High Performance Programming
Uppsala University – Spring 2020
Report for Assignment 3 by Team xx

Li Ju, Shu-pei Huang

23rd February 2020

I Introduction

1 Problem description

Gravitation is a natural phenomenon by which all things with mass or energy—including planets, stars, galaxies, and even light,are brought toward (or gravitate toward) one another. As one of the 4 fundamental interactions of physics, gravitation plays a vital importance role in physical world. In this assignment, a code will be implemented to calculate the evolution of N particles in a gravitational simulation. Given an initial set of particles, the program should calculate the motion that approximates the revolution of a galaxy.

For simplicity of presenting the results, the simulation is done in two partial dimensions, so the position of each particle will be given by two coordinates (x and y). There will be N particles with positions in a $L \times W$ dimensionless domain ($L = W = 1$ in this assignment).

The particle masses, initial positions and velocities will be read from an input file when the program starts, then simulated for a given number of timesteps, and in the end of the final masses, positions and velocities will be written to a result file.

2 Physics principle

The program is aimed to simulate a N-particle system governed by gravity. Key equations will be used are described as following:

1. Newton's law of gravity $f_{ij} = -Gm_i m_j \mathbf{r}_{ij}/r_{ij}^3$, where G is the gravitational constant, m_i and m_j are the masses of the particles, r_{ij} is the distance between the particles, and $\mathbf{r}_{ij}$ is the vector that gives the position of particle i relative to particle j. However, in reality, "particles" normally have finite extension in space. To deal with this, we introduce a slightly modified force that corresponds to so-called Plummer spheres as: $f_{ij} = -Gm_i m_j \mathbf{r}_{ij}/(r_{ij} + \epsilon_0)^3$;
2. Newton's second law $\mathbf{a} = \mathbf{F}/m$, where a is the acceleration of the particle, F is the force the particle is exerted and m is the mass of the particle;

3. $\mathbf{u} = \mathbf{u}_0 + \mathbf{a}\Delta t$, where Δt is the time step, $\mathbf{u}$ is the velocity after the time step, $\mathbf{u}_0$ is the velocity before the time step;

3 Simulation Settings

In the simulation program, constants and parameters are set as following:
$G = 100/N$
$\epsilon_0 = 10^{-3}$
$\Delta t = 10^{-5}$

4 Computer Details

To be added...

II Serial Implementation, Analysis and Optimization

1 Implementation

For assignment 3, the simulation is implemented serially.

Algorithm design After analyzing the problem, a primary peuso code is designed and shown as following.

Algorithm Gravitation Simulation

Require: Given valid sufficient parameters
Ensure: Valid data can be read from input file
  while max step not reached **do**
    Use position information to update the matrix of $N \times N$, in which gravity between each pair of particles are stored;
    for each in particles **do**
      Calculate resultant force from gravity matrix;
      Calculate acceleration of current particle;
      Update the particle information: velocity and position;
    end for
  end while
  Save final status information of particles to a result file.

Algorithm optimization Before writing the code, it is always good to analyze and optimize the as-designed algorithm. The time complexity of the algorithm is of $\mathcal{O}(N^2)$, because the largest computation cost is required bt the step in which resultant force of each particle is calculated by summing up forces exerted by all other particles. The memory complexity of the algorithm is also of $\mathcal{O}(N^2)$, because a force matrix of size $N \times N$ is maintained to store interactions between each pair of particles.

Though in assignment 3, complexity of $\mathcal{O}(N^2)$ is acceptable, still we can make some optimization on the algorithm.
1. According to Newton's third Law, force that particle i exerted to particle j is numerically equal to the force that particle j exerted to particle i, but the direction is opposite. Therefore, it is not required to calculate all values in the force matrix. Instead, only values in a half of the matrix is needed. When calculating the resultant force of a particle, we only to add those which are not stored by adding their reaction force negatively;
2. When forces between each pair of particles, masses of both particles are multiplied. However, when resultant force of a specific particle is calculated and acceleration of the particle is calculating, the mass of the particle needs to be divided. Therefore, to reduce redundant calculation, only $C_{ij} = -G\boldsymbol{r}_{ij}/(r_{ij} + \epsilon_0)^3$ are calculated and stored in the matrix as a coefficient matrix. When accumulating resultant accelerations with coefficients, for each coefficients, the mass of particle j is multiplied, as $\boldsymbol{a}_i = \sum_{j=0,j\neq i}^{j=N-1} C_i m_j$.

The algorithm optimization reduces computation cost of the code be reducing the redundant calculation. The primary implementation is based on the optimized algorithm.

Algorithm implementation During the implementation, to achieve a good performance even on the primary version, following rules are followed:

1. To reducing times of system allocating memory, a one dimensional array is allocated to be the force matrix, so that the one time function call for memory allocating is enough, instead of calling malloc function N times;
2. Try not to call functions, which will introduce a new stack to the memory, in the inner loop, e.g. use a*a*a instead of pow(a,3);
3. Avoid if statement, which may reduce pipeline of the CPU's efficiency, in the inner loop;
4. Improve data locality, which means in inner loop, visit data that are in nearby memory.

2 Performance Analysis

Error evaluation To evaluate the simulation error comparing with references, several tests are conducted with following details:
[**input files**]: *ellipse_N_00010.gal, ellipse_N_00100.gal, ellipse_N_00500.gal, ellipse_N_01000.gal, ellipse_N_02000.gal and ellipse_N_03000.gal*;
[**time step**]: *200, 200, 200, 200, 200* and *100*, respectively;
[**GCC flags**]: *-lm* only;
[**Optimization options**]: No GCC optimization flags are used;
[**computer details**]: shown in part I.

The result of error evaluation test is shown as Table(1):

Total running time evaluation To evaluate the total running time of the simulation program, test are conducted with exact the same test cases as we used in previous paragraph. For each test case, 10 times' parallel running are conducted, to minimum the random testing error.

Number of particles	Number of steps	Position maxdiff
00010	200	0.000000000000
00100	200	0.000000000000
00500	200	0.000000005978
01000	200	0.000000008404
02000	200	0.000000005425
03000	100	0.000000001922

Table 1: 1st implementation error analysis

For each test case, the minimum running time of 10 data are chosen to be the actual running time. The results of the running time evaluation are listed as Table(2):

Firstly, how the computational time depends on N is required to be analysed. Though we have

Number of particles	Number of steps	User time/s	System time/s
00010	200	0.001	0.000
00100	200	0.033	0.000
00500	200	0.821	0.004
01000	200	3.776	0.008
02000	200	16.886	0.016
03000	100	19.515	0.044

Table 2: 1st implementation: total running time

known that our algorithm is of complexity of $\mathcal{O}(N^2)$, a regression analysis is conducted between user time and N^2 to validate this fact. The analysis excludes input file of $N = 3000$, because the time step the test is 100, which is different from other test cases. As Figure(1) shows, P value of the regression is extremely small, which indicates that the linear relation is extremely possible to exist. Also, as coefficient R squared shows, the regression function is extremely likely to be linear. We can conclude from the two facts that the the computational time depends on N^2 linearly.

Time analysis of each part After the analysis of total running time, we go further for the analysis of the time cost for each part of the program: to figure out which part of the program is the bottleneck of the running performance. The code mainly contains following parts: read data, allocate memory, update force matrix in each time step, update particle attributes in each time step and save result & release memory. As reading data and saving data are not costly and unavoidable, here remaining three parts are tested, including memory allocation time, sum of force matrix updating time and sum of particle attributes updating time. The test are based on the input file of $N = 2000$ and time step of 200. Other setting details are exactly the same as settings of previous error testing.

10 parallel tests are conducted and the results are shown as Table(3). Obviously, allocating memory does not take much time in the running process. Comparing with total running time analysis, the sum of means of force matrix update time and particle attribute update time almost equals to the total running time, while the standard deviations of both are relatively small. Therefore, it can be concluded that most of the running time are cost by the two inner loops, force matrix updating and particle attribute updating parts. These two parts should be

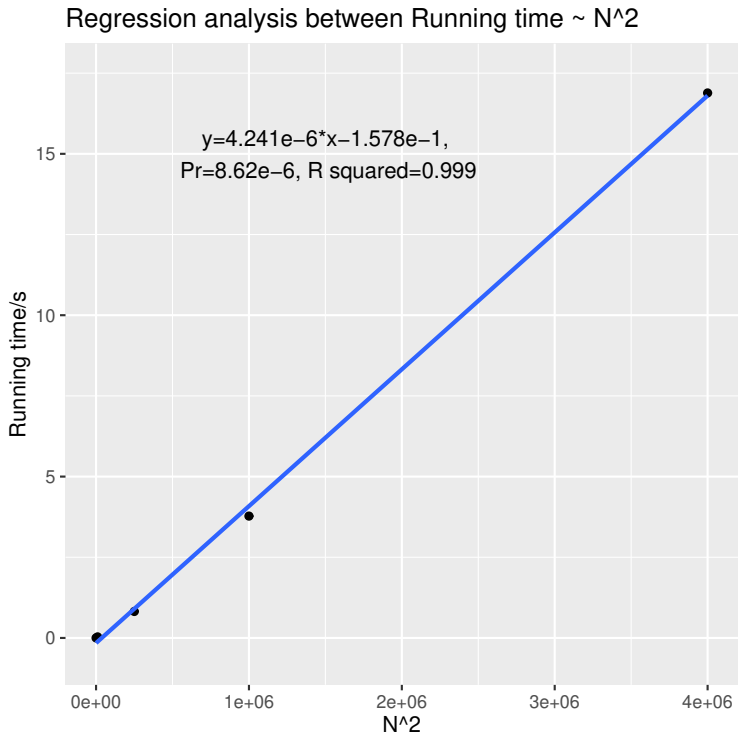

Figure 1: Regression between running time versus N^2.

mainly focused on during the program optimization.

Time item	Mean/ms	SD/ms	Min/ms
Memory allocation time	0.005	0.001	0.004
Force matrix update time	9276.443	251.625	9153.167
Particle attribute update time	8272.835	221.029	8127.255

Table 3: 1st implementation: running time of different parts

3 Serial Optimization

Based on the analysis before, serial optimization on the program is conducted.

Precision improvement attempt In the 1st implemented code, for the step of calculating acceleration of a particle, forces are accumulated. Accumulating floating may cause accumulated rounding error, which may reduce the precision of the simulation. To reduce rounding error, Kahan summation algorithm is applied.
However the algorithm did not improve the precision of our program. After our discussion, the possible reason is that: Kahan summation algorithm is suitable for summing a series of small value numbers to a big sum number. However, in this case, a great number of forces to be added are a series of both positive and negative numbers which have big absolute values comparing with the sum. Therefore, Kahan algorithm is not able to improve the precision of our simulation.

Algorithm Kahan Summation Algorithm

```
sum = 0.0, error = 0.0, temp = 0.0;
while i in length(double_array) do
    temp = sum + (double_array[i] - error);
    error = temp - sum - (double_array[i] - error);
    sum = temp;
end while
```

Turning on GCC optimization options To do less work, we have reduced function calling in inner loop, reduced computation by calculating half of the force matrix and avoided using denormalized numbers in the implementation. To improve the performance further, GCC optimization options are turned on. Optimization flags "-O2, -O3 and -Ofast" are tested and compared. Again, the test are under the same settings as we used in error analysis part and 10 parallel test for each test cases are conducted. The minimum time for in 10 parallel tests of each test case are chosen and listed as Table(4) shows.

Number of particles	Time step	No optimization/s	-O2/s	-O3/s	-Ofast/s
00010	200	0.001	0.001	0.001	0.001
00100	200	0.022	0.006	0.006	0.007
00500	200	0.821	0.192	0.187	0.171
01000	200	3.776	0.867	0.867	0.776
02000	200	16.886	5.254	5.252	4.884
03000	100	19.515	6.953	6.957	6.576

Table 4: User time of different GCC optimization options

As we can see from the results, turning on the GCC optimization options does improve the performance of the program greatly (reducing time cost more than 60%), without causing any precision loss. O2 and O3 options almost have the same performance, while Ofast option is slightly more efficient. Therefore, for all later testing and optimization, Ofast options are turned on.

Keyword optimization Keywords in C program are able to help compiler do more efficient optimization for the program. Generally keyword "constant" and "restrict" are two most commonly used ones. In our program, function are rarely used. Thus, restrict is not necessary to be introduced. Keyword "constant" are added to variables of number of particles and Δt. In our program, ϵ_0 and G are written in macro, thus they are not necessary to be rewritten. The test are under the same settings as that of error analysis part but with optimization flag "-Ofast". The results of the test are shown as Table(5) The results indicate that the time consumption does not vary much before and after the optimization. It can be concluded that adding keywords to variables does not help compiler improve the performance of our program.

Pipeline optimization In the Implementation part, we have stated that our first implementation follows the rule that do not use if statement in the inner loop, to make pipeline of CPU work more efficiently. Here valgrind is used to test whether our efforts do improve the performance. The program with input file of $N = 3000$ and time step is 100 is run by valgrind to check the CPU misprediction rate. Our program has a small misprediction rate of 0.1%. Therefore, it is for sure that CPU pipeline is used in an efficient way, without many cache misses.

Number of particles	Time step	No optimization/s	Keyword optimized/s
00010	200	0.001	0.001
00100	200	0.007	0.007
00500	200	0.171	0.170
01000	200	0.776	0.789
02000	200	4.884	4.856
03000	100	6.576	6.573

Table 5: User time of keyword optimized and non-keyword optimized program

For CPU pipeline optimization, another available way is to fuse loops. In our program, there are two inner loops which can be fused. However, the second inner loop, in which accelerations are calculated, depends on the first loop, in which force matrix is updated. To fuse two loops, double computation should be done, rather than computing a half of the force matrix. This is a trade-off between "doing less work" and "doing the work faster". Therefore, two approaches are compared and the results are shown as Table(6). Both program are compiled with optimization flag of "-Ofast". From results we can state that separated loop with less computation works

Number of particles	Time step	separated loop/s	fused loop/s
00010	200	0.001	0.001
00100	200	0.007	0.006
00500	200	0.171	0.117
01000	200	0.776	0.470
02000	200	4.884	1.879
03000	100	6.576	2.111

Table 6: User time of programs containing separated or fused loop

works slower than program with fused loop and doubled computation cost. This indicates that for nested loops over large numbers, much time are cost on repetitively visiting variables, rather than computing process. Therefore, all afterwards further testing and optimization are based on the program with fused loop.