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：吴迪

年级	19	专业（方向）	计算机科学与技术
学号	**19335174**	姓名	**施天予**
开始日期	**2021-5-2**	完成日期	**2021-5-3**

一、 实验题目

根据 7-Development.pdf 课件在nbody问题或者tsp问题中**二选一**进行实现，要求实现一个串行版本和MPI，OpenMP，Pthreads中的任意**两种**版本。

本次实验我选择**nbody**问题，并实现了**串行版本**和**OpenMP**，**Pthreads**版本。

二、 实验内容

串行版本

n体问题的串行版本比较简单，我按照书上二维n体问题的基本方法做了改进就完成了。因为我发现给定的标准输出文件都是15位有效数字，所以我写了一个digit函数用于计算一个浮点数整数部分的位数，再计算得到小数控制位数输出。

源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>

#define  DIM      3         // 维数
#define  X        0
#define  Y        1
#define  Z        2
#define  N        1024      // 粒子数
#define  G        1         // 引力常量
#define  dT       0.005     // 时间差
#define  n_steps  20        // 迭代数

#define  GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
}

double start,stop;
double masses[N];           // 质量
double pos[N][DIM];         // 位置
double vel[N][DIM];         // 速度
double forces[N][DIM];      // 作用力
```

```c
void init() {   // 从文本文件中输入
    FILE *fp = fopen("nbody_init.txt", "r");
    if (fp == NULL) {
        printf("File cannot open! ");
        exit(0);
    }
    for (int i = 0; i < N; i++) {
        fscanf(fp, "%lf", &masses[i]);
        fscanf(fp, "%lf", &pos[i][X]);
        fscanf(fp, "%lf", &pos[i][Y]);
        fscanf(fp, "%lf", &pos[i][Z]);
        fscanf(fp, "%lf", &vel[i][X]);
        fscanf(fp, "%lf", &vel[i][Y]);
        fscanf(fp, "%lf", &vel[i][Z]);
    }
    fclose(fp);
}

int digit(double x) {   // 用于判断一个数整数部分的位数
    int k = 0;
    int y = abs((int)x);
    while (y > 0) {
        y /= 10;
        ++k;
    }
    return k;
}

void print() {  // 输出到文本文件
    FILE *fp = fopen("nbody_serial_out.txt", "w");
    if (fp == NULL) {
        printf("File cannot open! ");
        exit(0);
    }
    for (int i = 0; i < N; i++) {
        fprintf(fp, "%.*lf ", 15-digit(masses[i]), masses[i]);      // 控制输出为
15位有效数字
        fprintf(fp, "%.*lf ", 15-digit(pos[i][X]), pos[i][X]);
        fprintf(fp, "%.*lf ", 15-digit(pos[i][Y]), pos[i][Y]);
        fprintf(fp, "%.*lf ", 15-digit(pos[i][Z]), pos[i][Z]);
        fprintf(fp, "%.*lf ", 15-digit(vel[i][X]), vel[i][X]);
        fprintf(fp, "%.*lf ", 15-digit(vel[i][Y]), vel[i][Y]);
        fprintf(fp, "%.*lf ", 15-digit(vel[i][Z]), vel[i][Z]);
        fprintf(fp, "\n");
    }
    fclose(fp);
}

void Compute_force(int q) {     // 计算作用力
    double x_diff, y_diff, z_diff, dist, dist_cubed;
    for (int k = 0; k < N; k++) {
        if (k == q) continue;
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        z_diff = pos[q][Z] - pos[k][Z];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff + z_diff*z_diff);
        dist_cubed = dist*dist*dist;
```

```
            forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
            forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
            forces[q][Z] -= G*masses[q]*masses[k]/dist_cubed * z_diff;
        }
}

void Compute_POSandVEL(int q) {     // 计算速度和位置
    vel[q][X] += dT/masses[q]*forces[q][X];
    vel[q][Y] += dT/masses[q]*forces[q][Y];
    vel[q][Z] += dT/masses[q]*forces[q][Z];
    pos[q][X] += dT*vel[q][X];
    pos[q][Y] += dT*vel[q][Y];
    pos[q][Z] += dT*vel[q][Z];
}

int main(int argc, char* argv[]) {
    GET_TIME(start);
    init();
    for (int step = 1; step <= n_steps; step++) {   // 迭代20次
        memset(forces, 0, sizeof(forces));
        for (int part = 0; part < N; part++)
            Compute_force(part);
        for (int part = 0; part < N; part++)
            Compute_POSandVEL(part);
    }
    print();
    GET_TIME(stop);
    printf("Run time: %e\n", stop-start);
}
```

OpenMP

OpenMP的实现也比较简单。因为当多个线程或进程试图访问I/O缓冲区时，输出顺序不可预测，所以只用一个线程完成所有的I/O操作。

```
#   pragma omp parallel num_threads(thread_count) \
        default(none) private(step, part) \
        shared(forces, masses, pos, vel, thread_count)
    for (int step = 1; step <= n_steps; step++) {   // 迭代20次
        //memset(forces, 0, sizeof(forces));
#       pragma omp for schedule(static, N / thread_count)   // 并行for循环（块划分）
        for (part = 0; part < N; part++) {
            forces[part][X] = 0;
            forces[part][Y] = 0;
            forces[part][Z] = 0;
        }
#       pragma omp for schedule(static, N / thread_count)   // 并行for循环（块划分）
        for (part = 0; part < N; part++)
            Compute_force(part);
#       pragma omp for schedule(static, N / thread_count)   // 并行for循环（块划分）
        for (part = 0; part < N; part++)
            Compute_POSandVEL(part);
    }
```

OpenMP的实现基本只有上面这段做了改变。首先用# pragma omp parallel创建线程，再用# pragma omp for进行并行循环。不同于直接使用# pragma omp parallel for，这样写只创建一次线程，并在每次内部循环的执行中重用它们，减少运行时间。step和part是循环变量，所以是私有的；forces，masses，pos，vel，thread_count都是全局共享变量。因为n体问题的基本算法采用块划分法会有更好的性能，所有要在每个for循环前加上# pragma omp for schedule(static, N / thread_count)的调度子句。值得注意的是，在外部循环的每一次迭代中，不能使用memset将forces清零，因为不同线程运行速度不同，这样做会导致得到的结果与正确结果十分接近，但每次都不一样。# pragma omp for在循环末尾有一个隐式的路障，用它将forces清零，既正确又高效。

源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <omp.h>
#include <sys/time.h>

#define  DIM      3         // 维数
#define  X        0
#define  Y        1
#define  Z        2
#define  N        1024      // 粒子数
#define  G        1         // 引力常量
#define  dT       0.005     // 时间差
#define  n_steps  20        // 迭代数

#define  GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
}

double  masses[N];          // 质量
double  pos[N][DIM];        // 位置
double  vel[N][DIM];        // 速度
double  forces[N][DIM];     // 作用力
double  x_diff, y_diff, z_diff, dist, dist_cubed, start, stop;
int     step, part, thread_count = 4;

void init() {   // 从文本文件中输入
    FILE *fp = fopen("nbody_init.txt", "r");
    if (fp == NULL) {
        printf("File cannot open! ");
        exit(0);
    }
    for (int i = 0; i < N; i++) {
        fscanf(fp, "%lf", &masses[i]);
        fscanf(fp, "%lf", &pos[i][X]);
        fscanf(fp, "%lf", &pos[i][Y]);
        fscanf(fp, "%lf", &pos[i][Z]);
        fscanf(fp, "%lf", &vel[i][X]);
        fscanf(fp, "%lf", &vel[i][Y]);
        fscanf(fp, "%lf", &vel[i][Z]);
    }
    fclose(fp);
}
```

```c
int digit(double x) {    // 用于判断一个数整数部分的位数
    int k = 0;
    int y = abs((int)x);
    while (y > 0) {
        y /= 10;
        ++k;
    }
    return k;
}

void print() {  // 输出到文本文件
    FILE *fp = fopen("nbody_OpenMP_out.txt", "w");
    if (fp == NULL) {
        printf("File cannot open! ");
        exit(0);
    }
    for (int i = 0; i < N; i++) {
        fprintf(fp, "%.*lf ", 15-digit(masses[i]), masses[i]);      // 控制输出为
15位有效数字
        fprintf(fp, "%.*lf ", 15-digit(pos[i][X]), pos[i][X]);
        fprintf(fp, "%.*lf ", 15-digit(pos[i][Y]), pos[i][Y]);
        fprintf(fp, "%.*lf ", 15-digit(pos[i][Z]), pos[i][Z]);
        fprintf(fp, "%.*lf ", 15-digit(vel[i][X]), vel[i][X]);
        fprintf(fp, "%.*lf ", 15-digit(vel[i][Y]), vel[i][Y]);
        fprintf(fp, "%.*lf ", 15-digit(vel[i][Z]), vel[i][Z]);
        fprintf(fp, "\n");
    }
    fclose(fp);
}

void Compute_force(int q) {     // 计算作用力
    double x_diff, y_diff, z_diff, dist, dist_cubed;
    for (int k = 0; k < N; k++) {
        if (k == q) continue;
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        z_diff = pos[q][Z] - pos[k][Z];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff + z_diff*z_diff);
        dist_cubed = dist*dist*dist;
        forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
        forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
        forces[q][Z] -= G*masses[q]*masses[k]/dist_cubed * z_diff;
    }
}

void Compute_POSandVEL(int q) {     // 计算速度和位置
    vel[q][X] += dT/masses[q]*forces[q][X];
    vel[q][Y] += dT/masses[q]*forces[q][Y];
    vel[q][Z] += dT/masses[q]*forces[q][Z];
    pos[q][X] += dT*vel[q][X];
    pos[q][Y] += dT*vel[q][Y];
    pos[q][Z] += dT*vel[q][Z];
}

int main(int argc, char* argv[]) {
    if (argc == 2)
        thread_count = strtol(argv[1], NULL, 10);
```

```c
    if (N % thread_count != 0) {
        fprintf(stderr, "thread_count %d can not divide N!\n", thread_count);
        exit(0);
    }
    GET_TIME(start);
    init();
#   pragma omp parallel num_threads(thread_count) \
        default(none) private(step, part) \
        shared(forces, masses, pos, vel, thread_count)
    for (int step = 1; step <= n_steps; step++) {   // 迭代20次
        //memset(forces, 0, sizeof(forces));
#       pragma omp for schedule(static, N / thread_count)   // 并行for循环（块划分）
        for (part = 0; part < N; part++) {
            forces[part][X] = 0;
            forces[part][Y] = 0;
            forces[part][Z] = 0;
        }
#       pragma omp for schedule(static, N / thread_count)   // 并行for循环（块划分）
        for (part = 0; part < N; part++)
            Compute_force(part);
#       pragma omp for schedule(static, N / thread_count)   // 并行for循环（块划分）
        for (part = 0; part < N; part++)
            Compute_POSandVEL(part);
    }
    print();
    GET_TIME(stop);
    printf("Run time: %e\n", stop-start);
}
```

Pthreads

Pthreads的实现相对复杂一些。与OpenMP相同，I/O操作都由一个线程完成。因为Pthreads中没有类似于parallel for的指令，所以必须显式地决定哪个循环变量对应于线程的计算。我编写了函数Loop_schedule来决定循环变量的初始值、最终值、增量，该函数的输入是调用该函数的线程编号、线程的数目、指明采用块调度还是循环调度的参数。实际上，因为n体问题的基本算法采用块划分法会有更好的性能，所以我只使用了块调度。

```c
void Loop_schedule(int my_rank, int thread_count, int sched, int* first_p, int*
last_p, int* incr_p) {
    if (sched == CYCLIC) {      // 循环调度，每个线程从自己线程编号开始，每隔
thread_count个元素取一个
        *first_p = my_rank;
        *last_p = N;
        *incr_p = thread_count;
    }
    else {                      // 块调度，每个线程平均取N/thread_count个
        *incr_p = 1;
        *first_p = my_rank * N / thread_count;
        *last_p = *first_p + N / thread_count;
    }
}
```

OpenMP的parallel for指令有隐含的路障，如果在Pthreads实现中，简单地按照线程划分循环的迭代，那么在内部for循环的末尾不会有路障，由此会产生竞争状态。因此，必须在内部循环可能产生竞争状态的地方显式地使用路障，可以用条件变量和互斥量实现。

```
void Barrier() {    // 用条件变量和互斥量实现路障
    pthread_mutex_lock(&b_mutex);
    b_thread_count++;
    if (b_thread_count == thread_count) {
        b_thread_count = 0;
        pthread_cond_broadcast(&b_cond_var);
    }
    else
        while (pthread_cond_wait(&b_cond_var, &b_mutex) != 0);
    pthread_mutex_unlock(&b_mutex);
}
```

路障必须在每个内部循环之间添加，如果缺少某一个路障，都会带来不可预测的结果。

```
void* Thread_compute(void* rank) {  // 线程操作
    long my_rank = (long)rank;
    int first, last, incr;  // 循环变量的初始值、最终值、增量
    int step, part;
    Loop_schedule(my_rank, thread_count, BLOCK, &first, &last, &incr);
    for (step = 1; step <= n_steps; step++) {   // 迭代20次
        memset(forces, 0, sizeof(forces));
        Barrier();  // 路障
        for (part = first; part < last; part += incr)
            Compute_force(part);
        Barrier();  // 路障
        for (part = first; part < last; part += incr)
            Compute_POSandVEL(part);
        Barrier();  // 路障
    }
    return NULL;
}
```

源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <pthread.h>
#include <sys/time.h>

#define  DIM      3         // 维数
#define  X        0
#define  Y        1
#define  Z        2
#define  N        1024      // 粒子数
#define  G        1         // 引力常量
#define  dT       0.005     // 时间差
#define  n_steps  20        // 迭代数
#define  BLOCK    0         // 块调度
#define  CYCLIC   1         // 循环调度

#define  GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
```

```
}

double          start,stop;
double          masses[N];          // 质量
double          pos[N][DIM];        // 位置
double          vel[N][DIM];        // 速度
double          forces[N][DIM];     // 作用力
int             thread_count;       // 线程数
int             b_thread_count;     // 路障计数器
pthread_mutex_t b_mutex;            // 路障互斥量
pthread_cond_t  b_cond_var;         // 路障条件变量

void init() {   // 从文本文件中输入
    FILE *fp = fopen("nbody_init.txt", "r");
    if (fp == NULL) {
        printf("File cannot open! ");
        exit(0);
    }
    for (int i = 0; i < N; i++) {
        fscanf(fp, "%lf", &masses[i]);
        fscanf(fp, "%lf", &pos[i][X]);
        fscanf(fp, "%lf", &pos[i][Y]);
        fscanf(fp, "%lf", &pos[i][Z]);
        fscanf(fp, "%lf", &vel[i][X]);
        fscanf(fp, "%lf", &vel[i][Y]);
        fscanf(fp, "%lf", &vel[i][Z]);
    }
    fclose(fp);
}

int digit(double x) {   // 用于判断一个数整数部分的位数
    int k = 0;
    int y = abs((int)x);
    while (y > 0) {
        y /= 10;
        ++k;
    }
    return k;
}

void print() {  // 输出到文本文件
    FILE *fp = fopen("nbody_Pthreads_out.txt", "w");
    if (fp == NULL) {
        printf("File cannot open! ");
        exit(0);
    }
    for (int i = 0; i < N; i++) {
        fprintf(fp, "%.*lf ", 15-digit(masses[i]), masses[i]);      // 控制输出为
15位有效数字
        fprintf(fp, "%.*lf ", 15-digit(pos[i][X]), pos[i][X]);
        fprintf(fp, "%.*lf ", 15-digit(pos[i][Y]), pos[i][Y]);
        fprintf(fp, "%.*lf ", 15-digit(pos[i][Z]), pos[i][Z]);
        fprintf(fp, "%.*lf ", 15-digit(vel[i][X]), vel[i][X]);
        fprintf(fp, "%.*lf ", 15-digit(vel[i][Y]), vel[i][Y]);
        fprintf(fp, "%.*lf ", 15-digit(vel[i][Z]), vel[i][Z]);
        fprintf(fp, "\n");
    }
    fclose(fp);
```

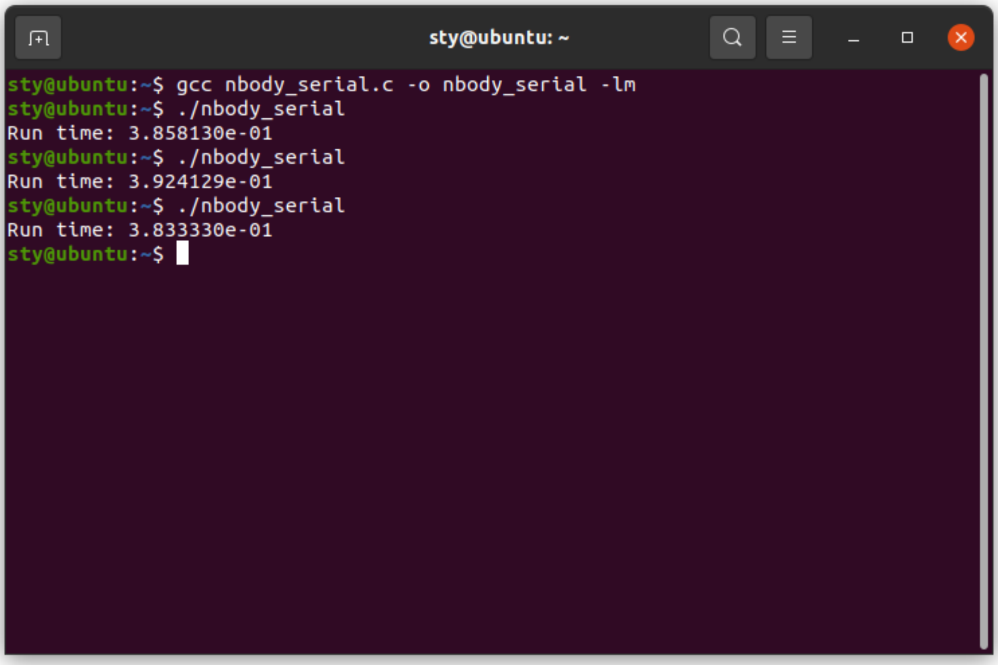
```c
}

void Loop_schedule(int my_rank, int thread_count, int sched, int* first_p, int*
last_p, int* incr_p) {
    if (sched == CYCLIC) {      // 循环调度，每个线程从自己线程编号开始，每隔
thread_count个元素取一个
        *first_p = my_rank;
        *last_p = N;
        *incr_p = thread_count;
    }
    else {                      // 块调度，每个线程平均取N/thread_count个
        *incr_p = 1;
        *first_p = my_rank * N / thread_count;
        *last_p = *first_p + N / thread_count;
    }
}

void Barrier_init() {   // 初始化路障
    b_thread_count = 0;
    pthread_mutex_init(&b_mutex, NULL);
    pthread_cond_init(&b_cond_var, NULL);
}

void Barrier() {    // 用条件变量和互斥量实现路障
    pthread_mutex_lock(&b_mutex);
    b_thread_count++;
    if (b_thread_count == thread_count) {
        b_thread_count = 0;
        pthread_cond_broadcast(&b_cond_var);
    }
    else
        while (pthread_cond_wait(&b_cond_var, &b_mutex) != 0);
    pthread_mutex_unlock(&b_mutex);
}

void Barrier_destroy() {    // 销毁路障
    pthread_mutex_destroy(&b_mutex);
    pthread_cond_destroy(&b_cond_var);
}

void Compute_force(int q) {     // 计算作用力
    double x_diff, y_diff, z_diff, dist, dist_cubed;
    for (int k = 0; k < N; k++) {
        if (k == q) continue;
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        z_diff = pos[q][Z] - pos[k][Z];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff + z_diff*z_diff);
        dist_cubed = dist*dist*dist;
        forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
        forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
        forces[q][Z] -= G*masses[q]*masses[k]/dist_cubed * z_diff;
    }
}

void Compute_POSandVEL(int q) {     // 计算速度和位置
    vel[q][X] += dT/masses[q]*forces[q][X];
    vel[q][Y] += dT/masses[q]*forces[q][Y];
```

```
        vel[q][Z] += dT/masses[q]*forces[q][Z];
        pos[q][X] += dT*vel[q][X];
        pos[q][Y] += dT*vel[q][Y];
        pos[q][Z] += dT*vel[q][Z];
    }

void* Thread_compute(void* rank) {  // 线程操作
    long my_rank = (long)rank;
    int first, last, incr;  // 循环变量的初始值、最终值、增量
    int step, part;
    Loop_schedule(my_rank, thread_count, BLOCK, &first, &last, &incr);
    for (step = 1; step <= n_steps; step++) {   // 迭代20次
        memset(forces, 0, sizeof(forces));
        Barrier();  // 路障
        for (part = first; part < last; part += incr)
            Compute_force(part);
        Barrier();  // 路障
        for (part = first; part < last; part += incr)
            Compute_POSandVEL(part);
        Barrier();  // 路障
    }
    return NULL;
}

int main(int argc, char* argv[]) {
    long thread;
    pthread_t* thread_handles;
    if (argc == 2)
        thread_count = strtol(argv[1], NULL, 10);
    if (N % thread_count != 0) {
        fprintf(stderr, "thread_count %d can not divide N!\n", thread_count);
        exit(0);
    }
    thread_handles = (pthread_t*)malloc(thread_count * sizeof(pthread_t));
    Barrier_init();     // 初始化路障
    GET_TIME(start);
    init();
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Thread_compute,
(void*)thread);   // 启动线程
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);     // 停止线程
    print();
    GET_TIME(stop);
    printf("Run time: %e\n", stop-start);
    Barrier_destroy();  // 销毁路障
    free(thread_handles);
    return 0;
}
```

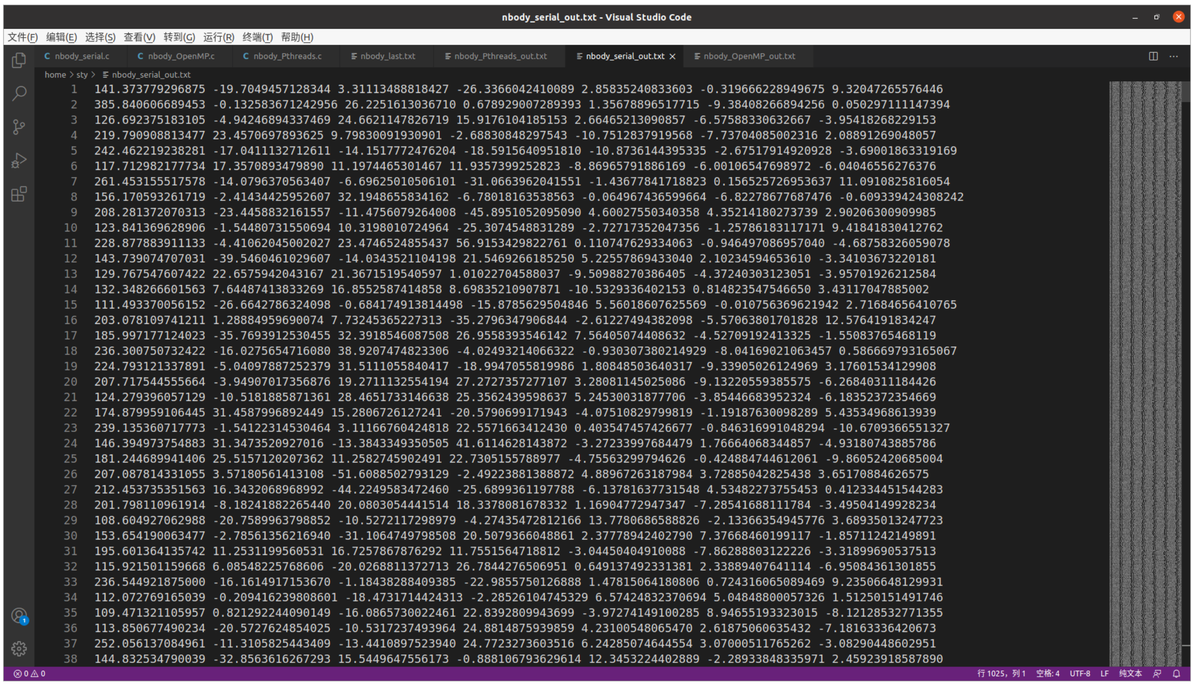
三、 实验结果

串行版本

编译运行



输出文件截图

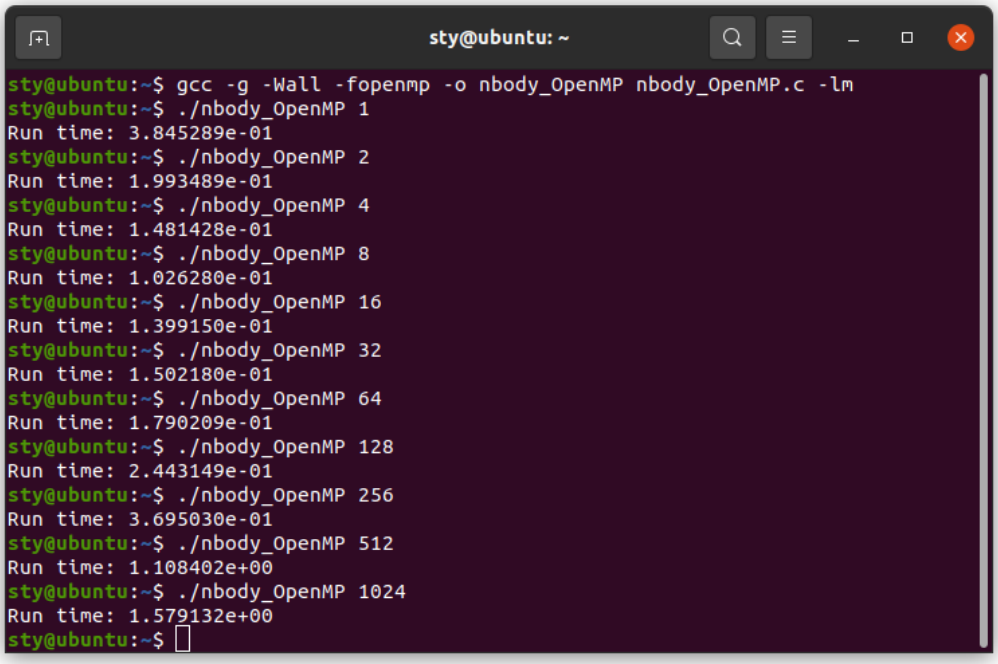


结果分析

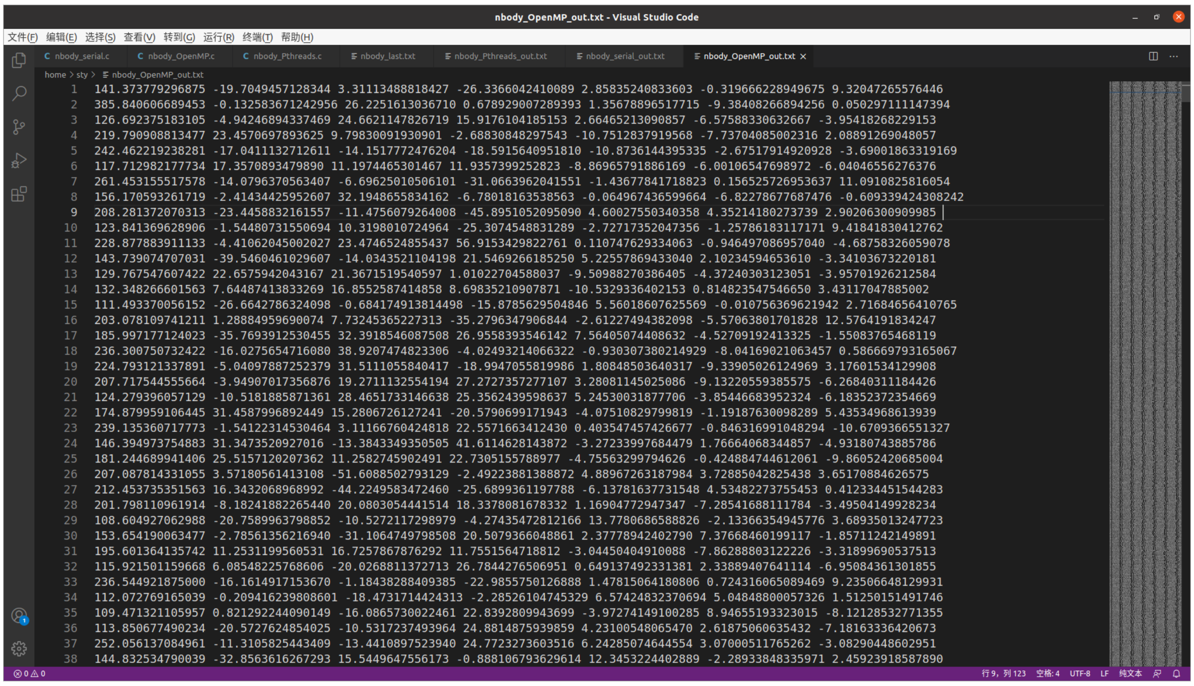
n体问题串行版本的结果与给定的标准输出文件基本相同。位置pos的三个值与给定的标准输出文件误差为10^-8数量级，速度vel的三个值与给定的标准输出文件误差为10 ^-6数量级。

OpenMP

编译运行



输出文件截图

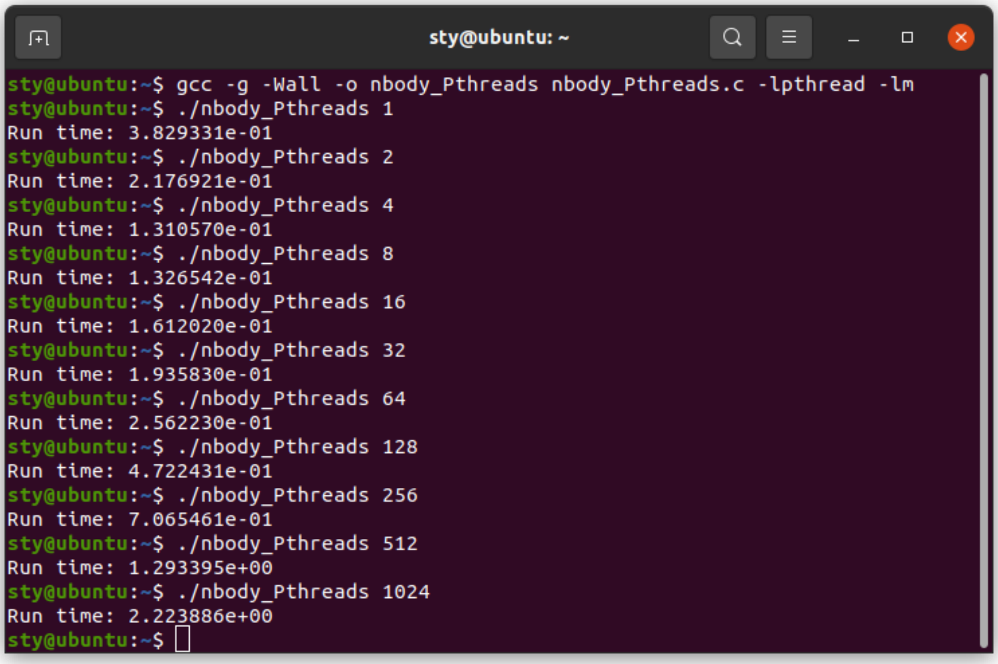


结果分析

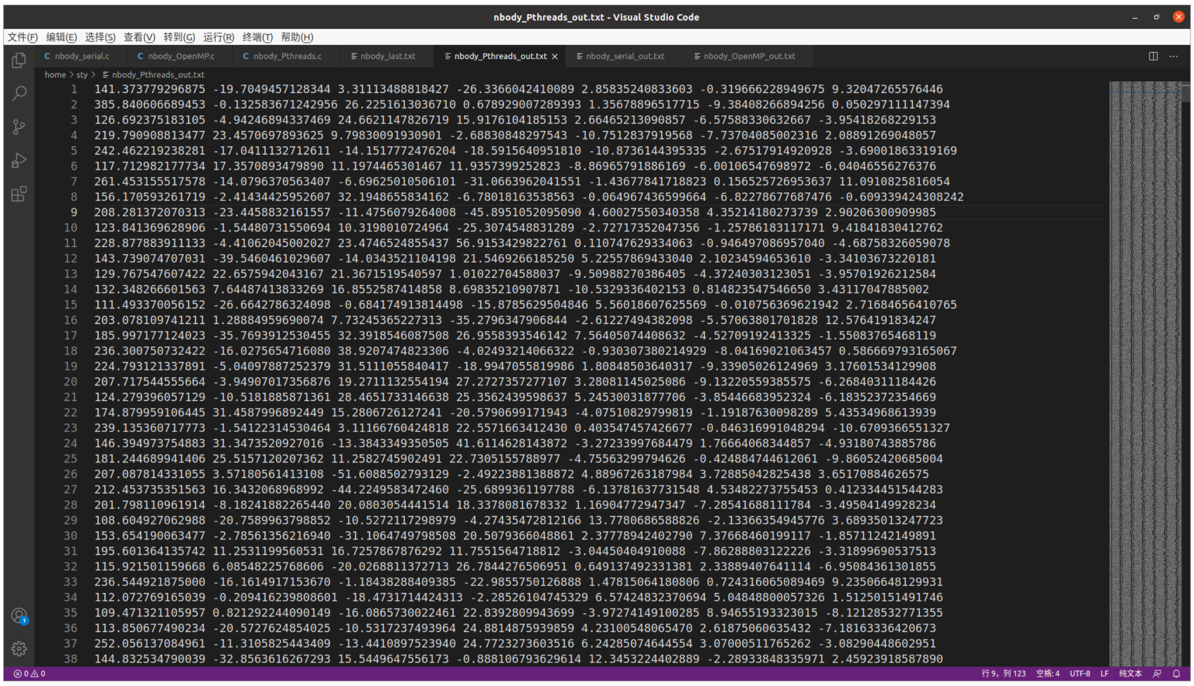
OpenMP的输出文件nbody_OpenMP_out.txt与串行版本nbody_serial_out.txt完全相同，说明OpenMP实现正确。可以发现，在数据规模N = 1024的情况下，OpenMP在线程数thread_count = 8时，运行最快。

Pthreads

编译运行



输出文件截图



结果分析

Pthreads的输出文件nbody_Pthreads_out.txt与串行版本nbody_serial_out.txt完全相同，说明Pthreads实现正确。可以发现，在数据规模N = 1024的情况下，Pthreads在线程数thread_count = 4和thread_count = 8时，运行最快。