array of particle structs will result in each particle's information being allocated nearby in memory. This array of particles can then be passed to functions with the "__restrict" keyword to enable possible optimizations.

During implementation, it became apparent that calculating the forces between particles was redundant. Instead of calculating forces, we calculate the resulting accelerations of the particles to avoid first multiplying by mass and then dividing by mass later. Furthermore, the brightness information about a particle is not used in any calculation. We chose to store brightness in a separate array used only for writing to output file, keeping the particle struct as trim as possible. The resulting particle struct we chose to implement thus contains a particle's position, velocity, acceleration, and mass, which are all needed in the simulation calculations:

```
1 struct particle {
2     double x, v_x   // x position and velocity
3     double y, v_y   // y position and velocity
4     double a_x, a_y // x and y acceleration
5     double mass     // particle mass
6 }
```

When calculating the force and thus the acceleration for a particle, we chose to make use of Newton's third law. Because the force between two particles acts equally on both particles with inverted signs, we can calculate the acceleration of two particles in every particle pair iteration. That is, instead of using two nested for-loops that loop over every particle as:

```
for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
        // Calculate acceleration of particle i from the
        // interaction with particle j
    }
    // Update position of particle i
}
```

we can loop over every particle pair:

```
for(int i = 0; i < N; i++) {
    for(int j = i+1; j < N; j++) {
        // Calculate acceleration of particle i and j from the
        // interaction between them
    }
    // Update position of particle i
}
```

This approximately halves the work necessary to update the particles' accelerations. Moreover, this implementation avoids calculating the nill-force when i==j without using a conditional, splitting the loop, or performing unnecessary math. However, a separate loop is needed to initialize the acceleration to 0. Instead of looping over all particles and setting their accelerations to 0 before doing any calculations, we chose to implement a loop for the particular first case i == 0 when the acceleration is not initialized. After that, we can safely assume the acceleration is initialized for each particle. Our final calculation for-loops thus become:

```
// Set acceleration of particle 0 to 0
for(int j = 1; j < N; j++) {
        // Set acceleration of particle j to 0
        // calculate acceleration of particle 0 and j from the
        // interaction between them
}
// Update position of particle 0

for(int i = 1; i < N; i++) {
    for(int j = i+1; j < N; j++) {
        // Calculate acceleration of particle i and j from the
        // interaction between them
    }
    // Update position of particle i
}
```

Performance and discussion

The program was compiled and run on an Intel Core i5-5257U CPU @ 2.70GHz processor in macOS Mojave. The compiler used was Apple LLVM version 10.0.0 (clang-1000.11.45.5), with optimization flags `-O3` and `-march=native`.

The best time found for 3000 particles (from input file ellipse_N_03000.gal) with 100 timesteps and step size 0.00001 was 2.183 seconds, timed with the build-in timing command `time` in the macOS terminal (bash). All timing results presented are the real-time runtimes of the program.

The algorithm execution time is $O(N^2)$ for input size N, as can be seen in figure 1. The reason for the seemingly poor performance when $N < 100$ is likely due to the timing function being inaccurate for such small intervals; the runtime of the program is dominated by system calls, not actual algorithm work.

There are some optimizations which were attempted but discarded as they either offered no improvement or increased the execution time of the program. At first we believed that blocking the double for-loop used to update the particles would improve data locality and thus allow the program to utilize the cache better. This however, did not improve the execution speed for any block size tested and only added more iterator overhead. This is likely due to the input size being small enough to fit the entire problem in cache memory. For $N = 3000$ particles the input size in bytes is

$$(nr\ of\ doubles\ in\ struct) \cdot sizeof(double) \cdot N = 7 \cdot 8 \cdot 3000 = 168Kb,$$

excluding the constants and assuming that there is no padding between each particle struct. For reference, the "smart cache" of an Intel Core i5-5257U is 3Mb.

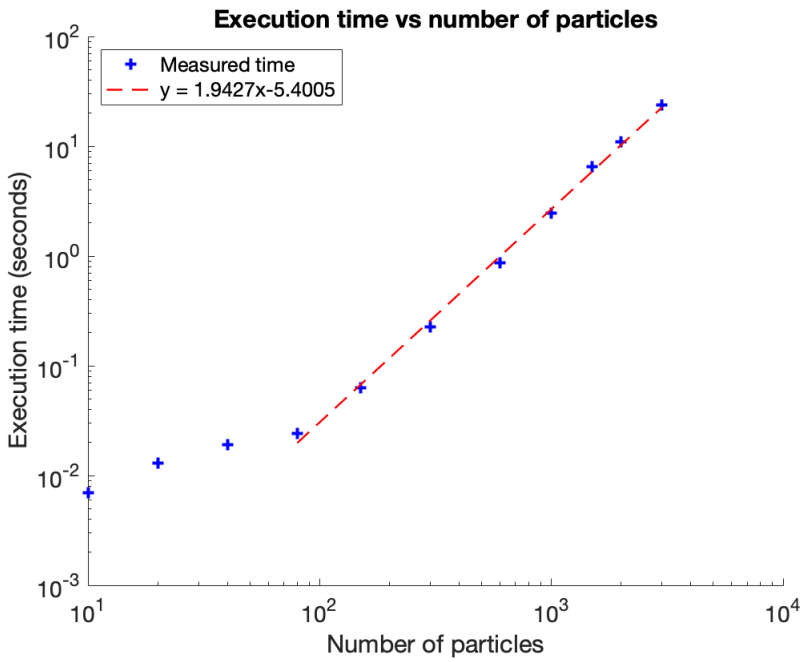

Figure 1: The execution time plotted against number of particles included in the simulation. The $O(N^2)$ complexity is shown by the straight line with slope approximately 2.

Another failed attempted optimization was to split the particle struct into separate arrays (using a structure of array instead of an array of structures), to be able to declare the mass array as constant using the keyword const. This greatly reduces the locality of the data. For example, the values stored for particle 1 and particle 2 are split into different arrays, so while the x coordinates for particles 1 and 2 are all close together, they are far away in memory from the y coordinates, velocities, etc. Defining the data structures in this manner proved less efficient than storing the values of each particle close together in memory. Only moving the mass out of the particle struct to declare it as const also proved slower than including it normally. We also attempted to move only the acceleration out of the particle structs into a separate array, so as to be able to use the memset function to zero the acceleration before each iteration. Even this proved less efficient than simply including it in the struct and looping through each particle. Lastly, we tested compiling with the `-ffast-math` flag. While this still gave the exact correct solution for all test cases compared to the provided answers, it offered no performance increase. Strangely enough it actually seemed to slightly decrease performance.