UPPSALA
UNIVERSITET

DEPARTMENT OF MATHEMATICS - Course 2020

HIGH PERFORMANCE PROGRAMMING - ASSIGNMENT 3: The Gravitational N-Body Problem

KIERAN BARBER

PANAGIOTIS PAPIAS

Teacher
Sverker Holmgren

Contents

1 The Problem

For this assignment, we are going to implement a program that will calculate the evolution of N particles in a gravitational simulation, whereby we are given an initial set of particles. The simulation will be done in 2 spatial dimensions with an x and y coordinate.

To help describe the evolution of these particles, we will make use of Newton's law of gravitation in two dimensions, which states that the force exerted on particle i by particle j is given by

$$\boldsymbol{f}_{ij} = -\frac{Gm_i m_j}{\|\boldsymbol{r}_{ij}\|^3}\boldsymbol{r}_{ij}.$$

where G is the gravitational constant, m_i and m_j are the masses of the particles, and $\boldsymbol{r}_{ij}$ is the distance vector given by $\boldsymbol{x}_i - \boldsymbol{x}_j$ with x_i being the position of particle i.

We will make use of the following force equation (Plummer Sphere Force Equation) to describe the forces on the particles. The force on particle i is given by

$$\boldsymbol{F}_i = -Gm_i \sum_{j=0, j!=i}^{N-1} \frac{m_j}{(\|\boldsymbol{r}_{ij}\| + \epsilon_0)^3}\boldsymbol{r}_{ij}.$$

where ϵ_0 is a smoothing parameter and in the computations, we used the value $\epsilon = 10^{-3}$.

To update the particle positions, we use the Euler Symplectic Time Integration method. The equations describing this are

$$\begin{cases} \boldsymbol{a}_i^n = \frac{\boldsymbol{F}_i^n}{m_i}; \\ \boldsymbol{u}_i^{n+1} = \boldsymbol{u}_i^n + \Delta t \boldsymbol{a}_i^n; \\ \boldsymbol{x}_i^{n+1} = \boldsymbol{x}_i^n + \Delta t \boldsymbol{u}_i^{n+1}, \end{cases}$$

where Δt is the time step size, $\boldsymbol{a}_i$ is the acceleration of particle i, $\boldsymbol{u}_i$ is the velocity and $\boldsymbol{x}_i$ is the position of particle i. In the computations, we used $\Delta t = 10^{-5}$. The value of G depends on N and is given by $G = 100/N$.

2 The Solution

We began by writing our code in a file we named galsim.c. This is where all the functionality that we require is stored. We used code from graphics to be able to visualise the results of the simulation.

Within the galsim.c file, we write our main program that will enable us to calculate the evolution of a system of N particles. This file begins with calling the relevant libraries we require. Below this, we take in the variables for the graphics, gravity and the smoothing constant epsilon. To improve runtime, we have set epsilon as a constant.

Below this, we have our structs, one for the various vectors we will use and one for the particles attributes. The vector struct takes two doubles, one for x and one for y. The particle struct takes two vectors for position and and velocity, and two doubles, one for mass and brightness. These have been ordered so that they are in the same order as the data that's read in from the gal-files.

Below this, we introduced print functions that we would use for debugging. This includes print_vector_info, print_particle_info and print_all_particle_info. After this, we include a function we then call in the main function to get computation times for each optimisation.

We now look to the building of the force functions. Since the Plummer spheres force equation has smoothing involved, we looked to implement this. To do this, we built each component of the equation in separate functions. The first function to be built was $\boldsymbol{r}_{ij}$. We only computed this in one way. The next to compute was r_{ij}^3. This was implemented in two different ways.

Version 1

```
double norm3(vector_t v) {
    return pow((sqrt(pow((v.x), 2) + pow((v.y), 2)) + epsilon), 3);
}
```

Version 2 - 4

```
double norm3(vector_t v) {
    double n3;

    n3 = sqrt(v.x*v.x + v.y*v.y) + epsilon;
    n3 = n3 * n3 * n3;
    return n3;
}
```

With the above versions of norm3, we then implemented them into two versions of a function to calculate the force on particle i from particle j. To do this, we had to calculate in both the x and y directions. Initially, we chose to include all terms for both of these directions.

Version 1 - 2

```
vector_t force(particle_t pi, particle_t pj) {
    vector_t f;

    f.x = -(G * pi.mass * pj.mass * r_vector(pi.position, pj.position).x)
            / norm3(r_vector(pi.position, pj.position));
    f.y = -(G * pi.mass * pj.mass * r_vector(pi.position, pj.position).y)
            / norm3(r_vector(pi.position, pj.position));
    return f;
}
```

We decided then that we would calculate any repetitions in each equation just once, so that this may improve computation time.

Version 3 - 4

```
vector_t force(particle_t pi, particle_t pj) {
    vector_t f;
    vector_t r;
    double a;

    r = r_vector(pi.position, pj.position);
    a = -(G * pi.mass * pj.mass) / norm3(r);

    f.x = a * r.x;
    f.y = a * r.y;

    return f;
}
```

Now that we have functions to compute the force applied to each particle, we can now loop through each particle, and compute the total force applied to each particle.

Version 1 - 3

```
vector_t total_force(particle_t* p, int i, int N) {
    vector_t f = {0.0,0.0};

    for(int j = 0; j < N; j++) {
        if(i != j) {
            vector_t f_temp = force(p[j], p[i]);
            f.x += f_temp.x;
            f.y += f_temp.y;
        }
    }
    return f;
}
```

One final improvement we applied to this was to note that since we had the forces computed from particle i to particle j, we also had the negative of this, so we then had the force applied from particle j to particle i.

Version 4

```
void compute_forces(vector_t* forces, particle_t* particles, int N) {

    for(int i = 0; i < N; i++) {
        forces[i].x = 0.0;
        forces[i].y = 0.0;
    }

    for(int i = 0; i < N; i++) {

        for(int j = i + 1; j < N; j++) {
                vector_t f = force(particles[j], particles[i]);
                forces[i].x += f.x;
                forces[i].y += f.y;
                forces[j].x -= f.x;
                forces[j].y -= f.y;
        }

    }
}
```

One final thing that we used to help us was to implement timings directly into the Makefile code so that when it came to computing wall timings at the end with different optimisations, we could do this directly from the Makefile.

3 CPU Configuration

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               142
Model name:          Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:            10
CPU MHz:             700.041
CPU max MHz:         3400,0000
CPU min MHz:         400,0000
BogoMIPS:            3600.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            6144K
NUMA node0 CPU(s):   0-7

(Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
```

4 Performance and Discussion

4.1 Timings

Version	-OO	-O3	-Ofast
V1	199.702909	110.818015	3.372535
V2	94.216680	4.225227	4.065753
V3	45.380386	3.521552	3.271567
V4	24.352817	2.538566	2.454337

Given the optimisations described through code changes in the previous section, we have above, our timings for various optimisation flags.

Starting with Version 1, we can see that setting no optimisation flags to the code leaves us computing for a significant amount of time. this is almost halved by taking the -O3 flag. The -Ofast flag takes a significantly shorter amount of time to compute this system. A potential reason we can see here is that it sees the use of the pow function and realises it just simply needs to compute $x * x$ rather than x^2 for instance.

The second version of the code that we produced moved from calculating norm3 in the version we described above. This gave significant increases in time performance for the -O0 and -O3 flags, however, it actually increased computation time for the -Ofast flag. We have not understood why this is the case.

For Version 3, we altered the way in which we computed the force for each particle. Instead of calling functions for both the x and y directions, we decided to take out the elements of each equation that are the same and compute them before we computed the force in each of the x and y directions, thus meant that we were doing half of the original calculations. Again, this gave significant improvements in performance for -O0. Proportionately, it gave big performance improvements for -O3 and -Ofast.

The final Version took into account that we could compute the forces in half the time. This is achieved since once we know the force applied to particle i from particle j, we know that the force applied to particle j from particle i is just the negative. This gave significant performance improvements with each flag.

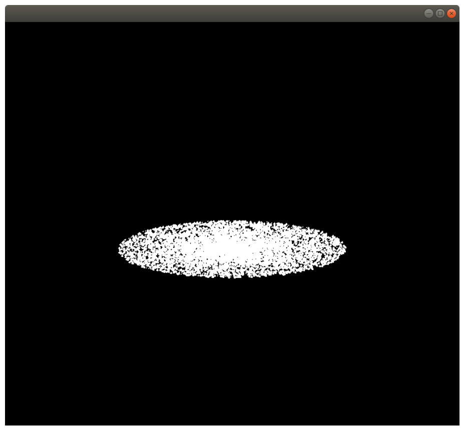

Figure 1: A simulation of a galaxy

4.2 Complexity

Finally, below we have a plot outlining the computation times measured against $O(N^2)$.

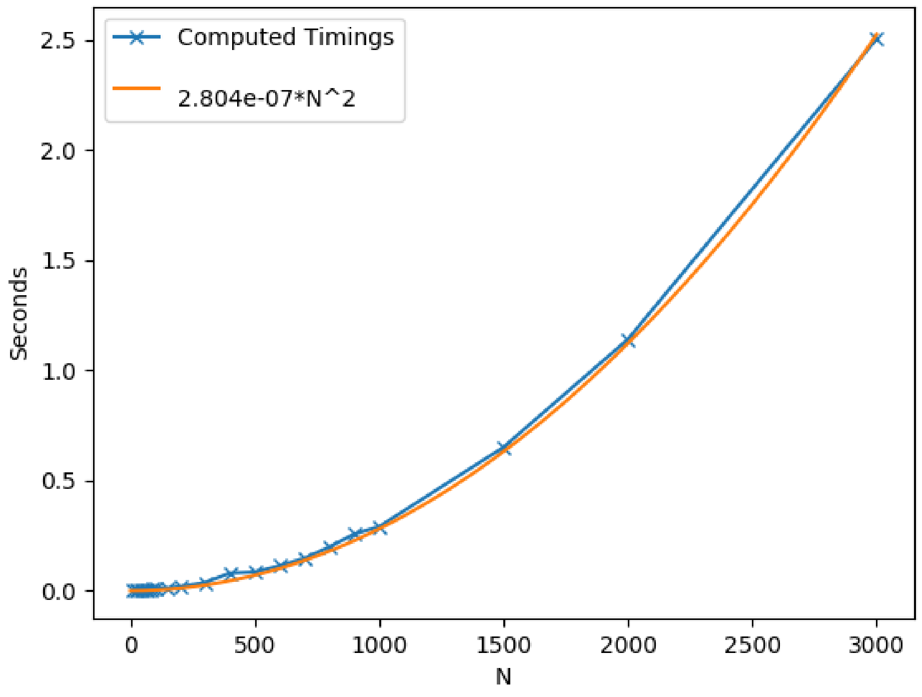

Figure 2: Time Complexity Plot