Project 2 Nbody simulation

Name: Yuxi Wang(5131309040), Yiming Zhang(5140219368)

1 Introduction

In physics and astronomy, an N-body simulation is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity (see n-body problem). N-body simulations are widely used tools in astrophysics, from investigating the dynamics of few-body systems like the Earth-Moon-Sun system to understanding the evolution of the large-scale structure of the universe. In physical cosmology, N-body simulations are used to study processes of non-linear structure formation such as galaxy filaments and galaxy halos from the influence of dark matter. Direct N-body simulations are used to study the dynamical evolution of star clusters.
In this project, we will show you how to do N-body simulation in parallel by using OpenMP and also how to simulate N-body with Barnes-Hut algorithm.

2 Code Implementation

Both of these two methods could use OpenMP to get the parallel version very easy. For the naive method one method to parallel it, is just divide the outer loop. Thus every thread only need to do a part of the calculation and they do it in parallel. More specifically, we also divided the number bodies accord to the number of threads and also create some private variables, since we used **# pragma omp parallel numThreads threadCount** as the parallel way. This parallel block include the computer force part and also the update postion part. It also can use **parallel for** to achieve this parallelization.
As for the parallelization of Barnes-Hut algorithm, same as the method we used in naive version. Just divide the loop of bodies, then every thread will take care its own bodies.

2.1 Naive Method

We have implemented OpenMP to nbody naive method. The way is just to parallel two for loops. One is the computation of forces. The other is to update the velocity and position of each body. Each calculation is independent. Thus, simply divide these loops and assign responsibility to each process, then that is the implementation.
Code is listed in **Appendix.A**.

2.2 Improvement to the Naive Method

The naive method uses a complete two-nested loop to calculate the force between two bodies. In fact, it's redundant because in this way all forces are calculated twice. For example, the case when $i = 2$ and $j = 4$ is the same as the case when $i = 4$ and $j = 2$. Intuitively we want each force to be calculated only once.
Based on this idea, we change

```
for(i = 0; i< world->num_bodies; ++i)
    for(j = 0; j< world->num_bodies; ++j)
```

to

```
for(i = 0; i< world->num_bodies; ++i)
    for(j = i + 1; j< world->num_bodies; ++j)
```

2.3 the Barnes-Hut algorithm

Here is the introduction of this algorithm http://physics.princeton.edu/ fpretori/Nbody/intro.htm
The main point of the algorithm is to construct a 4-childs-tree to simulate the world. And if there are some stars are 'close' to each other, and 'far' away from the star A, just consider these star is one stuff when calculate the force from these stars to star A.
There are two main parts of this algorithm. 1. Build the tree recursively. 2. Compute the force for every star. **Thus we need two new structs:**
1. 'BHTree' to represent the a node(rectangle unit) in the space. It have 4 BHTree nodes and a body (a real body or a equivalent star cloud) and also a Quad.
2. 'Quad' save the a rectangle unit information, it include the left lower vertex' postion and the width, height of this space.
And we need several functions:
1. function insert(BHTree, body), this function is used to insert the a body to the the node it belongs to. Use this function just insert the bodies one by one to the root node, it can build a BHTree for us. And in this function there are three possible cases:
case 1: it is empty leaf node, so just put the body in.
case 2: it is interior node, then insert the body to its child node recursively.
case 3: it is a leaf node with a body in it, then take this body out and create new childs node for this node, then insert these two bodies to the its child node.
2. function computeForce(body, BHTree, fx, fy), this function is used to compute the force from the tree to the body. Just call it one time use a body and root of the BHtree as parameter, it will add the force to the fx and fy. There is also three cases to compute it recursively:
case 1: the node have one body init and it is exactly the body itself then just do nothing and return.
case 2: if the node is a 'small' Quad and 'far' from the body or it is a leaf node including other body, just use the node's body to compute the force for the parameter body.

case 3: if the node is a interior node not 'small' or not 'far' enough, then we need compute the force from its child node recursively.
3. function ooforce(body a, body b, fx, fy), use this function to compute the force between two bodies.

Code is listed in **Appendix.B**.

3 Experiment

Table 1 shows the result of timing the naive Method.

Table 1: Run Times of naive Method

comm_sz	N-body Number						
	100	150	200	250	300	350	400
Without OpenMP	2.77	6.32	10.48	20.97	40.85	54.7	67.08
1	3.19	7.05	12.48	22.82	35.16	46.17	63.37
2	1.62	4.43	7.1	17.84	27.44	30.69	39.89
5	1.24	3.19	5.55	13.23	19.68	27.85	36.94

Table 2 shows the result of timing the Barnes-Hut Method. We set THETA as 0.05 to balance the efficiency and effectiveness.

Table 2: Run Times of Barnes-Hut Method

comm_sz	N-body Number						
	100	150	200	250	300	350	400
Without OpenMP	7.65	16.4	33.62	55.19	98.74	119.97	129.72
1	9.44	27.59	43.54	64.21	99.31	108.10	159.33
2	4.77	9.10	17.28	21.17	34.18	42.11	49.82
5	2.79	5.47	6.96	10.67	13.48	18.58	22.74

4 Analysis

We tested the program with different number of threads. We decided to measure the cost of OpenMP. Thus, the program without OpenMP was also included in our experiment.

4.1 Naive Method

Table 3 shows the speedups of the parallel naive Method.

Table 3: Speedups of naive Method

comm_sz	N-body Number						
	100	150	200	250	300	350	400
Without OpenMP	1.15	1.12	1.19	1.09	0.86	0.84	0.94
1	1	1	1	1	1	1	1
2	1.97	1.59	1.76	1.28	1.28	1.50	1.59
5	2.57	2.21	2.25	1.72	1.79	1.66	1.71

Figure 1 is the trend of speedups.

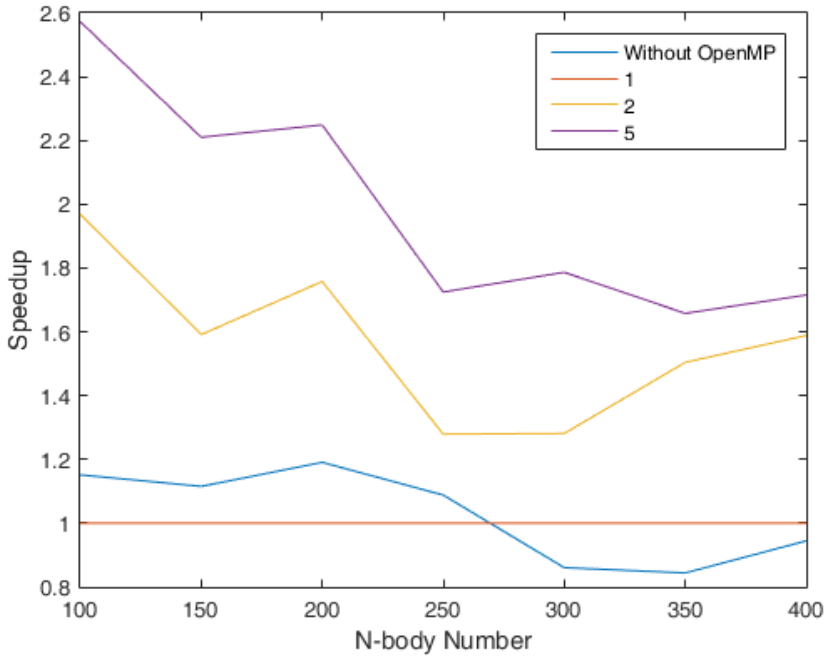

Figure 1: Speedups of parallel naive Method

With these data and figure, it is easy to notice some interesting and meaningful features.
Observation 1 If we increase the number of threads, the run-time decreases, which means the speedup increases.
Explanation This meets our expectation. Parallel program gets speedup by assigning loads to different threads.

Observation 2 For **larger** N-body number or **more** threads, this program's speedup is considerably **smaller than** ideal speedup.
Explanation When the number of thread is 5, the speedup is less than 3. Considering the assigned tasks, the first job is to calculate the forces between any two bodies by implementing the two nested loops. In this case, cache miss will always occur. The larger the number of threads is, the more cache miss there is. It has a dramatic effect on the performance of this program.

Observation 3 Compared with "Without OpenMP" and "1", when N-body number is relatively small, 1-thread version is obviously **slower** than Without-OpenMP version.
Explanation This observation proves that OpenMP cost some time in any parallel program, like communication cost, setup cost.

4.2 the Barnes-Hut algorithm

Table 4 shows the speedups of the parallel Barnes-Hut Method.

Table 4: Speedups of Barnes-Hut Method

comm_sz	N-body Number						
	100	150	200	250	300	350	400
Without OpenMP	1.23	1.68	1.30	1.16	1.01	0.90	1.23
1	1	1	1	1	1	1	1
2	1.98	3.03	2.52	3.03	2.91	2.57	3.20
5	3.38	5.04	6.26	6.02	7.57	5.82	7.01

Figure 2 is the trend of speedups.

Observation 1 If we increase the number of threads, the run-time decreases, which means the speedup increases.
Explanation It also meets our expectation.
Observation 2 For **larger** N-body number, this program's speedup is even **higher than** ideal speedup. Thai is to say, we achieved superlinear speedup.
Explanation One possible reason for this phenomenon is that in this algorithm the lager number of bodies it has, the more cache hit it happens when do in parallel. For instance, 1000 bodies may have more number of star cloud like stuffs than 100 bodies. Thus a lot of star in the 1000 bodies far from the star cloud will reuse the star cloud's node data. Thus if we want to produce a cache hit, we need to reuse the data before it be replaced. Thus, the more threads we have the higher possibility to compute a body's force which can reuse the data. That's why the speed up dramatically increased with the number of the bodies even achieved the superlinear speedup.

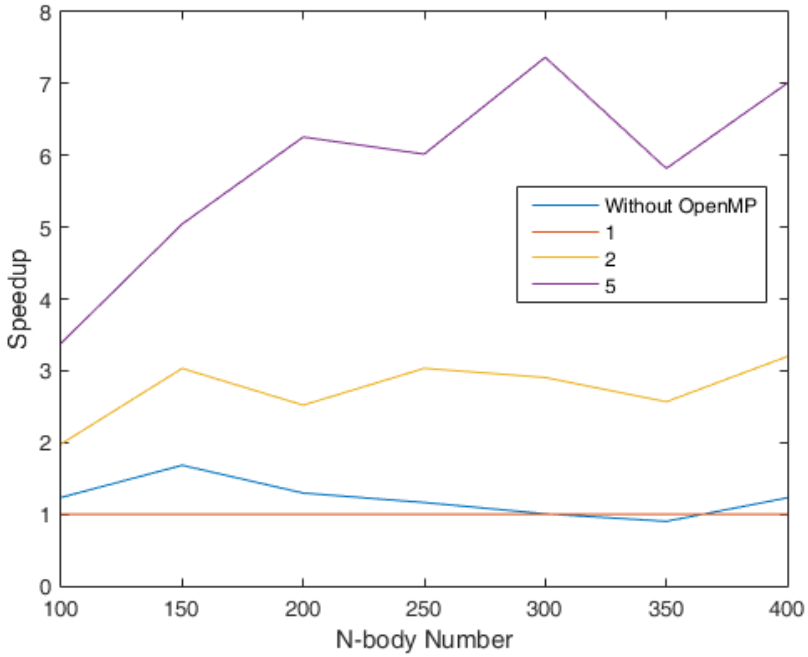

Figure 2: Speedups of parallel Barnes-Hut Method

4.3 Comparison between Two Methods

If we fix the N-body number and the number of threads, our experiment result shows the parallel Barnes-Hut Method performs better than the parallel naive Method.

5 Reference

http://www.cs.princeton.edu/courses/archive/fall03/cs126/assignments/barnes-hut.html
http://blog.csdn.net/baimafujinji/article/details/53036473
http://portillo.ca/nbody/code/
http://www.docin.com/p-1457353180.html
http://maskray.me/blog/2012-09-10-parallel-n-body

6 Appendix

A Naive Method with OpenMP

```
1 void position_step(struct world *world, double time_res, int thread_count) {
2     int i, j;
```

```
3     double d, d_cubed, diff_x, diff_y;
4
5     /* The forces array stores the x and y components of the total force
      acting
6       * on each body. The forces are index like this:
7       *     F on body i in the x dir = F_x[i]
8       *     F on body i in the y dir = F_y[i] */
9     double *force_x = (double*)malloc(sizeof(double) * world->num_bodies);
10  double *force_y = (double*)malloc(sizeof(double) * world->num_bodies);
11    // initialize all forces to zero
12    force_x = memset(force_x, 0, sizeof(double) * world->num_bodies);
13  force_y = memset(force_y, 0, sizeof(double) * world->num_bodies);
14
15  # pragma omp parallel num_threads(thread_count)
16  {
17    int my_rank = omp_get_thread_num();
18    int thread_count = omp_get_num_threads();
19    int local_bodies = world->num_bodies/thread_count;
20    int local_index = local_bodies*my_rank;
21    int k;
22    int i;
23    int j;
24    double d, d_cubed;
25    /* Compute the net force on each body */
26    for (k = 0; k < local_bodies; ++k) {
27      for (j = 0; j < world->num_bodies; ++j) {
28        i = k+local_index;
29        if (i == j) {
30          continue;
31        }
32        // Compute the x and y distances and total distance d between
33        // bodies i and j
34        double diff_x = world->bodies[j].x - world->bodies[i].x;
35        double diff_y = world->bodies[j].y - world->bodies[i].y;
36        d = sqrt((diff_x * diff_x) + (diff_y * diff_y));
37
38        if (d < 25) {
39          d = 25;
40        }
41        d_cubed = d * d * d;
42        // Add force due to j to total force on i
43        force_x[i] += GRAV * (world->bodies[i].m * world->bodies[j].m
44            / d_cubed) * diff_x;
45        force_y[i] += GRAV * (world->bodies[i].m * world->bodies[j].m
46            / d_cubed) * diff_y;
47      }
48    }
49
50    // Update the velocity and position of each body
51
52    for (k = 0; k < local_bodies; ++k) {
```

```
      i = k + local_index;
      // Update velocities
      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;
      // Update positions
      world->bodies[i].x += world->bodies[i].vx * time_res;
      world->bodies[i].y += world->bodies[i].vy * time_res;
    }
  }

}
```

B the Barnes-Hut algorithm with OpenMP

```
#define THETA 0.05

struct body {
    double x, y; // position
    double vx, vy; // velocity
    double m; // mass
    double r; // radius of the particle
};


struct Quad {
  double x, y;
  double xlen, ylen;
};


void initQ(struct Quad *q, double a, double b, double c, double d){
  q->x = a;
  q->y = b;
  q->xlen = c;
  q->ylen = d;
};

bool contains(struct Quad *q, double a, double b) {
  if (a >= q->x && b >= q->y && a < q->x + q->xlen && b < q->y + q->ylen)
    return true;
  return false;
}


struct BHTree {
  struct body body;
  struct Quad quad;
  struct BHTree *NW;
  struct BHTree *NE;
  struct BHTree *SW;
  struct BHTree *SE;
```

```
};

void initB (struct BHTree *p, struct Quad q){
  p->body.x = p->body.y = 0;
  p->body.vx = p->body.vy = 0;
  p->body.m = p->body.r = 0;
  p->quad = q;
  p->NE = p->NW = p->SW = p->SE = NULL;
}

void insert_to_child(struct BHTree *p, struct body b);

void insert(struct BHTree *p, struct body b) {
  //printf("%f\n", b.x);
  struct body a = p->body;
  p->body.m += b.m;
  p->body.x = (p->body.x + b.x * b.m) / p->body.m;
  p->body.y = (p->body.y + b.y * b.m) / p->body.m;
  if (a.m == 0) {
  } else if (!(p->NW == NULL && p->NE == NULL && p->SW == NULL & p->SE == NULL
    )) {
    insert_to_child(p, b);
  } else {
    insert_to_child(p, a);
    insert_to_child(p, b);
  }
}

void insert_to_child(struct BHTree *p, struct body b){
  struct Quad *q = malloc(sizeof(struct Quad));
  if (b.x < p->quad.x + p->quad.xlen / 2) {
    if(b.y < p->quad.y + p->quad.ylen / 2) {
      if(p->SW == NULL) {
        p->SW = malloc(sizeof(struct BHTree));
        initQ(q, p->quad.x, p->quad.y, p->quad.xlen/2,
            p->quad.ylen/2);
        initB(p->SW, *q);
      }
      insert(p->SW, b);
    } else {
      if(p->NW == NULL) {
        p->NW = malloc(sizeof(struct BHTree));
        initQ(q, p->quad.x, p->quad.y + p->quad.ylen/2, p->quad.xlen/2,
            p->quad.ylen/2);
        initB(p->NW, *q);
      }
      insert(p->NW, b);
    }
  } else {
    if(b.y < p->quad.y + p->quad.ylen / 2) {
      if(p->SE == NULL) {
```

```
87          p->SE = malloc(sizeof(struct BHTree));
88          initQ(q, p->quad.x+p->quad.xlen/2, p->quad.y, p->quad.xlen/2,
89              p->quad.ylen/2);
90          initB(p->SE, *q);
91        }
92        insert(p->SE, b);
93      } else {
94        if(p->NE == NULL) {
95          p->NE = malloc(sizeof(struct BHTree));
96          initQ(q, p->quad.x + p->quad.xlen/2, p->quad.y + p->quad.ylen/2, p->
    quad.xlen/2,
97              p->quad.ylen/2);
98          initB(p->NE, *q);
99        }
100       /*printf("m is %f\n", b.m);
101       printf("x, y is %f %f\n", b.x, b.y);
102       printf("Select NE\n");
103       */
104       insert(p->NE, b);
105     }
106   }
107   free(q);
108 }
109
110 struct world {
111     struct body *bodies;
112     struct BHTree *tree;
113     int num_bodies;
114 };
115
116 double max(double x, double y){if (x < y) return y; return x;}
117
118 void ooforce(struct body a, struct body b, double *fx, double *fy);
119
120 void computeForce(struct body b, struct BHTree *tree, double *fx, double *fy)
    {
121   double s = max(tree->quad.xlen, tree->quad.ylen);
122   double d = sqrt(pow(tree->body.x-b.x,2) + pow(tree->body.y-b.y,2));
123   bool isleaf = tree->NW == NULL && tree->NE == NULL && tree->SW == NULL &&
124     tree->SE == NULL;
125   if (isleaf && tree->body.x == b.x && tree->body.y == b.y) {
126     // same body
127   } else if (s / d < THETA || isleaf){
128     ooforce(b, tree->body, fx, fy);
129   } else {
130     if (tree->SE != NULL) computeForce(b, tree->SE, fx, fy);
131     if (tree->NW != NULL) computeForce(b, tree->NW, fx, fy);
132     if (tree->NE != NULL) computeForce(b, tree->NE, fx, fy);
133     if (tree->SW != NULL) computeForce(b, tree->SW, fx, fy);
134   }
135 }
```

```
136
137 void ooforce(struct body a, struct body b, double *fx, double *fy) {
138   // Compute the x and y distances and total distance d between
139   // bodies i and j
140   double diff_x = b.x - a.x;
141   double diff_y = b.y - a.y;
142   double d = sqrt((diff_x * diff_x) + (diff_y * diff_y));
143
144   if (d < 25) {
145     d = 25;
146   }
147   double d_cubed = d * d * d;
148   // Add force due to j to total force on i
149   *(fx) += GRAV * (a.m * b.m / d_cubed) * diff_x;
150   *(fy) += GRAV * (a.m * b.m / d_cubed) * diff_y;
151 }
152
153
154 void position_step(struct world *world, double time_res, int thread_count) {
155     int i, j;
156     double d, d_cubed, diff_x, diff_y;
157
158
159     /* The forces array stores the x and y components of the total force
        acting
160      * on each body. The forces are index like this:
161      *     F on body i in the x dir = F_x[i]
162      *     F on body i in the y dir = F_y[i] */
163     double *force_x = (double*)malloc(sizeof(double) * world->num_bodies);
164   double *force_y = (double*)malloc(sizeof(double) * world->num_bodies);
165     // initialize all forces to zero
166     force_x = memset(force_x, 0, sizeof(double) * world->num_bodies);
167   force_y = memset(force_y, 0, sizeof(double) * world->num_bodies);
168
169   struct Quad *q = malloc(sizeof(struct Quad));
170   initQ(q, 0, 0, WIDTH, HEIGHT);
171   world->tree = malloc(sizeof(struct BHTree));
172   initB(world->tree, *q);
173   for (i = 0; i < world->num_bodies; ++i){
174     insert(world->tree, world->bodies[i]);
175   }
176
177   # pragma omp parallel num_threads(thread_count)
178   {
179     int my_rank = omp_get_thread_num();
180     int thread_count = omp_get_num_threads();
181
182     int local_bodies = world->num_bodies/thread_count;
183     int local_index = local_bodies*my_rank;
184     int elem;
185
```

```
186     for (i = 0; i < local_bodies; ++i) {
187       computeForce(world->bodies[i+local_index], world->tree,
188           &force_x[i+local_index], &force_y[i+local_index]);
189     }
190 
191     // Update the velocity and position of each body
192 
193     for (i = 0; i < local_bodies; i++) {
194       elem = i + local_index;
195       // Update velocities
196       world->bodies[elem].vx += force_x[elem] * time_res / world->bodies[elem
    ].m;
197       world->bodies[elem].vy += force_y[elem] * time_res / world->bodies[elem
    ].m;
198 
199       // Update positions
200       world->bodies[elem].x += world->bodies[elem].vx * time_res;
201       world->bodies[elem].y += world->bodies[elem].vy * time_res;
202     }
203   }
204 }
```