High Performance Programming
Uppsala University – Spring 2020
Report for Assignment 4 by Team xx

Li Ju, Shu-pei Huang

10th March 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;

4. Barnes-Hut approximation: For many problem settings, the number of operations required for computing the force in the N-body problem can be substantially reduced by taking advantage of the idea that the force exerted by a group of objects on object i can be approximated as the force exerted by one object with mass given by the total mass of the group located at the center of gravity of the group of objects.

3 Simulation Settings

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

4 Computer Details

To be added...

II Serial Implementation, Analysis and Optimization

1 Implementation

For assignment 4, the simulation is implemented serially.

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

Algorithm Gravitation Simulation: main function

Require: Given valid sufficient parameters
Ensure: Valid data can be read from input file
  Save all particle data to an array of struct particle
  while max step not reached **do**
    Construct a quadtree for all particles
    for each in particles **do**
      Calculate acceleration of current particle with information from quadtree;
      Update the particle information in particle array: velocity and position;
    end for
    release all quadtree sources
  end while
  Save final status information of particles to a result file.

Algorithm Gravitation Simulation: quatree node insertion

Require: particle mass, particle position and tree to be inserted on
  if current node is empty **then**
    Create a new node and save particle mass and positions in
  else if current node is leaf node **then**
    Create and insert two nodes, one for particle which are from current leaf node, another for the input particle;
    Status of current node is set to be non-leaf
    Mass of current node is set to be the sum of two particles
    Position of current node is set to be the center of the grid
  else
    Insert the input particle to the children of current node
    Mass of input particle is added to mass of current ndoe
  end if

Algorithm Gravitation Simulation: acceleration calculation

Require: Node n, particle p on which forces are exerted, variable acceleration
  Calculate the distance between particle p and node n
  if node n is not leaf node **then**
    if θ is less than constant THETA **then**
      Approximation is applied and calculate the acceleration of particle p exerted by node n
    else
      Call this function for each child of node n
    end if
  else if distance is less than 10^{-10} **then**
    Calculate the acceleration of particle p exerted by leaf node n, which is not particle p itself
  end if

Algorithm analysis Comparing with naive algorithm for galaxy simulation, here Barnes-Hut approximation is applied by using a quadtree. The complexity of naive algorithm is of $\mathcal{O}(N^2)$, while the complexity of Barnes-Hut algorithm is of $\mathcal{O}(N \log N)$.
However it is important to know that though Barnes-Hut algorithm is of a lower complexity, it is not guaranteed that Barnes-Hut approximation cost less time than naive algorithm. Several reasons exist:

1. For each step, a function is called, which takes much computational cost, while in naive algorithm only a float multiplying is needed for each step;

2. Inside each called function, a series of if statements are used, to check if particles in this node could be approximated as a group. As we know, if statement takes more clock cycles, especially when they are mis-predicted.

However, it can be concluded that if the number of particle increase, the cost of Barnes-Hut algorithm increase at a lower speed comparing with naive algorithm.

2 Performance Analysis

Error control To control the error comparing with reference output data, constant THETA requires optimization. Here is assignment 4, the simulation error is supposed to be less than 10^{-3}. Several tests are conducted with following details:
[**input files**]: *ellipse_N_02000.gal*;
[**time step**]: *200*;
[**GCC flags**]: *-lm* only;
[**Optimization options**]: No optimization flag;
[**computer details**]: shown in part I.

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

Number of particles	Number of steps	THETA	Position maxdiff
2000	200	0.50	0.005337273260
2000	200	0.30	0.002505661751
2000	200	0.20	0.000410095951
2000	200	0.25	0.000885759450
2000	200	0.26	0.001152830565

Table 1: Assignment 4: error control

The results indicate that when the constant THETA is at 0.25, the error is controlled within 10^{-3}. Therefore, THETA is chosen as 0.25 for afterwards tests and optimization.

Total running time evaluation To evaluate the total running time of the simulation program, tests are conducted with the chosen constant THETA. Test cases are listed as following. For each test case, 10 times' parallel running are conducted, to minimum the random testing error. For each test case, the minimum running time of 10 data are chosen to be the actual running time.
[**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, 200*;
[**GCC flags**]: *-lm*;
[**Optimization options**]: No optimization flag;
[**computer details**]: shown in part I.

The results of the running time evaluation are listed as Table(2):

Number of particles	Number of steps	User time/s	System time/s
00010	200	0.002	0.000
00100	200	0.053	0.000
00500	200	0.544	0.008
01000	200	1.543	0.008
02000	200	4.071	0.024
03000	200	7.211	0.032

Table 2: Assignment 4: total running time

Firstly, how the computational time depends on N is required to be analysed. Though we have known that our algorithm is of complexity of $\mathcal{O}(N \log N)$, a regression analysis is conducted between user time and $N \log N$ to validate this fact. As Figure(1) shows, P value of the regression is relatively small, which indicates that the linear relation is 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 $N \log N$ linearly.

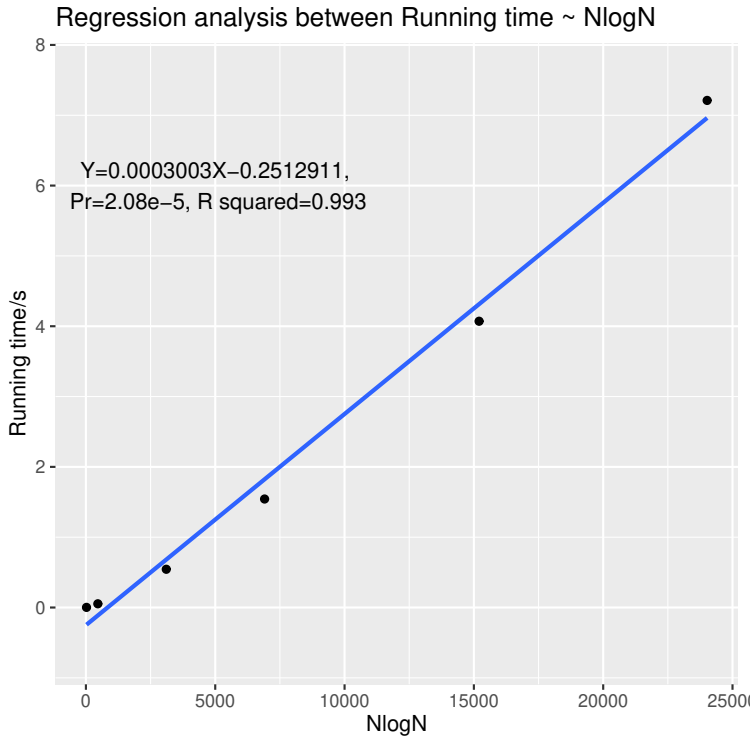

Figure 1: Regression between running time versus $N \log N$.

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, create quadtree in each time step, calculate the acceleration and 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 creating quadtree and calculate the acceleration. 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). As we cab see from the table, tree creation does not take much time in the running process. Comparing with total running time analysis, almost all time cost are on recursively calculating the accelerations. Therefore, this function is the main part that should be mainly focused on during the program optimization. Also tree creati0on also takes around 10% of total time cost. Optimization may also be required.

Time item	Mean/ms	Min/ms
Tree creation time	244	243
Particle array update time	3944	3892

Table 3: Assignment 4: running time of different parts

3 Serial Optimization

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

Optimization on GCC optimization options To improve the performance, GCC optimization options are turned on. Optimization flags "-O1, -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	-O1/s	-O2/s	-O3/s	-Ofast/s
00010	200	0.002	0.002	0.002	0.002
00100	200	0.053	0.052	0.049	0.042
00500	200	0.544	0.542	0.526	0.526
01000	200	1.543	1.522	1.504	1.511
02000	200	4.071	4.054	4.021	4.044
03000	200	7.211	7.151	7.142	7.108

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 slightly (reducing time cost around 1%), without causing any precision loss. All flags almost have the same performance, while O3 option is slightly more efficient. Therefore, for all later testing and optimization, O3 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 tests, both keywords are introduced. Keyword "constant" are added to proper variables and function parameters. The test are under the same settings as that of error analysis part but with optimization flag "-O3t". The results of the test are shown as Table(5) The results indicate that the time consumption does not vary much before and after

Number of particles	Time step	No optimization/s	Keyword optimized/s
00010	200	0.002	0.002
00100	200	0.049	0.046
00500	200	0.526	0.543
01000	200	1.504	1.528
02000	200	4.021	4.074
03000	100	7.142	7.183

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

the optimization. It can be concluded that adding keywords to variables does not help compiler improve the performance of our program.