Assignment 3

Bernat Sopena Gilboy & Xabier García Andrade

UPPSALA UNIVERSITET

10th of February, 2019

Contents

1 The Problem **3**

2 Solution **3**

3 Performance Discussion **4**

3.1 Performance measurement 4

3.2 Original Algorithm: discussion, timing and complexity 5

3.3 Serial optimizations 9

3.3.1 Algorism Improvements 9

3.3.2 Switching data types to const 10

3.3.3 Loop fusion 10

3.3.4 Loop unrolling 11

3.3.5 restrict keyword 11

3.3.6 Next steps 11

3.4 Optimized version: Complexity 11

4 Apendix A: lscpu result **12**

1 The Problem

The N-body problem has a long history since Lagrange (1700) that gave analytical solutions for $N =$, and has been of utmost importance in cosmology and other areas of physics. We aim to solve the newtonian equations of movement of system not subjected to external forces of N particles with given initial position and velocities. To this end we use the *Euler Symplectic Method*

$$u_i^{n+1} = u_i^n + \Delta t a_i^n \tag{1}$$
$$x_i^{n+1} = x_i^n + \Delta t u_i^{n+1} \tag{2}$$

Where $a_i^n u_i^n$ x_i^n denote the acceleration, velocity and position of particle i at time step n.

2 Solution

Here goes algorithm discussion and description. (Describe the data structures, the structure of your codes, and how the algorithms are implemented.

Are there other options, and why did you not use them?)

i.e all we did before optimization

We read the initial positions, mass, initial velocities and brightness and store them in a $N \times 6$ array. This array is allocated dynamically.

We follow the algorithm proposed in the assignment statement. The force upon particle i is given by:

$$F_i = -Gm_i \sum_i \frac{m_j}{r_{ij}^2} \hat{r}_{ij} \tag{3}$$

For each time step, we define $R_{ij} = \frac{1}{r_{ij}^2}\hat{r}_{ij}$. Note that $(R_{ij})_{ij}$ is an anti-symmetric matrix, hence we only need to calculate and store its inferior triangle. Then we can recover the full matrix by $R_{ij} = -R_{ji}$ when $j > i$.

This values are then multiplied by the corresponding masses and summed to get the acceleration over particle i, by doing this instead of working directly with the force matrix $(F_{ij})_{ij}$ we avoid doing a float multiplication in the innermost loop and a float division in the next inmediate outerloop. The downside is that our code becomes less intuitive. The effect of this improvements in the algorithm is discussed in the sequel.

Once this matrix is known we can apply Euler simplectic method as explained above.

Our code is composed of only the `main()` function.

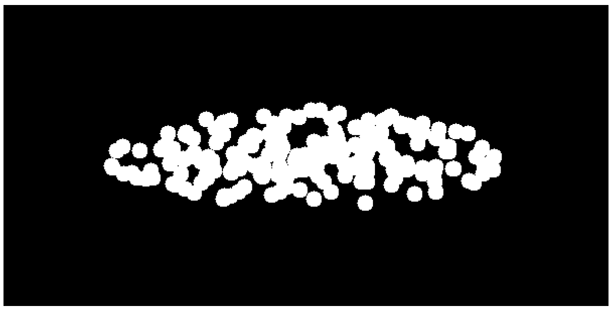

Figure 1: Simulation with $N = 300$

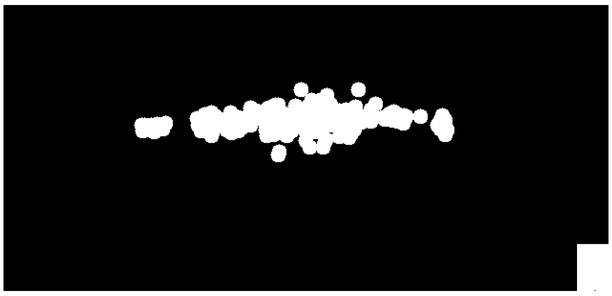

Figure 2: Simulation with $N = 300$

3 Performance Discussion

3.1 Performance measurement

We may mention that in order to measure the performance of every modification, we executed the script 100 times and calculated the mean and the standard deviation of the executiuon times. Then, each time is expressed as the mean value and a confidence interval spanned by its standard deviation up to two significant figures (so that it is consistent with common error theory conventions).

We now proceed to discuss how the error of the physical quantities that we measured (time) was treated. The obtained execution time measurements are used to calculate other values, namely speedup, whose error has to be propagated according to the following formula:

$$\sigma_{(y)} = \sqrt{\sum_{i}^{n} \left[\left(\frac{\partial y}{\partial x_i} \right)^2 \cdot \sigma_i^2 \right]} \tag{4}$$

Where σ_i stands for the error of the ith variable. In all of our cases, it would be equal to the standard deviation of the time samples.

The binary file that was used as input file to test our programmes was **ellipse_N_03000.gal**, which was as well compared with its corresponding reference output **ellipse_N_03000_after100step** It is worth mentioning as well that all timings are shown after compiling with the optimization flags gcc -O3 -march=native -ffast-math. In all cases, different optimization flags were tried, but always ended up getting the optimal results with the aforementioned. The final output error did not change significantly in any of the cases.

In order to measure the speedup, we will define a quantity S as:

$$S = \frac{T_{original}}{T_{optimized}} \tag{5}$$

Using (4) we can propagate the uncertainty in the quotient to be sure that the speedup is not due to statistical fluctuations:

$$\sigma_{(S)} = \sqrt{\left(\frac{-T_{original}}{T^2_{optimized}}\right)^2 \sigma^2_{(optimized)} + \left(\frac{1}{T_{optimized}}\right)^2 \sigma^2_{(original)}} \tag{6}$$

3.2 Original Algorithm: discussion, timing and complexity

We first discuss the efficiency and the performance of the straight-forward implementation. The following snippet shows the algorithm:

```
for (int k = 0 ; k<n_steps ; k++){

    for (int i = 0; i<N ; i++){
      //calculates new positions in just one step
      double sum_x = 0;
      double sum_y = 0;
      double force_x = 0;
      double force_y = 0;
      double acceleration_x,acceleration_y,new_velocity_x,new_velocity_y ,
   new_position_x , new_position_y;
      for (int j = 0; j<(N) ; j++){
        if (j != i){
          /* first we calculate the coordinates with respect
          to the initial frame of reference, then
          the denominator and finally multiply the result
          by the mass of the particle and the distance vector.
          */
          double x_direction = arr[i][0] - arr[j][0];
          double y_direction = arr[i][1] - arr[j][1];
          double denominator = pow((sqrt((x_direction)*(x_direction) + (
   y_direction)*(y_direction))+epsilon_0),3);
          sum_x += arr[j][2]*x_direction/denominator;
          sum_y += arr[j][2]*y_direction/denominator;
        }

      }
      //Calculate the force on each dimension
      force_x = -G*arr[i][2]*sum_x;
      force_y = -G*arr[i][2]*sum_y;

      //Apply euler method
      acceleration_x = force_x/arr[i][2];
      acceleration_y = force_y/arr[i][2];
      new_velocity_x = arr[i][3] + delta_t*acceleration_x;
      new_velocity_y = arr[i][4] + delta_t*acceleration_y;
      new_position_x = arr[i][0] + delta_t*new_velocity_x;
      new_position_y = arr[i][1] + delta_t*new_velocity_y;

      //Store updated values for particle i
      new_arr[i][0] = new_position_x;
      new_arr[i][1] = new_position_y;
      new_arr[i][2] = new_velocity_x;
      new_arr[i][3] = new_velocity_y;


    }
    //Update the pos and vel arrays
    for (int i = 0; i<N ; i++){
      arr[i][0] = new_arr[i][0];
      arr[i][1] = new_arr[i][1];
      arr[i][3] = new_arr[i][2];
      arr[i][4] = new_arr[i][3];
    }
  }
```

Listing 1: Straight-forward implementation.

Time (s)	Number of elements
0.0277± 0.0027	10
0.0523 ± 0.0047	20
0.216± 0.012	80
0.619 ± 0.025	200
1.566 ± 0.044	400
4.05 ± 0.11	800
5.74 ± 0.17	1000
35.76 ± 0.21	3000

In this first attempt, we are updating the position and velocity of the particles in both directions in every step. We have two nested loops inside of the one iterating over the number of steps. One of the loops fixes a particle i, while the inner most loop iterates over the rest of the particles, thus calculating the force acting over the ith particle. After calculating the force, we can obtain velocities and positions by simple algebraic manipulations. This results must be stored in an auxiliary array, so that the force acting over the next particle is still computed with the original configuration of particles. After every step, the auxiliary array must be set to our original array.

Measurements:

$$t = (35.76 \pm 0.21) \quad s$$

In order to find the time complexity of this implementation, we measure usr time for different number of elements and then find the function that fits to the experimental points the best. All the following samples were executed up to 100 number of steps with $\Delta t = 10^{-5}$.

After trying with different functions, we found that the most accurate one was a quadratic function (as expected by the algorithm complexity) of the form:

$$y = ax^2 + bx + c$$

The following plot shows both the experimental points and our quadratic fit:

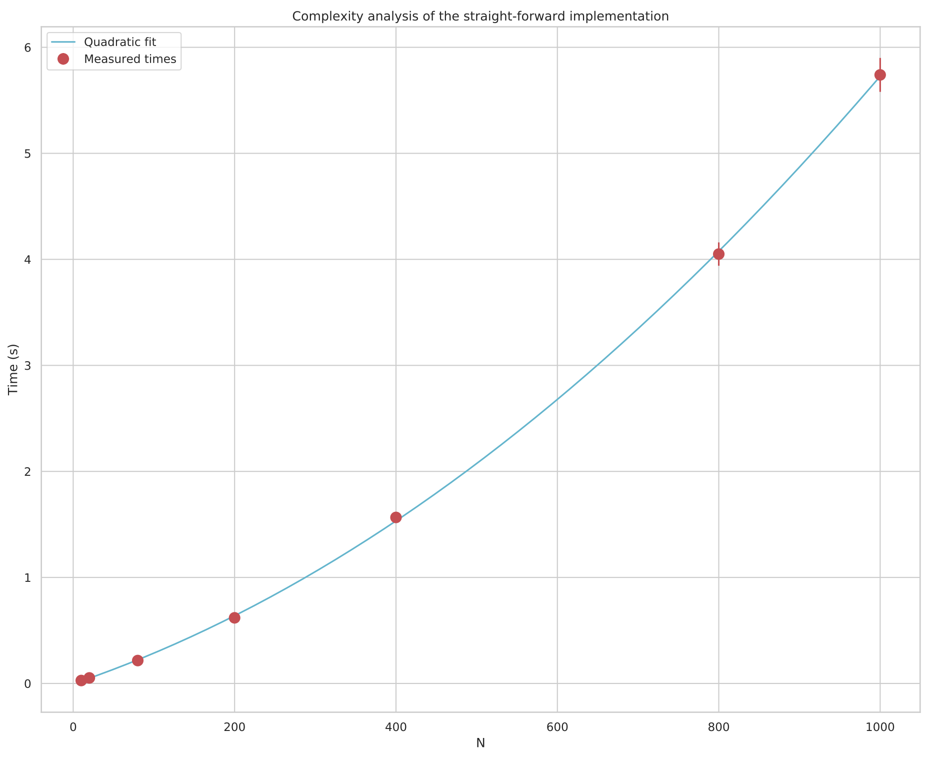

Figure 3: Inital complexity

The coefficients obtained and the R^2 estimator are:

$$a = (3.15 \pm 0.11) \cdot 10^{-6}\ s \qquad b = (2.58 \pm 0.11) \cdot 10^{-3} s$$

$$c = (-3.6 \pm 1.1) \cdot 10^{-3}\ s \qquad R^2 = 0.99991$$

3.3 Serial optimizations

All tests have been done with the following inputs: `N ellipse_N_03000.gal 200 0.00001 0` the cpu specifications are attached in Apendix A. The files are compiled with gcc and flags `-O3 -march=native -ffast-math`.

The original time is:

$$T_original = (54.20 \pm 0.078)s \tag{7}$$

This was obtained by running the program 10 times with the input discussed above and taking the mean and the sdt of the results.

3.3.1 Algorism Improvements

The original algorithm was changed by the one described in ??. This gave us a `usr time` of (52.68 ± 0.18)s . This gives a speedup of

$$S = 1.028 \pm 0.004 \tag{8}$$

```
 for (int k = 0 ; k<n_steps ; k++){
   for (int i = 0; i<N ; i++){
     for (int j = 0; j<=i ; j++){
         /* first we calculate the coordinates with respect
         to the initial frame of reference, then we calculate the Rij matrix.
         */
         double x_direction = arr[i][0] - arr[j][0];
         double y_direction = arr[i][1] - arr[j][1];
         double denominator = pow((sqrt((x_direction)*(x_direction) + (
  y_direction)*(y_direction))+epsilon_0),3);
         acc_matrix_x[i][j] = -G*x_direction/denominator;
         acc_matrix_y[i][j] = -G*y_direction/denominator;
     }
   }
   //Calculate total acceleration resulting on particle i from
   // Rij matrix and j!=i masses.
   for (int i = 0 ; i < N ; i++){
     double total_acc_x = 0;
     double total_acc_y = 0;
     //get total acceleration up to column i.
     for (int j = 0 ; j <= i ; j++){
       total_acc_x = total_acc_x + arr[j][2]*acc_matrix_x[i][j];
```

```
        total_acc_y = total_acc_y + arr[j][2]*acc_matrix_y[i][j];
      }
      //get the rest from column i, and row j > i. Sign change is needed
      for (int j = i+1 ; j < N ; j++){
        total_acc_x = total_acc_x-arr[j][2]*acc_matrix_x[j][i];
        total_acc_y = total_acc_y-arr[j][2]*acc_matrix_y[j][i];
      }
      //Apply euler simplectic method
      arr[i][3] = arr[i][3] + delta_t*total_acc_x;
      arr[i][4] = arr[i][4] + delta_t*total_acc_y;
      arr[i][0] = arr[i][0] + delta_t*arr[i][3];
      arr[i][1] = arr[i][1] + delta_t*arr[i][4];
    }
  }
```

Listing 2: Algorithm improvement

3.3.2 Switching data types to const

Micro-optimizations are usually carried out by the compiler itself, specially since all our tests are done with the -O3 flag optimization. In this case, we tried switching to const the variables that were not change throughout the script (number of particles, number of steps , Δt , etc);

```
const char *file_name = args[2];
const int N = atoi(args[1]);
const int n_steps = atoi(args[3]);
const double delta_t = atof(args[4]);
const float G = 100/(double)N;
const float epsilon_0 = 0.001;
```

Listing 3: Using const data type.

$$T_{original} = (0.03276 \pm 0.00025)\ s \qquad T_{optimized} = (0.03182 \pm 0.00014)\ s$$

Calculating the speedup and propagating the uncertainty:

$$S = (1.0295 \pm 0.0089)$$

Then we can conclude that this improvement does not grant us a large speedup, but its value is larger than the confidence interval spanned by its uncertainty, so it is statistical significant and we should stick to it.

3.3.3 Loop fusion

At the moment where we allocate arrays dynamically, we noticed that we could avoid doing one the the for loops if we fused them. The difference in efficiency is the following. $T = (65.18 \pm 0.43)s$. There was no speedup in this case so we discarded the changes.

Time (s)	Number of elements
0.000386± 8.87e-05	10
0.000723 ± 0.00014	30
0.00412± 0.00055	100
0.04301 ± 0.0043	300
0.714 ± 0.041	1000

3.3.4 Loop unrolling

The for loops can be unrolled manually. We decided for an unrollfactor of 4. This gave a usr time of $T = 36.93 \pm 0.067$, the resulting speedup

$$S = 1.467 \pm 0.003 \tag{9}$$

3.3.5 restrict keyword

Adding the restrict keyword on top of the loop unrolling the used pointers gave the time of $T = 36.913 \pm 0.067$. Hence a total speedup of

$$S = 1.468 \pm 0.003 \tag{10}$$

This was our best result so far.

3.3.6 Next steps

There is a part of our code that transposes a matrix and changes its sign. This could benefit from the optimization technique explained in lab 03, blocking. Sadly we ran out of time to explore this possibility.

3.4 Optimized version: Complexity

Measurments:

$$t = (0.714 \pm 0.041)$$

Again we fit for a quadratic function. The following plot shows both the experimental points and the fit.

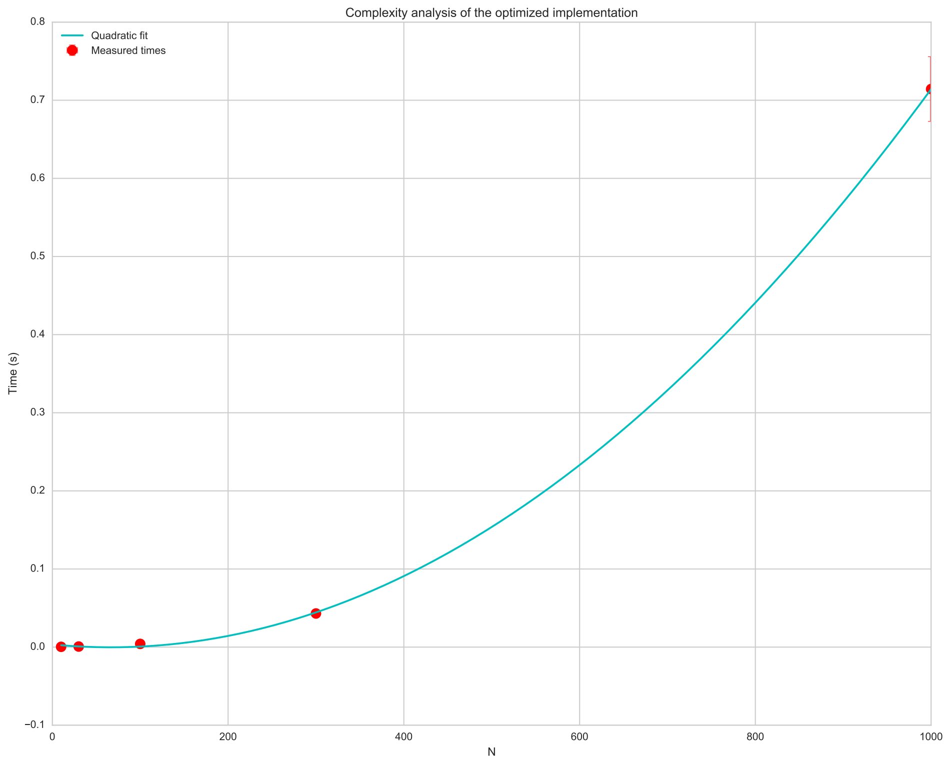

Figure 4: Final complexity

The coefficients obtained and the R^2 estimator are:

$$a = (8.203 \pm 0.017) \cdot 10^{-7}\ \ s \qquad b = (-1.946 \pm 0.18) \cdot 10^{-4} s$$

$$c = (3.4e \pm 2.2e) \cdot 10^{-3}\ \ s \qquad R^2 = 0.99991$$

4 Apendix A: lscpu result

All the timings and tests were performed on the host computer arrhenius.it.uu.se. Specifications:

- OS : Ubuntu 18.04
- GCC version : 7.3.0
- Architecture: x86_64
- CPU op-mode(s): 32-bit,64-bit

- Byte Order: Little Endian
- CPU(s): 16
- On-line CPU(s) list: 0-15
- Thread(s) per core: 2
- Core(s) per socket: 4
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 26
- Model name: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
- Stepping: 5
- CPU MHz: 1636.709
- CPU max MHz: 2268,0000
- CPU min MHz: 1600,0000
- BogoMIPS: 4533.87
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 8192K