北京邮电大学软件学院

2019-2020 学年第一学期实验报告

课程名称： 并行计算

项目名称： 天体运动模拟

项目完成人：

姓名： 平雅霓 **学号：** 2017211949

指导教师： 卢本捷

日　　期： **2019 年 12 月 14 日**

一、 实验目的

以分治的方法设计并实现天体运动模拟。

二、 实验内容

1．在二维平面上以牛顿经典力学模拟宇宙运行模型。

2．假设有 10000 个星体。

3．星体的质量、位置、初始速度以随机的方式生成。

4．每秒钟进行一次计算。

5．以点绘图的方式描述天体的位置及其变化。

三、 实验环境

1. 两台或以上的 windows 或 linux 等。

2. Visual studio 2017

四、 实验要求

1. 建立 Barnes Hut 模型

2. 以天体位置为依据建立 4 叉树。

3. 计算质量与中心

4. 计算每个点的受力与加速度。

5. 更新天体位置。

6. 观察最后的宇宙运行情况。

五、 MPICH 实验步骤

1. 问题描述与分析

本实验要求制作一个简化的牛顿力学宇宙局部模型。

太空中有 N 个天体，任意两个天体的作用力为万有引力，每个天体都会受到其他物体的影响，根据牛顿第二运动定律且不考虑考虑相对论的影响（质量的改变，空间的弯曲等）：

$$F = \frac{Gm_a m_b}{r^2}$$

其准确的运行模式：

$$F = ma$$

微分方程组：$F = m\frac{dv}{dt}$　　$\mathrm{V} = \frac{dx}{dt}$

注意 F，V，X 均为向量（多个天体）。

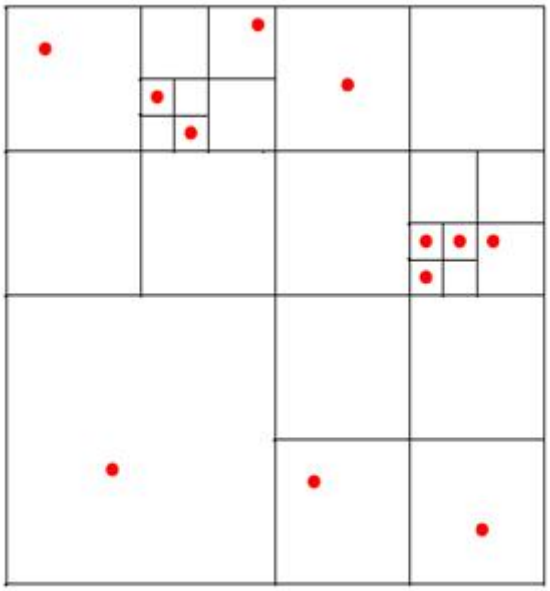
在求解的时候，我们需要将运动轨迹进行离散化模拟，时间被切割分为特定的间隔 t_0，t_1，$\Delta\mathrm{t}$ 则牛顿方程就变为了 $F = \frac{\mathrm{m}(v^{t+1} - v^t)}{\Delta t}$，其速度变为 $\mathrm{v}^{\mathrm{t}+1} = v^t + \frac{F\Delta t}{m}$，而位置变为了 $\mathrm{x}^{t+1} - x^t = v\Delta t$，这是一个天体的变化过程，其他每个天体都按照这样的方式运动，下一个时段的时候重新开始。

对于一个天体而言，F,V,X 均为三维向量，需要在三个方向上分解，在此模型下平衡态最终会坍缩为一个点。

使用并行化的方法来模拟天体运动，首先我们要对初始形体进行简单的划分，每个处理器对应一组天体，天体之间的受力情况，通过消息传递。在实验中，我采用了近似处理的方法，计算了一堆形体的中心的质量来代

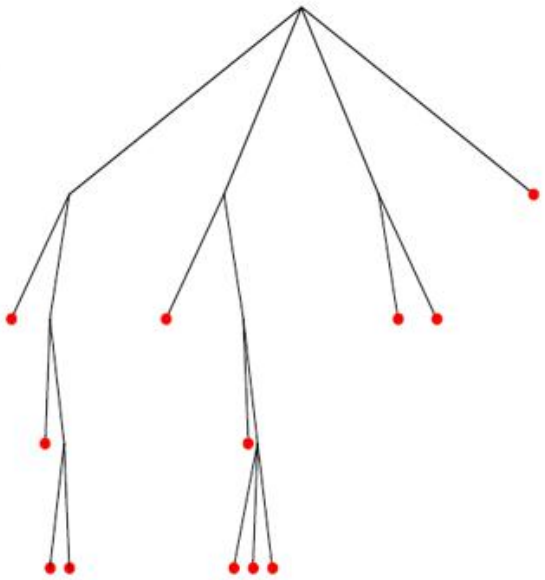
表要计算的形体以外的星体质量，采用 Barnes-Hut 算法的思想，进行空间划分。因为要将空间划分为 8 个子立方体，三位空间结构不便于用图形化的方式进行展示，所以只进行了 x，y 轴的计算，使用 2D 的方式表示出了星体的变化过程。

2. 原理解释



此图是 Barnes-Hut 算法的简单展示图，图中展示了划分的规则，即先将大正方形划分为 4 个小正方形，然后将小正方形继续用同样的方式进行划分，这个想法像极了递归，在实现的时候可以采用递归的方式来进行建立四叉树。图中的红点表示此划分中含有星体，根据 Barnes-Hut 算法，整个空间分为 8 个子立方体。如果某子立方体不包含物体，则被删除，否则保留下来。如果子立方体包含超过一个物体，则递归地划分。直至只包

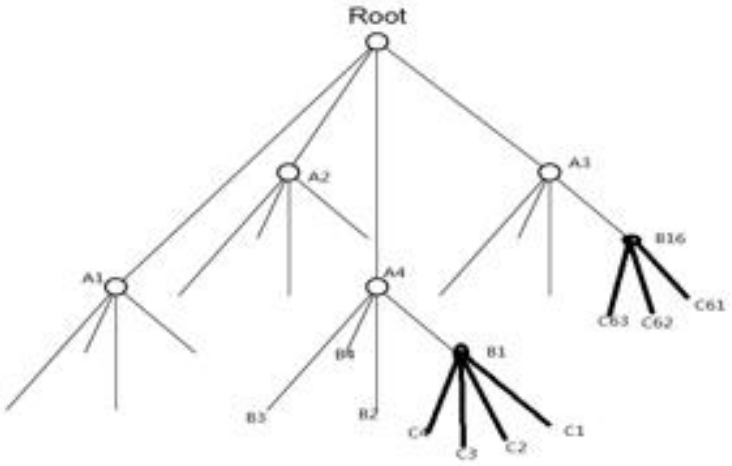
含一个物体为止。



Partial quadtree

此图为构建的四叉树样式。下面我们拿八叉树立方体来解释原理。

每个子立方体的总质量为 $\sum_{i=0}^{7} m_i$，每个子立方体的重心为 $C = \frac{1}{M}\sum_{i=0}^{7} c_i m_i$，当树构建好后，每个子立方体的质量和重心可以通过遍历树的方式用上面的式子计算，并存储在节点上。每个物体受到的力只需要从根节点开始遍历树即可，底层立方体内的点，以单个天体的方式计算，上层直至顶层的节点均以立方体的方式计算，这样的话计算复杂度会极度减小为 $O(\mathrm{n}\log n)$。



如图，计算 C1 点（叶子节点）的受力。只需计算：

A1，A2, A3 第一级立方体

B2，B3，B4 第二级立方体

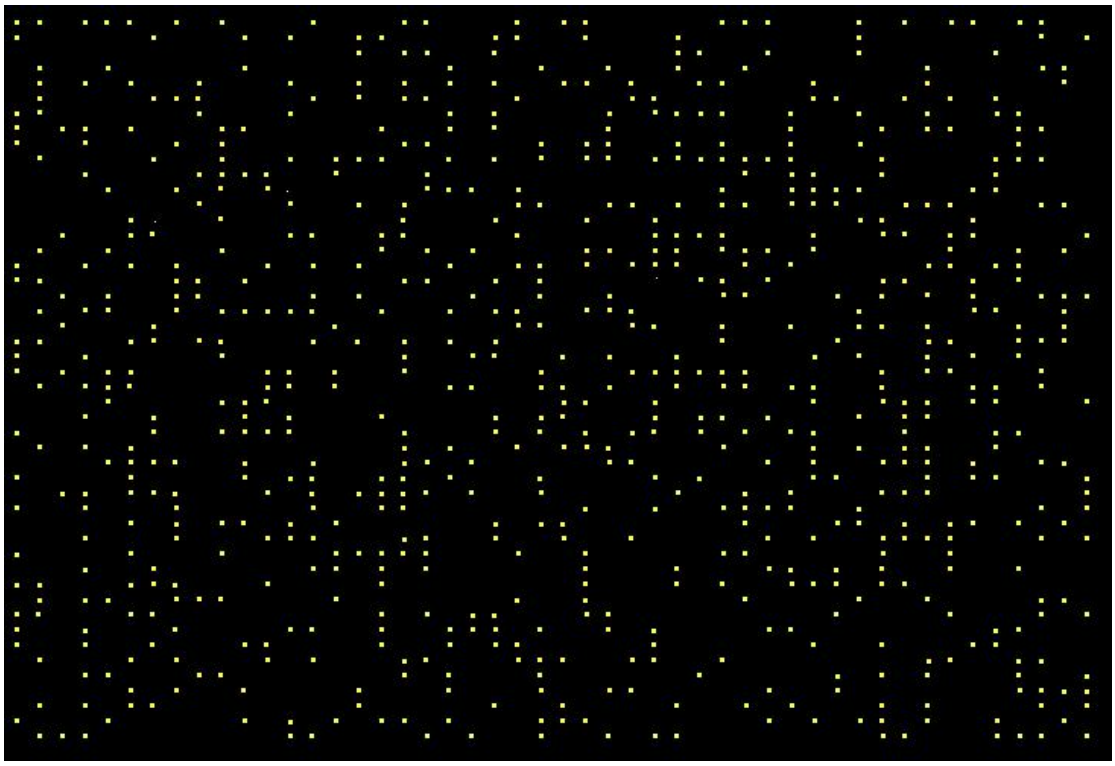
C2，C3，C4 叶子节点

复杂度为 logn，n 个物体全部算一遍，复杂度为 nlogn。

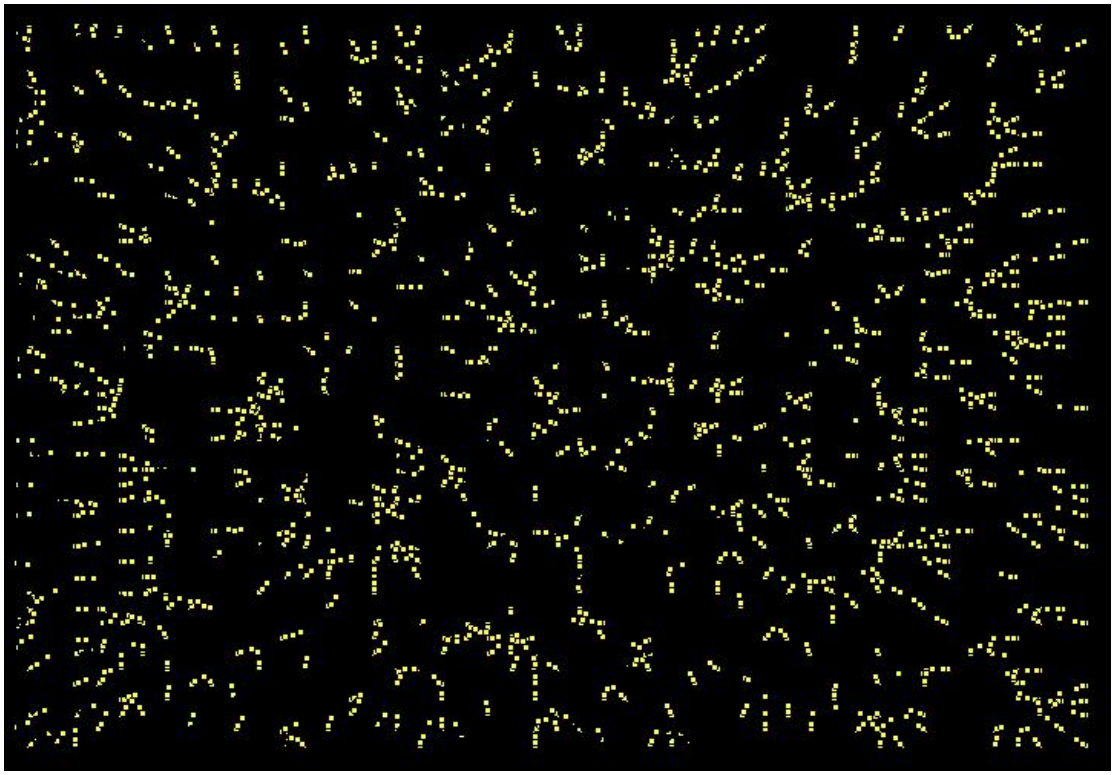
3. 运行结果分析

（1） 初始化界面，随机生成星体的位置。

随机生成 1000*size 个点(size 为进程数)，坐标随机分布在（0-48，0-48）的范围内，质量随机生成。随机生成的点因为有一个确定的范围，可以看出。点的分布十分均匀。



(2) 500 次迭代后，聚拢的趋势变得比较明显。



4. 代码实现过程

```
typedef struct {
    double x, y, z;
    double mass;
} Particle;
```

结构体定义了划分的单位，并在定义结构体变量时使用 Particle。其中有 x、y、z 属性，分别表示划分粒子的三维坐标，它们的类型是 double，mass 是划分单位的质量，数据类型也是 double。

```
/* 储存力和速度*/
typedef struct {
    double xold, yold, zold;
    double fx, fy, fz;
} ParticleV;
```

结构体定义了每个粒子的受力和速度。xold、yold、zold 是三维坐标中每一个粒子在位置改变前的坐标。fx、fy、fz 是三维坐标中每一个力方向的分力大小。

```
typedef struct {
    MPI_Comm     mycomm;
    int          left, right;
    MPI_Datatype type;
    Particle     *buf1, *buf2, *buf;
    MPI_Request  requests[2];
    int          stage, last_stage, maxlen;
    int          typesize;
} MPE_Pipe;
```

结构体定义了每个处理器的信息。结构体名称为 MPE_Pipe，每个处理器对应一组天体，mycomm 是进程管理器，left 和 right 是星体移动的方向，type 是 MPI_datatype，包括了 MPI 中的一系列数据类型，buf1、

buf2、buf 是缓存，requests 是请求，stage 和 last_stage 是处理需要进行的阶段。

```
void MPE_Pipe_create(MPI_Comm comm, MPI_Datatype type, int maxsize, void **pipe)
{
    MPE_Pipe  * news = (MPE_Pipe *)malloc(sizeof(MPE_Pipe));
    int  trues = 1, size, dsize, maxbuf;
    MPI_Aint dextent;

    /* 创建多个进程 */
    MPI_Comm_size(comm, &size);
    MPI_Cart_create(comm, 1, &size, &trues, 1, &news->mycomm);    //笛卡尔拓扑
    MPI_Cart_shift(news->mycomm, 0, 1, &news->left, &news->right);

    MPI_Type_size(type, &dsize);
    MPI_Type_extent(type, &dextent);
    if (2 * dsize <= int(dextent)) {
        fprintf(stderr,
            "Datatype needs to be (nearly) contiguous; size = %d and extent = %d\n",
            dsize, dextent);
        free(news);
        *pipe = 0;
        return;
    }
    news->typesize = dextent;
    MPI_Allreduce(&maxsize, &maxbuf, 1, MPI_INT, MPI_MAX, comm);
    news->maxlen = maxbuf;
    maxbuf *= dextent;
    news->buf = (Particle *)malloc(2 * maxbuf);
    news->buf1 = news->buf;
    news->buf2 = (Particle *)((char *)news->buf1 + maxbuf);
    news->type = type;
    news->stage = 0;
    news->last_stage = size - 1;
    *pipe = (void *)news;
}
```

此函数用来创建处理器，传入的参数为 MPI_Comm 进程管理器、MPI_Datatype 数据类型，buf 的最大容量，以及最终创建的处理器的存储变量 pipe。

在此函数中，先为指针变量*news 申请了 MPI_Pipe 大小的内存空间，用来初始化处理器，最后将初始化之后的处理器赋值给 pipe，至此一个处理器 pipe 初始化成功。

使用 MPI_Comm_size(comm, &size)得到所有参加运算的进程的个数，并存储在变量 size 中。

使用 MPI_Cart_create(comm, 1, &size, &trues, 1, &news -> mycomm) 创建笛卡尔拓扑，MPI 提供两种拓扑，即笛卡儿拓扑和图拓扑，分别用来表示简单规则的拓扑和更通用的拓扑。对于每一维，说明进程结构是否是周期性的。 MPI_CART_CREATE 返回一个指向新的通信域的句柄，这个句柄与笛卡尔拓扑信息相联系。如果 reorder =false，那么在新的进程组中每一进程的标识数就与在旧进程组中的标识数相一致，否则，该调用会重新对进程编号，该调用得到一个 ndims 维的处理器阵列，每一维分别包含 dims[0] dims[1] ... dims[ndims-1]个处理器。

MPI_Cart_shift(news->mycomm, 0, 1, &news->left, &news->right);

News->mycomm 是带有笛卡尔结构的通信域句柄，0 是 direction，即需要平移的坐标维，1 为 disp，偏移量，news->left 为源进程的卡氏坐标，news->right 为目标进程的卡氏坐标。

之后使用 MPI_Type_size(type,&dsize)来获取形参 type 数据类型的大小。

使用MPI_Type_extent(type, &dextent)来获取形参type的数据类型的范围，将值的大小存储在变量 dextent 中。

MPI_Pipe 变量 pipe 的属性 typesize 即 dextent 的值。

然后使用 MPI_Allreduce(&maxsize, &maxbuf, 1, MPI_INT, MPI_MAX, comm)函数，所有进程都执行归约操作 相当于每一个进程都执行了一次 MPI_Reduce。

下面的语句为 news 其他属性的初始化，其中 buf 的大小为 maxbuf 的两倍，buf1 的大小等于 buf 的大小，buf2 为 3 倍的 maxbuf。

最后将 news 的内容赋值给形参 pipe，至此 pipe 处理器初始化完毕。

```
void MPE_Pipe_start(void *pipe, void *mybuf, int len, int qcopy)
{
    MPE_Pipe *p = (MPE_Pipe *)pipe;
    if (p->stage != 0) {
        fprintf(stderr, "Can only start pipe when pipe is empty\n");
        return;
    }
    if (p->last_stage == 0) {
        return;
    }
    MPI_Irecv(p->buf1, p->maxlen, p->type, p->left, 0, p->mycomm,
        &p->requests[0]);
    if (qcopy) {
        memcpy(p->buf2, mybuf, len * p->typesize);
        mybuf = p->buf2;
    }
    MPI_Isend(mybuf, len, p->type, p->right, 0, p->mycomm, &p->requests[1]);
}
```

MPE_Pipe_start()函数用来向 pipe 中传递数据。传递进来的参数为 pipe，mybuf，len，qcopy。

如果 pipe 的 stage 不为 0，则表示处理器中为空，不再往下执行，函数返回。如果 last_stage 为 0 表示数据传递结束，函数返回。

MPI_Irecv(p->buf1, p->maxlen, p->type, p->left, 0, p->mycomm, &p->requests[0])函数用来接受数据。

MPI_Isend(mybuf, len, p->type, p->right, 0, p->mycomm,

&p->requests[1])函数用来发送数据。

```
void MPE_Pipe_push(void *pipe, Particle **recvbuf, int *recvlen)
{
    MPE_Pipe *p = (MPE_Pipe *)pipe;
    MPI_Status  statuses[2];
    Particle *tmp;

    if (p->last_stage == 0) return;

    MPI_Waitall(2, p->requests, statuses);
    MPI_Get_count(&statuses[0], p->type, recvlen);
    *recvbuf = p->buf1;

    if (++p->stage == p->last_stage) {
        p->stage = 0;
        return;
    }

    /* 开始下一个循环 */
    tmp = p->buf1;
    p->buf1 = p->buf2;
    p->buf2 = tmp;
    MPI_Irecv(p->buf1, p->maxlen, p->type, p->left, 0, p->mycomm,
        &p->requests[0]);
    MPI_Isend(p->buf2, *recvlen, p->type, p->right, 0, p->mycomm,
        &p->requests[1]);
}
```

MPE_Pipe_push()函数用来建树，传入的形参为 pipe 处理器，缓存 recvbuf，接受到的数据长度 recvlen。

MPI_Waitall(2, p->requests, statuses)函数必须等到非阻塞通信对象表中所有的非阻塞通信对象相应的非阻塞操作都完成后才返回，用来等待所有给定的通信结束，第一个参数为非阻塞通信对象的个数， 第二个参数 array_of_requests 非阻塞通信完成对象数组(句柄数组)，第三个参数为 array_of_statuses 状态数组(状态数组类型)。

MPI_GET_COUNT( )函数返回的是以指定的数据类型为单位，接收操作接收到的数据的个数，第一个参数 status 接收操作返回的状态(状态类型) ，第二个参数 datatype 接收操作使用的数据类型(句柄)，第三个参数 count 接收到的以指定的数据类型为单位的数据个数(整型)。

下面开始下一个循环，知道建立四叉树完毕。

```
void MPE_Pipe_free(void **pipe)
{
    MPE_Pipe *p = (MPE_Pipe *)*pipe;

    if (p->stage != 0) {
        fprintf(stderr, "无法清除处理器，需要%d个阶段都执行完毕\n",
            p->last_stage - p->stage);
        return;
    }
    MPI_Comm_free(&p->mycomm);
    free(p->buf);
    *pipe = 0;
}
```

MPE_Pipe_free()函数来清除处理器。实现较为简单。

下面的 ComputeForces( )函数用来计算星体受力和加速度。

```
//计算每个点的受力与加速度
void ComputeForces(Particle  *particles, ParticleV *pv, int npart, Particle *recvbuf, int rlen, double    *max_f)
{
    int i, j;
    double xi, yi, mi, rx, ry, mj, r, fx, fy;
    double xnew, ynew, rmin;

    /* 计算受力*/
    for (i = 0; i < npart; i++) {
        rmin = 100.0;
        xi = particles[i].x;
        yi = particles[i].y;
        fx = 0.0;
        fy = 0.0;
        for (j = 0; j < rlen; j++) {
            rx = xi - recvbuf[j].x;
            ry = yi - recvbuf[j].y;
            mj = recvbuf[j].mass;
            r = rx * rx + ry * ry;
            if (r == 0.0) continue;
            if (r < rmin) rmin = r;
            /* 计算受力*/
            r = r * sqrt(r);
            fx += mj * rx / r;
            fy += mj * ry / r;
        }

        pv[i].fx -= fx;
        pv[i].fy -= fy;
        fx = sqrt(fx*fx + fy * fy) / rmin;
        if (fx > *max_f) *max_f = fx;
    }
}
```

因为此实验中我只展现了 2d 的变化效果，因此只需要计算 x 轴和 y 轴方向的受力和加速度。

计算受力时对每个星体进行计算。计算时使用了向量的计算方式。

计算结束后更新星体的受力大小。

```
//界面
void PrintParticles(Particle *particles, Particle *p2, int npart, double t)
{
    int i;
    for (i = 0; i < npart; i++) {
        double ball_x = 400, ball_y = 100;
        setcolor(BLACK);
        setfillcolor(BLACK);
        fillcircle(ball_x + p2[i].x * 10, ball_y + p2[i].y * 10, 2);
        setcolor(BLACK);
        setfillcolor(YELLOW);
        fillcircle(ball_x + particles[i].x * 15, ball_y + particles[i].y * 10, 2);
    }
}
```

此函数是界面展示，其中展示了每个星体的位置，并将其在画布上展示出来。

图形库使用的是 easyX，黄色表示星体，黑色是背景色。

下方皆为 main 函数中内容

```
int main(int  argc, char **argv)
{
    srand((unsigned)time(NULL));
    Particle  particles[MAX_PARTICLES], temParticles[MAX_PARTICLES];
    ParticleV pv[MAX_PARTICLES];
    Particle  *recvbuf;
    int       rank, size, npart, i, j;
    int       step, rlen;
    int       totpart, cnt;
    MPI_Datatype particletype;
    double    time;
    double    dt, dt_old;
    double    t;
    double    a0, a1, a2;
    void      *pipe;
    int       debug_flag = 1;

    initgraph(1500, 600);
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Particle particles[ ]，temParticles[ ]为每个节点中的粒子。

ParticleV pv[ ]为每个粒子的速度结构体。

time 为计算时间。

dt 为计算公式中的 dt，用来计算力和加速度。

a0，a1，a2 为星体的加速度，用来计算总的加速度。

initgraph(1500, 600)为初始化画布，大小为长 1500，高 600 像素。

MPI_Init(&argc, &argv)初始化进程。

MPI_Comm_rank(MPI_COMM_WORLD, &rank)获取进程编号。

MPI_Comm_size(MPI_COMM_WORLD, &size)获取进程数量。

```
npart = 1000 * size;
printf("%d",size);
if (argc > 1)
    npart = atoi(argv[1]) / size;
if (npart > MAX_PARTICLES)
    MPI_Abort(MPI_COMM_WORLD, 1);

MPI_Allreduce(&npart, &totpart, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

cnt = 100000;

MPI_Type_contiguous(4, MPI_DOUBLE, &particletype); //创建一个连续的数据类型
MPI_Type_commit(&particletype);                    //提交一个类型
/* Generate the initial values */
```

npart 为总的星体数，npart=1000*size，因为是事先约定总数最大为4000，意味着进程不能超过 4 个。

使用 MPI_Allreduce(&npart, &totpart, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD)函数将所有进程的划分相加。

MPI_Type_contiguous(4, MPI_DOUBLE, &particletype)用来创建一个连续的数据类型，它得到的新类型是将一个已有的数据类型按顺序依次连续进行复制后的结果。第一个参数为 count，表示复制个数(非负整数) ，第二个参数为 oldtype，是旧数据类型(句柄) ，第三个参数为 newtype 新数据类型(句柄)。

MPI_Type_commit(&particletype)用来提交这个新产生的类型。

```
for (i = 0; i < npart; i++) {
    particles[i].x = rand() % 48;
    particles[i].y = rand() % 48;
    particles[i].z = rand() % 48;
    /* 星体的质量*/
    particles[i].mass = 1 * 2.582e-8;
    /* 初始化星体的受力*/
    pv[i].xold = particles[i].x;
    pv[i].yold = particles[i].y;
    pv[i].zold = particles[i].z;
    pv[i].fx = 0;
    pv[i].fy = 0;
    pv[i].fz = 0;
}

dt = 0.001;
dt_old = 0.001;
```

这一部分用来初始化星体的位置、质量和受力，位置为随机数来生成，范围为 0~48，即在这个三维空间中，最多可以容纳 48*48*48=9216 个点。

星体的质量皆初始化为 2.582e-8，pv 中的位置初始化为 particle 的位置，并且受力初始化为 0。

dt、dt_old 为 0.001。

```
/*创建处理器 */
MPE_Pipe_create(MPI_COMM_WORLD, particletype, npart, &pipe);
time = MPI_Wtime();
t = 0.0;
```

紧接着创建处理器，并记录当前时间。

```
while (cnt--) {
    double max_f, dt_est, new_dt, dt_new;
    /* 加载初始化发送的缓冲区*/
    MPE_Pipe_start(pipe, particles, npart, 1);

    /* 计算加速度 */
    a0 = 2.0 / (dt * (dt + dt_old));
    a2 = 2.0 / (dt_old * (dt + dt_old));
    a1 = -(a0 + a2);

    /* 计算受力*/
    max_f = 0;
    ComputeForces(particles, pv, npart, particles, npart, &max_f);

    /* 计算相互作用力*/
    for (step = 1; step < size; step++) {
        /*创建树 */
        MPE_Pipe_push(pipe, &recvbuf, &rlen);
        /* 计算受力 */
        ComputeForces(particles, pv, npart, recvbuf, rlen, &max_f);
    }

    for (i = 0; i < npart; i++) {
        temParticles[i].x = particles[i].x;
        temParticles[i].y = particles[i].y;
    }
```

循环开启处理器，并计算每个星体的加速度和受力。

如果 size>1 则创建树，并且计算受力。

```
Sleep(1000);
for (i = 0; i < npart; i++) {
    double xi, yi;
    xi = particles[i].x;
    yi = particles[i].y;
    particles[i].x = (pv[i].fx - a1 * xi - a2 * pv[i].xold) / a0;
    particles[i].y = (pv[i].fy - a1 * yi - a2 * pv[i].yold) / a0;
    pv[i].xold = xi;
    pv[i].yold = yi;
    pv[i].fx = 0;
    pv[i].fy = 0;
}

t += dt;
```

Sleep(DWORD dwMilliseconds) 单位为毫秒，1s 更新一次星体的位置。

```
if (debug_flag && rank == 0)
    PrintParticles(particles, temParticles, npart, t);
```

将星体显示在画布上。

```
    dt_est = 1.0 / sqrt(max_f);

    if (dt_est < 1.0e-6) dt_est = 1.0e-6;
    MPI_Allreduce(&dt_est, &dt_new, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);

    if (dt_new < dt) {
        dt_old = dt;
        dt = dt_new;
        if (debug_flag && rank == 0)
            printf("#New time step is %f\n", dt);
    }
    else if (dt_new > 4.0 * dt) {
        dt_old = dt;
        dt *= 2.0;
        if (debug_flag && rank == 0)
            printf("#New time step is %f\n", dt);
    }
}
```

重新计算时间评估。

使用 MPI_Allreduce( )函数计算 dt_new》

```
    time = MPI_Wtime() - time;
    if (rank == 0) {
        printf("#Computed %d particles in %f seconds\n", totpart, time);
    }

    MPE_Pipe_free(&pipe);
    MPI_Type_free(&particletype);

    MPI_Finalize();
    system("pause");
}
```

计算运行时间 time。

最后清楚 pipe 处理器和粒子的类型，最终结束进程。

六、 调试心得

通过本次实验，我对并行计算的基本思想有了更深入的了解，并且学会了如何使用分治的思想模拟天体运行。通过计算星体的受力、速度等影响因素，将星体之间的运动过程使用 easyX 图形库展现了出来，并且最终将程序运行成功，并对实验结果进行了一系列的分析。