**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&黄聃

|  |  |  |  |
| --- | --- | --- | --- |
| 年级 | **2019级** | 专业（方向） | **计算机科学与技术** |
| 学号 | **18329015** | 姓名 | **郝裕玮** |
| 开始日期 | **2021.5.6** | 完成日期 | **2021.5.11** |

1. **实验题目**

nbody问题

1. **实验内容**

实验题目内容为：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，pthread中的任意两种版本。

在经过学习和对比之后，我决定实现nbody问题的串行，OpenMP和pthread的三种版本。

**（1）串行版本**

具体分析已全部放在代码注释中，代码如下所示：

#include<iostream>

#include<cmath>   //公式里需要用到sqrt开根函数

#include<fstream> //该头文件用于读写文件

#include<iomanip> //该头文件用于控制数据精度

#include<sys/time.h> //标准日期时间头文件，用于计算运行时间

using namespace std;

#define timestep 20     //迭代次数

#define dT 0.005        //时间间隔

#define G 1             //引力常数

#define MAX 1024        //各信息的数组大小均设置为1024，与粒子个数相同

#define particles 1024   //数据集中共有1024个粒子

#define GET\_TIME(now) { \

   struct timeval t; \

   gettimeofday(&t, NULL); \

   now = t.tv\_sec + t.tv\_usec/1000000.0; \

}

//最后一行将us转换为s，统一单位

//该结构体用于计算程序串行部分运行时间

struct information

{

   double x;

   double y;

   double z;

}pos[MAX],v[MAX];//初始化两个数组，分别为每个粒子的位置和速度

                 //数组内部每个元素存储了对应粒子在x轴，y轴，z轴上的对应分量

int main()

{

   int i,j,k,s;//循环变量

   double x\_diff,y\_diff,z\_diff;//两粒子之间在三条坐标轴上的相对距离

   double dist,dist\_cubed;//两粒子之间的绝对距离和绝对距离的立方

   double x\_f,y\_f,z\_f;//两粒子之间作用力在三条坐标轴上的分量

   double m[MAX];//每个粒子的质量

   double start,end;//用于计算串行部分的开始时间和结束时间

   ifstream myfile("C:\\Users\\93508\\Desktop\\.vscode\\.vscode\\nbody\_init.txt");//打开数据集文件

   for(i=0;i<=particles-1;i++){

      myfile>>m[i]>>pos[i].x>>pos[i].y>>pos[i].z>>v[i].x>>v[i].y>>v[i].z;//根据题目要求将每一行的数据写入到各数组对应的结构体成员中

   }

   GET\_TIME(start);//串行开始

   for(k=1;k<=timestep;k++){//迭代次数为timestep次（20）

      for(i=0;i<=particles-1;i++){//遍历每个粒子

         x\_f=0;

         y\_f=0;

         z\_f=0;

         //每次遍历该粒子以外的粒子之前需要先将作用力清空，防止与上一个粒子在三个方向上所受的作用力相叠加

         for(j=0;j<=particles-1;j++){//遍历该粒子以外的所有粒子

         //因为该粒子以外的所有粒子都对它有作用力

            if(i==j){//该粒子本身可以跳过

               continue;

            }

            x\_diff=pos[i].x-pos[j].x;

            y\_diff=pos[i].y-pos[j].y;

            z\_diff=pos[i].z-pos[j].z;

            //计算两粒子在三条坐标轴上的相对距离

            dist=sqrt(x\_diff\*x\_diff+y\_diff\*y\_diff+z\_diff\*z\_diff);//计算两粒子之间的绝对距离

            dist\_cubed=dist\*dist\*dist;//计算距离的立方

            x\_f-=G\*m[i]\*m[j]/dist\_cubed\*x\_diff;

            y\_f-=G\*m[i]\*m[j]/dist\_cubed\*y\_diff;

            z\_f-=G\*m[i]\*m[j]/dist\_cubed\*z\_diff;

            //计算两粒子之间作用力在三条坐标轴上的分量

         }

         v[i].x+=dT\*x\_f/m[i];

         v[i].y+=dT\*y\_f/m[i];

         v[i].z+=dT\*z\_f/m[i];

         //利用dv=f\*dT/m(动量定理)来更新粒子在三条坐标轴上的速度分量

      }

      for(s=0;s<=particles-1;s++){//遍历每个粒子

         pos[s].x+=dT\*v[s].x;

         pos[s].y+=dT\*v[s].y;

         pos[s].z+=dT\*v[s].z;

         //利用d=d0+v\*dT来更新粒子在三条坐标轴上的坐标

      }

   }

   GET\_TIME(end);//获取串行部分结束时间

   cout<<end-start<<endl;//输出运行时间

   ofstream outfile;

   outfile.open("C:\\Users\\93508\\Desktop\\.vscode\\.vscode\\serial\_result.txt",ios::out);

   //准备将结果写入serial\_result文件中

   for(i=0;i<=particles-1;i++){

      outfile<<setprecision(15)<<m[i]<<' '<<pos[i].x<<' '<<pos[i].y<<' '<<pos[i].z<<' '<<v[i].x<<' '<<v[i].y<<' '<<v[i].z<<endl;

      //将数据保留15位小数并按照要求把每个粒子的数据按顺序写入每行，一行数据代表一个粒子

   }

   myfile.close();

   outfile.close();

   //关闭文件

   system("pause");

}

**（2）OpenMP版本**

OpenMP版本大体内容与串行版本保持一致，唯一的不同是在三个循环前均增加了#pragma omp指令。同时需要注意修改编译参数（eg.在VSCode上编译时，需要在tasks.json中的args里添加一行”-fopenmp”）。

    "args": [

        "-g",

        "${file}",

        "-fopenmp",//需要添加的

        "-o",

        "${fileBasenameNoExtension}.exe"

    ], // 编译命令参数

绝大部分分析已全部放在代码注释中，代码如下所示：

#include<iostream>

#include<cmath>   //公式里需要用到sqrt开根函数

#include<fstream> //该头文件用于读写文件

#include<iomanip> //该头文件用于控制数据精度

#include<sys/time.h> //标准日期时间头文件，用于计算运行时间

using namespace std;

#define timestep 20     //迭代次数

#define dT 0.005        //时间间隔

#define G 1             //引力常数

#define MAX 1024        //各信息的数组大小均设置为1024，与粒子个数相同

#define particles 1024   //数据集中共有1024个粒子

#define GET\_TIME(now) { \

   struct timeval t; \

   gettimeofday(&t, NULL); \

   now = t.tv\_sec + t.tv\_usec/1000000.0; \

}

//最后一行将us转换为s，统一单位

//该结构体用于计算程序串行部分运行时间

struct information

{

   double x;

   double y;

   double z;

}pos[MAX],v[MAX];//初始化两个数组，分别为每个粒子的位置和速度

                 //数组内部每个元素存储了对应粒子在x轴，y轴，z轴上的对应分量

int main()

{

   int i,j,k,s;//循环变量

   double x\_diff,y\_diff,z\_diff;//两粒子之间在三条坐标轴上的相对距离

   double dist,dist\_cubed;//两粒子之间的绝对距离和绝对距离的立方

   double x\_f,y\_f,z\_f;//两粒子之间作用力在三条坐标轴上的分量

   double m[MAX];//每个粒子的质量

   double start,end;//用于计算串行部分的开始时间和结束时间

   int thread\_count;//线程数

   ifstream myfile("C:\\Users\\93508\\Desktop\\.vscode\\.vscode\\nbody\_init.txt");//打开数据集文件

   for(int i=0;i<=particles-1;i++){

      myfile>>m[i]>>pos[i].x>>pos[i].y>>pos[i].z>>v[i].x>>v[i].y>>v[i].z;//根据题目要求将每一行的数据写入到各数组对应的结构体成员中

   }

   thread\_count=8;//可在这里选择或修改并行线程数

   GET\_TIME(start);//获取omp并行部分开始时间

# pragma omp parallel num\_threads(thread\_count) \

private(x\_f,y\_f,z\_f,i,j,k,s,x\_diff,y\_diff,z\_diff,dist,dist\_cubed) \

shared(m,pos,v)

//用parallel指令在最外层循环前创建thread\_count个线程的集合。

//同时将作用力（x\_f,y\_f,z\_f），循环计数变量（i,j,k,s），距离相关变量（x\_diff,y\_diff,z\_diff,dist,dist\_cubed）均设置为private（这些变量是每个线程独有的，不可被其他线程访问）

//而m,pos,v这三个数组则设置为shared，因为粒子的这三个信息会被每个线程访问（并且pos和v的信息在不断更新，所以更加需要被共享）

   for(k=1;k<=timestep;k++){

#        pragma omp for schedule(static)

//使用for指令，告诉OpenMP用已有的线程组来并行化for循环。

//schedule(static)含义为：对for循环并行化进行任务调度

//对于static:这种调度方式非常简单。假设有n次循环迭代，t个线程，那么给每个线程静态分配大约n/t次连续的迭代。

         for(i=0;i<=particles-1;i++){//遍历每个粒子

            x\_f=0;

            y\_f=0;

            z\_f=0;

            //每次遍历该粒子以外的粒子之前需要先将作用力清空，防止与上一个粒子在三个方向上所受的作用力相叠加

            for(j=0;j<=particles-1;j++){//遍历该粒子以外的所有粒子

               if(i==j){//该粒子本身可以跳过

                  continue;

               }

               x\_diff=pos[i].x-pos[j].x;

               y\_diff=pos[i].y-pos[j].y;

               z\_diff=pos[i].z-pos[j].z;

               //计算两粒子在三条坐标轴上的相对距离

               dist=sqrt(x\_diff\*x\_diff+y\_diff\*y\_diff+z\_diff\*z\_diff);//计算两粒子之间的绝对距离

               dist\_cubed=dist\*dist\*dist;//计算距离的立方

               x\_f-=G\*m[i]\*m[j]/dist\_cubed\*x\_diff;

               y\_f-=G\*m[i]\*m[j]/dist\_cubed\*y\_diff;

               z\_f-=G\*m[i]\*m[j]/dist\_cubed\*z\_diff;

               //计算两粒子之间作用力在三条坐标轴上的分量

            }

            v[i].x+=dT\*x\_f/m[i];

            v[i].y+=dT\*y\_f/m[i];

            v[i].z+=dT\*z\_f/m[i];

            //利用dv=f\*dT/m(动量定理)来更新粒子在三条坐标轴上的速度分量

         }

#     pragma omp for schedule(static) //解释与上方的omp for相同

      for(s=0;s<=particles-1;s++){//遍历每个粒子

         pos[s].x+=dT\*v[s].x;

         pos[s].y+=dT\*v[s].y;

         pos[s].z+=dT\*v[s].z;

         //利用d=d0+v\*dT来更新粒子在三条坐标轴上的坐标

      }

   }

   GET\_TIME(end);//获取omp并行部分结束时间

   cout<<end-start<<endl;//输出运行时间

   ofstream outfile;

   outfile.open("C:\\Users\\93508\\Desktop\\.vscode\\.vscode\\omp\_result1.txt",ios::out);

   //准备将结果写入omp\_result文件中

   for(i=0;i<=particles-1;i++){

      outfile<<setprecision(15)<<m[i]<<' '<<pos[i].x<<' '<<pos[i].y<<' '<<pos[i].z<<' '<<v[i].x<<' '<<v[i].y<<' '<<v[i].z<<endl;

      //将数据保留15位小数并按照要求把每个粒子的数据按顺序写入每行，一行数据代表一个粒子

   }

   myfile.close();

   outfile.close();

   //关闭文件

   system("pause");

}

对于# pragma omp parallel和# pragma omp for结合使用的补充解释：

与parallel for指令不同的是，for指令并不创建任何线程。它使用已经在parallel块中创建的线程。且在循环的末尾有一个隐式的路障，所以代码的结果将与在内部使用两个# pragma omp parallel for的结果保持一致。

**（3）pthread版本**

Pthreads版本需要在VSCode上进行额外配置（参考网址：https://blog.csdn.net/CSDN\_WHB/article/details/81475233）。同时需要注意修改编译参数（eg.在VSCode上编译时，需要在tasks.json中的args里添加一行”-lpthread”）：

    "args": [

        "-g",

        "${file}",

        "-fopenmp",

        "-o",

        "${fileBasenameNoExtension}.exe",

        "-lpthread"//需要添加的

    ], // 编译命令参数

绝大部分分析已全部放在代码注释中，代码如下所示：

#include<iostream>

#include<cmath>   //公式里需要用到sqrt开根函数

#include<fstream> //该头文件用于读写文件

#include<iomanip> //该头文件用于控制数据精度

#include<sys/time.h> //标准日期时间头文件，用于计算运行时间

#include<pthread.h>

using namespace std;

#define timestep 20     //迭代次数

#define dT 0.005        //时间间隔

#define G 1             //引力常数

#define MAX 1024        //各信息的数组大小均设置为1024，与粒子个数相同

#define particles 1024   //数据集中共有1024个粒子

#define GET\_TIME(now) { \

   struct timeval t; \

   gettimeofday(&t, NULL); \

   now = t.tv\_sec + t.tv\_usec/1000000.0; \

}

//最后一行将us转换为s，统一单位

//该结构体用于计算程序串行部分运行时间

struct information

{

   double x;

   double y;

   double z;

}pos[MAX],v[MAX];//初始化两个数组，分别为每个粒子的位置和速度

                 //数组内部每个元素存储了对应粒子在x轴，y轴，z轴上的对应分量

long thread\_count;         // 并行线程数

int b\_thread\_count;       // 线程互斥锁计数器

pthread\_mutex\_t mutex;        // 线程互斥量，也即互斥锁

pthread\_cond\_t cond\_var;      // 线程条件变量

double m[MAX];//每个粒子的质量

void Barrier\_init(void){//通过条件变量等待法来同步所有线程

    b\_thread\_count=0;//计数器初始化为0

    pthread\_mutex\_init(&mutex,NULL);//初始化互斥锁mutex

    pthread\_cond\_init(&cond\_var,NULL);//初始化条件变量conda\_var

 }

void Barrier(void){//设置路障

    pthread\_mutex\_lock(&mutex);//线程调用该函数来获得临界区的访问权

    //互斥量可以用来限制每次只有1个线程能进入临界区。互斥量保证了一个线程独享临界区

    //其他线程在有线程已经进入该临界区的情况下，不能同时进入。

    b\_thread\_count++;//计数器+1

    if(b\_thread\_count==thread\_count){//说明所有线程均已获得互斥锁

        b\_thread\_count=0;//计数器清零

        pthread\_cond\_broadcast(&cond\_var);//唤醒所有等待条件变量cond\_var的线程

        //也即被阻塞在当前锁mutex上的线程会被唤醒。

    }

    else{

        while(pthread\_cond\_wait(&cond\_var, &mutex)!=0); //详见下方补充

    }

    pthread\_mutex\_unlock(&mutex);//线程退出临界区

}

void\* loop\_schedule(void\* rank){

    long long my\_rank=(long long)rank;//如果指针类型的大小和表示进程编号的整数类型不同，在编译时就会受到警告

    //所以我们需要进行强制类型转换

    long long my\_n=MAX/thread\_count;//计算每个线程中需要分配的循环次数

    long long my\_first\_i=my\_n\*my\_rank;//计算每个线程中开始计算的第一项

    long long my\_last\_i=my\_first\_i+my\_n;//计算每个线程中需要计算的最后一项、

    int i,j,k,s;//循环变量

    double x\_diff,y\_diff,z\_diff;//两粒子之间在三条坐标轴上的相对距离

    double dist,dist\_cubed;//两粒子之间的绝对距离和绝对距离的立方

    double x\_f,y\_f,z\_f;//两粒子之间作用力在三条坐标轴上的分量

    for(k=1;k<=timestep;k++){//迭代次数为timestep次（20）

        for(i=my\_first\_i;i<=my\_last\_i-1;i++){//遍历该线程所分配到的粒子

            x\_f=0;

            y\_f=0;

            z\_f=0;

            //每次遍历该粒子以外的粒子之前需要先将作用力清空，防止与上一个粒子在三个方向上所受的作用力相叠加

            for(j=0;j<=particles-1;j++){//遍历该粒子以外的所有粒子

            //因为该粒子以外的所有粒子都对它有作用力

                if(i==j){//该粒子本身可以跳过

                continue;

                }

                x\_diff=pos[i].x-pos[j].x;

                y\_diff=pos[i].y-pos[j].y;

                z\_diff=pos[i].z-pos[j].z;

                //计算两粒子在三条坐标轴上的相对距离

                dist=sqrt(x\_diff\*x\_diff+y\_diff\*y\_diff+z\_diff\*z\_diff);//计算两粒子之间的绝对距离

                dist\_cubed=dist\*dist\*dist;//计算距离的立方

                x\_f-=G\*m[i]\*m[j]/dist\_cubed\*x\_diff;

                y\_f-=G\*m[i]\*m[j]/dist\_cubed\*y\_diff;

                z\_f-=G\*m[i]\*m[j]/dist\_cubed\*z\_diff;

                //计算两粒子之间作用力在三条坐标轴上的分量

            }

            v[i].x+=dT\*x\_f/m[i];

            v[i].y+=dT\*y\_f/m[i];

            v[i].z+=dT\*z\_f/m[i];

            //利用dv=f\*dT/m(动量定理)来更新粒子在三条坐标轴上的速度分量

        }

        Barrier();//设置路障

        for(s=my\_first\_i;s<=my\_last\_i-1;s++){//遍历该线程所分配到的粒子

            pos[s].x+=dT\*v[s].x;

            pos[s].y+=dT\*v[s].y;

            pos[s].z+=dT\*v[s].z;

            //利用d=d0+v\*dT来更新粒子在三条坐标轴上的坐标

        }

        Barrier();//设置路障

    }

    return NULL;

}

int main()

{

    int i,j,k,s;//循环变量

    ifstream myfile("C:\\Users\\93508\\Desktop\\.vscode\\.vscode\\nbody\_init.txt");//打开数据集文件

    for(i=0;i<=particles-1;i++){

      myfile>>m[i]>>pos[i].x>>pos[i].y>>pos[i].z>>v[i].x>>v[i].y>>v[i].z;//根据题目要求将每一行的数据写入到各数组对应的结构体成员中

    }

    pthread\_t\* thread\_handles;//pthread\_t 用于声明线程ID

    double start,end;//用于计算pthread并行部分的开始时间和结束时间

    long long thread;

    thread\_count=8;//可在这里选择或修改并行线程数

    Barrier\_init();//路障初始化，注意要放在创建线程数组之前！

    thread\_handles = (pthread\_t\*) malloc (thread\_count\*sizeof(pthread\_t));//创建线程数组

    GET\_TIME(start); //获取pthread并行部分开始时间

    for(thread=0;thread<thread\_count;thread++){

        pthread\_create(&thread\_handles[thread],NULL,loop\_schedule,(void\*)thread);

//创建线程

        //第一个参数为指向线程标识符的指针。

        //第二个参数用来设置线程属性。

        //第三个参数是线程运行函数的起始地址。

        //最后一个参数是运行函数的参数。

    }

    for(thread=0;thread<thread\_count;thread++){

        pthread\_join(thread\_handles[thread], NULL);

        //函数pthread\_join用来等待一个线程的结束。

        //第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。

        //这个函数是一个线程阻塞的函数，调用它的线程将一直等待到被等待的线程结束为止

        //当函数返回时，被等待线程的资源被收回。

        //也就是说主线程中要是加了这段代码，就会在该代码所处的位置卡住，直到这个线程执行完毕才会继续往下运行。

    }

    GET\_TIME(end);//获取pthread并行部分结束时间

    cout<<end-start<<endl;//输出运行时间

    ofstream outfile;

    outfile.open("C:\\Users\\93508\\Desktop\\.vscode\\.vscode\\pthread\_result.txt",ios::out);

    //准备将结果写入pthread\_result文件中

    for(i=0;i<=particles-1;i++){

      outfile<<setprecision(15)<<m[i]<<' '<<pos[i].x<<' '<<pos[i].y<<' '<<pos[i].z<<' '<<v[i].x<<' '<<v[i].y<<' '<<v[i].z<<endl;

      //将数据保留15位小数并按照要求把每个粒子的数据按顺序写入每行，一行数据代表一个粒子

    }

    myfile.close();

    outfile.close();

    //关闭文件

    system("pause");

}

对上述代码的补充：

关于互斥量：因为处于忙等待的线程仍然在持续使用CPU，所以忙等待不是限制临界区访问的最理想方法。这里，有两个更好的方法:互斥量和信号量。而互斥量是互斥锁的简称，它是一个特殊类型的变量，通过某些特殊类型的函数，互斥量可以用来限制每次只有一个线程能进入临界区。互斥量保证了一个线程独享临界区，其他线程在有线程已经进入该临界区的情况下，不能同时进入。

关于

        while(pthread\_cond\_wait(&cond\_var, &mutex)!=0);

pthread\_cond\_wait的作用是通过互斥量mutex\_p来阻塞线程，直到其他线程调用pthread\_cond\_signa1或者pthread\_cond \_broadcast来解锁它。当线程解锁后，它重新获得互斥量。

那为什么在被唤醒之后还要再次进行条件判断（即为什么要使用while循环来判断条件）？是因为可能有“惊群效应”。我们假设有多个线程都在等待这个条件，而同时只能有一个线程进行处理，那么此时就必须要再次条件判断，以确保只有一个线程进入临界区处理。

所以用while来保证临界区内只有一个线程在处理。

1. **实验结果**

线程不变，迭代次数增加：

迭代20次（8线程）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 串行时间 | OpenMP运行时间 | Pthread运行时间 | OpenMP加速比 | Pthread加速比 |
| 0.457774 | 0.130446 | 0.130771 | 3.509 | 3.501 |

迭代200次（8线程）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 串行时间 | OpenMP运行时间 | Pthread运行时间 | OpenMP加速比 | Pthread加速比 |
| 4.37327 | 1.20676 | 1.23047 | 3.624 | 3.554 |

迭代2000次（8线程）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 串行时间 | OpenMP运行时间 | Pthread运行时间 | OpenMP加速比 | Pthread加速比 |
| 43.5725 | 22.6479 | 22.2641 | 1.924 | 1.957 |

由上可知，随着迭代次数的增加，OpenMP和Pthread的加速效率会先提高再下降。但OpenMP和Pthread相对于串行的加速比始终几乎一致，两者并无显著差异。

先升高后下降的原因是：因为线程不变，而随着迭代次数的增加，每个线程的负载就会加重。所以当处于线程负载能力范围内时，加速比会随迭代次数的增加而增加；而在超出线程负载能力范围之后，加速比就会随着迭代次数的增加而降低。

迭代次数不变，线程增加：

迭代200次（4线程）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 串行时间 | OpenMP运行时间 | Pthread运行时间 | OpenMP加速比 | Pthread加速比 |
| 4.35834 | 1.45708 | 1.47844 | 2.991 | 2.948 |

迭代200次（8线程）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 串行时间 | OpenMP运行时间 | Pthread运行时间 | OpenMP加速比 | Pthread加速比 |
| 4.50996 | 1.18379 | 1.21345 | 3.810 | 3.717 |

迭代200次（32线程）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 串行时间 | OpenMP运行时间 | Pthread运行时间 | OpenMP加速比 | Pthread加速比 |
| 4.35734 | 1.40324 | 1.44886 | 3.105 | 3.007 |

由上可知，随着线程数的适当增加，OpenMP和Pthread的加速效率也会提高。且OpenMP和Pthread相对于串行的加速比几乎一致，两者并无显著差异。

但若线程过多，也会导致加速效率降低。这很显然也是由运行程序的电脑配置决定的，由于我的电脑是4核8线程，所以显然8线程的表现会最好，若继续分配额外的线程，则必然会导致加速比下降。



因为每一时刻，都只会有8个线程在进行计算，而并行的结束是以所有的线程运行结束为标志，因此，最后那个线程（或者说最慢的那个线程）将决定并行结束的时间，所以线程越多，那么排队就越长，并行结束的时间相应就会增长。