中山大学计算机学院本科生实验报告

课程名称：超级计算机原理与操作

任课教师：中山大学吴迪、黄聃老师

年级	19	专业（方向）	计算机科学与技术
学号	18302007	姓名	陈浩深
开始日期	2021/5/5	完成日期	2021/5/8

一、实验题目

编程实现nbody问题，要求实现一个串行版本和MPI，OpenMP，pthread中的任意两种版本。

本次实验中，我实现的并行版本为**OpenMP**和**pthread**

二、实验内容

输入文件为nbody_init.txt，输入每一行为一个body，每列分别是质量，x轴位置，y轴位置，z轴位置，x轴速度，y轴速度，z轴速度

补充说明：参考输出为nbody_last.txt，文件格式与输入文件相同
参考输出的计算方式为先更新速度再更新位置，与先更新位置再更新速度相比差距不大，两种方法都可以

输出文件格式应与输入文件一致，为20轮迭代后的结果

相关参数设置为dT=0.005，G=1，迭代次数为20

三、实验结果

1. 串行版本

main:

main函数基本框架是在在每次迭代中计算出粒子的受力情况，并通过受力算出粒子在dT内的速度变化和位移变化：

```
/* initial data */
for(int i = 0; i < NUM_ITERATION; ++i) {
	ComputeForce();
	UpdateVelocityAndPosition();
}
/* ouput data */
```

ComputeForce:

参考课本提出的reduced方法，首先计算每个粒子的受力情况，对于每个粒子q，只计算粒子k(k > q)对q的作用力f_qk，然后通过f_qk = -f_kq得出粒子q对粒子k的作用力

```
//compute all force using reduced method
void ComputeForce() {
	//set 0 each time
	memset(force, 0, sizeof(force));

	for(int q = 0; q < NUM_BODY; ++q) {
		//k > q
		for(int k = q + 1; k < NUM_BODY; ++k) {
			Body * bq = &nbody[q], * bk = &nbody[k];

			COMPUTE_TYPE x_diff = bq->xpos - bk->xpos,
					y_diff = bq->ypos - bk->ypos,
					z_diff = bq->zpos - bk->zpos;

			COMPUTE_TYPE dist = sqrt(x_diff*x_diff + y_diff*y_diff + z_diff*z_diff);
			COMPUTE_TYPE dist_cubed = dist * dist * dist;
			COMPUTE_TYPE gmm_dist3 = -G*(bq->m)*(bk->m) / dist_cubed;

			COMPUTE_TYPE force_qk_X = gmm_dist3 * x_diff,
					force_qk_Y = gmm_dist3 * y_diff,
					force_qk_Z = gmm_dist3 * z_diff;

			force[q][X] += force_qk_X;
			force[q][Y] += force_qk_Y;
			force[q][Z] += force_qk_Z;
			//symmetric
			force[k][X] -= force_qk_X;
			force[k][Y] -= force_qk_Y;
			force[k][Z] -= force_qk_Z;
		}
```

```
    }
}
```

UpdateVelocityAndPosition:

循环所有粒子根据受力情况算出粒子的速度，按照作业要求有需先算出粒子变化后的速度，再利用该速度算出粒子的位移

```
//update all velocity and position in nbody[]
void UpdateVelocityAndPosition() {
      for(int i = 0; i < NUM_BODY; ++i) {
            Body * bi = &nbody[i];
            //update velocity
            bi->vx += force[i][X] / bi->m * dT;
            bi->vy += force[i][Y] / bi->m * dT;
            bi->vz += force[i][Z] / bi->m * dT;
            //update position
            bi->xpos += bi->vx * dT;
            bi->ypos += bi->vy * dT;
            bi->zpos += bi->vz * dT;
      }
}
```

Check:

```
#ifdef DEBUG
      Check();
#endif
```

编写Check函数，将所得结果与nbody_last.txt中数据对比，这里的是相对误差的最大值来作为判断的依据

- relative_error_i = / (my_answer_i - standard_answer_i) / standard_answer_i
- return max_off_all(relative_error_i...)

```
//compare with nbody_last.txt to check correctness
void Check() {
      const char * filename = "nbody_last.txt";
      FILE *fp = fopen(filename, "r");
      if(fp == NULL) {
            fprintf(stderr, "can not open file: %s\n", filename);
            exit(1);
      }

      COMPUTE_TYPE relative_max = 0;

      for(int i = 0; i < NUM_BODY; ++i) {
            Body refdata;
```

```
            ReadNBody(&fp, &refdata);

            MaxOfAll(&relative_max, 6,
                  GetRelativeError(nbody[i].xpos, refdata.xpos),
                  GetRelativeError(nbody[i].ypos, refdata.ypos),
                  GetRelativeError(nbody[i].zpos, refdata.zpos),
                  GetRelativeError(nbody[i].vx, refdata.vx),
                  GetRelativeError(nbody[i].vy, refdata.vy),
                  GetRelativeError(nbody[i].vz, refdata.vz)
            );
      }

       printf("max relative error = %.15Lf\n", relative_max);
       fclose(fp);
}
```

MaxOfALL:

使用可变参数函数来对多个数据取max值

```
//using max difference to judge
void MaxOfAll(COMPUTE_TYPE * max, int n, ...) {
       va_list argptr;
       va_start(argptr, n);
       for (int i = 0; i < n; ++i) {
             COMPUTE_TYPE tmp = va_arg(argptr, COMPUTE_TYPE);
             *max = *max > tmp ? *max : tmp;
      }
       va_end(argptr);
}
```

运行过程与结果：

编译: (-D DEBUG 为可选参数)

```
gcc nbody_serial.c -o serial_run -D DEBUG
```

运行:

```
.\serial_run
```

运行结果：

```
\proj> gcc nbody_serial.c -o serial_run -D DEBUG
\proj> .\serial_run.exe
run time: 3.002291e-001
max relative error = 0.000000058855085
```

可以看到串行程序的运行时间约为0.3s，且计算结果与nbody_last.txt中的参考数据相近

2. OpenMp版本

分析：

1. openmp的omp代码块中有隐式路障，因此不需要自己实现同步的机制。课堂上学习的时候，我们利用openmp处理的都是对单个变量的并行操作(reduce)，在这次的实验中需要处理多个变量，因此这里并不太适合使用openmp提供的对单个变量的操作。

2. openmp版本的nbody问题主要利用了openmp提供的对for循环的并行机制，用类似pthread的并行编程方法，即将不同的变量分配给不同的线程并计算当前线程所负责的区域的数据，最后进行汇总。即创建一个loc_force[NUM_THREAD*NUM_BODY][NUM_DIMENTION]数组，每个线程负责loc_force[my_rank *n: my_rank* (n+1)]的范围的计算，最后将结果汇总到总的force数组中，这样每个线程负责的范围不会有交集，无需考虑竞争的问题，提高并行的效率。

3. 程序使用了静态的线程分配，通过修改nbody_omp.c中的NUM_THREAD的值声明线程数

```
#define NUM_THREAD 8
```

main:

首先定义openmp作用的块，对每个线程先获取my_rank，然后循环更新粒子所受的力以及速度和位移的改变

```
# pragma omp parallel num_threads(NUM_THREAD)
{
      int my_rank = omp_get_thread_num();
      for(int i = 0; i < NUM_ITERATION; ++i) {
            /*
                  update pariticles
            */
     }
}
```

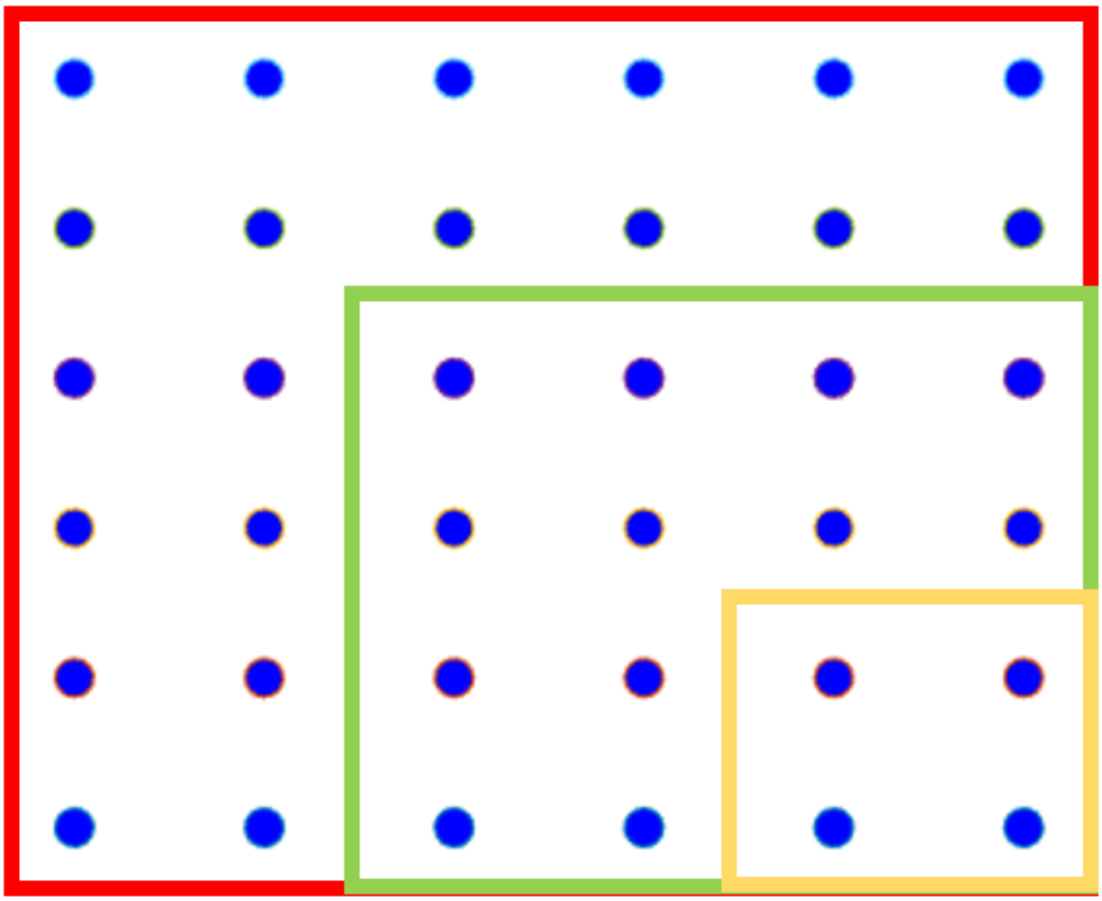
每次计算时，需要先将loc_force的值置零：

```
for (int idx = 0; idx < NUM_THREAD*NUM_BODY; ++idx)
                  loc_force[idx][X] = loc_force[idx][Y] = loc_force[idx][Z] = 0.0;
```

在隐式路障作用下，所用线程同步到下一个omp for语句，计算粒子q所受的作用力，并将中间结果储存在loc_force + my_rank*NUM_BODY的位置中。这里**ComputeForce**函数与串行版本中基本相似，唯一的不同点是串行版本的**ComputeForce**计算的是所有粒子的所受力，串行版本中传入参数q计算粒子q所受力的作用并对所有粒子调用该函数并行计算

```
# pragma omp for schedule(dynamic)
            for (int q = 0; q < NUM_BODY-1; q++)
                  ComputeForce(loc_force + my_rank*NUM_BODY, q);
```

默认情况下，omp的schedule策略为static，即将任务迭代逐个地分配到各个线程中，假设任务中共有6个粒子，3个线程，则static方法对于reduced method任务分配结果如图：



可以看到对于迭代起始的线程，分配到的计算量远高于迭代末尾的线程，而计算的时间取决于最慢的线程，因此openmp默认的static调度用在这里是不合适的，所以改为schedule(dynamic)，即逐个将任务分配到每一个线程，若某个线程执行完了就继续分配新的任务

接下来需要将每个线程loc_force的值汇总到force中，这里的omp for放在外层循环中，即对变量q所在循环进行并行处理，而不是对变量thread所在循环处理，这样避免了竞争的情况出现，这里因为不是像reduced method一样按对角线分配任务量，因此直接使用默认的static调度，可以较为平均地分配计算任务

```
//reduce all loc_force to force
# pragma omp for
for(int q = 0; q < NUM_BODY; ++q) {
      force[q][X] = force[q][Y] = force[q][Z] = 0.0;
      for(int thread = 0; thread < NUM_THREAD; ++thread) {
            force[q][X] += loc_force[thread*NUM_BODY + q][X];
            force[q][Y] += loc_force[thread*NUM_BODY + q][Y];
            force[q][Z] += loc_force[thread*NUM_BODY + q][Z];
     }
}
```

计算出作用力后，更新每个粒子的速度和位移变化，同样地，**UpdateVelocityAndPosition**函数与串行方法中基本相似，只是传入了参数q，这样便于在main函数中使用并行

```
 //update state in parallel
# pragma omp for
for(int q = 0; q < NUM_BODY; ++q)
      UpdateVelocityAndPosition(q);
```

运行过程与结果：

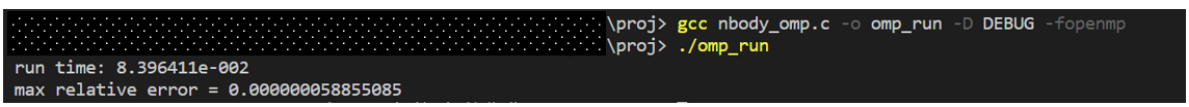
编译: (-D DEBUG 为可选参数)

```
gcc nbody_omp.c -o omp_run -D DEBUG -fopenmp
```

运行:

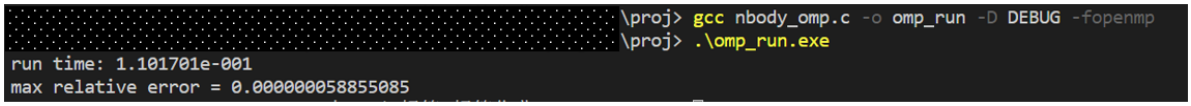
```
.\omp_run
```

运行结果：



可以看到openmp版本的并行程序的运行时间约为0.08s，加速比约为**3.58**，且计算结果与nbody_last.txt中的参考数据相近 (注：原本以为两个版本的计算所得的相对误差会有区别，但多次实验并检查是否有编程错误后发现，两程序所得的相对误差的确相等)

将openmp的schedule策略从dynamic改为(static,1)后测试：



可以看到并行的效率略低于dynamic版本(注：因为两版本运行时间其实差别不大，也会出现dynamic效率低于(static,1)的情况，但在多次实验情况下，绝大部分时候dynamic的调度方法会获得更高的效率)

3. pthread版本

分析：

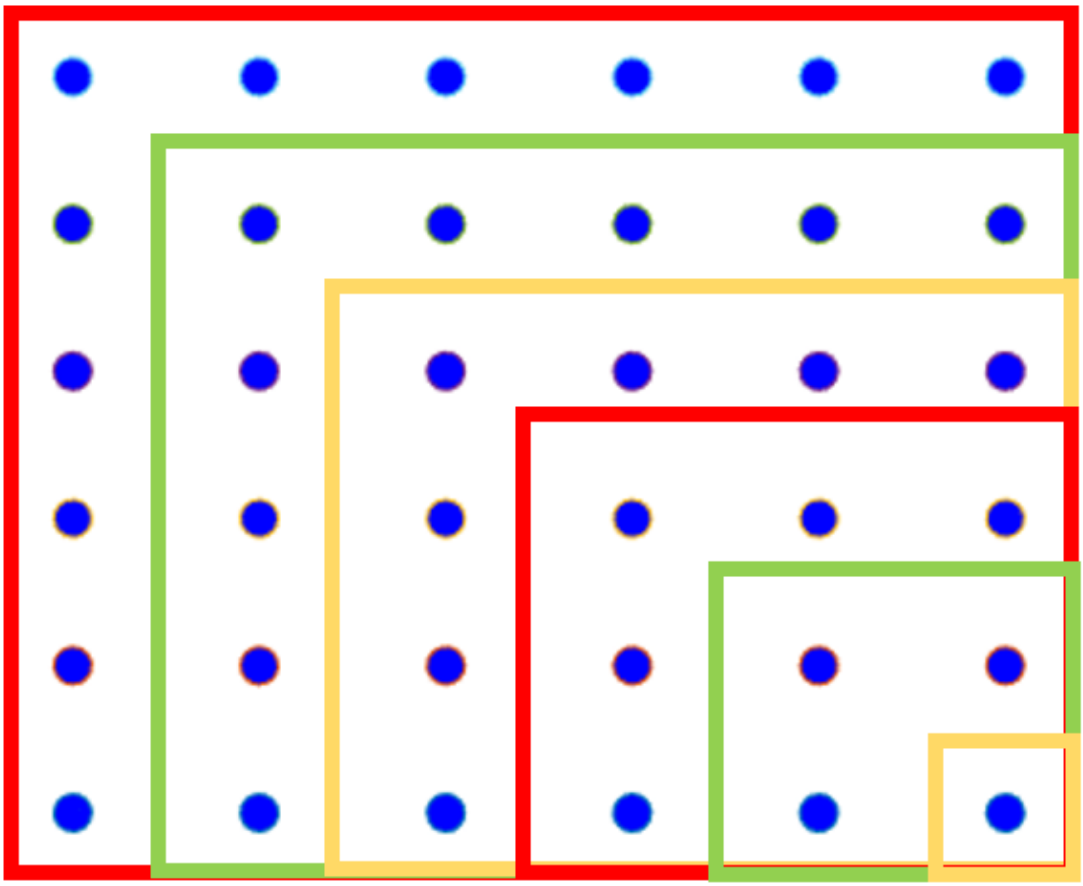
1. pthread并行编程时，需要分配好足够的线程以执行任务，因此首先定义了threadHandle数组静态分配好线程

```
pthread_t threadHandle[NUM_THREAD];//thread pool
```

2. 参考课本有，pthread并行编程的基本思路是将任务自定义分配好给不同的线程，但是pthread并没有提供如# pragma omp for schedule(dynamic) 等语句，因此需要自己编程实现：

```
//cyclic schedule
void Schedule(int my_rank, int * my_first, int * my_last, int * my_incr) {
      *my_first = my_rank;
      *my_last = NUM_BODY;
      *my_incr = NUM_THREAD;//i += thread_num
}
```

这里实现的是循环调度，即实现了# pragma omp for schedule(static,1) 的功能，每次分配给线程一个任务直至所有线程分配完，此时开始再一次循环分配：如图：



这种调度不如dynamic分配的效果平均，但比起block的调度有了较好的优化。

3. pthread并行编程并没有提供隐式路障，因此需要编程实现线程同步的方法，这里编写了Barrier函数：

```
//synchronize al; threads
void Barrier() {
      pthread_mutex_lock(&barrier_cnt_mut);
      //begin critical area
      ++barrier_thread_cnt;
      if(barrier_thread_cnt == NUM_THREAD) {//last thread
            barrier_thread_cnt = 0;
            pthread_cond_broadcast(&barrier_cond_var);//release all
     }
      else {
            while(pthread_cond_wait(&barrier_cond_var, &barrier_cnt_mut) != 0)
                  /* do nothing */;
     }
      //end critical area
      pthread_mutex_unlock(&barrier_cnt_mut);
}
```

Barrier的实现思想是，设置计数器barrier_thread_cnt并初始化为0，每有一个线程进入Barrier就自增barrier_thread_cnt，然后调用pthread_cond_wait等待条件变量barrier_cond_var。当所有线程都进入Barrier后，将barrier_thread_cnt重新置为0，并调用pthread_cond_broadcast修改barrier_cond_var变量，结束所有进程的阻塞。

在调用Barrier的过程中会有多个线程修改barrier_thread_cnt的值，因此还加入了互斥锁barrier_cnt_mut变量，避免竞争引发的错误。

4. 程序使用了静态的线程分配，通过修改nbody_pth.c中的NUM_THREAD的值声明线程数

```
#define NUM_THREAD 8
```

main:

main函数的核心部分如下，开始时需要初始化Barrier函数中所用到的计数器，互斥锁以及条件变量，然后通过调用**pthread_create**为所有线程分配任务(**UpdateState**)，计算结束后通过调用**pthread_join**回收所有的线程。

```
BarrierInit();
//create all threads
for(int thread = 0; thread < NUM_THREAD; ++thread)
	pthread_create(&threadHandle[thread], NULL, UpdateState, (void*)thread);

for(int thread = 0; thread < NUM_THREAD; ++thread)
	pthread_join(threadHandle[thread], NULL);

BarrierDestroy();
```

UpdateState:

UpdateState函数比较长，因此解析直接写在函数的注释中

```
//update velocity and position in current thread
void * UpdateState(void * rank) {
	/* 通过获取rank变量的值得出线程id，方便进行Schedule */
	long my_rank = (long) rank;
	int my_first, my_last, my_incr;

	/* 通过调用Schedule分配好线程的任务 */
	//schedule current thread's work
	Schedule(my_rank, &my_first, &my_last, &my_incr);

	/* 与openmp不同，在调用UpdateState时就已经为所有线程分配好NUM_ITERATION次迭代的任务，
	   但是根据题目给出的信息可以知道，所有的线程必须同步计算好所有粒子的状态变化，因此需要引入同步，等待所有线程完成同一轮迭代
	 */
	for(int i = 0; i < NUM_ITERATION; ++i) {
		/* 首先为每个线程负责的loc_force区域置零 */
		//initial loc_force with 0
		memset(loc_force + my_rank*NUM_BODY, 0, NUM_BODY * sizeof(COMPUTE_TYPE[NUM_DIMENTION]));

		/* 置零后，当前线程可直接开始ComputeForce的计算，这里ComputeForce函数的定义与openmp版本相同 */
```

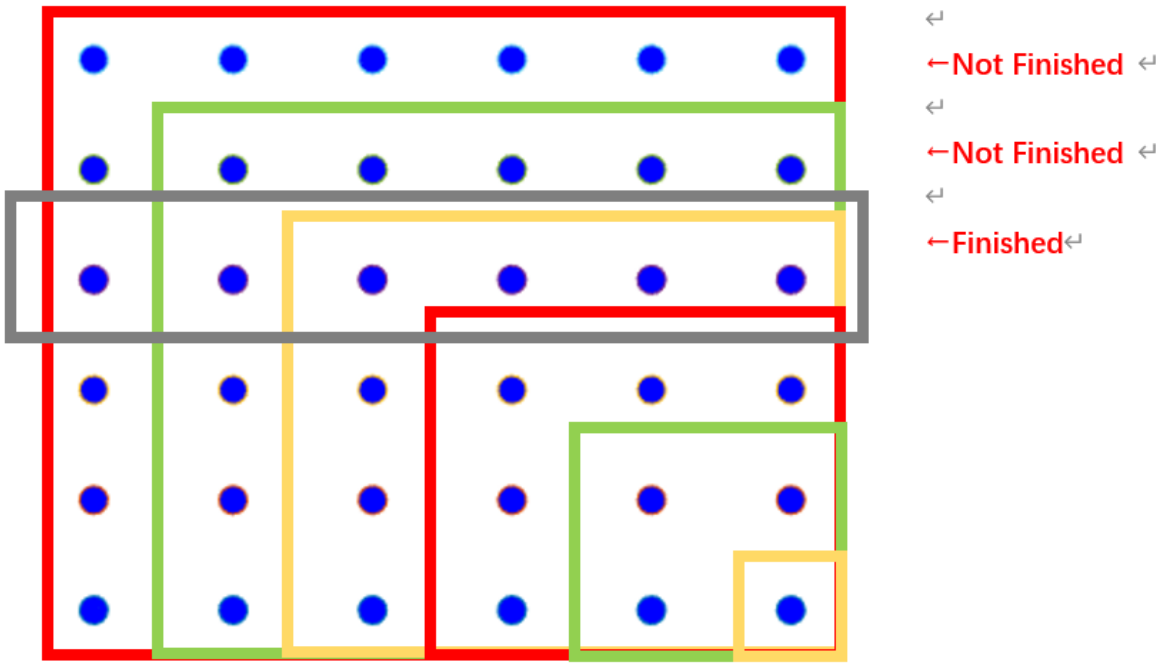
```
            for(int q = my_first; q < my_last; q += my_incr) {
                  ComputeForce(loc_force + my_rank*NUM_BODY, q);
            }

            /* 这里需要同步等待所有线程完成计算loc_force，否则将会出现下图中的错误：线程1, 2还没有完成计算，但是线程3已经完成计算并汇总所有的loc_force这时，部分loc_force还没有完成计算，将产生错误的结果
            */
            Barrier();//necessary

            /* 和openmp中的reduce方法相同，已经使用Schedule提前分配好每个线程负责的粒子q，因此不会出现竞争的情况 */
            //reduce loc_force to force
            for(int q = my_first; q < my_last; q += my_incr) {
                  force[q][X] = force[q][Y] = force[q][Z] = 0.0;
                  for(int thread = 0; thread < NUM_THREAD; ++thread) {
                        force[q][X] += loc_force[thread*NUM_BODY + q][X];
                        force[q][Y] += loc_force[thread*NUM_BODY + q][Y];
                        force[q][Z] += loc_force[thread*NUM_BODY + q][Z];
                  }
            }

            /* 由于粒子的更新需要知道所有其他粒子产生的作用力，因此线程需要同步等待force的计算完成，之后开始更新粒子的速度与位移，UpdateVelocityAndPosition与openmp中实现方法相同 */
            Barrier();//finish computing all force
            //update velocity and position in current thread
            for(int q = my_first; q < my_last; q += my_incr) {
                  UpdateVelocityAndPosition(q);
            }

            /* 粒子的位移变化会影响相互间作用力的大小，因此需要等待所有粒子更新位移后，才能够开始新一轮迭代 */
            Barrier();//update all, start next iteration
      }
}
```



运行过程与结果：

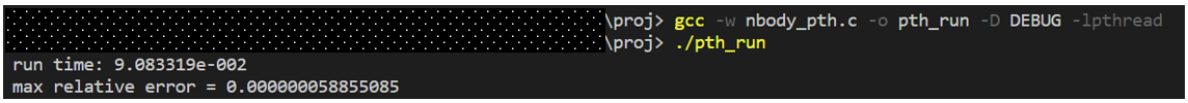
编译: (-D DEBUG 为可选参数)

```
gcc -w nbody_pth.c -o pth_run -D DEBUG -lpthread
```

运行:

```
./pth_run
```

运行结果：



可以看到openmp版本的并行程序的运行时间约为0.09s，加速比约为**3.31**，且计算结果与nbody_last.txt中的参考数据相近