High Performance Programming
Uppsala University – Spring 2020
Report for Assignment 3-6 by Team 48

Li Ju, Shu-Pei Huang

22nd 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:

a) $G = 100/N$

b) $\epsilon_0 = 10^{-3}$

c) $\Delta t = 10^{-5}$

d) Accuracy Control: $Error < 10^{-3}$

4 Computer Details

For assignment 3 and 4, the tests are conducted on the virtual host of AWS. The detailed information of the computer is listed below:

a) Operating system: Ubuntu 18.04.3 LTS

b) CPU model: Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz

c) Number of virtual CPU(s): 2

d) CPU MHz: 2499.992

e) Compiler: gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1 18.04.1)

For assignment 5 and 6, because the virtual machine of AWS is limited to 2-core, to do better multi-core test, tests are conducted on the host *arrhenius.it.uu.se*. The detailed information of the computer is listed below:

a) Operating system: Ubuntu 18.04.4 LTS

b) CPU model: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz

c) Number of CPU(s): 16

d) CPU MHz: 1600.180

e) Compiler: gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1 18.04.1)

II Solution

1 Assignment 3: Naive Algorithm

Algorithm design
For assignment 3, the simulation is implemented serially with naive algorithm. The algorithm is designed and shown as Algorithm: Naive Gravitation Simulation.

Algorithm Naive Gravitation Simulation

Require: Given sufficient and valid 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.

Data structure
To implement the naive algorithm, a data structure named **particle** is defined, to save the information of each particle, including **position_x**, **position_y**, **mass**, **velocity_x**, **position_y** and **brightness**. Also a structure named **force** is defined, to store the 2-dimensional forces in, containing **x** and **y**.

Algorithm analysis
The time complexity of the algorithm is of $\mathcal{O}(N^2)$, because the largest computation cost is required in the step of resultant force of each particle being 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.

Algorithm optimization
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 is able to lower the computation cost by reducing the redundant calculation at algorithm-level. Therefore, the following 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 act as 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 introduces 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 Assignment 4: Barnes-Hut's algorithm (serial)

Algorithm design
For assignment 4, the simulation is implemented serially with Barnes-Hut's algorithm. The pseudo code of the algorithm is shown as Algorithm BH Gravitation Simulation: main function, Algorithm BH Gravitation Simulation: quatree node insertion and Algorithm BH Gravitation Simulation: acceleration calculation.

Algorithm BH 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.

Data structure
Except for the data structure **particle**, which have been described in Assignment 3, two more data structures are defined, including **node** for quadtree node storage and a 2-dimensional vector structure **vec2d**.
node contains a char **is_leaf** indicating the node is leaf node or not, 2 **vec2d**s for the boundary

Algorithm BH 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 BH 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

of the node, 1 **vec2d** for the mass centre of the node, 1 double for the mass of particles in the node and 4 node pointers pointing to the children of the node.
vec2d contains two double **x** and **y** to represent two coordinates of a 2-dimensional vector.

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.

Still, we can conclude that if the number of particles increase, the cost of Barnes-Hut algorithm increase at a lower speed comparing with naive algorithm.

3 Assignment 5: Barnes-Hut's algorithm (parallel with pthread)

Algorithm design
For assignment 5, Barnes-Hut's algorithm is implemented parallel with pthread. There are two main tasks in the algorithm for each time step, including task 1: tree creation and task 2: acceleration calculation & particle information update. In our algorithm, task 1 is done in main thread and task 2 is done in several threads (noted as thread group **para**) parallel. The control flow between main thread and **para** is nested: each time main thread finished tree creation, it "tells" **para** to do acceleration calculation and waits for them. When all threads in **para** finish acceleration calculation, they pass the token to main thread and wait for it. In our implementation, instead of creating new threads and destroying threads (which are cost-consuming) for each time step, the whole work are done in the as-created threads and nested barrier and condition are used.
The as-designed algorithm is shown as Algorithm BH Gravitation Simulation: pthread-acce and BH Gravitation Simulation: pthread-main.

Data structure
Except for the data structure used in Assignment 4, a struct named **threaddata** is used to pass data to thread function. In **threaddata**, there are integers for thread id, number of threads in **para**, number of particles, doubles for Δt and θ, address of the tree and particle array.

4 Assignment 6: Barnes-Hut's algorithm (parallel with OpenMP)

Algorithm design
For assignment 6, Barnes-Hut's algorithm is implemented parallel with OpenMP. Based on code of Assignment 4, for each time step, there are two main parts, tree creation and acceleration calculation & particle information update. It is hard to create tree parallel because each node

Algorithm BH Gravitation Simulation: pthread-acce

Require: Paremeters from the main thread, including tree address, total number of threads, total number of particles, Δt, θ and address of particle array
  while 1 **do**
    Lock the mutex and wait for the condition
    if end == 1 **then**
      Unlock the mutex and return
    end if
    unlock the mutex
    Calculate the acceleration and update particle information
    Start barrier wait
  end while

Algorithm BH Gravitation Simulation: pthread-main

Require: Given valid sufficient parameters
Ensure: Valid data can be read from input file
  Save all particle data to an array of struct particle
  Create the quadtree
  Create all threads of **para**
  Lock the mutex
  if All threads are created and waiting **then**
    Broadcast the condition and release mutex
  end if
  while step is less than max step **do**
    Start barrier wait
    while 1 **do**
      if All threads in **para** are prepared and waiting **then**
        Break the loop
      end if
    end while
    Release the tree and re-create the tree
    Broadcast all threads in **para** and unlock the mutex
  end while
  Start barrier wait
  if All threads in **para** are prepared and waiting **then**
    Set end=1
  end if
  Join all threads in **para**
  Save the data in file
  Release all resources

can only be inserted after the previous nodes have been added. Therefore, only the second part can be done parallel. Here OpenMP is used for parallelization for this part, in which both acceleration calculation and particle information update are done.

III Performance Analysis & Optimization

1 Assignment 3: Naive Algorithm

Total running time evaluation
To evaluate the total running time of the simulation program, test cases and details are shown as follown. For each test case, 10 times' 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. The results of the running time evaluation are listed as Table(1):
[**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, 100*;
[**GCC flags**]: *-lm, -std=c99*;
[**Optimization options**]: No GCC optimization flags are used;

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 1: Assignment 3: 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 $\mathcal{O}(N^2)$, 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 $N = 2000$

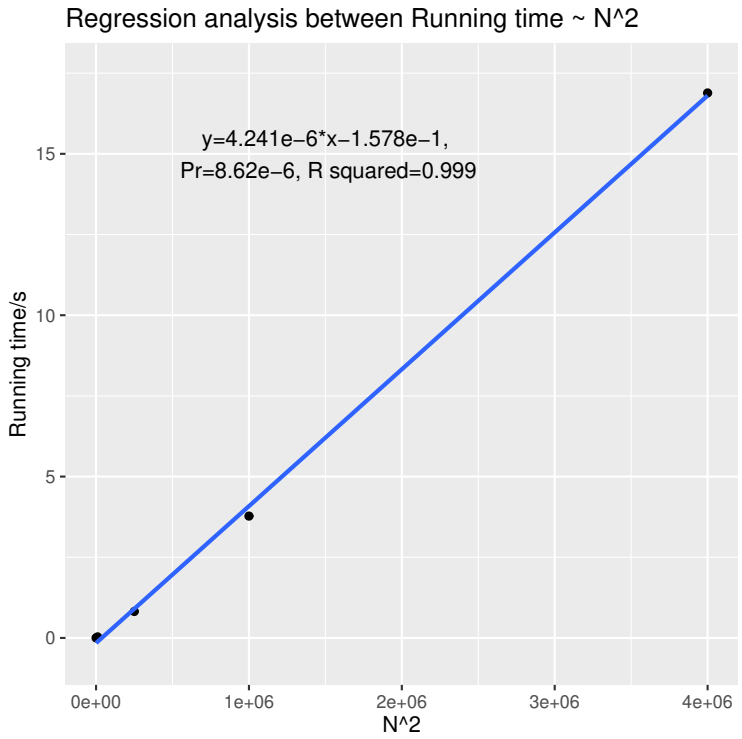

Figure 1: Regression between running time versus N^2.

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(2). 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 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 2: 1st implementation: running time of different parts

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(3) 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 3: 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(4) The results indicate that the time consumption does not vary much before

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 4: User time of keyword optimized and non-keyword optimized program

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.

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(5). 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 5: 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.

2 Assignment 4: Barnes-Hut's algorithm (serial)

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;

The result of error control test is shown as Table(6):
The results indicate that when the constant THETA is at 0.25, the error is controlled within

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 6: Assignment 4: error control

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, -std=c99*;
[**Optimization options**]: No optimization flag;
The results of the running time evaluation are listed as Table(7):

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 7: 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(2) 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.

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(8). 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 creation also takes around 10% of total time cost. Optimization may also be required.

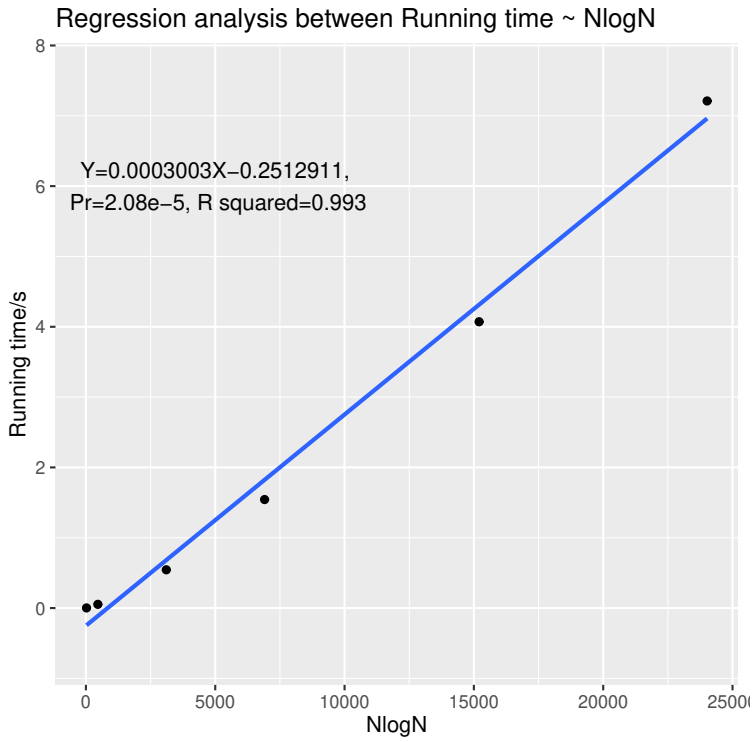

Figure 2: Regression between running time versus $N \log N$.

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

Table 8: Assignment 4: running time of different parts

Turning 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(9) shows.

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 "-O3". The results of the test are shown as Table(10) The results indicate that the time consumption does not vary much before and after the optimization. It can be

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 9: User time of different GCC optimization options

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 10: User time of keyword optimized and non-keyword optimized program

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

3 Assignment 5: Barnes-Hut's algorithm (parallel with pthread)

Time analysis
In assignment 5, the Barnes-Hut's algorithm is implemented parallel with pthread. To figure the best performance and how multithreads code can improve the performance, several test cases are conducted on the host *arrhenius.it.uu.se* as described in Introduction part.All the test cases are listed as following. For each test case, 3 times' parallel running are conducted, to minimum the random testing error and the minimum real time of 10 data are chosen to be the actual running time.
[**input files**]: *ellipse_N_00100.gal, ellipse_N_01000.gal, ellipse_N_05000.gal, ellipse_N_10000.gal*;
[**time step**]: *200, 200, 200, 200*;
[**number of threads**]: *1, 2, 4, 6, 8, 10, 12, 14, 16* for each input file;
[**GCC flags**]: *-lm, -std=c99, -lpthread*;
[**Optimization options**]: *-Ofast*;

With Table(11), a speedup versus number of threads figure is plotted. Speedup is defined as $speedup = T_{old}/T_{new}$. As Figure(3) shows, all the speedup point is lower than idea speedup. Also, when the number of particles N is small, the increase of number of threads will not significantly improve the time cost of the code, while the number of particles N is large, the speedup increase quickly as number of threads increase. More precisely, if $N = 10000$ is picked up as the example, we can see from the figure that the speedup keeps increasing as the number of threads increase, from 65s to 10s, indicating that more than 6 times speed improve. However, when the threads is larger than 16, the speedup decrease slightly. Similar problem arises when $N = 5000$.

#threads	N=100	N=1000	N=5000	N=10000
01	0.240	3.501	28.231	66.115
02	0.216	2.131	15.723	35.391
04	0.168	1.330	8.960	19.886
06	0.181	1.118	6.788	14.833
08	0.162	1.110	6.038	14.431
10	0.177	0.957	6.363	13.314
12	0.163	0.894	5.700	11.936
14	0.195	0.844	5.242	10.869
16	0.173	0.805	4.650	10.399
18	0.185	0.966	5.478	11.321

Table 11: Time cost of number of threads with different N

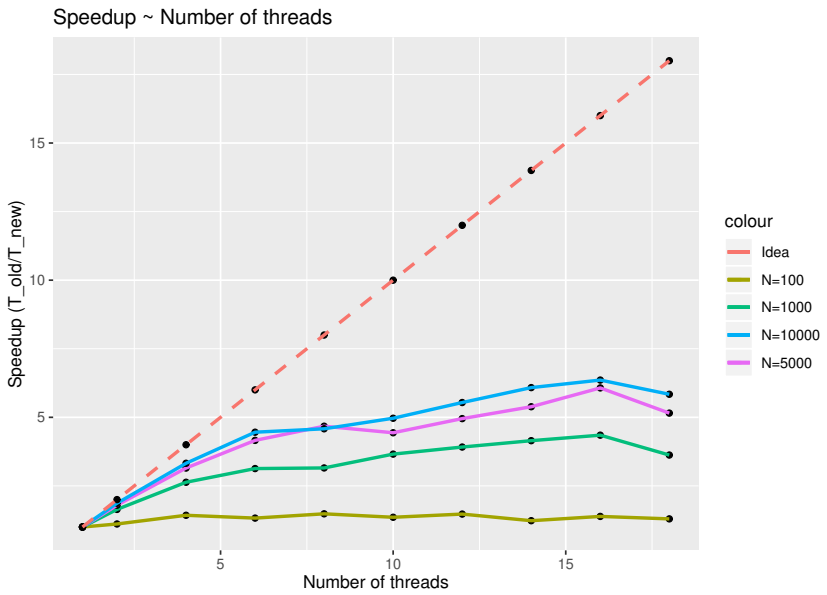

Figure 3: Speedup versus number of threads with different N (pthread)

Two piece of information can be inferred from the figure:

1. When N is relatively small, the cost increase of more number of threads is similar to the cost reduction of more threads contributed, this is why increasing number of threads does not significantly reduce the time cost/increase speedup. This is also the reason why all speedup is less than idea speedup;

2. The optimal number of threads for large N is approximately 16, which is the number of the physical core of the computer. When the number of the threads exceed the number of core, multiple threads will be assigned to a single CPU core, which might reduce the performance slightly.

Load balance analysis
In our implementation, all the acceleration calculation and particle array update work are assigned to different threads equally, and the load on every thread should be balanced. Therefore, there is no need to do further work to balance the load between threads.

4 Assignment 6: Barnes-Hut's algorithm (parallel with OpenMP)

Time analysis

In assignment 6, the Barnes-Hut's algorithm is implemented parallel with OpenMP. Like Assignment 5, several test cases are conducted on the host *arrhenius.it.uu.se* as described in Introduction part.All the test cases are listed as following. For each test case, 3 times' parallel running are conducted, to minimum the random testing error and the minimum real time of 10 data are chosen to be the actual running time.
[**input files**]: *ellipse_N_00100.gal, ellipse_N_01000.gal, ellipse_N_05000.gal, ellipse_N_10000.gal*;
[**time step**]: *200, 200, 200, 200*;
[**number of threads**]: *1, 2, 4, 6, 8, 10, 12, 14, 16* for each input file;
[**GCC flags**]: *-lm, -std=c99, -fopenmp*;
[**Optimization options**]: *-Ofast*;

#threads	N=100	N=1000	N=5000	N=10000
01	0.207	3.135	27.018	65.025
02	0.329	1.816	14.970	35.187
04	0.150	1.070	8.408	19.791
06	0.162	0.868	7.857	14.888
08	0.138	0.889	6.961	12.751
10	0.155	0.787	6.169	13.810
12	0.125	0.829	5.791	12.213
14	0.140	0.739	4.726	11.030
16	0.183	1.546	4.377	10.325
18	0.174	0.883	5.268	11.384

Table 12: Time cost of number of threads with different N

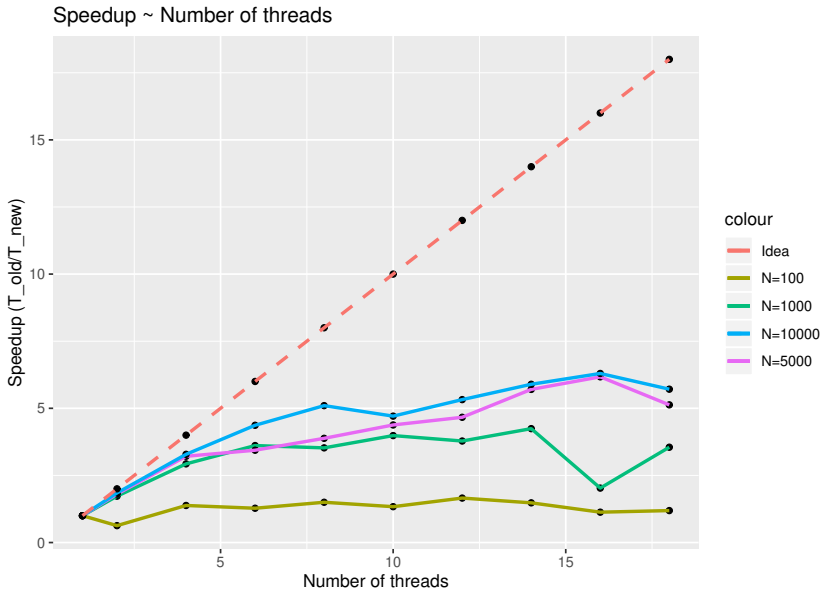

Figure 4: Speedup versus number of threads with different N (OpenMP)

With Table(12), a speedup versus number of threads figure is plotted. As Figure(4) shows, all the speedup point is lower than idea speedup. Also, like parallelization with pthread, when the number of particles N is small, the increase of number of threads will not significantly improve the time cost of the code, while the number of particles N is large, the speedup increase quickly as number of threads increase. Again, if $N = 10000$ is picked up as the example, we can also see from the figure that the speedup keeps increasing as the number of threads increase, from 65s to 10s, indicating that more than 6 times speed improve. However, when the threads is larger than 16, the speedup decrease slightly. Similar problem arises when $N = 5000$.
The conclusions for Assignment 5's parallelization is similar, that with larger N, optimal number of threads is better to be figured and the optimal number is the number of physical cores of the host.
Comparing with pthread, the performance of OpenMP is almost the same as that of pthread, thought there is a slight unsteady on OpenMP. A possible cause is that, OpenMP is highly abstract to be suitable for any situations, while pthread used here is specifically for this program, with a specific control flow and data protection, therefore, it is more predictable and steady here in this program.

Load balance analysis
In our implementation, all the acceleration calculation and particle array update work are assigned to different threads equally, and the load on every thread should be balanced. Therefore, there is no need to do further work to balance the load between threads.

5 Conclusion

In the project, the galaxy with multiple particles is simulated, with both naive algorithm and Barnes-Hut's algorithm. Performance analysis are conducted and based on the performance results, optimization on the program is also further conducted, including compiler flags optimization, pipeline optimization, keyword optimization and parallelization with pthread and OpenMP. All the techniques of serial and parallel optimization improves the performance of the greatly with speedup of 6 and 6, respectively. Therefore, in total, the program is able to produce simulation with tolerable error 36 times faster than naive algorithm. The program is able to do galaxy simulation in either single core or multi-core computer in an efficient and precise way, by adjusting the number of threads and θ with respect to target error tolerance.

Contribution
Li Ju and Shu-Pei Huang contribute equally to the project, including algorithm design, code implementation and report.