Assignment 3, Philipp Noel von Bachmann

February 15, 2022

1 Problem

In the N-body Problem, we want to predict the movement of n bodies that interact with each other over time. The interaction is given by the force they excert on each other, which in turn changes the acceleration of each body, thus the velocity and position. The problem is modeled by assuming pairwise interaction between the bodies and taking their sum as the total force excerted onto each body.

2 Solution

2.1 Data structure

In general, we have the following 4 characteristics of each body:

- velocity
- position
- mass
- brightness

Mass and brightness are represented as double, since they are both continuous. As position and velocity are both vectors, they are represented as a struct of coordinates. Since we model the problem in 2D, each vector thus comprises of an x and y coordinate. The characteristics are collected in a struct called body. This format also allows for easy reading of the input gal file, since we can directly read each block into a body struct.

In later parts 3.6.1, we also keep track of the accelerations of each body in an array of coordinates. We opt not to integrate this into the body struct to be able to retain easy writing and reading of files.

2.2 Structure

The baseline code consists of the following modules/functions:

- utility functions for coordinates: compute direction and compute norm
- compute force: computes the force that acts onto a body
- compute body update: computes the update step for each body

- step: handles the update step, calls the subroutines. Only this function actually changes the state of our bodys.

In the final code, we restructured it to the following, see mostly section 3.6.1 for details.

- utility functions for coordinates: compute direction and compute norm
- compute acceleration: computes the acceleration that acts onto a body
- update body: computes the update step for each body and applies it
- step: calls the subroutines and iterates over all bodys

3 Performance and Discussion

3.1 Hardware information

- CPU: Apple M1 with 4 performance cores and 4 efficiency cores
- GPU: Apple M1 8 core.
- Compiler: Slang, Version 13.0.0

3.2 Performance Measurement script

The performance was measured with a simple shell script to enable reproducability. In the script we vary the number of bodies. In the end we also check that the code is correct with the **compare_gals** file. Our time measurement was done within the code with the help of the time **timeoftheday** function. We only started the time measurement immediatly before calling the step loop and stopped the measurement directly after the loop.

3.3 Optimization and Results

3.3.1 Baseline

steps	10	100	1000
time (s)	0.007	0.065	0.648

bodies	10	100	500	1000
time (s)	0.003	0.065	1.615	6.502

Here, we see that our computation time scales linearly with the number of steps, which we would expect since steps are in an outer loop. In comparison, we see that the number of bodies doesn't scale linearly with time. This makes sense since our naive algorithm has complexity $O(n^2)$.

3.4 Using optimizer flags

- -O2

bodies	10	100	500	1000
time (s)	0.000	0.012	0.291	1.164

- -O3

bodies	10	100	500	1000
time (s)	0.001	0.012	0.290	1.163

- -03 -ffast-math

bodies	10	100	500	1000
time (s)	0.000	0.002	0.037	0.146

We can see that by enabling optimization flags, we can improve the performace of our code significantly. If we test our problem on ellipse_N_00100_after200steps.gal, even enabling ffast-math does not change the correctness of the result. Since we might loose precision in general however, we opt to only use -O3 in all further optimizations.

3.5 Optimizing Instructions

We can make the following small optimizations:

- Remove division by m_i for a_i by also removing it in F_i, however this doesn't change the computation time.
- Give inline hint to our vector utility function like norm, however no improvement since compiler does it probably by itself.

We can reduce the time of computing r^3 by explicitly writing it out instead of using the **pow** function. This leads to a big improvement:

bodies	10	100	500	1000
time (s)	0.000	0.002	0.038	0.151

3.6 Memory optimization

3.6.1 Reducing memory usage

Instead of creating a new array of bodies for each timestep, we can also update our bodies in place. This however means that we need to restructure our code such that we first compute all accelerations, and then updating them. Surprisingly this doesn't lead to a big performance improvement, our time reduced to 0.149 for 1000 bodys which is just 2 ms. One of the reasons could be that while we create fever objects, we need to split our update loop in two on order to first compute all accelerations and then compute all body updates.

3.6.2 Optimizing cache usage

We can see in the formula for force

$$F_i = -Gm_i \sum_{j \neq i} \frac{m_j}{r_{ij}^2} \hat{r}_i j \tag{1}$$

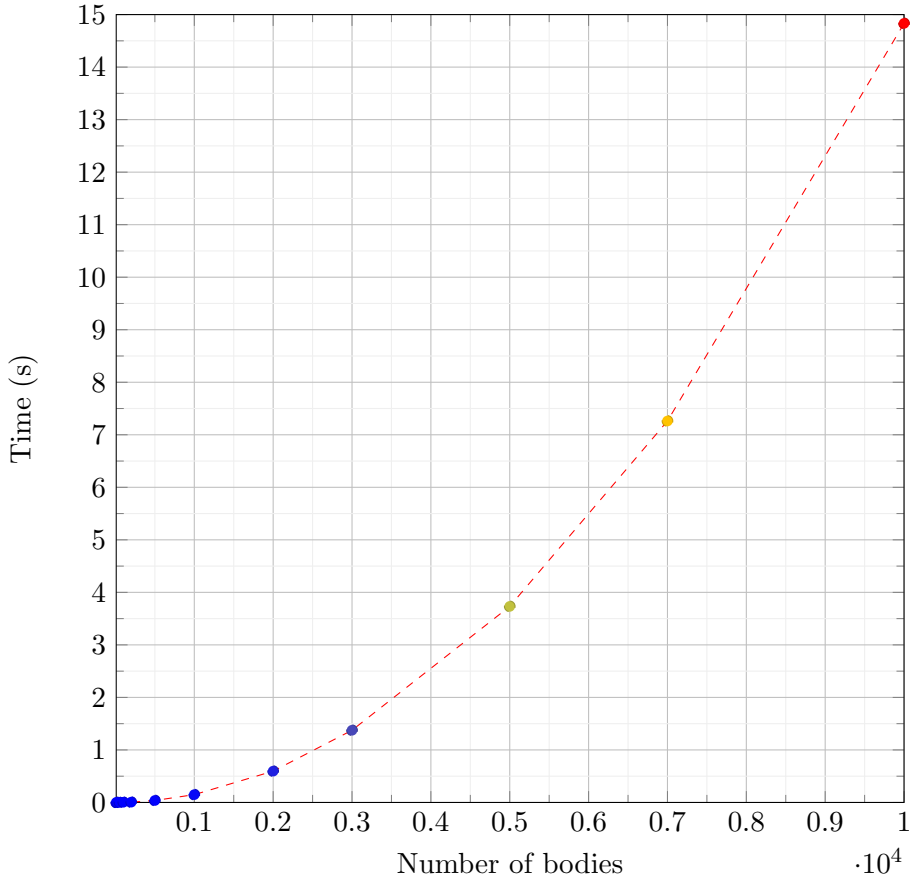
that we our sum goes over all j. If we have a very big number of bodies, the first body in the cache will be replaced by the time we come to the end of the sum. Thus if we want to acces the first body in the next iteration again, we need to re-read this line. Instead, we can also compute the sum block-wise over all js. However this reformatting doesn't lead to an improvement, instead it slows down for smaller block sizes. One reason could be that by adding blocks we added more for loops which make it harder for the compiler to optimize the code.

3.7 Instruction Level Parallelism

We can get rid of the if $i \neq j$ statement in the F_i 1 computation loop by dividing the loop up. This leads to a minor improvement of 0.006s for 1000 bodies.

3.8 Final code time measurement

In the end, we ended up with a code that included all optimizations except of 3.6.2. This lead to the following performance:



We can clearly see that this curve resembles an polynomial increasing cost, $O(n^2)$.

4 References

- For the timing, the **get_wall_seconds** from Lab06, exercise 7
- For data reading, the function **read_doubles_from_file** from **compare_gal_files**, but slightly modified