ASSIGNMENT 3:
THE GRAVITATIONAL N-BODY PROBLEM

High Performance Programming 1TD062, 10 hp

Erik Bjerned & Mattias Persson

2023-02-06

UPPSALA
UNIVERSITET

Contents

1 The Problem

The gravitational N-body problem is based around solving how N particles affect the positions of all other particles with respect to their own gravity. Basically simulating a galaxy-type system where all particles affect every other particle. A particle in this case exists in a 2-dimensional domain, and consists of 6 data-points, x and y coordinates, velocity in x and y, mass and brightness. The simulation consists of updating each particle and its variables each timestep according to some governing equations, these equations resembles the normal gravitational equations but with a few tweaks to increase stability and some changes to constants to fit inside the set domain ($[0,1] \times [0,1]$). The following equations were the starting point for the simulation.

The force equation

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

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

The symplectic Euler method

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

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

2 The Solution

2.1 Theory

At first a simple algorithm was implemented, looping through N^2 calculations to calculate the force and further the acceleration all particles. When starting to optimize the program, this became the part that was the most expensive. So as a step in optimizing this task, a smarter approach to calculating the acceleration had to be taken. The following theory gives the background for this implementation and it still grows as a $\mathcal{O}(N^2)$ algorithm but more specifically the acceleration calculations are reduced down to $\mathcal{O}(N^2/2)$

$$\mathbf{F}_i = -Gm_i \sum_{j=0,\ j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}_{ij},\ \mathbf{a}_i = \frac{\mathbf{F}_i}{m_i} \tag{5}$$

$$\implies \mathbf{a}_i = -G \sum_{j=0,\ j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}_{ij} \tag{6}$$

We can use the fact that

$$\frac{\mathbf{r}_{ij}}{(r_{ij}+\epsilon_0)^3} = -\frac{\mathbf{r}_{ji}}{(r_{ji}+\epsilon_0)^3} \tag{7}$$

to reduce the number of calculations. This is easily proven but left out for since we are being brief. Because of this symmetry it is possible to reduce the iterations of the acceleration calculations by half. This is especially important as r_{ij} is expensive to compute due to the square root involved in its computation.

2.2 Implementation

The program can be divided into a few parts which will be explained in the coming sections. Timing of the implementation was done with the function `get_wall_seconds()` from a previous lab [1].

1. Initialization
2. Solve loop
    2.1. Calculation of acceleration
    2.2. Solve for new positions
3. Write output

2.2.1 Initialization

A part from reading input variables and assigning them, this part is mainly reading the input data into memory and allocating memory for the acceleration calculation. The input data is read into an array of size $6N$ as there are six attributes per particle and in the programme each attribute will be accessed by offsetting the index of the particle with a integer between zero and five. This could also be done with an array of a defined struct, each containing data for a particle. But in this case where the particles have few attributes which all are doubles it is not necessary to use a struct for code clarity. Whether it is a array of structs or an array of doubles, it probably will not affect performance noticeably at run time. The compiler most likely reduces both alternatives to offsets of memory addresses, compilation time might be affected it will not be a great difference and is not to be taken in to consideration. There may be some difference in runtime when reading input and writing output as it can be written straight from an array or read straight into an array, however this is a quite small part of the total time, especially when calculating for large N.

2.2.2 Solve loop

The solve loop is divided into two sections because of the modification to the algorithm in section 2.1 causes the need to calculate the acceleration of all particles before starting to solve for the positions. The performance of this is is discussed in a later section.

2.2.3 Calculation of acceleration

For the actual implementation of calculating the acceleration this is translated into this pseudo code

```
for i less than N:
    for j less than i:
        acceleration[i] = -G*m_j*vec_r_ij/(r_ij+eps)^3
        acceleration[j] = G*m_i*vec_r_ij/(r_ij+eps)^3
```

Instead of having only two lines for assigning values of the acceleration, it is in reality four as the acceleration has two components, x and y. In this algorithm we see that since we calculate forces between two objects, we do not need to check every individual pair of particles. The contribution of gravitational force is equal, and the acceleration change is proportional to the mass of the two particles.

2.2.4 Solve for new positions

For solving the new positions, the given *Symplectic Euler method* is implemented in a separate `for`-loop. Because of the given method, it is possible to simply update the relevant values in the data array sequentially without the use of temporary variables. Firstly the velocity of each particle is updated using said sympletic euler method, and then the actual positions are updated accordingly.

2.2.5 Write output

Since the data in this implementation is stored in a single, continuous array it can be written with a single `fwrite`-call.

3 Performance & Discussion

The program was performance tested on the university's Linux machine `vitsippa`. Every combination of settings are tested three times and the test with shortest runtime is recorded in the tables below

3.1 Code optimization

Within the code some standard serial optimization techniques are used. For instance loop invariants. Some of the values used in the inner loop (j-loop), namely the mass and position of the outer loop (i-loop) variables. Setting these parameters reduces instructions within the j-loop by three. Some expensive algorithms, for instance calculating $(r_{ij}+\epsilon)^3$ using the function `pow` is avoided, since just multiplying it three times with itself is faster. Usage of division is also done sparingly, since it is more computationally expensive than multiplying, since $G/(r_{ij}+\epsilon_0)^3$ is used calculating the acceleration in x and y directions for both the i and j variable, computing this beforehand and multiplying is quicker than dividing with $(r_{ij}+\epsilon_0)^3$ 4 times. The most important optimization is probably the one discussed in the theory section, using the symmetry of force between particles, this because of the reduction in `sqrt` calls.

Some other techniques using keywords were tried to see if they would give any improvement. Usage of the keyword `const` on every input and non-changing variable resulted in no improvement, this probably stems from the small amount of calls to the main computing function. Using `inline` for the functions also has no impact, most probably because that the compiler already does this optimization.

Table 1: Code optimization.Tested on `ellipse_N_03000.gal` for 100 timesteps with $\Delta t = 10^{-5}$ with flag `-O2 -ffast-math -march=native`. Time measured over entire program execution.

Keyword	**Time**
<none>	5.8400
`const`	5.8403
`restrict`	5.8417
`inline`	5.9740

Half inner loop, reduces iterations and expensive square roots

Table 2: Code optimization. Tested on `ellipse_N_03000.gal` for 100 timesteps with $\Delta t = 10^{-5}$ with flag `-O0`. Time measured over entire program execution.

Keyword	**Time**
<none>	32.1721
`const`	32.1685
`restrict`	32.2298
`inline`	32.1710

Another optimization that was done was to improve how the array containing the acceleration data was reset at each iteration. This is necessary to do since the calculation is only additive and it also done to reset any residual values in the allocated memory when performing the initially calculation. The naive approach is looping through the array and setting it to zero. This is often not considered as slow, but as it is needed to be done each time step there is a need to find a better solution. As seen previously in a lab `memset` was used as a faster alternative [2]. It does in fact result in a performance increase in the order of 1 *ms* per iteration with 3000 particles. There is however an even faster alternative, which is including this in the calculation loop, which is the alternative used and this further improved performance a reduction of 0.7 *ms* per iteration with 3000 particles.

3.2 Optimization flags

Using different optimization flags has large impact on performance. In the table below a comparison of different standard optimization flags are made using a data set of 3000 particles and 100 time steps. Using `-O0` is the default, and using the most basic `-O1` makes a huge difference in run time. The flags that gave the best result was `-O2 -ffast-math -march=native` which can be seen in bold in the table below. It is interesting that there is a time increase when switching `-O2` for `-O3` but it is difficult to say what is causing this as the compiler and the flags are complex.

Table 3: Optimization flags. Tested on `ellipse_N_03000.gal` for 100 timesteps with $\Delta t = 10^{-5}$. No special keywords. Time measured over entire program execution.

Flags	Time
`-O0`	32.1900
`-O1`	8.3986
`-O2`	7.1928
`-O3`	7.2655
`-Ofast`	7.2233
`-Os`	11.0113
`-O2 -ffast-math`	6.7691
`-O2 -ffast-math -march=native`	**5.8400**
`-O3 -ffast-math`	7.2250
`-O3 -ffast-math -march=native`	5.9034
`-Ofast -march=native`	5.8980

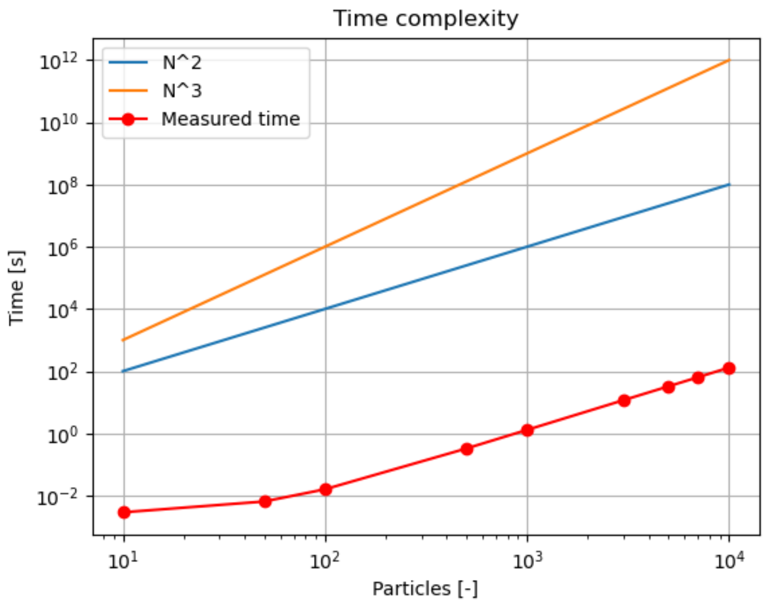

Figure 1: Time complexity of solution. Done with 200 timesteps with $\Delta t = 10^{-5}$

The complextity of the implementation is $\mathcal{O}(N^2)$ as expected. This can be seen in figure 1. For small $N < 100$ the complexity is a bit less than $\mathcal{O}(N^2)$, as the computation time is above to what is expected. In this case, it is more likely that the other parts of the program, such as reading input data or writing the output is the biggest contributor to computation time. The two mentioned depend on N but most certainly has some overhead time cost when opening and closing files which are independent of N causing this differing result. The computation time for the acceleration and solving is in this case so small that other parts of the program becomes the bottleneck.

References

[1] Course material. "`get_wall_seconds()`". In: *Lab 3* Task 9 (2023), p. 7.

[2] Course material. "SERIAL OPTIMIZATION PART 1, REDUCING INSTRUCTIONS". In: *Lab 5* Task 4 (2023), p. 7.

Appendix

A Serially optimized galsim.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>

double* read_input(char* path, int n);
void acceleration(double* data_array, double* acc_array, int n);
void solver(double* data_array, double* acc_array, double dt, int n);
void write_output(double* data_array, double* acc_array, int n, char* path);
double get_wall_seconds();



int main(int argc, char* argv[]){
  double time = get_wall_seconds();
  double start = time;

  if(argc != 6){
    printf("Not enough arguments. Input arguments: 'N', 'filename', 'nsteps',...
     'delta_t', 'graphics'\n");
    return -1;
  }


  int n_particles = atoi(argv[1]);
  char* input_path = argv[2];
  int n_steps = atoi(argv[3]);
  double dt = atof(argv[4]);
//  char graphics = *argv[5];
  double* acc_arr = (double*)malloc(n_particles*2*sizeof(double));

  printf("Initialized in %lf s\n", get_wall_seconds()-time);
  time = get_wall_seconds();

  double* data_arr = read_input(input_path, n_particles);

  if (data_arr == NULL) return -1;

  printf("Input read in %lf s\n", get_wall_seconds()-time);
  time = get_wall_seconds();

  for(int i = 0; i < n_steps; i++){
    acceleration(data_arr, acc_arr, n_particles);
    solver(data_arr, acc_arr, dt, n_particles);
  }

  printf("\n");
  printf("Problem solved in %lf s, %lf per timestep with %i particles\n", ...
   get_wall_seconds()-time, (get_wall_seconds()-time)/n_steps, n_particles);
  time = get_wall_seconds();

  write_output(data_arr, acc_arr, n_particles, "result.gal");
  printf("Output written in %lf s\n", get_wall_seconds()-time);

  free(data_arr);
  free(acc_arr);
  printf("Program finished in %lf s. Exiting...\n", get_wall_seconds()-start)...
   ;

  return 0;
}

void acceleration(double* data_array, double* acc_array, int n){
  double x_i, y_i, x_j, y_j,m_i, m_j, r_ij;
  double G = 100/(double)n;
  double eps = 0.001;
  int i, j;
  double denom;

  for(i = 0; i < 6*n; i +=6){
    // Assign current values to mitigate carry over from previous timestep. ...
   Faster than memset
    acc_array[2*(i/6)] = 0;
    acc_array[2*(i/6)+1] = 0;

    // Fetch j-loop invariant values
    x_i = data_array[i];
    y_i = data_array[i+1];
    m_i = data_array[i+2];

    for(j = 0; j < i; j+=6){
      x_j = data_array[j];
      y_j = data_array[j+1];
      m_j = data_array[j+2];

      r_ij = sqrt((x_i-x_j)*(x_i-x_j)+(y_i-y_j)*(y_i-y_j));
      denom = G/((r_ij+eps)*(r_ij+eps)*(r_ij+eps));

      // Update x and y components of the acceleration
      acc_array[2*(i/6)] += -m_j*(x_i-x_j)*denom;
      acc_array[2*(i/6)+1] += -m_j*(y_i-y_j)*denom;
      acc_array[2*(j/6)] += m_i*(x_i-x_j)*denom;
      acc_array[2*(j/6)+1] += m_i*(y_i-y_j)*denom;
    }
  }
}


void solver(double* data_array, double* acc_array, double dt, int n){
  int i;
  // Symplectic Euler for the n particles
  for(i = 0; i < 6*n; i+=6){
    // Velocity update
    data_array[i+3] = data_array[i+3] + dt*acc_array[2*(i/6)];
    data_array[i+4] = data_array[i+4] + dt*acc_array[2*(i/6)+1];

    // Position update
    data_array[i] = data_array[i] + dt*data_array[i+3];
    data_array[i+1] = data_array[i+1] + dt*data_array[i+4];
  }
}


```

```c

double* read_input(char* path, int n){
  FILE* file = fopen(path, "rb");
  if (file == NULL){
    printf("ERROR: File path not valid\n");
    return NULL;
  }
  // Each particle has 6 attributes --> [pos_x, pos_y, m, v_x, v_y, b, ...]

  double* input_data = (double*)malloc(6*n*sizeof(double));
  if(fread(input_data, sizeof(double), 6*n, file) != 6*n){
    printf("ERROR: Mismatch between specified number of particles and read ...
   number of particles\n");
    return NULL;
  }
  fclose(file);
  return input_data;

}

void write_output(double* data_array, double* acc_array, int n, char* path){
  FILE* file = fopen(path, "wb");

  if(fwrite(data_array, sizeof(double) ,6*n, file) != 6*n){
    printf("ERROR: Mismatch between specified number of particles and written...
    number of particles\n");
    return;
  }
  fclose(file);
}

double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}
```