UPPSALA
UNIVERSITET

HIGH PERFORMANCE PROGRAMMING

Assignment 6

Sy-Hung Doan, Marius Nitzsche

March 13, 2019

1 The problem

The goal of this Assignment was to simulate the evolution of possible galaxies. Based on Newton's law of gravitation in two dimensions the following equation will help compute new positions and velocities of stars after a certain time step.

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

m_i is the mass of a particle i, r_{ij} is the distance between two particles and $\mathbf{r}_{ij}$ is the two dimensional vector, which describes the positions of the particles dependent on each other. With the help of the symplectic Euler time integration method and the result of the above mentioned formula new positions and velocities can be updated. Additionally, the Barnes-Hut algorithm is used to reach a faster approximation. It joins points to a single point if the factor

$$\theta = \frac{\text{width of box}}{\text{distance between center of box and particle}} \tag{2}$$

is small enough for nodes in a quadtree.
Furthermore, the performance is improved using OpenMP. It is applied on calculating the force on each particle and updating new positions.
Initial positions and velocities are given in a file; our program is supposed to read them and save the updated information after computing a certain amount of time steps. The constants have the following values:

- $\epsilon_0 = 10^{-3}$
- Gravity $G = 100/N$ (N is the amount of particles)
- timestep $\Delta t = 10^{-5}$

The program is able to run with and without graphics, which demonstrate the current state of the simulated galaxy.

2 The solution

2.1 Data structure

We use `struct`'s to store the essential information of a particle like its positions and its mass. To store the velocity and the brightness of a particle, we use `array`'s, which separates work. We use the `struct` force to store a force vector's components. `struct` is also used for the quadrangle tree.

```
typedef struct point {
    double px;  // position x of the point
    double py;  // position y of the point
    double m;   // mass of the point
} point;
```

```c
typedef struct force {
    double fx;  // force along x-axis
    double fy;  // force along y-axis
} force;


typedef struct quad quad;
struct quad {
    point* p;   // the particle inside the quadrangle

    double w;   // width of this quadrangle
    double cx;  // position x of the center of the quadrangle
    double cy;  // position y of the center of the quadrangle

    point* core;    // represents the center of mass
                    // px : position x of the center of mass
                    // py : position x of the center of mass
                    // m : the mass of the quadrangle

    quad *child[4]; // pointers to 4 child quadrangles.
};
```

2.2 Code structure

Our code is divided into 4 main parts:

The first part contains `#include`'s and some coefficients and constants that are used later on.

Next part is the definitions of some helper functions like reading data from the input file, executing with or without graphics, calculating the data at the next time step and writing data to the corresponding output file.

What follows is the `main` function. The last part is helper functions' bodies.

2.3 Algorithm implementation

First, after reading data from the input file, we obtain an array of `point`'s. It then goes through the function `quad_insert`, which inserts point by point into an empty quad-tree.

After that, `quad_mass` is called to calculate the accumulative mass and its position of each quadrangle. The center of mass of a quadrangle recursively depends on its child quadrangles' center of mass.

Then, a loop, which iterates through the array of examined `points`, calls `quad_force`, which takes a `point` and a quad-tree and gives the force on that `point` by `point`'s or quadrangles in that tree.

Making use of the array of `force`'s returned from the above function, we update the velocity of each point and then its new position. This function utilizes the θ_{max} rule. If θ of the examined point and the current quadrangle is smaller than or equal to the defined θ_{max}, it will not go down to its children and the current quadrangle is treated as a `point`.

```
force quad_force(quad* qt, point* p, double theta_max)
{
    force f = {0, 0};

    if(qt == NULL)
        return f;

    // if this point is exactly the same as the point in the quadrangle,
    // return zero force
    if(qt->p == p)
        return f;

    double rx = p->px - qt->core->px;  // distance along x-axis
    double ry = p->py - qt->core->py;  // distance along y-axis

    double rij = sqrt(rx * rx + ry * ry);   // distance between two
     ↪  objects

    // if the quadrangle is innermost
    // or the theta rule is satisfied
    if(qt->p != NULL || qt->w <= theta_max * rij)
    {
        double rij_e0_3 = (rij + 0.001) * (rij + 0.001) * (rij + 0.001);

        f.fx = qt->core->m * rx / rij_e0_3;
        f.fy = qt->core->m * ry / rij_e0_3;

        return f;
    }

    // calculate forces of each child and sum it up
    for(int i = 0; i < 4; i++)
    {
        force fc = quad_force((qt->child)[i], p, theta_max);
        f.fx += fc.fx;
        f.fy += fc.fy;
    }

    return f;
}
```

3 Performance and Discussion

3.1 Serial Optimization

All of our test were made on our own computer, it has the following specs:

```
Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz
Ubuntu 18.04 LTS

L1d cache:  32K
L1i cache:  32K
L2 cache:   256K
L3 cache:   6144K

gcc (Ubuntu 7.3.0-16ubuntu3) 7.3.0
```

In the beginning we made tests to determine the parameter θ_{max}. The following Flags were used at those tests: `-O3 -funroll-all-loops`.

The data `ellipse_N_02000.gal` was used and 200 time steps were simulated. We increased θ_{max} after every test until it reached the fault tolerance which is 10^{-3} and calculated by comparing our results to the provided results. The optimal θ_{max} for our code is **0.256**.

The following records were made when we found that our code performed well enough. We did not put too much attention to the execution time at this time. We shall discuss the optimization very soon.

	Theta-max	Execution time	Difference
Test 1	0.000	67.93256	0.00000
Test 2	0.100	17.69874	0.00013
Test 3	0.200	8.87998	0.00041
Test 4	0.300	5.71884	0.00251
Test 5	0.250	6.94221	0.00089
Test 6	0.260	6.66523	0.00115
Test 7	0.255	6.78976	0.00095
Test 8	0.256	6.73123	0.00096
Test 9	0.257	6.76034	0.00101

Table 1: Table to compare Theta-max

We have gone through different steps to optimize. However, to keep it succinct and brief, we list how we came up to the final code. At each stage, we always tried to keep the difference be smaller than 10^{-3}.

1. Original implementation of the quad-tree
   Exec time: 6.731233

2. Use flags: `-O3 -funroll-loops -march=native` for just the biggest `gcc` compilation (the one that directly generates `galsim`)
   Exec time: 5.162397

3. Use flags: `-O3 -funroll-loops -march=native` for all `gcc` compilations
   Exec time: 3.687321

4. Use flags: `-O3 -funroll-loops -march=native` for all `gcc` compilations
   Exec time: 3.687325

5. Use multiplication instead of division as much as possible and reduce if-else statement
   Exec time: 2.946810

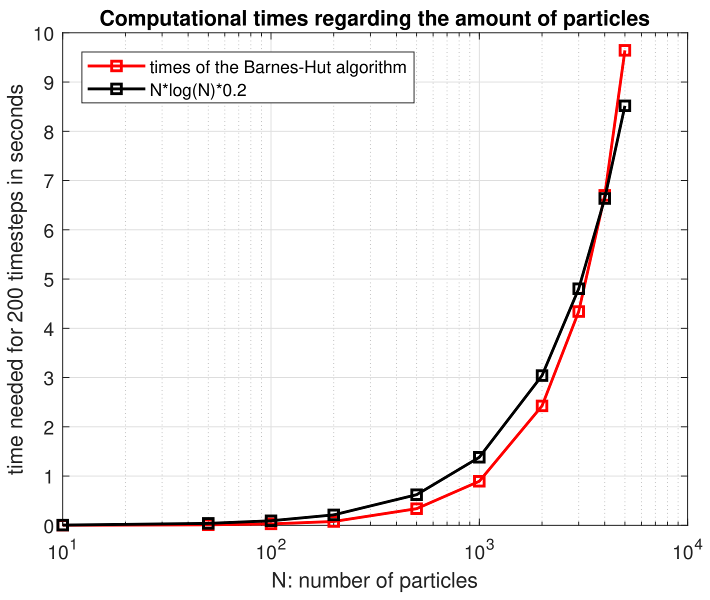
6. Modify the structure of `quad` from 3 `double` numbers storing information of the center of mass right in `struct quad` to currently a pointer to an exteral point.
   Exec time: 2.678212

7. Use flags: `-Ofast` instead of `-O3`
   Exec time: 2.375275

At some points, `gprof` helped us a ton to find out which function took most the execution time and `valgrind memcheck` detected if we had forgotten to free up the memory.

Once we reached this final code, we tried with different problem sizes, which is shown in the following graph. We use 200 time steps for all these cases.

Figure 1: Computational times for different values of N up until 5000

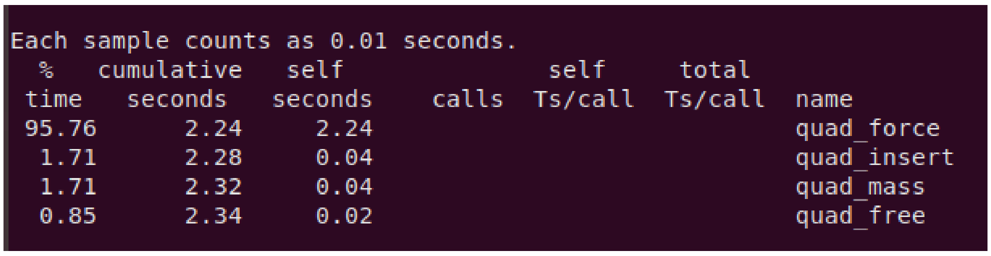


In the above graphic it is clear that the computational times increase similar to a $N \cdot \log(N) \cdot 0.2$ curve, which was expected. We are very satisfied with our times; even with 5000 particles and 200 time steps the simulation takes less than ten seconds. Our optimizations were very successful.

3.2 Parallel Optimization using OpenMP

We tried to inspect the timing to see which function takes up the most time using `gprof`. Below is the data it provided with the problem:

```
N=2000, nsteps=200, theta_max=0.256, max_diff = 0.000962522914
```



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 95.76      2.24     2.24                             quad_force
  1.71      2.28     0.04                             quad_insert
  1.71      2.32     0.04                             quad_mass
  0.85      2.34     0.02                             quad_free
```

Figure 2: Computational times for N=2000 with 200 time steps

The `quad_force` function is obviously the one we need to optimize. The others seem to be good and we handle them later.

However, the time we see above is the total time that `quad_force` function was called. It seems like we cannot use OpenMP inside the function since it does not have a big loop or something like that (we use recursion in that function), therefore we decided to optimize the loop which called that function (in `next_time_step` function).

What we did was we divided the main loop in `next_time_step` function into several chunks corresponding to the number of threads. It is similar to the method that the teachers used in some previous examples. We use OpenMP for the loop that updates the new position of a particles as well, even though it is not something that reduces the execution time very much.

Below is the code after our parallelization using OpenMP.

```c
#pragma omp parallel num_threads(n_threads)
{
    #pragma omp for schedule(dynamic)
    for(i = 0; i < N; i++)
    {
        // calculate the force on each particle
        f[i] = quad_force(qtree, &(list_points[i]), theta_max2);
    }
}

#pragma omp parallel num_threads(n_threads)
{
    #pragma omp for schedule(dynamic)
    for(i = 0; i < N; i++)
    {
        // update the new velocity and new position of each particle
        velo_x[i] += - G * delta * f[i].fx;
        velo_y[i] += - G * delta * f[i].fy;
        list_points[i].px += delta * velo_x[i];
        list_points[i].py += delta * velo_y[i];
    }
}
```

In order to measure the performance of this change, the next figure will demonstrate the speedup of the OpenMP code compared to the serial code from Assignment 4:

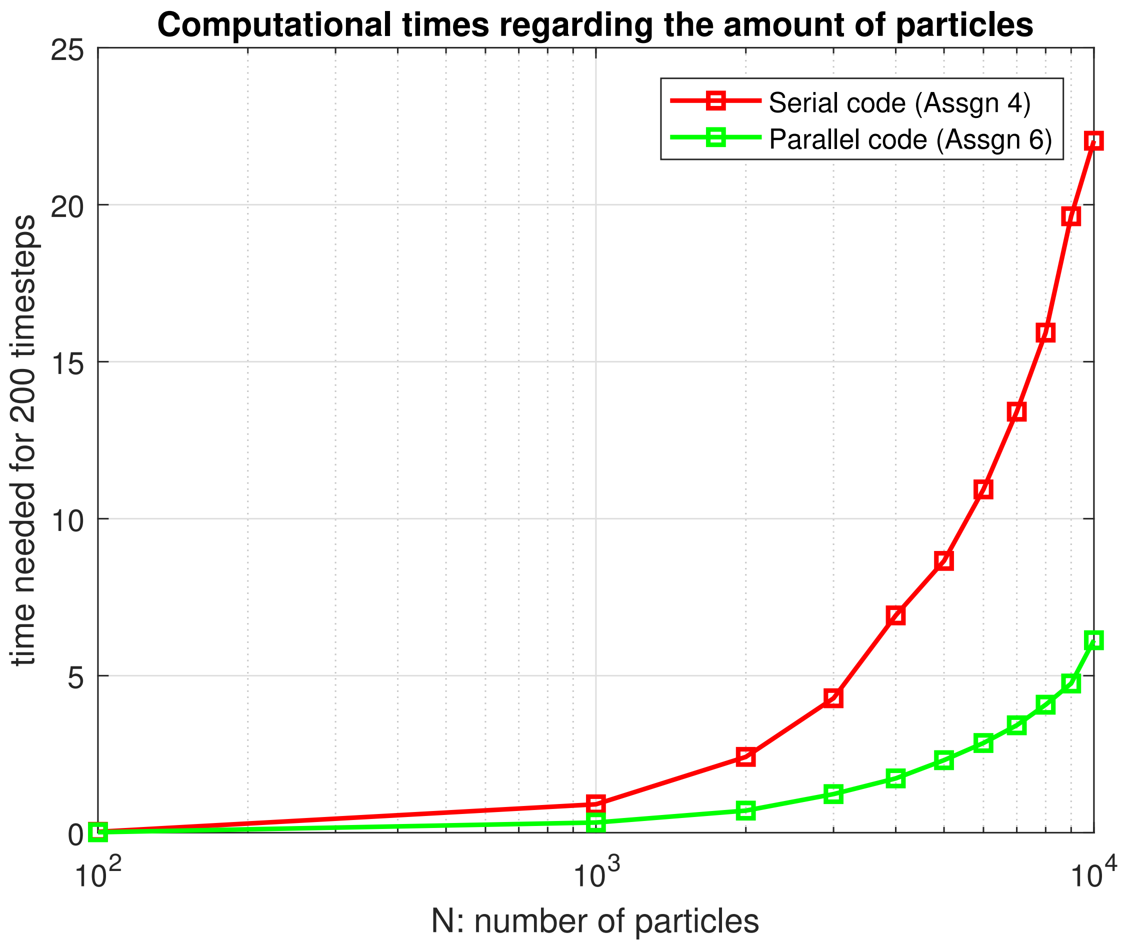


Figure 3: Comparison of PThreads and OpenMP

Figure 3 shows, taht the parrallel code is around four times as fast as the serial code. For big choices of N this is a huge improvement. It is clear that parallel code is worth it if it is applicable. Now we will compare PThreads to OpenMP. For that we look at the computational time for the following problems with different number of threads.

1. `N=2000, nsteps=200, theta_max=0.256, max_diff = 0.000962522914`
2. `N=3000, nsteps=100, theta_max=0.584, max_diff = 0.000987675851`

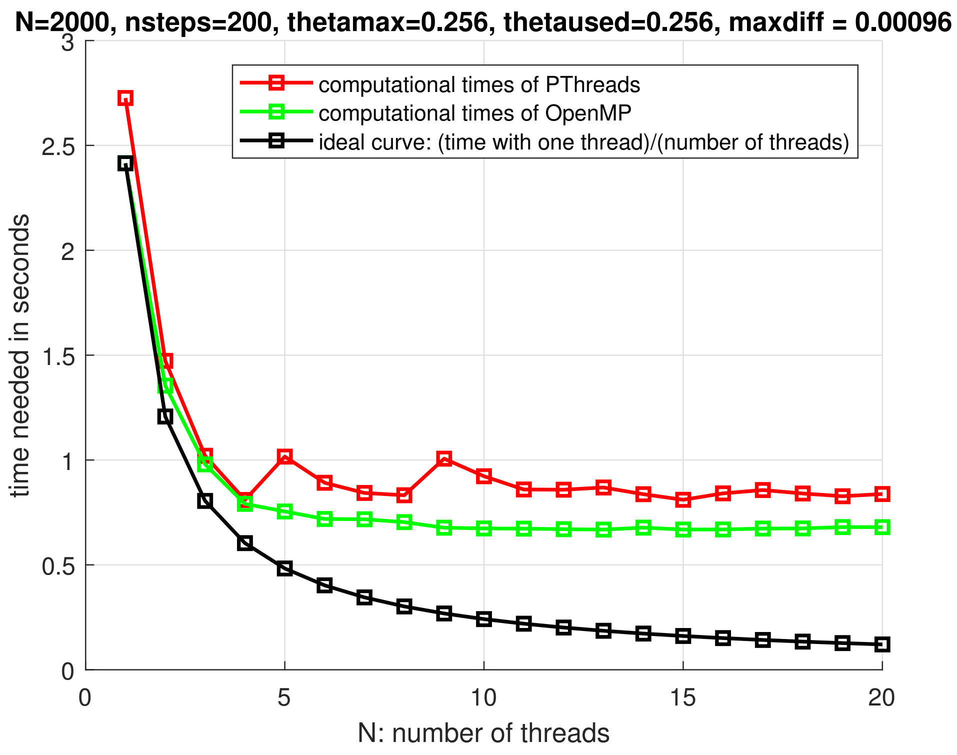


Figure 4: Comparison of PThreads and OpenMP

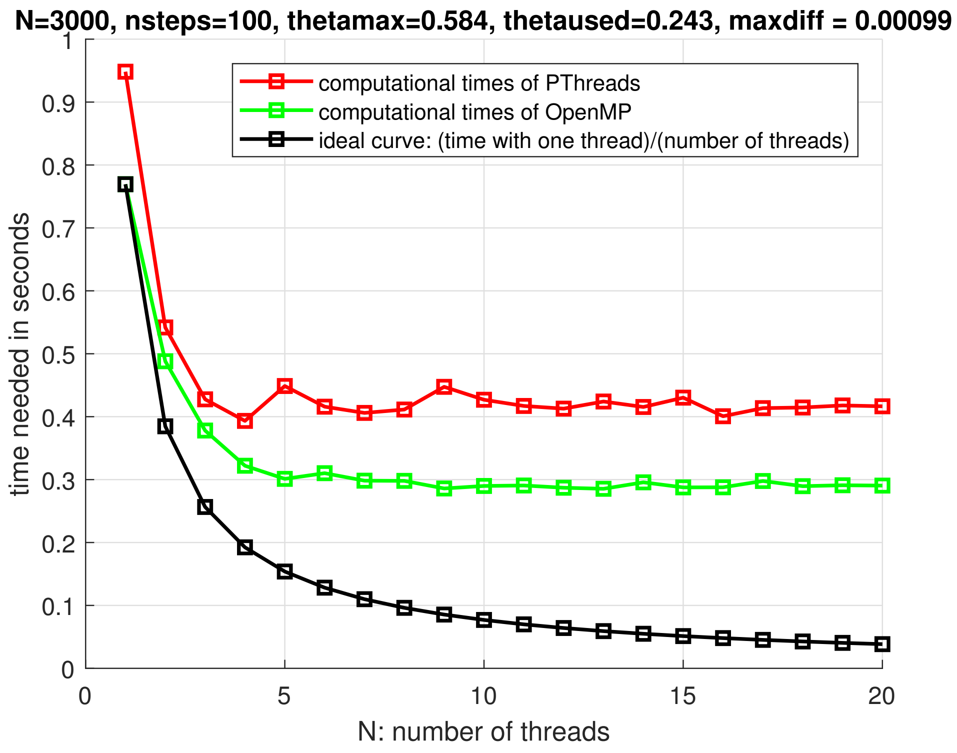


Figure 5: Comparison of PThreads and OpenMP

Figures 4 and 5 demonstrate that our OpenMP implementation is more efficient than our PThreads implementation. For any number of threads it computes faster. However, the ideal curve shows that there is still a fixed overhead.
OpenMP is a smoother curve; there are no sudden jumps. The minimal computational time is reached at around eight threads; more threads do not lead to substantial improvements.

4 Work contribution

We paired programming in order to create fewer bugs and make sure both understanding the codes.