CS433 Parallel and distributed programming

N-body simulation

Tianhao Li 5140309349
Chao Gao 5142029014

2016.12.1

1. Introduction

An N-body simulation approximates the motion of particles, often specifically particles that interact with one another through some type of physical forces. The input to the problem is the mass, position, and velocity of each particle at the start of the simulation, and the output is typically the position and velocity of each particle at a sequence of user-specified times.
First we have a serial N-body solver, and in this project we'll try to parallelize it by using OpenMP.

2. Environment

CPU: I7-4700HQ 4 cores 8 threads
Operating System: Linux Mint 18
Compiler: gcc 5.0
Editor: Atom

3. File Structure

There are 9 program files in our projects.
version0 **original**
- nobody.c

version1 **basic**
- version1_1.c ———— basic OpenMP version(naive way)
- version1_2.c ———— body slimming version
- version1_3.c ———— blocked version

version2 **two-world**
- version 2_1.c ———— basic version
- version 2_2.c ———— body slimming version
- version 2_3.c ———— blocked version

version3 **Barnes-Hut**
- version 3_1.c ———— serial version
- version 3_2.c ———— parallel version

4. Implementation

We mainly modify the core function “**position_step**”, which calculates the forces of bodies and updates the position of bodies.

4.1 naive way

The very naive thinking is that we just add OpenMP parallel directives directly to the original program.Besides, we set the num of threads in the main function.

```
#pragma omp parallel for
```

```
omp_set_num_threads(threads);
```

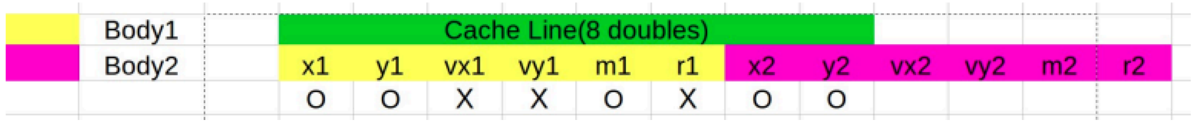
```
#pragma omp parallel for
    for (int i = 0; i < world->num_bodies; i++) {
        double d, d_cubed, diff_x, diff_y;
        for (int j = 0; j < world->num_bodies; j++) {
            if (i == j) {
                continue;
            }
            // Compute the x and y distances and total distance d between
            // bodies i and j
            diff_x = world->bodies[j].x - world->bodies[i].x;
            diff_y = world->bodies[j].y - world->bodies[i].y;
            d = sqrt((diff_x * diff_x) + (diff_y * diff_y));
            if (d < 25) {
                d = 25;
            }
            d_cubed = d * d * d;
            // Add force due to j to total force on i
            force_x[i] += GRAV * (world->bodies[i].m * world->bodies[j].m
                                  / d_cubed) * diff_x;
            force_y[i] += GRAV * (world->bodies[i].m * world->bodies[j].m
                                  / d_cubed) * diff_y;
        }
    }
// Update the velocity and position of each body
#pragma omp parallel for
    for (int i = 0; i < world->num_bodies; i++) {
        world->bodies[i].vx += force_x[i] * time_res / world->bodies[i].m;
        world->bodies[i].vy += force_y[i] * time_res / world->bodies[i].m;

        world->bodies[i].x += world->bodies[i].vx * time_res;
        world->bodies[i].y += world->bodies[i].vy * time_res;
    }
```

Therefore, now OpenMP has parallelized the for loops which computes the forces and updates the position of bodies. Code is as above.
However, at first, the problem occurs that the parallelized version does not have a better performance. Firstly, we guess that the reason is the existence of the barrier, all threads have to wait until all are finished. This is not well parallelized. But, after our deep thinking, the execution time of each thread is almost the same because of the **load balancing**. Finally, we figure out that in the for loop, we should use the local j by simply using **int j** in the loop. Because if we use the j as the global variable, for every thread to use j, it becomes a **critical section**, which will cause non-necessary waiting cost.

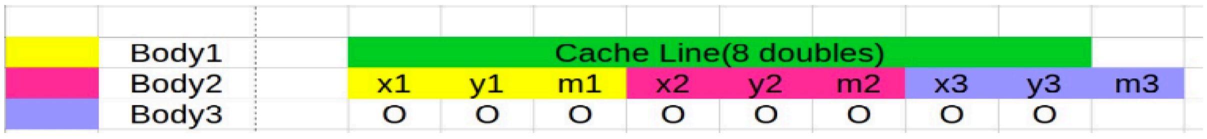
4.2 body slimming version

The first optimization we think about is to reduce the number of variables in the **body struct**. Because this can optimize reading data, at one time we can read more data, so that we can increase the hit rate because we access data continuously. When we parallel the position_step function, we see that r, vx, vy are not necessarily needed in the struct, so we take them out. Or when the cache miss occurs, we should load all the data which contains lots of unnecessary data. Below is the schematic diagram.
First, we load Body1, X represents the element is of no use. And next time if we want to access m2, there is a cache miss.



🟨	Body1		Cache Line(8 doubles)											
🟪	Body2		x1	y1	vx1	vy1	m1	r1	x2	y2	vx2	vy2	m2	r2
			O	O	X	X	O	X	O	O				

But after modification, we get a cache until we access m3.



🟨	Body1		Cache Line(8 doubles)									
🟥	Body2		x1	y1	m1	x2	y2	m2	x3	y3	m3	
🟦	Body3		O	O	O	O	O	O	O	O		

As for the code, Just some changes to the declaration and the update of the velocity. The others are the same as the first version. The program is as below.

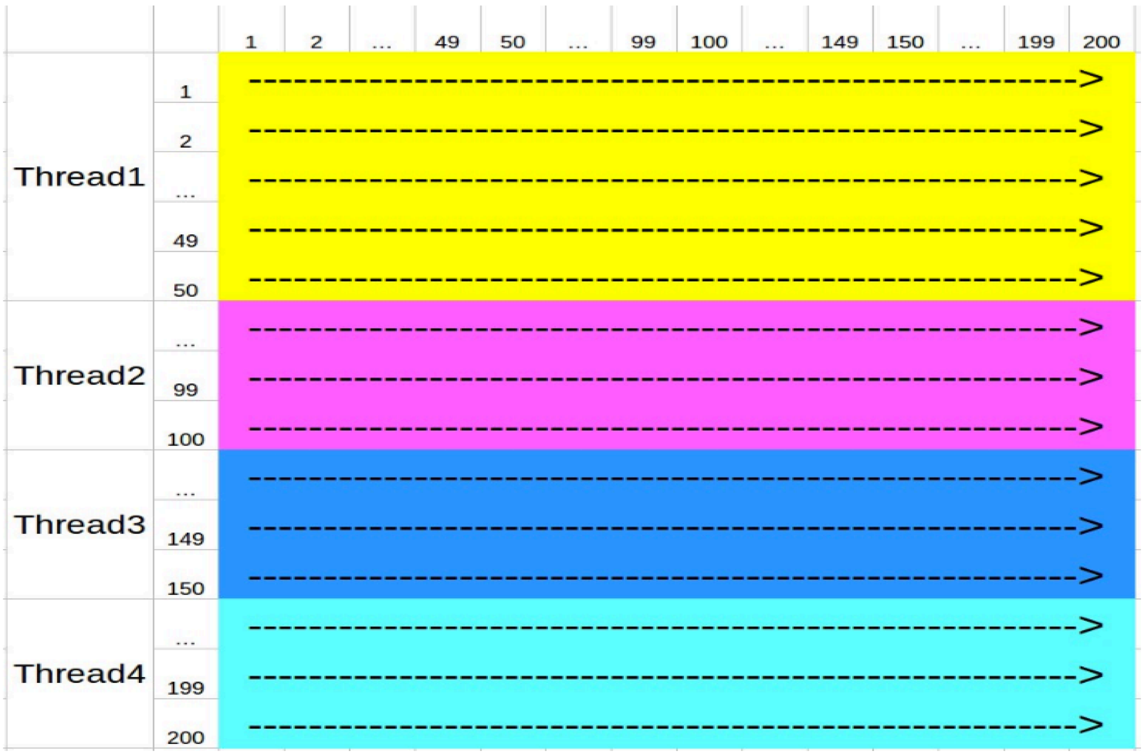
```
struct body {
    double x, y; // position
    double m; // mass
};
struct speed{
    double vx,vy;
};
struct speed* speed;
double * radius;

// Update velocities
speed[i].vx += force_x[i] * time_res / world->bodies[i].m
speed[i].vy += force_y[i] * time_res / world->bodies[i].m;
```

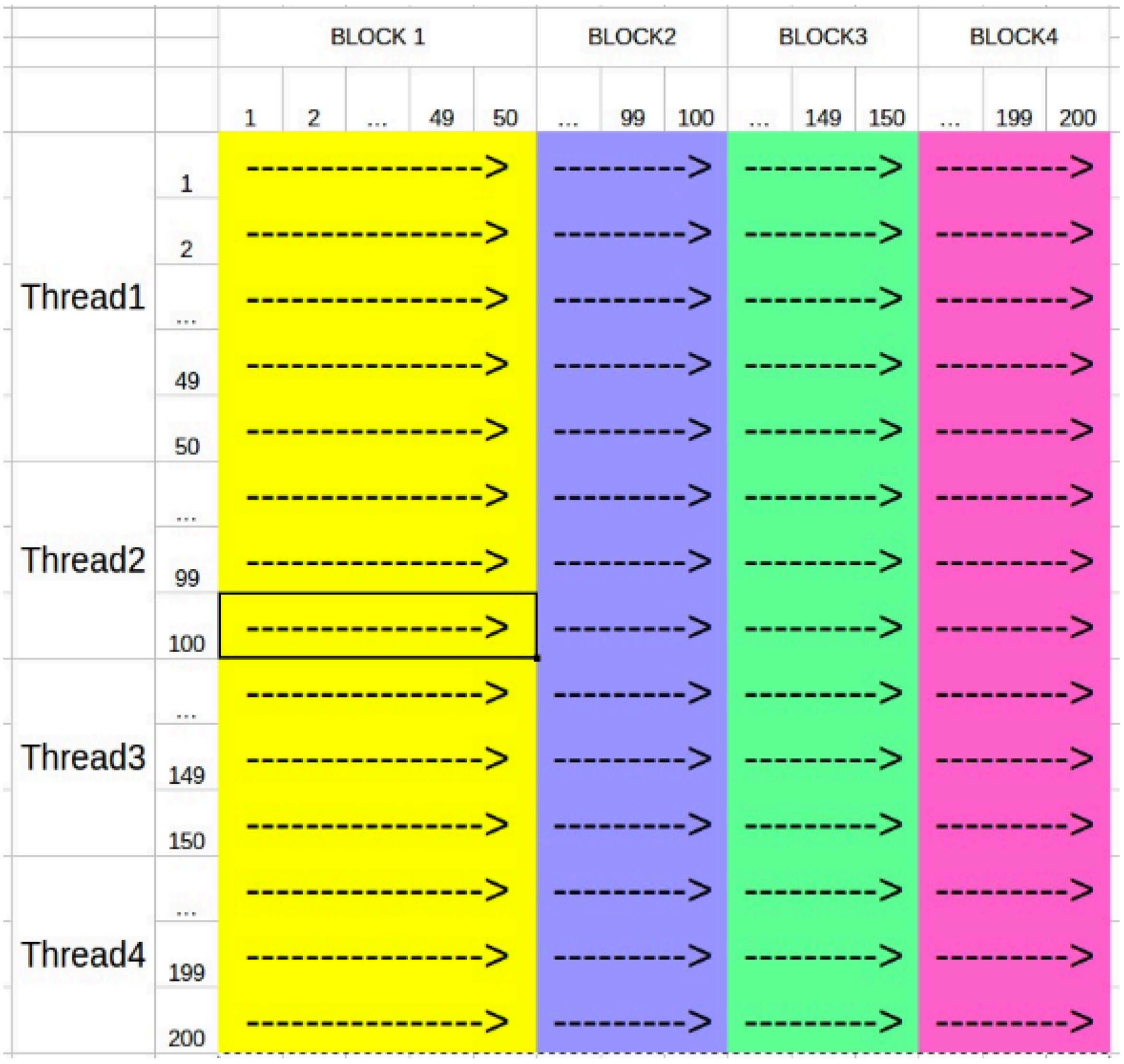
4.3 blocked version

The second optimization is the blocked version. The cache can not load all the bodies at one time, so we can divide the body array into several blocks. Therefore, according to the principle of locality, we can have a more centralized access, which will reduce the times of memory access.

Below is the schematic diagram



After dividing into 4 blocks, each thread accesses the blocks in the order of Block1, Block2, Block3, Block4.



		BLOCK 1					BLOCK2			BLOCK3			BLOCK4		
		1	2	...	49	50	...	99	100	...	149	150	...	199	200
Thread1	1	--------------->					--------->			--------->			--------->		
	2	--------------->					--------->			--------->			--------->		
	...	--------------->					--------->			--------->			--------->		
	49	--------------->					--------->			--------->			--------->		
	50	--------------->					--------->			--------->			--------->		
Thread2	...	--------------->					--------->			--------->			--------->		
	99	--------------->					--------->			--------->			--------->		
	100	--------------->					--------->			--------->			--------->		
Thread3	...	--------------->					--------->			--------->			--------->		
	149	--------------->					--------->			--------->			--------->		
	150	--------------->					--------->			--------->			--------->		
Thread4	...	--------------->					--------->			--------->			--------->		
	199	--------------->					--------->			--------->			--------->		
	200	--------------->					--------->			--------->			--------->		

The changes to the position_step function is the for loop calculating the forces. We add a for loop for dividing blocks. The program is as below.

```
int BLOCK_SIZE = world->num_bodies / BLOCK_NUM;
int my_rank = omp_get_thread_num();
int num_of_threads = omp_get_num_threads();
int thread_size = world->num_bodies / num_of_threads;
for (int k = 0; k < BLOCK_NUM; ++k) {
    for (int i = my_rank * thread_size; i < (my_rank + 1) * thread_size; i++) {
        double d, d_cubed, diff_x, diff_y;
        for (int j = BLOCK_SIZE * k; j < BLOCK_SIZE * (k + 1); j++) {
            if (i == j) {
                continue;
              }
              // Compute the x and y distances and total distance d between
              // bodies i and j
              diff_x = world->bodies[j].x - world->bodies[i].x;
              diff_y = world->bodies[j].y - world->bodies[i].y;
              d = sqrt((diff_x * diff_x) + (diff_y * diff_y));

              if (d < 25) {
                       d = 25;
               }
              d_cubed = d * d * d;
               // Add force due to j to total force on i
              force_x[i] += GRAV * (world->bodies[i].m * world->bodies[j].m
                                         / d_cubed) * diff_x;
              force_y[i] += GRAV * (world->bodies[i].m * world->bodies[j].m
                                         / d_cubed) * diff_y;
          }
      }
 }
```

4.4 two-world version

This is totally a new idea from the previous version. Let's recall the naive version, we parallelize the force calculation and the update of position. However, there is a barrier between this two operations, that is to say, we have to wait for all threads to finish calculating the force, after that, we can start to update the body position.

Now we have an idea without waiting. We can combine the two operations together. We can create **another world** to store the information. This is like Frame Buffer which we have learned in the computer graphics.
If we are now in the first world, we update the position to the second world. And other threads can calculate the force according to the position information in the first world. And in the next iteration, we can use the position in the second world to calculate the force, and update the position to the first world.

```
 if(iteration%2==0){
     diff_x = world->bodies[j].x - world->bodies[i].x;
     diff_y = world->bodies[j].y - world->bodies[i].y;
     }
     else{
     diff_x = world->bodies[j].x_back - world->bodies[i].x_back;
     diff_y = world->bodies[j].y_back - world->bodies[i].y_back;
   }
```

```
if (iteration%2== 0) {
   // Update positions
    world->bodies[i].x_back = world->bodies[i].x + world->bodies[i].vx * time_res;
    world->bodies[i].y_back = world->bodies[i].y + world->bodies[i].vy * time_res;
    } else {
      world->bodies[i].x = world->bodies[i].x_back + world->bodies[i].vx * time_res;
      world->bodies[i].y = world->bodies[i].y_back + world->bodies[i].vy * time_res;
}
```

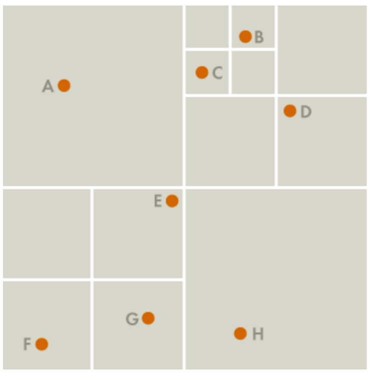
Therefore, now there is no data dependency, we don't have to wait.

The same as the above three versions, the **two-world version** also has three versions, that is basic version, body-slimming version, blocked version. The implementation is similar. At this time we don't have a repeated elaboration.

4.5 Barnes-Hut algorithm

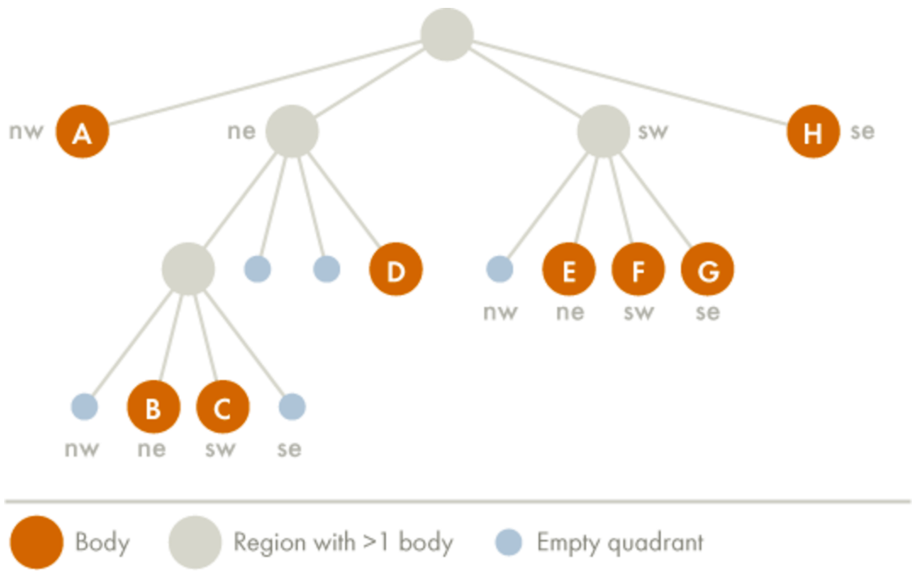
This optimization is on the algorithm level, not like the cache level as previous version shows.
The main reference:
http://blog.csdn.net/baimafujinji/article/details/53036473
https://github.com/robmdunn/nbody

Barnes-Hut algorithm is an algorithm using the physical conception **centroid.** The implementation is based on the quad-tree. And the operations related are constructing the tree, inserting a body and calculate the force.



divide the space into four parts.

every leaf node in the tree represents the single body, the rest nodes represents groups of bodies, including the total mass and centroid.



Constructing the tree

1. If node x does not contain a body, put the new body b here.
2. If node x is an internal node, update the center-of-mass and total mass of x.
3. Recursively insert the body b in the appropriate quadrant.

If node x is an external node, say containing a body named c, then there are two bodies b and c in the same region. Subdivide the region further by creating four children. Then, recursively insert both b and c into the appropriate quadrant(s). Since b and c may still end up in the same quadrant, there may be several subdivisions during a single insertion. Finally, update the center-of-mass and total mass of x. Because the code is too long, we don't show it.

Calculating the force

1. If the current node is an external node (and it is not body b), calculate the force exerted by the current node on b, and add this amount to b's net force.

2. Otherwise, calculate the ratio s/d. If s/d < θ, treat this internal node as a single body, and calculate the force it exerts on body b, and add this amount to b’s net force.
3. Otherwise, run the procedure on each of the current node’s children.

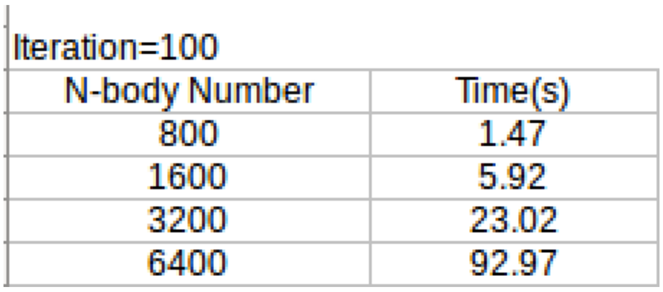
```
if( (((r/nodep->diag) > ratiothreshold) || (nodep->bodyp))&&(nodep->bodyp!=bodyp) )
    {
        accel = (10.0) * nodep->totalmass / r /r /r;

        bodyp->ax += accel*dx;
        bodyp->ay += accel*dy;
    } else {
        if(nodep->q1) { treesum(nodep->q1, bodyp, ratiothreshold); }
        if(nodep->q2) { treesum(nodep->q2, bodyp, ratiothreshold); }
        if(nodep->q3) { treesum(nodep->q3, bodyp, ratiothreshold); }
        if(nodep->q4) { treesum(nodep->q4, bodyp, ratiothreshold); }
    }
```

Besides, for the parallel version of this algorithm, we just add the basic OpenMP directives to the serial version as the version1_1 goes.

5. Performance Analysis

1. First, simply test the original serial version.


Iteration=100

N-body Number	Time(s)
800	1.47
1600	5.92
3200	23.02
6400	92.97

The main cost should be the force calculation part because of the nested for loop.

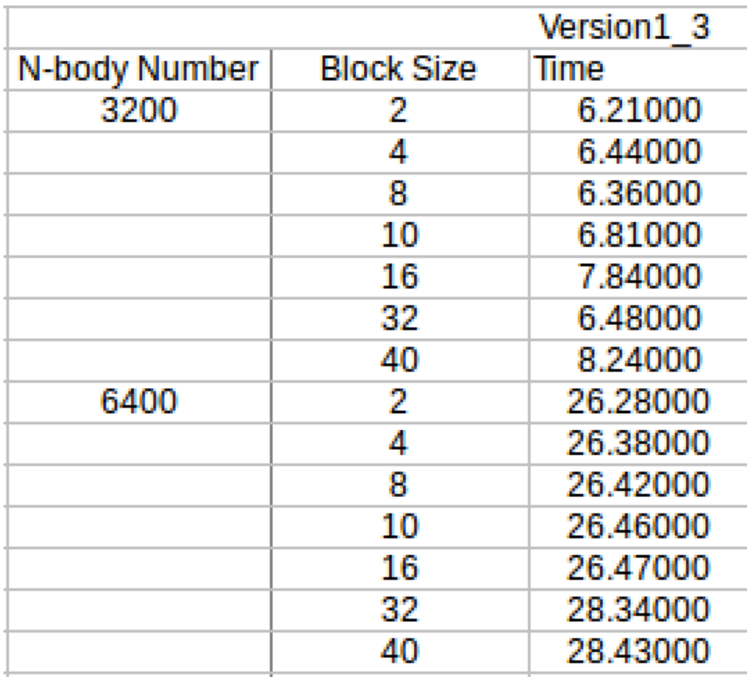
2. For the basic OpenMP version, we have the result table as below.
we can observe that the parallelized version is much faster than the serial version. And because my computer’s hardware environment is 4-core and 8-thread, the result shows that when the thread is set to 8, the cost is the cheapest.

N-body Number	Thread Number	Version1(s)	N-body Number	Thread Number	Version1(s)
800	1	1.58000	1600	1	6.32000
	2	0.82000		2	3.23000
	3	0.59000		3	2.32000
	4	0.59000		4	1.91000
	5	0.54000		5	2.24000
	6	0.51000		6	1.95000
	7	0.48000		7	1.86000
	8	0.78000		8	1.95000
	9	0.59000		9	2.05000
	10	0.56000		10	1.96000
3200	1	25.1800	6400	1	108.250
	2	13.0200		2	52.0800
	3	8.80000		3	35.3700
	4	6.85000		4	27.9300
	5	8.92000		5	34.5900
	6	7.77000		6	31.0600
	7	7.14000		7	31.3900
	8	6.44000		8	28.5700
	9	7.26000		9	32.6600
	10	7.30000		10	29.9800

3. For the body slimming version, we have the result table as below:

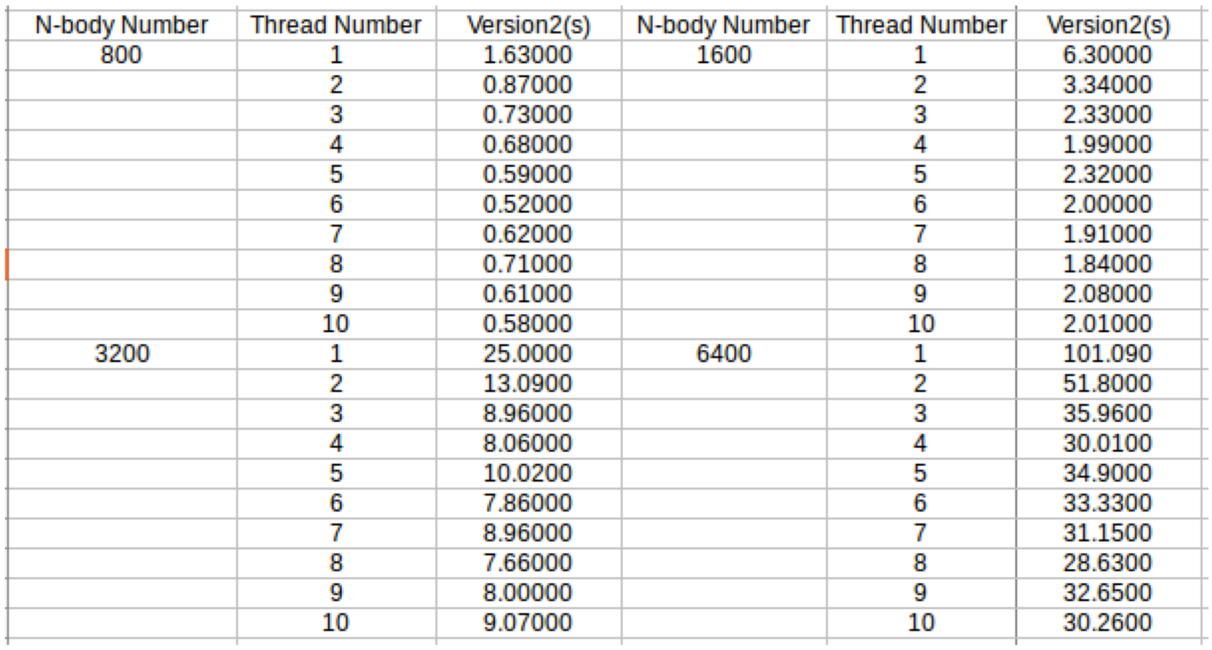
N-body Number	Thread Number	Version1_2(s)	N-body Number	Thread Number	Version1_2(s)
800	1	1.62000	1600	1	6.31000
	2	0.89000		2	3.21000
	3	0.69000		3	2.27000
	4	0.55000		4	2.20000
	5	0.58000		5	2.31000
	6	0.52000		6	1.99000
	7	0.56000		7	1.95000
	8	0.69000		8	1.93000
	9	0.57000		9	2.05000
	10	0.57000		10	1.95000
3200	1	25.0500	6400	1	100.220
	2	12.8900		2	51.8700
	3	9.39000		3	35.7200
	4	6.89000		4	29.0400
	5	9.13000		5	35.3100
	6	7.89000		6	31.1400
	7	8.95000		7	29.5200
	8	7.85000		8	29.5000
	9	7.55000		9	31.8100
	10	7.52000		10	27.8600

4. For the blocked version, we only test the 8-thread case, and we can get the table. We can get that when block size is 8, the speed is the fastest.



		Version1_3
N-body Number	Block Size	Time
3200	2	6.21000
	4	6.44000
	8	6.36000
	10	6.81000
	16	7.84000
	32	6.48000
	40	8.24000
6400	2	26.28000
	4	26.38000
	8	26.42000
	10	26.46000
	16	26.47000
	32	28.34000
	40	28.43000

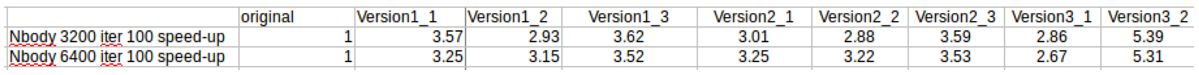
5. For the two-world version, we only show the basic parallel version.



N-body Number	Thread Number	Version2(s)	N-body Number	Thread Number	Version2(s)
800	1	1.63000	1600	1	6.30000
	2	0.87000		2	3.34000
	3	0.73000		3	2.33000
	4	0.68000		4	1.99000
	5	0.59000		5	2.32000
	6	0.52000		6	2.00000
	7	0.62000		7	1.91000
	8	0.71000		8	1.84000
	9	0.61000		9	2.08000
	10	0.58000		10	2.01000
3200	1	25.0000	6400	1	101.090
	2	13.0900		2	51.8000
	3	8.96000		3	35.9600
	4	8.06000		4	30.0100
	5	10.0200		5	34.9000
	6	7.86000		6	33.3300
	7	8.96000		7	31.1500
	8	7.66000		8	28.6300
	9	8.00000		9	32.6500
	10	9.07000		10	30.2600

6. For Barnes Hut algorithm, we only test 8-thread version in the parallel version. Actually it has a very good performance.

Finally, we can get the speed-up table(we only care about the **8-thread case for all versions, and only 8-blocksize for blocked version**), and in this table, we compute speed-up as **original time / current time**.



	original	Version1_1	Version1_2	Version1_3	Version2_1	Version2_2	Version2_3	Version3_1	Version3_2
Nbody 3200 iter 100 speed-up	1	3.57	2.93	3.62	3.01	2.88	3.59	2.86	5.39
Nbody 6400 iter 100 speed-up	1	3.25	3.15	3.52	3.25	3.22	3.53	2.67	5.31

This is what we get.
Blocked version has a optimization.
But body slimming version do not have a good performance when it is multithreading, but in serial program there exists optimizations. We think that maybe in the multithreading program, the load balancing amortizes the miss penalty.
Besides, two-world version does not change a lot, we think it is because the task that every thread does is almost the same, so the threads will accomplish their jobs almost at the same time, so actually there is not much waiting time for the basic OpenMP version, so we can not get a much better performance.
But for Barnes Hut algorithm, it is pretty good. Its complexity is O(nlogn). When the data size becones larger, the optimization is remarkable.

6. Summary

From this project, actually we learn a lot about parallel programming by using OpenMP, and have a deeper understanding about cache level optimization. Besides, thanks TA for answering our questions.