N-body Simulation using OpenMP, Pthread and MPI

Xin Huang
Dept. of CST, THU
ID: 2011011253

September 12, 2013

Abstract

An **N-body Simulation** is a simulation of a dynamical system of particles under the influence of gravity. [**wiki__nbs**] Here the author compare efficiency among programs using three kinds of parallel backends: *OpenMP* , *pthread* , and *MPI* .

This is *homework 4* for course *Parallel Programming*

Keyword N-body, simulation, parallel

Contents

1 Instruction **2**

2 Design & Approach **2**
2.1 Definitions . . . 2
2.2 Overview . . . 2
2.3 The Engine . . . 2
2.3.1 SimpleEngine . . . 2
2.3.2 BSPTree . . . 3
2.4 The Worker . . . 4
2.4.1 *OpenMP* . . . 4
2.4.2 *pthread* . . . 4
2.4.3 *MPI* . . . 5

3 Result & Analysis **5**
3.1 *OpenMP* . . . 6
3.2 *pthread* . . . 7
3.3 *MPI* . . . 8
3.4 Brief Summary . . . 9

4 Experience **9**

A Specific Result **10**

B Source Code **12**

1 Instruction

You can simply follow *make*$- >$ *make run* to run the program. For more detailed instruction, seed Appedix **??**

2 Design & Approach

2.1 Definitions

- **Body** The bodies are defined as sphere. A body has its mass, position and velocity. All of them are in three dimension. But the demo showed in 2 dimension.
- **Engine** The engine is responsible for the calculation of the gravitational effect assigned to the single body and the collision between bodies.
- **Worker** The worker is responsible for the scheduling the task of calculation of the collision and gravity. In this program, there are three method to achieve this goal: **OpenMP**, **pthread**, **MPI**.

2.2 Overview

Simulation consists of multiple rounds. In each round, program simulates the state of the system within Δt seconds, with the approximation of considering the motion as uniform linear motion within a round.

Each round consists of two phases:

1. Firstly, we calculate the net force as well as the acceleration of one body, and then get the velocity of each. Then we update the new position where the body should be.
2. Secondly, we need to consider the collisions between two bodies. We know it may be true that more than two bodies can collide at the same time. However, too many collision at the same time may be trouble for programming for the velocity is unable to calculate. So we assume every time of update only one collision occurs. The method is to calculate the least time that the bodies will collide and calculate that time. And there is a lot of calculation to solve the quadric equations. It takes so much time.

2.3 The Engine

There are two engines available:

2.3.1 SimpleEngine

SimpleEngine will calculate accelaration **precisely**: accumulate all forces exerted on it using formula

$$F_{net} = \Sigma \frac{G \cdot m \cdot m_{other}}{r^2}$$

and the accelaration with

$$a_{net} = \Sigma \frac{F_{net}}{m} = \Sigma \frac{G \cdot m_{other}}{r^2}$$

then the new velocity will be

$$v\prime = v + a_{net} * \Delta t$$

The method use a loop to enumerate all bodie and collisions with boundaries. And calculate the least time of the next one collision. Due to the method enumerate all bodies and in each loop check all the other bodies. this method gives an $O(n^2 + n^2) = O(n^2)$ algorithm for n bodies.

2.3.2 BSPTree

Comparing to Octree, BSPTree (Binary Space Partitioning Tree) is a data structure can arrange bodies in a unlimited space, while the former only deals with bounded space. It is easier for BSPTree to calculate the position quickly.

A node in tree holds information of the subtree rooted on that node. Here are details of the tree:

Axis Aligned Bounding Box(AABox)

AABox is a cuboid which can described the bodies set. AABox has three perpendicular axis: x, y, z. The objects are in this Axis Aligned Bounding Box.

It's generally used in 3D games.

Construction of the tree

The tree is recursively constructed from the root with all bodies given.

1. An AABox need to hold the bodies and it's attributes, such as the mass and the center mass of a box. It can make the calculation quicker.
2. Select one axis, and find out the center of the bodies in the AABox on this axis.
3. Find out the id of the body in the middle, divide the bodies in the AABox into two equal parts. The two parts are regarded as the two child nodes of the current node. The plane divides the box is also in that node.
4. recursively construct the tree with the two parts of the bodies.

With $O(n)$ algorithm to find the median, and $O(\log n)$ depth of the tree, construction of the tree can be done in $O(n \log n)$ time. And with apparently $O(n)$ nodes in the tree with constant information stored on each node, space complexity is $O(n)$.

Acceleration Calculation

For each body, we calculate the accelaration its own exerted by other bodies start from the root of the tree as follow.

1. There are a lot of calculation in the accumulation. So we need to do some tricks on it.

   We can consider a box of bodies as a mass point when the box is so far away from the given body. If a node is a leaf node containing one body or not a leaf but so far away. We can just calculate the mass center to find out the acceleration.

   Here we define the insignificance θ of a node to a body as:

   $$\theta = \frac{V}{D^3}$$

   where V is the volume of AABox of the node, and D is the distance between CM of the node and the body.

   We also set a threshold η empirically, and consider nodes whose $\theta < \eta$ as insignificant to the given body. The large η, the faster and imprecise the calculation will be.
2. Else consider the acceleration separately on each child node.

With certain value of η, we can expect a much less time for each body to obtain its acceleration, expecting a $O(n \log n)$ time complexity.

Body Collision

We can assume the motion in δt is linear for the body collision. And we can quickly detect whether there is a collision by distending the AABox a length from time 0 to time δt.

For each body, we find its earliest collision may happen with another body in δt time in the tree start with root node as follow:

1. construct a new AABox which is the AABox of current node but with a expand in width of the body's diameter.
2. if l_i does not intersect the new AABox of current node, no further inspection is needed, thus return to its root node.
3. if current node is a leaf node, calculate the collision.
4. if l_i intersect the new AABox of current node, inspect both of its child node for further investigation.

Also, only a few amount of node need to be checked, much less time is consumed. If the speed of the body is not that large comparing to the volume of AABox, this algorithm yields a time complexity of $O(n \log n)$

We assume two body b_i and b_j will hit at time t, then we have equations

$$\begin{cases} \mathbf{p_i\prime} = \mathbf{p_i} + \mathbf{v_i} \cdot t \\ \mathbf{p_j\prime} = \mathbf{p_j} + \mathbf{v_j} \cdot t \\ |\mathbf{p_i} - \mathbf{p_j}| = r_i + r_j \end{cases}$$

where $\mathbf{p_i}, \mathbf{v_i}, r_i$ are the position, velocity and radius of b_i respectively. This is a quadric equation set, and we can simply choose the smallest opposite t as the solution. If such t does not exist or $t > \Delta t$, then the two body will not collide.

Finally, when knowing the two body will collide, the response to collision of two body b_1 and b_2 is calculated as follow[**col_rspns**]:

$$\begin{aligned} \mathbf{u} &= \frac{\mathbf{p_1} - \mathbf{p_2}}{|\mathbf{p_1} - \mathbf{p_2}|} \\ u_1 &= \mathbf{u} \cdot \mathbf{v_1} \\ \mathbf{v_{1x}} &= \mathbf{u} \cdot u_1 \\ \mathbf{v_{1y}} &= \mathbf{v_1} - \mathbf{v_{1x}} \\ u_2 &= -\mathbf{u} \cdot \mathbf{v_2} \\ \mathbf{v_{2x}} &= -\mathbf{u} \cdot u_2 \\ \mathbf{v_{2y}} &= \mathbf{v_2} - \mathbf{v_{2x}} \\ \mathbf{v_1\prime} &= \mathbf{v_{1x}} \frac{m_1 - m_2}{m_1 + m_2} + \mathbf{v_{2x}} \frac{2 \cdot m_2}{m_1 + m_2} + \mathbf{v_{1y}} \\ \mathbf{v_2\prime} &= \mathbf{v_{1x}} \frac{2 \cdot m_1}{m_1 + m_2} + \mathbf{v_{2x}} \frac{m_2 - m_1}{m_1 + m_2} + \mathbf{v_{2y}} \end{aligned}$$

where $v_1\prime$ and $v_2\prime$ are velocity after collision.

2.4 The Worker

Three workers are:

2.4.1 *OpenMP*

There are several phases can be parallelized in openmp. Calculation of the velocity, detection of the collision, Get new velocity after collision, and the accelaration. They can be respectively calculated using the openmp pragma notation. And in this program it's not as efficient as the previous task when using the dynamic schedule. The reason may be it costs more to fork for there are many parallel for.

2.4.2 *pthread*

For two phases in each round, dynamically distribute calculation tasks (both calculation of acceleration and collision) among threads using a task scheduler.

2.4.3 *MPI*

The program is running in *quasi-master-slave* mode. Due to the amount of calculation for each body is roughly equal, equal-amount assignment of calculation tasks is hereby employed. All processes are responsible for calculation, but only the so-called master process plots the result. For continuous rendering of frames, slave process stucks in calculation procedure, and repeat the configuration receiving, acceleration calculation, collision calculation loop until master process emit the quit signal.

3 Result & Analysis

The method for estimate the performance is to measure the average fps for each. The running time is limited to 20 seconds with different number of bodies. Efficiency is tested by exhaustively calculating the total count of frames in 20 seconds.

The initial setting of the body is that, one body with huge mass in the center at rest, and the remaining body with small mass lies around the body in a circle, with a tangential velocity to the center.

δt is set to 0.01 seconds.

Efficiency of the program with same nubmer of bodies is calculated by

$$E = \frac{f \cdot m}{F \cdot n}$$

where f denote the average FPS, m denote minmum number of workers used, F denote the average FPS with minimum number of workers and n denote the number of workers used in program.

3.1 *OpenMP*

	64	128	256	512	1024	2048
2	74.8401	18.8998	9.14185	0.0981635	0.00440611	0.192445
4	613.039	22.9485	9.24054	0.671141	0.041141	0.110224
6	1667.23	19.2788	9.19495	0.366862	0.647701	0.638978
8	3086.59	26.938	9.44499	0.668698	0.176445	0.314027
10	1423.68	25.8071	8.3823	0.28479	0.381717	0.737644
12	6981.5	40.5635	9.37031	0.822444	0.876184	0.608787

The openmp code does not perform so well on the clusters. I have tried the static, dynamic, and guided method, but the results are similar. It's a little puzzled that the performance is better than pthread when the datasize is small. Maybe too many bodies influence the speed of the server and cannot achieve a good performance.

3.2 *pthread*

	256	512	1024	2048	4096	8192	16384
2	935.603	432.178	211.608	89.9185	38.7845	16.9449	6.611
4	993.3	464.954	261.598	121.194	57.1943	26.9298	10.8913
6	977.601	481.452	286.857	135.886	69.062	33.4665	14.0311
8	1021.7	535.096	316.403	149.378	76.0468	37.4775	16.2468
10	1011.6	521.524	309.085	151.862	79.2643	39.978	17.578
12	1001.3	540.323	314.834	0.00551279	79.9021	40.6736	18.2717

	256	512	1024	2048	4096	8192	16384
2	0.61	0.63	0.66	0.71	0.73	0.82	0.95
4	0.42	0.44	0.47	0.45	0.55	0.63	0.84
6	0.24	0.25	0.27	0.38	0.42	0.51	0.72
8	0.15	0.19	0.20	0.30	0.38	0.40	0.58
10	0.12	0.18	0.22	0.25	0.32	0.30	0.43
12	0.10	0.16	0.21	0.26	0.30	0.28	0.41

The pthread method's performance is better than the openmp method when the dataset is larger. However the efficiency is low when the processors grows. The reason may be the use of dynamic schedule is frequent and the burden of load balance is larger.

3.3 *MPI*

	1024	2048	4096	8192	16384
2	211.608	89.9185	38.7845	16.9449	6.611
12	261.598	101.194	37.1943	17.9298	8.8913
24	266.857	115.886	39.062	19.4665	9.0311
36	216.403	119.378	36.0468	20.4775	10.2468
48	209.085	111.862	39.2643	21.978	11.578
60	214.834	105.58495	39.9021	21.6736	10.2717

The most unreasonable result comes from program programed with *MPI* . *MPI* runs perfectly with reasonable speed up on single machine, but it seems that, *MPI* does not work on clusters for some unknown reason. It is true that the program actually used desired amount of processes (from the logging information), but with **NO** speed up. I run it again and yields the same result.

3.4 Brief Summary

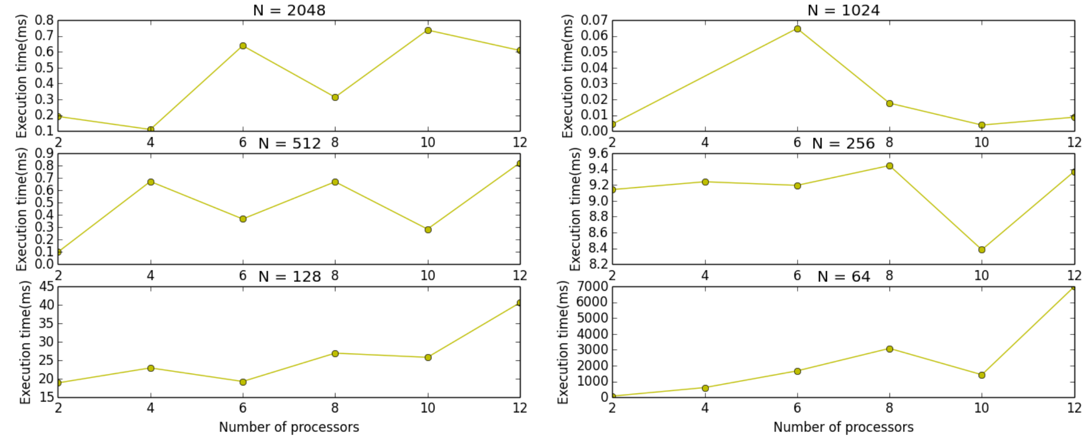
In this homework I met a spate of unanticipated situations of the performance. **OpenMP** performs not well on the large dataset. The efficiency is low in pthread's best performance. And I tried for several methods but they don't work either. The algorithm is also important in parallel programming. First I test the simple method to calculate the velocity and accelaration. The performance was even much worse that I had to terminate the program for the running time is too long. (for it's $O(n^2)$) The BSPTree performs much better on the large dataset. For the time complexity is $O(nlogn)$.

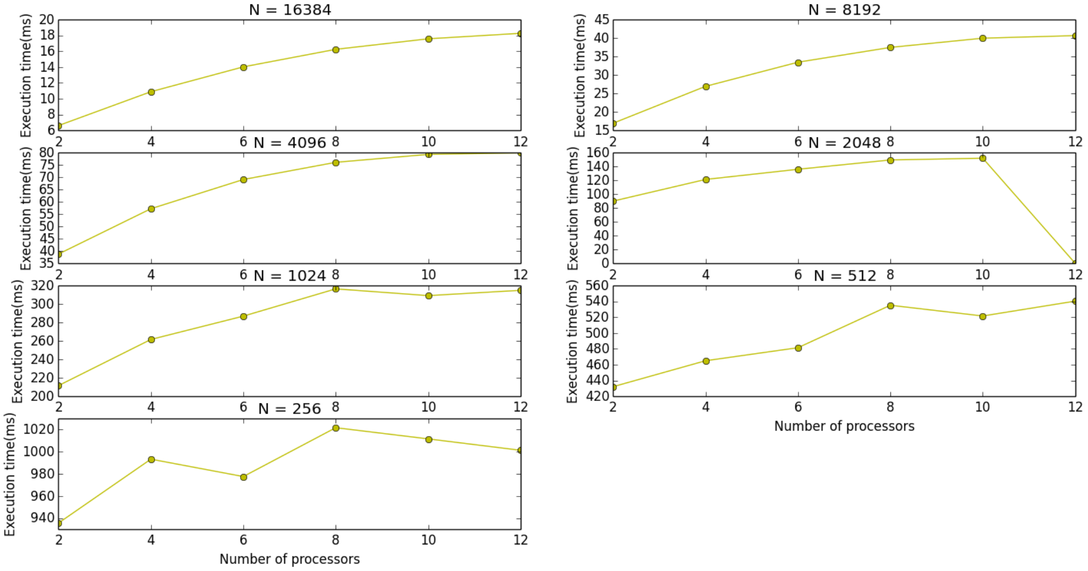
4 Experience

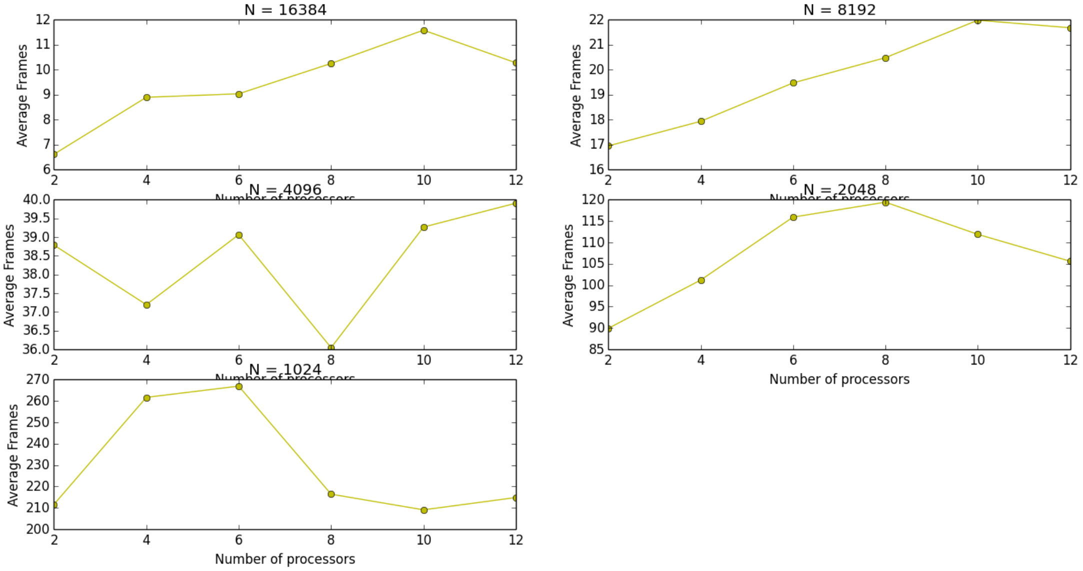
Writing a parallelize program is not a foolproof. The parallelize method need to be tested and be considered the scalability. This program performs not as good as I expected.

When algorithm used its self is complex, the efficiency gained by parallelize the program may be critically low. But a more efficient algorithm is always better than more efficient use of machine, since in any aspect, the running time is much smaller than algorithm with high time complextiy, and overall consumption on resources are much smaller.

A Specific Result






N = 16384
Average Frames
12
11
10
9
8
7
6
2
4
6
8
10
12
Number of processors
N = 8192
Average Frames
22
21
20
19
18
17
16
2
4
6
8
10
12
Number of processors
N = 4096
Average Frames
40.0
39.5
39.0
38.5
38.0
37.5
37.0
36.5
36.0
2
4
6
8
10
12
Number of processors
N = 2048
Average Frames
120
115
110
105
100
95
90
85
2
4
6
8
10
12
Number of processors
N = 1024
Average Frames
270
260
250
240
230
220
210
200
2
4
6
8
10
12
Number of processors

B Source Code

../../src/body.hh

```
#ifndef __BODY__
#define __BODY__

#include"collision.hh"

class ColorRGB{
    public:
        double r, g, b;
        ColorRGB(){r=g=b=255;}
        ColorRGB(double r, double g, double b):r(r), g(g), b(b){}
};

class Body{
    public:
        ColorRGB color;
        int id;
        Position pos, v;
        double mass;
        double r;

        double lengthsqr(const Body &b1, const Body &b2){
            double dx = b1.pos.x - b2.pos.y;
            double dy = b1.pos.y - b2.pos.y;
            return dx*dx+dy*dy;
        }

        bool is_collide(const Body &b){
            if (lengthsqr(*this, b) > this->r*this->r + b.r*b.r)
                return false;
            return true;
        }

        bool intersect(const Body &b) const{
            double distsqr = (pos - b.pos).lengthsqr();
            double rsum = (r+b.r)*(r+b.r);
            return distsqr < rsum;
        }

        void set_pos(const Position &p){this->pos = p;}
        void move(const Position &dpos){pos += dpos;}
        void advance(const double &dtime){pos += v*dtime;}
        double boundary(int cid, int type){
            double t;
            if (cid == 0) t = pos.x;
            else if (cid == 1) t = pos.y;
            else t = pos.z;
            return t + (type == 0 ? -1:1) * r;
        }



};

bool body_collide(Body &b1, Body &b2, double dtime, bool modify_b = false);
Collision body_clsdtct(Body &id0, Body &id1, double dtime);
Position calc_acc(const Position &p0, const Position &p1, double m1);

#endif
```

../../src/body.cc

```
/*
 * $File: body.cc
 * $Date: Wed Sep 11 17:14:43 2013 +0000
 * $Author: Xinyu Zhou <zxytim[at]gmail[dot]com>
 */


#include "body.hh"
#include "collision.hh"
#include <cassert>


static double sqr(double x) { return x * x; }
bool body_collide(Body &a, Body &b_, double dtime, bool modify_b)
{
//    log_printf("col: (%d,%d), %Lf", a.id, b.id, dtime);
    a.advance(dtime);
    Body *b, btmp;
    if (modify_b)
        b = &b_;
    else b = &btmp, btmp = b_;

    b->advance(dtime);

    double dist = (a.pos - b->pos).lengthsqr(),
           rsum = sqr(a.r + b->r);

    if (dist <= rsum)
    {
        double delta = (rsum - dist) / 2 * 1.1 + EPS;
        Position dir01 = b->pos - a.pos;
        a.set_pos(a.pos - dir01 * delta);
        b->set_pos(b->pos + dir01 * delta);
    }
    //assert(dist > rsum);
    assert(!a.intersect(*b));

    double m1, m2, x1, x2;
    Position v1, v2, v1x, v2x, v1y, v2y, x(a.pos - b->pos);


    x.unitize();
    v1 = a.v;
    x1 = x.dot(v1);
    v1x = x * x1;
    v1y = v1 - v1x;
    m1 = a.mass;

    x = x * -1;
    v2 = b->v;
    x2 = x.dot(v2);
    v2x = x * x2;
    v2y = v2 - v2x;
    m2 = b->mass;

    double s = m1 + m2;
    a.v = v1x * (m1 - m2) / s + v2x * (2 * m2) / s + v1y;
    b->v = v1x * (2 * m1) / s + v2x * (m2 - m1) / s + v2y;

    return true;
}

static bool solve_quadratic_equation(const double &a, const double &b, const double &c,
        double &root0, double &root1)
{
    if (fabs(a) < EPS)
        return false;
    double delta = b * b - 4 * a * c;
    if (delta >= 0)
    {
        double sq = sqrt(delta),
```

```cpp
               d = 1 / (2 * a);
        root0 = (-b - sq) * d;
        root1 = (-b + sq) * d;
        return true;
    }
    return false;
}

Collision body_clsdtct(Body &b0, Body &b1, double dtime)
{
    int id0 = b0.id,
        id1 = b1.id;
    if (id0 == id1)
        return Collision(-1, -1, -1);
    Position AB = b1.pos - b0.pos,
           vab = b1.v - b0.v;
    double rab = (b0.r + b1.r);
    double a = vab.dot(vab),
           b = 2 * vab.dot(AB),
           c = AB.dot(AB) - rab * rab;

    double t0, t1, t;
    if (!solve_quadratic_equation(a, b, c, t0, t1))
        return Collision(-1, -1, -1);
    t = t0;
    if (t < 0) t = t1;
    if (t < 0 || t > dtime) return Collision(-1, -1, -1);
    if (t > EPS)
        t -= EPS / 2;
    if (t > dtime)
        t = -1;
    return Collision(id0, id1, t);
}

Position calc_acc(const Position &p0, const Position &p1, double m1)
{
    Position dir = p1 - p0;
    double dist_sqr = dir.lengthsqr(),
           magnitude = G * m1 / dist_sqr;

    return dir / sqrt(dist_sqr) * magnitude;
}

/**
 * vim: syntax=cpp11 foldmethod=marker
 */
```

../../src/nbody.hh

```cpp
#ifndef __NBODY__
#define __NBODY__
#include"collision.hh"
#include"body.hh"
#include<cstdlib>

struct NBody{
    Position min_coord, max_coord; //Range from min to max for n body
    NBody(){}
    NBody(const Position &min, const Position &max):min_coord(min), max_coord(max){}
    double length(int axis) const{
        return max_coord[axis] - min_coord[axis];
    }
};

struct NBodyConfig{
    Body *body;
    int nbody;
    NBody domain;
    void init(const NBody &domain, int n);
    void readfile(const char* filename);

    NBodyConfig(){body = NULL;}
    ~NBodyConfig(){if(body) delete[]body;}
};

#endif
```

../../src/nbody.cc

```
#include"body.hh"
#include"nbody.hh"
#include<vector>
#include<cstdio>
#include<cstdlib>
#include<string.h>


static double Rand(double max){
    return max * (rand() / (RAND_MAX + 1.0));
}

static double RandRange(double low, double high){
    return low + Rand(high - low);
}

static ColorRGB RandColor(){
    return ColorRGB(Rand(1), Rand(1), Rand(1));
}

void NBodyConfig::init(const NBody &domain, int n){

    Position center = (domain.min_coord + domain.max_coord) / 2;
    nbody = n, this->domain = domain;
    body = new Body[n];

    body[0].id = 0;
    body[0].pos = center;
    body[0].v = Position(10, 30, 0);
    body[0].mass = 100000;
    body[0].r = 30;
    body[0].color = ColorRGB(0, 0, 0);

    double R = 200, V = 150;
    int start = 1;
    for (int i = start; i < n; i++){
        double angle = 2 * M_PI / (n-start)*i;
        body[i].pos = center + Position(R*cos(angle), R*sin(angle), 0);
        body[i].v = Position(V*sin(angle), V*cos(angle), 0);
        body[i].mass = i*5;
        body[i].r = pow(body[i].mass, 1/3.0);
        body[i].color = RandColor();
        body[i].id = i;
    }
}

void NBodyConfig::readfile(const char* filename){
    FILE *rfile;
    if (!strcmp(filename, "-"))
        rfile = stdin;
    else rfile = fopen(filename, "r");
    std::vector<Body> bvec;
    Body b;
    nbody = 0;
    while(fscanf(rfile, "%lf %lf %lf %lf %lf %lf %lf %lf", &b.pos.x, &b.pos.y, &b.pos.z, &b.v.x, &b.v.y, &b.v.z, &b.mass, &b
        b.color = RandColor();
        b.id = nbody;
        bvec.push_back(b);
        nbody++;
    }
    body = new Body[nbody];
    for (int i = 0; i < bvec.size(); i++)
        body[i] = bvec[i];
    if (rfile != stdin)
        fclose(rfile);
}
```

../../src/AABox.hh

```
#ifndef __AABOX__
#define __AABOX__

#include"body.hh"
#include<cstdlib>
#include<cmath>

#define REAL_MAX 10000000000.0
#define REAL_MIN -10000000000.0

struct AABox{
    Position min_coord, max_coord;
    inline double volume()const{
        double ret = 1;
        for (int i = 0; i < 3; i++){
            ret *= max_coord[i] - min_coord[i];
        }
        return ret;
    }

    void reset(){
        min_coord = Position(REAL_MAX, REAL_MAX, REAL_MAX);
        max_coord = Position(REAL_MIN, REAL_MIN, REAL_MIN);
    }

    bool in_box(const Position &pos) const{
        for (int i = 0; i < 3;i++){
            if (pos[i] > max_coord[i] || pos[i] < min_coord[i])
                return false;
        }
        return true;
    }

    AABox enhance(double width) const{
        AABox ret(*this);
        for (int i = 0; i < 3; i++){
            ret.max_coord.coord(i) += width;
            ret.min_coord.coord(i) -= width;
        }
        return ret;
    }

    bool collide(const Position &pos, const Position &vel, double dtime)const{
        if (in_box(pos))
            return true;
        Position eps = vel.unit() * EPS;
        for (int axis = 0; axis < 3; axis++){
            double v = vel[axis];
            if (fabs(v) < EPS) continue;

            double dmin = min_coord[axis] - pos[axis];
            double tmin = dmin / v;
            if (tmin > 0 && tmin <= dtime && in_box(pos+vel*tmin+eps)) return true;
            double dmax = max_coord[axis] - pos[axis];
            double tmax = dmax / v;
            if  (tmax > 0 && tmax <= dtime && in_box(pos+vel*tmax+eps)) return true;
        }
        return false;
    }
    bool collide_info(const Body &body, double &time, int &axis, int &mag){
        time = REAL_MAX;
        for (int i = 0; i < 3; i++){
            double v = body.v[i];
            if (fabs(v) < EPS) continue;
            double dmax = max_coord[i] - body.pos[i] - body.r;
            double tmax = dmax / v;
            if (tmax > 0 && tmax < time){
                time = tmax; axis = i; mag = 1;
            }
            double dmin = min_coord[i] - body.pos[i] + body.r;
            double tmin = dmin / v;
            if (tmin > 0 && tmin < time){
                time = tmin; axis = i; mag = 0;
            }
        }
        return time != REAL_MAX;
    }
};

#endif
```

../../src/engine.hh

```
#ifndef __ENGINE__
#define __ENGINE__

#include"nbody.hh"
#include"collision.hh"

class Engine{
    public:
        virtual void init(const NBodyConfig &conf) = 0;
        virtual Collision collision_detect(int id, double dtime) = 0;
        virtual void calc_vel(int id, const double &threshold, const double &dtime) =0;
};

#endif
```

../../src/simple.hh

```
#ifndef __SIMPLE__
#define __SIMPLE__

#include"engine.hh"
#include"AABox.hh"

class SimpleEngine : public Engine{
    public:
        virtual void init(const NBodyConfig &conf);
        virtual Collision collision_detect(int id, double dtime);

        virtual void calc_vel(int id, const double &threshold, const double &dtime);

        NBodyConfig conf;
        AABox boundary;
        Body *body;
};


#endif
```

../../src/simple.cc

```
#include"simple.hh"

void SimpleEngine::init(const NBodyConfig &conf){
    this->conf = conf;
    boundary.min_coord = conf.domain.min_coord;
    boundary.max_coord = conf.domain.max_coord;

    body = conf.body;
}

void SimpleEngine::calc_vel(int id, const double &threshold, const double &dtime){
    Position accel(0, 0, 0);
    for (int i = 0; i < conf.nbody; i++){
        if (i != id)
            accel += calc_acc(body[id].pos, body[i].pos, body[i].mass);
    }
    body[id].v += accel * dtime;
}

Collision SimpleEngine::collision_detect(int id, double dtime){
    int axis, mag;
    double time = REAL_MAX;
    Collision col;
    col.time = REAL_MAX;
    col.item0 = id;
    if (boundary.collide_info(body[id], time, axis, mag)){
        if (time < dtime){
            col.item1 = -1;
            col.time = time;
            col.axis = axis;
            col.mag = mag;
        }
    }
    for (int i = 0; i < conf.nbody; i++) //make col least
        if(i != id){
            Collision tcol = body_clsdtct(body[id], body[i], dtime);
            if (tcol.time < dtime && tcol.time > 0 && tcol.time < col.time)
                col = tcol;
        }

    if (col.time > dtime){
        col.time = -1; col.item1 = -1;
    }
    return col;
}
```

../../src/bsp.hh

```
#ifndef __BSP__
#define __BSP__

#include"nbody.hh"
#include"AABox.hh"
#include"engine.hh"

class BSPTree : public Engine {
    public:
        virtual void init(const NBodyConfig &conf);
        virtual Collision collision_detect(int id, double dtime);
        virtual void calc_vel(int id, const double &threshold, const double &dtime);

        BSPTree(){root = NULL;}
        ~BSPTree(){}

        struct Plane{
            int axis;
            double coord;
        };
        struct Node{
            Node *ch[2];
            int dep;
            Node(){ch[0] = ch[1] = NULL;}
            bool is_leaf() const {return ch[0] == NULL && ch[1] == NULL;}
            inline double volume() const {return box.volume();}
            Plane plane;

            AABox box;
            int id, n;
            Position cm;
            double mass;
        };
        void release (Node *root);
        AABox construct(int *ids, int n);

        Node *root;
        int m_nbody;
        int m_dep_max;
        Body *body;

        struct CmpBody{
            Body *body;
            int cid;
            CmpBody(Body *body, int cid):body(body), cid(cid){}
            bool operator () (int i, int j){
                return body[i].pos[cid] < body[j].pos[cid];
            }
        };

        AABox boundary;
        int find_middle(int *ids, int n, int cid);
        Node *build_tree(int *ids, int nbody, int dep);
        Collision do_col_dtct(Node *root, int id, double dtime);
        Collision body_clsdtct(int item0, int item1, double dtime);
        Position calc_acc(Node *root, int id, double threshold);

        static Position calc_single_acc(const Body &a, const Node *b);
        void check();
};

#endif
```

../../src/bsp.cc

```
#include "bsp.hh"

#include <cmath>
#include <algorithm>
#include <cstring>
#include <cassert>

using namespace std;

#define For(i, n) for (int i = 0; i < (n); i ++)

static int Rand(int max){
    return (int)(rand() / (RAND_MAX + 1.0) * max);
}

static int * alloc_ordered_array(int n)
{
    int *ret = new int[n];
    For(i, n) ret[i] = i;
    return ret;
}


```

```
void BSPTree::release(Node *root)
{
    if (!root)
        return;
    release(root->ch[0]);
    release(root->ch[1]);
    delete root;
}

void BSPTree::init(const NBodyConfig &conf)
{
    release(root);

    m_nbody = conf.nbody;
    body = conf.body;
    m_dep_max = log((double)m_nbody) / log(2.0) * 4;

    boundary.min_coord = conf.domain.min_coord;
    boundary.max_coord = conf.domain.max_coord;

    //memcpy(m_body, conf.body, sizeof(Body) * m_nbody);

    int *ids = alloc_ordered_array(this->m_nbody);
    root = build_tree(ids, m_nbody, 0);
    delete [] ids;
}

static double cube(const double &x) { return x * x * x; }

template<typename T>
static void checkmin(T &a, T &b){
    if (b < a) a = b;
}
template<typename T>
static void checkmax(T &a, T &b){
    if (b > a) a = b;
}

AABox BSPTree::construct(int *ids, int n)
{
    AABox box;
    box.reset();
    for (int i = 0; i < n; i ++)
    {
        int id = ids[i];
        for (int j = 0; j < 3; j ++)
        {
            if (box.min_coord.coord(j) > body[id].pos.coord(j) - body[id].r)
                box.min_coord.coord(j) = body[id].pos.coord(j) - body[id].r;
            if (box.max_coord.coord(j) < body[id].pos.coord(j) + body[id].r)
                box.max_coord.coord(j) = body[id].pos.coord(j) + body[id].r;
        }
    }

    return box;
}

int BSPTree::find_middle(int *ids, int n, int cid)
{
    //std::sort(ids, ids + n, CmpBody(body, cid));
    //return n / 2;

    int left = 0, right = n;

    int *buf = new int[n];
    int k = (n + 1)/ 2;

    while (left + 1 != right)
    {
        int mid = (left + right) >> 1;
        Body &b = body[ids[mid]];
        double bm = b.pos.coord(cid); // benchmark
        int head = 0, tail = right - left;
        for (int i = left; i < right; i ++)
            if (body[ids[i]].pos.coord(cid) < bm)
                buf[head ++] = ids[i];
            else buf[-- tail] = ids[i];
        while (tail < n && body[buf[tail]].pos.coord(cid) == body[buf[head]].pos.coord(cid))
            tail ++;
        memcpy(ids + left, buf, sizeof(int) * (right - left));
        if (k < head)
            right = left + head;
        else if (k > tail)
            left = left + tail, k -= tail;
        else
        {
            assert(left <= (n + 1) / 2 && (n + 1) / 2 < right);
            return (n + 1) / 2;
        }
    }

    delete buf;

    return right;
}


void BSPTree::check()
{
    return ;
    for (int i = 0; i < m_nbody; i ++)
    {
        assert(!isnan(body[i].pos.x));
        assert(!isnan(body[i].pos.y));
        assert(!isnan(body[i].pos.z));
    }
}

BSPTree::Node *BSPTree::build_tree(int *ids, int nbody, int dep)
{
    check();
    if (nbody == 0)
        return NULL;

    Node *root = new Node();
    root->dep = dep;
    root->box = construct(ids, nbody);
    root->n = nbody;

    root->cm = Position(0, 0, 0);
    root->mass = 0;
    For(i, nbody)
    {
        int id = ids[i];
        root->cm += body[id].pos * body[id].mass;
        root->mass += body[id].mass;
    }
    root->cm /= root->mass;

    if (nbody == 1)
    {
        root->id = *ids;
        return root;
    }

    root->plane.axis = Rand(3);
    int mid = find_middle(ids, nbody, root->plane.axis);
    assert(mid != nbody);
    root->plane.coord = (body[ids[mid]].pos[root->plane.axis] +
            body[ids[mid - 1]].pos[root->plane.axis]) / 2;

    root->ch[0] = build_tree(ids, mid, dep + 1);
    root->ch[1] = build_tree(ids + mid, nbody - mid, dep + 1);
    return root;
}

Collision BSPTree::collision_detect(int id, double dtime)
{
    check();
    Position pos = body[id].pos + body[id].v * dtime;
    double time = REAL_MAX;
    int axis, mag;
    Collision col;
    col.item0 = id;
    col.time = REAL_MAX - 1;
    if (boundary.collide_info(body[id], time, axis, mag))
        if (time > 0 && time < dtime)
        {
            col.item1 = -1;
            col.time = time;
            col.axis = axis;
            col.mag = mag;
        }

    Collision tcol = do_col_dtct(root, id, dtime);
    if (tcol.time > 0 && tcol.time < col.time)
        col = tcol;
    if (col.time > dtime || isinf(col.time))
    {
        col.time = -1;
        col.item1 = -1;
    }
    assert(!isinf(col.time));
    return col;
}

Collision BSPTree::do_col_dtct(Node *root, int id, double dtime)
{
    assert(body[id].id == id);
    Body &b = body[id];
    if (root->is_leaf())
        return body_clsdtct(id, root->id, dtime);

    if (!root->box.enhance(b.r).collide(b.pos, b.v, dtime))
        return Collision(-1, -1, -1);

    int chd = -1;
    if (root->ch[0] == NULL) chd = 1;
    else if (root->ch[1] == NULL) chd = 0;
    else if (b.pos[root->plane.axis] < root->plane.coord) chd = 0;
    else chd = 1;

    Collision ret = do_col_dtct(root->ch[chd], id, dtime);
    if (ret.item0 != -1)
        return ret;
    return do_col_dtct(root->ch[!chd], id, dtime);
}

static bool solve_quadratic_equation(const double &a, const double &b, const double &c,
        double &root0, double &root1)
{
    if (fabs(a) < EPS)
        return false;
    double delta = b * b - 4 * a * c;
    if (delta >= 0)
    {
        double sq = sqrt(delta),
               d = 1 / (2 * a);
        root0 = (-b - sq) * d;
        root1 = (-b + sq) * d;
        return true;
    }
    return false;
}

Collision BSPTree::body_clsdtct(int item0, int item1, double dtime)
{
    if (item0 == item1)
        return Collision(-1, -1, -1);
    Body &b0 = body[item0], &b1 = body[item1];
    Position AB = b1.pos - b0.pos,
           vab = b1.v - b0.v;
    double rab = (b0.r + b1.r);
    double a = vab.dot(vab),
           b = 2 * vab.dot(AB),
           c = AB.dot(AB) - rab * rab;

    double t0, t1, t;
    if (!solve_quadratic_equation(a, b, c, t0, t1))
        return Collision(-1, -1, -1);
    t = t0;
    if (t < 0) t = t1;
    if (t < 0 || t > dtime) return Collision(-1, -1, -1);
    if (t > EPS)
        t -= EPS / 2;
    return Collision(item0, item1, t);
}

void BSPTree::calc_vel(int id, const double &threshold, const double &dtime)
{
    Body &b = body[id];
    Position a = calc_acc(root, id, threshold);
    b.v += a * dtime;
    int asdf = 0;
}

Position BSPTree::calc_acc(Node *root, int id, double threshold)
{
    if (root == NULL)
        return Position(0, 0, 0);
    assert(!isnan(root->cm.x));
    Body &b = body[id];

    double cube_len = cube((root->cm - b.pos).length());
    double t = root->volume() / cube_len;
    if (root->is_leaf() || root->volume() / cube_len < threshold)
        return calc_single_acc(b, root);
    return calc_acc(root->ch[0], id, threshold)
        + calc_acc(root->ch[1], id, threshold);
}

Position BSPTree::calc_single_acc(const Body &a, const Node *b)
{
    if (b->is_leaf() && b->id == a.id)
        return Position(0, 0, 0);

    Position dir = b->cm - a.pos;
    double dist_sqr = dir.lengthsqr();
    double magnitude = G * b->mass / dist_sqr;

    Position ret = dir / sqrt(dist_sqr) * magnitude;
    if (dir.lengthsqr() < a.r * a.r)
        return ret*(-1);

    return ret;
}

/**
 * vim: syntax=cpp11 foldmethod=marker
 */
```

../../src/worker.hh

```
#ifndef __WORKER__
#define __WORKER__

#include<string>
#include"nbody.hh"
#include"engine.hh"

class Worker{
    public:
        int nworker;
        Engine *engine;
        virtual void init(int argc, char* argv[]){}
        virtual std::string name() const{}
        virtual int advance(const NBodyConfig &conf, double dtime){}
        void set_engine(Engine *engine){this->engine = engine;}
        void set_nworker(int n){this->nworker = n;}
        virtual int get_nworker(){return nworker;}
        virtual ~Worker(){}
};
#endif
```

../../src/openmp.hh

```
#ifndef __OPENMP__
#define __OPENMP__

#include"worker.hh"
#include"engine.hh"
#include<string>

class OpenmpWorker : public Worker
{
    public:
        bool *hash;
        Body *new_body;
        int work(const NBodyConfig &conf, int id, double dtime);

        virtual int advance(const NBodyConfig &conf, double dtime);
        virtual std::string name() const{return "openmp";}
};


#endif
```

../../src/openmp.cc

```
#include "openmp.hh"
#include <cassert>
#include <cstring>

int OpenmpWorker::advance(const NBodyConfig &conf, double dtime)
{
    int ncollide = 0;
    double threshold = 0.85;

```

```
#if 1
    double time_remain = dtime;
    while (time_remain)
    {
        assert(time_remain > 0);
        engine->init(conf);

#pragma omp parallel for num_threads(nworker)
        for (int i = 0; i < conf.nbody; i ++)
            engine->calc_vel(i, threshold, dtime);

        Collision min_col;
        min_col.time = REAL_MAX;
#pragma omp parallel for num_threads(nworker)
        for (int i = 0; i < conf.nbody; i ++)
        {
            Collision col =
                engine->collision_detect(i, dtime);
            if (col.item0 == -1)
                continue;
            if (col.time > 0 && col.time < min_col.time)
                min_col = col;
        }
        if (min_col.time < 0 || min_col.time > time_remain)
            break;
        ncollide ++;
        time_remain -= min_col.time;

#pragma omp parallel for num_threads(nworker)
        for (int i = 0; i < conf.nbody; i ++)
            if (i != min_col.item0 && i != min_col.item1)
                conf.body[i].advance(min_col.time);
        if (min_col.item1 == -1) // hit wall
            conf.body[min_col.item0].v[min_col.axis] *= -1;
        else
            body_collide(conf.body[min_col.item0], conf.body[min_col.item1], min_col.time, true);
    }
#pragma omp parallel for num_threads(nworker)
    for (int i = 0; i < conf.nbody; i ++)
        conf.body[i].advance(time_remain);

#else
    engine->init(conf);

#pragma omp parallel for num_threads(nworker)
    for (int i = 0; i < conf.nbody; i ++)
        engine->cal_new_velocity(i, threshold, dtime);

    Body *new_body = new Body[conf.nbody];

    memcpy(new_body, conf.body, sizeof(Body) * conf.nbody);
#pragma omp parallel for num_threads(nworker)
    for (int i = 0; i < conf.nbody; i ++)
    {
        int p = i;
        Collision col =
            engine->collision_detect(i, dtime);
        if (col.item1 != -1)
        {
            if (body_collide(new_body[p], conf.body[col.item1], col.time))
            {
                ncollide ++;
                new_body[p].advance(dtime - col.time);
            }
            else new_body[p].advance(dtime);
        }
        else if (col.time > 0)
        {
            ncollide ++;
            Body &b = new_body[p];
            b.advance(col.time);
            b.v[col.axis] *= -1;
            b.advance(dtime - col.time);
        }
        else new_body[p].advance(dtime);
    }

    memcpy(conf.body, new_body, sizeof(Body) * conf.nbody);

    delete [] new_body;

#endif
    //if (ncollide)
    //    log_printf("ncollide: %d", ncollide);

    return true;
}

/**
 * vim: syntax=cpp11 foldmethod=marker
 */
```

../../src/pthread.hh

```
#ifndef __PTHREAD__
#define __PTHREAD__

#include"worker.hh"

#include<pthread.h>

class PthreadWorker : public Worker{
    public:
        virtual int advance(const NBodyConfig &conf, double dtime);
        virtual std::string name() const {return "pthread";}
        void thread_calc_vel();
        void thread_calc_col();

        struct TaskSche{  //fetch the calc task
            public:
                pthread_mutex_t mutex;
                int cur, ntask, ntf_max;
                void init(int ntask, int ntf_max){
                    this->ntask = ntask; cur = 0; this->ntf_max = ntf_max;
                }
                int fetch_task(int &left, int &right){
                    pthread_mutex_lock(&mutex);
                    int n = ntask - cur;
                    if (n > ntf_max) n = ntf_max;
                    left = cur;
                    cur += n;
                    right = cur;
                    pthread_mutex_unlock(&mutex);
                    return n;
                }

        };

        TaskSche *taskpool;
        double dtime;
        double threshold;
        int ncollide;
        const NBodyConfig *conf;

        Body *body_buf;
};
```

```

#endif
```

../../src/pthread.cc

```
#include"pthread.hh"

#include<pthread.h>
#include<cstring>


struct PthreadArg{
    PthreadWorker *worker;
    int task_type;
};

static void *call_pthread(void *arg){ //every thread calc of vel and col
    PthreadArg *parg = static_cast<PthreadArg *>(arg);
    if (parg->task_type == 0)
        parg->worker->thread_calc_vel();
    else
        parg->worker->thread_calc_col();
    pthread_exit(NULL);
}

void PthreadWorker::thread_calc_vel(){
    int left, right;
    while (taskpool->fetch_task(left, right)){
        for (int i = left; i < right; i++)
            engine->calc_vel(i, threshold, dtime);
    }
}

void PthreadWorker::thread_calc_col(){
    int left, right;
    Body *body = conf->body;
    Body *new_body = body_buf;
    while(taskpool->fetch_task(left, right)){
        for (int i = left; i < right; i++){
            int p = i;
            Collision col = engine->collision_detect(p, dtime);
            if (col.item1 != -1){
                if (body_collide(new_body[p], body[col.item1], col.time)){
                    ncollide++;
                    new_body[p].advance(dtime - col.time);
                }
                else new_body[p].advance(dtime);
            }
            else if(col.time > 0){ //boundary
                ncollide ++;
                Body &b = new_body[p];
                b.advance(col.time);
                b.v[col.axis] *= -1;
                b.advance(dtime-col.time);
            }
            else new_body[p].advance(dtime);
        }
    }
}


int PthreadWorker::advance(const NBodyConfig &conf, double dtime){
    taskpool = new TaskSche();
    this->conf = &conf;
    this->dtime = dtime;
    threshold = 1;
    engine->init(conf);

    pthread_t *threads = new pthread_t[nworker];
    PthreadArg arg;
    arg.worker = this;
    arg.task_type = 0;
    int ntf_max = conf.nbody / nworker / 8;
    if (ntf_max < 10) ntf_max = 10;
    if (ntf_max > 100) ntf_max = 100;

    taskpool->init(conf.nbody, ntf_max);
    for (int i = 0; i < nworker; i++)
        pthread_create(threads + i, NULL, call_pthread, &arg);
    for (int i = 0; i < nworker; i++)
        pthread_join(threads[i], NULL);

    ncollide = 0;
    arg.task_type = 1;
    taskpool->init(conf.nbody, ntf_max);
    body_buf = new Body[conf.nbody];
    memcpy(body_buf, conf.body, sizeof(Body)*conf.nbody);
    for (int i = 0; i < nworker; i++)
        pthread_create(threads+i, NULL, call_pthread, &arg);
    for (int i = 0; i < nworker; i++)
        pthread_join(threads[i], NULL);
    memcpy(conf.body, body_buf, sizeof(Body)*conf.nbody);

    delete threads;
    delete taskpool;
    delete body_buf;
    return true;
}
```

../../src/mpi.hh

```
#ifndef __MPI__
#define __MPI__


#include"worker.hh"
#include"nbody.hh"

#include<mpi.h>

#define NEW_CONF 1
#define EXIT_CONF 0

enum MPITaskType{
    MPI_VEL,
    MPI_POS,
    MPI_END
};
struct MPITask{
    MPITaskType type;
    int left, right;
};

struct TaskConf{
    int type, nbody;
    double dtime;
};

class MPIWorker : public Worker{
    public:
    virtual void init(int argc, char* argv[]);
    virtual int advance(const NBodyConfig &conf, double dtime);
    virtual std::string name() const{return "MPI";}
    virtual int get_nworker() const {return npro;}

    MPIWorker();
    ~MPIWorker();

    MPI_Datatype MPI_TYPE_CONF, MPI_TYPE_BODY;
    NBodyConfig nbconf;
    int npro, pro_id;
    int ntask_assigned;
    int task_start, task_end;
    int decompose_task(int ntask, int &left, int &right);
    int get_task_per_proc(int ntask);
    void work_vel(double dtime);
    void work_col(double dtime);
    void advanceall(double dtime);
    void gather_body();

    bool master_conf(TaskConf conf, const NBodyConfig &nbconf);
    Body *body;
    int nbody;

    bool slave_conf(TaskConf &conf, NBodyConfig &nbconf);

};


#endif
```

../../src/mpi.cc

```
#include<mpi.h>
#include"mpi.hh"
#include<string.h>

#define ROOT_PROC 0


MPIWorker::MPIWorker(){}

void MPIWorker::init(int argc, char* argv[]){
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npro);
    nworker = npro;
    MPI_Comm_rank(MPI_COMM_WORLD, &pro_id);

    body = NULL;

    MPI_Type_contiguous(sizeof(TaskConf), MPI_CHAR, &MPI_TYPE_CONF);
    MPI_Type_commit(&MPI_TYPE_CONF);
    MPI_Type_contiguous(sizeof(Body), MPI_CHAR, &MPI_TYPE_BODY);
    MPI_Type_commit(&MPI_TYPE_BODY);
}

int MPIWorker::get_task_per_proc(int ntask){
    int task_per_proc = ntask/npro;
    int task_remain = ntask - task_per_proc *npro;
    if (task_remain)
        task_per_proc ++;
    return task_per_proc;
}

int MPIWorker::decompose_task(int ntask, int &left, int &right){
    if (ntask < npro){
        if (pro_id < ntask){
            left = pro_id;
            right = pro_id + 1;
            return 1;
        }
        else return 0;
    }

    int task_per_proc = get_task_per_proc(ntask);

    left = task_per_proc * pro_id;
    right = task_per_proc * (pro_id + 1);
    if (right > ntask)
        right = ntask;
    return right - left;
}

void MPIWorker::work_vel(double dtime){
    ntask_assigned = decompose_task(nbody, task_start, task_end);
    double threshold = 0.5;
    for (int i = task_start; i < task_end; i++)
        engine->calc_vel(i, threshold, dtime);
}

void MPIWorker::work_col(double dtime){
    Body *new_body = new Body[task_end - task_start];
    memcpy(new_body, body+task_start, sizeof(Body)*(task_end-task_start));
    for (int i = task_start; i < task_end; i++){
        int p = i - task_start;
        Collision col = engine->collision_detect(i, dtime);
        if (col.item1 != -1){
            if (body_collide(new_body[p], body[col.item1], col.time)){
                new_body[p].advance(dtime - col.time);
            }
            else new_body[p].advance(dtime);
        }
        else if (col.time > 0){
            Body &b = new_body[p];
            b.advance(col.time);
            b.v[col.axis] *= -1;
            b.advance(dtime - col.time);
        }
        else new_body[p].advance(dtime);
    }
    memcpy(body+task_start, new_body, sizeof(Body) * task_end-task_start);
    delete [] new_body;
}

void MPIWorker::gather_body(){
    int task_per_proc = get_task_per_proc(nbody);
    MPI_Allgather(body + task_start, task_per_proc, MPI_TYPE_BODY, body, task_per_proc, MPI_TYPE_BODY, MPI_COMM_WORLD);
}

void MPIWorker::advanceall(double dtime){
    for (int i = 0; i < nbody; i++)
        body[i].advance(dtime);
}

int MPIWorker::advance(const NBodyConfig &conf, double dtime){
    if (pro_id == ROOT_PROC){
        if (body == NULL){
            int real_nbody = get_task_per_proc(conf.nbody) * npro;
            body = new Body[real_nbody];
        }
        memcpy(body, conf.body, sizeof(Body) * conf.nbody);
        nbody = conf.nbody;

        TaskConf tconf;
        tconf.type = NEW_CONF;
        tconf.nbody = conf.nbody;
        tconf.dtime = dtime;

        nbconf = conf;
        nbconf.body = body;

        master_conf(tconf, nbconf);
        engine->init(nbconf);

        work_vel(tconf.dtime);
        gather_body();

        work_col(tconf.dtime);
        gather_body();

        memcpy(conf.body, body, sizeof(Body) * nbody);
        return true;
    }
    else {
        TaskConf tconf;
        while(slave_conf(tconf, nbconf)){
            engine->init(nbconf);
            work_vel(tconf.dtime);

            gather_body();

            work_col(tconf.dtime);
            gather_body();
        }
        return false;
    }
}

bool MPIWorker::master_conf(TaskConf conf, const NBodyConfig &nbconf){
    MPI_Bcast(&conf, 1, MPI_TYPE_CONF, ROOT_PROC, MPI_COMM_WORLD);
    if (conf.type == EXIT_CONF)
        return true;

    MPI_Bcast(body, nbconf.nbody, MPI_TYPE_BODY, ROOT_PROC, MPI_COMM_WORLD);

    return true;
}

bool MPIWorker::slave_conf(TaskConf &conf, NBodyConfig &nbconf){
    MPI_Bcast(&conf, 1, MPI_TYPE_CONF, ROOT_PROC, MPI_COMM_WORLD);
    if (conf.type == EXIT_CONF)
        return false;
    nbconf.nbody = conf.nbody;
    nbody = nbconf.nbody;

    if (body == NULL)
    {
        body = new Body[get_task_per_proc(conf.nbody) * npro];
        nbconf.body = body;
    }
    MPI_Bcast(body, nbconf.nbody, MPI_TYPE_BODY, ROOT_PROC, MPI_COMM_WORLD);
    return true;
}

MPIWorker::~MPIWorker(){
    if (pro_id == ROOT_PROC){
        TaskConf conf;
        conf.type = EXIT_CONF;
        master_conf(conf, nbconf);
    }
    MPI_Finalize();
}
```

../../src/gtk.hh

```
#ifndef __GTK__
#define __GTK__

#include<gtk/gtk.h>

```

```
#include<getopt.h>
#include<unistd.h>

#include<cstring>
#include<algorithm>
#include<cstdlib>
#include<cmath>
#include<queue>
#include<cassert>

#include"nbody.hh"
#include"bsp.hh"
#include"simple.hh"
#include"collision.hh"
#include"openmp.hh"
#include"pthread.hh"
#include"mpi.hh"
#include"timer.hh"

const double FPS = 1000;
const char *progname;
bool show_window = true;


struct Camera
{
    Position sight_pos;
    Position view_dir, up_dir;
    double frustum_dist;
    double fwidth, fheight; // configuration of the frustum
};

struct RenderConfig
{
    NBodyConfig nbody;

    Worker *worker;
    Camera camera;

    int wwidth, wheight; // window width and height

    int width() const { return wwidth; }
    int height() const { return wheight; }

    void set_worker(Worker *worker)
    { this->worker = worker; }

    bool advance(double dtime)
    {
        if (fabs(dtime) < EPS)
            return true;
//        assert(dtime > 0);
        return worker->advance(nbody, dtime);
    }

    void init() {}

} rconf;
Timer timer;

long long frame_count;
double runtime = 1e100;
Timer gtimer;

FpsCounter fps;

bool rendered = false;
```

```
void exit_on_timeout()
{
    if (rendered)
    {
        double rtime = gtimer.end() / 1000.0;
        printf("worker: %s\n", rconf.worker->name().c_str());
        printf("nworker: %d\n", rconf.worker->get_nworker());
        printf("running time: %lf\n", rtime);
        printf("total frames: %lld\n", frame_count);
        printf("average fps: %lf\n", frame_count / rtime);
    }
    delete rconf.worker;

    exit(0);
}

bool cmpZ(const Body &a, const Body &b)
{
    return a.pos.z + a.r > b.pos.z + b.r;
}
static gboolean delete_event(
        GtkWidget *,
        GdkEvent *,
        gpointer )
{
    return FALSE;
}

static void destroy(GtkWidget *, gpointer )
{
    gtk_main_quit();
}

static gboolean cb_timeout(GtkWidget *widget)
{
    if (widget->window == NULL)
        return FALSE;

    gtk_widget_queue_draw_area(widget, 0, 0, widget->allocation.width, widget->allocation.height);
    return TRUE;
}
int fcnt = 0;
static gboolean da_expose_callback(
        GtkWidget *widget,
        GdkEventExpose *,
        gpointer )
{
    if (timer.end() > EPS)
    {
        fps.count();
        frame_count ++;
        Timer tengine;
        tengine.begin();
        rconf.advance(timer.end() / 1000.0);
        //log_printf("engine time: %llums\n", tengine.end());
        timer.begin();
        if (gtimer.end() > runtime * 1000)
            exit_on_timeout();
    }

    if (!show_window)
        return FALSE;
    //return FALSE;
    cairo_t *cr = gdk_cairo_create(widget->window);

    // clear scene
    cairo_save (cr);
    cairo_new_path(cr);
    cairo_arc(cr, 0, 0, 900, 0, 2*M_PI);
    cairo_close_path(cr);
    cairo_set_source_rgb(cr, 255.0, 255.0, 255.0);
    cairo_set_operator (cr, CAIRO_OPERATOR_CLEAR);
    cairo_fill_preserve(cr);
    cairo_stroke(cr);
    cairo_paint (cr);
    cairo_restore (cr);

    NBodyConfig &nbody = rconf.nbody;
    Body *body = new Body[nbody.nbody];
    memcpy(body, nbody.body, sizeof(Body) * nbody.nbody);

    std::sort(body, body + nbody.nbody, cmpZ);
    Camera &camera = rconf.camera;

    cairo_save(cr);
    cairo_new_path(cr);
    cairo_arc(cr, 0.0, 0.0, 10000.0, 0, 2*M_PI);
    cairo_close_path(cr);
    cairo_set_source_rgb(cr, 255.0, 255.0, 255.0);
    cairo_fill_preserve(cr);
    cairo_stroke(cr);
    cairo_restore(cr);


    NBody &domain = rconf.nbody.domain;
    for (int i = 0; i < nbody.nbody; i ++)
    {
        Body &b = body[i];
        int x = b.pos.x / domain.length(0) * rconf.width(),
            y = b.pos.y / domain.length(1) * rconf.height();

        cairo_save(cr);

        cairo_new_path(cr);
        cairo_arc(cr, x, y, b.r, 0, 2 * M_PI);
        cairo_close_path(cr);

        ColorRGB &col = body[i].color;

        cairo_set_source_rgb(cr, col.r, col.g, col.b);
        cairo_fill_preserve(cr);
        cairo_stroke(cr);

        cairo_restore(cr);
    }


    delete [] body;
    cairo_set_source_rgb(cr, 0.0, 100.5, 0.5);;
    cairo_select_font_face(cr, "simsun",
            CAIRO_FONT_SLANT_NORMAL,
            CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size(cr, 20);
    cairo_move_to(cr, 420, 30);

    char sfps[20];
    snprintf(sfps, 20, "FPS: %.2lf", fps.fps());
    if (fcnt == 1000) {}
    cairo_show_text(cr, sfps);

    cairo_destroy(cr);
    return FALSE;
}

void show_init(int argc, char *argv[])
{
    gtk_init(&argc, &argv);
}

void show()
{
    GtkWidget *window;
    int border_width = 0;

    int window_width = rconf.width() + border_width * 2,
        window_height = rconf.height() + border_width * 2;
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_container_set_border_width(GTK_CONTAINER(window), border_width);
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_window_set_default_size(GTK_WINDOW(window), window_width, window_height);

    //gtk_widget_set_app_paintable(window, TRUE);
    gtk_widget_set_size_request(window, window_width, window_height);

    GtkWidget *da = gtk_drawing_area_new();
    gtk_widget_set_size_request(da, rconf.width(), rconf.height());

    g_signal_connect(window, "delete-event",
            G_CALLBACK(delete_event), NULL);
    g_signal_connect(window, "destroy",
            G_CALLBACK(destroy), NULL);

    /*
    gtk_widget_add_events(da, GDK_BUTTON_PRESS_MASK);
    g_signal_connect(da, "button-press-event",
            G_CALLBACK(cb_clicked), NULL);
    */

    g_signal_connect(da, "expose_event",
            G_CALLBACK(da_expose_callback), NULL);

    g_timeout_add(1000 / FPS, (GSourceFunc)cb_timeout, da);

    gtk_container_add(GTK_CONTAINER(window), da);
    gtk_widget_show_all(window);

    timer.begin();
    gtk_main();
}








int str2num(const std::string &str)
{
    int ret = 0, sign = 1, start = 0;
    if (str[0] == '-')
        sign = -1, start = 1;
    for (size_t i = start; i < str.length(); i ++)
        ret = ret * 10 + str[i] - '0';
    return ret * sign;
}

double str2real(const std::string &str)
{
    double ret;
    sscanf(str.c_str(), "%lf", &ret);
    return ret;
}
#endif
```

../../src/main.cc

```
#include"gtk.hh"
#include"nbody.hh"
#include"pthread.hh"
#include"openmp.hh"
#include"mpi.hh"
#include"bsp.hh"
#include"simple.hh"
#include"AABox.hh"

#include <string>
#include <algorithm>
#include <cstdlib>
#include <cmath>
#include <cstring>

Worker *worker_factory(const char *name)
{
    std::string w = name;
    if (w == "openmp")
        return new OpenmpWorker();
    if (w == "pthread")
        return new PthreadWorker();
    if (w == "mpi")
        return new MPIWorker();
    fprintf(stderr, "Unkown worker: %s\n", name);
    exit(-1);
}


int main(int argc, char *argv[])
{
    srand(2);
    progname = argv[0];

    int nworker = sysconf(_SC_NPROCESSORS_ONLN);
    option long_options[] = {
        {"nworker",        required_argument,    NULL, 'n'},
        {"size",        required_argument,    NULL, 's'},
        {"silent",        no_argument,        NULL, 'w'},
        {"parallel",    required_argument,    NULL, 'p'},
        {"input",        required_argument,    NULL, 'i'},
        {"nstar",        required_argument,    NULL, 'r'},
        {"runtime",        required_argument,    NULL, 't'},
        {"fps-test",    no_argument,        NULL, 'f'},
        {"interval",    required_argument,    NULL, 'l'},
    };

    int width = 600, height = 600;
    std::string para = "openmp", input = "";
    int nstar = 20;
    int opt;
    double time_interval = 0.01;
    bool fpstest = false;
    while ((opt = getopt_long(argc, argv, "l:fr:t:n:s:wp:i:", long_options, NULL)) != -1)
    {
        switch (opt)
        {
            case 'l':
                time_interval = str2real(optarg);
                break;
            case 'f':
                fpstest = true;
                break;
            case 't':
                runtime = str2real(optarg);
                break;
            case 'r':
                nstar = str2num(optarg);
                break;
            case 'n':
                nworker = str2num(optarg);
                break;

            case 's':
                {
                    int w, h;
                    if (sscanf(optarg, "%dx%d", &w, &h) != 2)
                        return 0;
                    width = w, height = h;
                }
                break;

            case 'w':
                show_window = false;
                break;

            case 'p':
                para = optarg;
                break;

            case 'i':
                input = optarg;
                break;
        }
    }

    show_init(argc, argv);

    NBody domain(Position(0, 0, 0), Position(600, 600, 100));
    rconf.nbody.init(domain, nstar);
    if (input.length())
        rconf.nbody.readfile(input.c_str());

    rconf.nbody.domain = domain;
    rconf.wwidth = width, rconf.wheight = height;

    Worker *worker = worker_factory(para.c_str());
    worker->init(argc, argv);
    worker->set_nworker(nworker);
    worker->set_engine(new BSPTree());
    rconf.set_worker(worker);
    rconf.init();
    gtimer.begin();
    if (fpstest)
    {
        while (true)
        {
            if (gtimer.end() > runtime * 1000)
                break;
            frame_count ++;
            if (!rconf.advance(time_interval))
                break;
            rendered = true;
        }
    }
    else if (rconf.advance(0.01))
        show();
    exit_on_timeout();
    return 0;
}
```

../../src/timer.hh

```
#include<queue>
#include<cstring>


typedef long long Time_t;

enum TimerPrecision
{
    TIMER_PRECISION_S,        // second
    TIMER_PRECISION_MS,        // millisecond
    TIMER_PRECISION_US        // microsecond
};

/*
 * Default timer precision is millisecond
 */
class Timer
{
    public:
        Timer();
        ~Timer();
        Time_t time() const;
        Time_t begin();
        Time_t end();
        Time_t duration();
        void sleep(Time_t time);

    private:
        TimerPrecision precision;
        Time_t time_begin;
        Time_t time_end;
};

class FpsCounter
{
    protected:
        std::queue<Time_t> que;

    public:
        FpsCounter();
        ~FpsCounter();
        void count();
        void reset();
        double fps();
};
```

../../src/timer.cc

```
#include"timer.hh"
#include<sys/time.h>
#include<unistd.h>

static Time_t time_prec_div[] = {1000000, 1000, 1};
static Time_t time_us();
static Time_t toMs(Time_t time, TimerPrecision prec);

Timer::Timer()
{
    precision = TIMER_PRECISION_MS;
}

Timer::~Timer()
{
}

Time_t Timer::time() const
{
    return time_us() / time_prec_div[precision];
}

Time_t Timer::begin()
{
    time_begin = time_us();
    return time_begin / time_prec_div[precision];
}

Time_t Timer::end()
{
    time_end = time_us();
    return duration();
}

Time_t Timer::duration()
{
    return (time_end - time_begin) / time_prec_div[precision];
}

void Timer::sleep(Time_t time)
{
    time = toMs(time, precision);
    usleep(time);
}


Time_t time_us()
{
    timeval tv;
    gettimeofday(&tv, 0);
    return (Time_t)tv.tv_sec * (Time_t)1000000 + (Time_t)tv.tv_usec;
}


Time_t toMs(Time_t time, TimerPrecision prec)
{
    if (prec == TIMER_PRECISION_S)
        return time * 1000;
    if (prec == TIMER_PRECISION_MS)
        return time;
    if (prec == TIMER_PRECISION_US)
        return time / 1000;
    return -1;
}


FpsCounter::FpsCounter()
{
}


FpsCounter::~FpsCounter()
{
}

void FpsCounter::count()
{
    que.push(Timer().time());
    while (que.back() - que.front() > 1000)
        que.pop();
}

void FpsCounter::reset()
{
    while (!que.empty())
        que.pop();
}

```

```cpp
double FpsCounter::fps()
{
    return que.size();
}
```