并行程序设计实践大作业

专业和班级：计算机2018-5

姓名：尹浩男

学号：201801051827

时间：2020年10月30日

**目录**

1. 并行程序的发展和当前应用概述
2. GPU编程实践的作业
3. MPICH编程实践的作业

## 一 并行程序的发展和当前应用概述

### 并行计算

（英语：**parallel computing**）一般是指许多指令得以同时进行的计算模式。在同时进行的前提下，可以将计算的过程分解成小部分，之后以[并发](https://zh.wikipedia.org/wiki/%E5%B9%B6%E5%8F%91%E8%AE%A1%E7%AE%97" \o "并发计算)方式来加以解决[[1]](https://zh.wikipedia.org/wiki/%E5%B9%B6%E8%A1%8C%E8%AE%A1%E7%AE%97#cite_note-1)。

电脑软件可以被分成数个运算步骤来运行。为了解决某个特定问题，软件采用某个算法，以一连串指令运行来完成。传统上，这些指令都被送至单一的[中央处理器](https://zh.wikipedia.org/wiki/%E4%B8%AD%E5%A4%AE%E5%A4%84%E7%90%86%E5%99%A8" \o "中央处理器)，以循序方式运行完成。在这种处理方式下，单一时间中，只有单一指令被运行(processor level: 比较微处理器，CISC, 和RISC，即流水线Pipeline的概念，以及后来在Pipeline基础上以提高指令处理效率为目的的硬件及软件发展，比如branch-prediction, 比如forwarding，比如在每个运算单元前的指令堆栈，汇编程序员对programm code的顺序改写)。并行运算采用了多个运算单元，同时运行，以解决问题。

### 基本体系结构

相对于[串行计算](https://zh.wikipedia.org/w/index.php?title=%E4%B8%B2%E8%A1%8C%E8%AE%A1%E7%AE%97&action=edit&redlink=1)，并行计算可以划分成时间并行和空间并行。时间并行即[指令流水化](https://zh.wikipedia.org/wiki/%E6%B5%81%E6%B0%B4%E7%BA%BF_(%E8%AE%A1%E7%AE%97%E6%9C%BA))，空间并行使用多个处理器执行并发计算，当前研究的主要是空间的并行问题。以程序和算法设计人员的角度看，并行计算又可分为[数据并行](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE%E5%B9%B6%E8%A1%8C)和[任务并行](https://zh.wikipedia.org/wiki/%E4%BB%BB%E5%8A%A1%E5%B9%B6%E8%A1%8C)。[数据并行](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE%E5%B9%B6%E8%A1%8C)把大的任务化解成若干个相同的子任务，处理起来比[任务并行](https://zh.wikipedia.org/wiki/%E4%BB%BB%E5%8A%A1%E5%B9%B6%E8%A1%8C)简单。

空间上的并行导致两类并行机的产生，按照麦克·弗莱因（Michael Flynn）的说法分为[单指令流多数据流](https://zh.wikipedia.org/wiki/%E5%8D%95%E6%8C%87%E4%BB%A4%E6%B5%81%E5%A4%9A%E6%95%B0%E6%8D%AE%E6%B5%81" \o "单指令流多数据流)（SIMD）和[多指令流多数据流](https://zh.wikipedia.org/wiki/%E5%A4%9A%E6%8C%87%E4%BB%A4%E6%B5%81%E5%A4%9A%E6%95%B0%E6%8D%AE%E6%B5%81" \o "多指令流多数据流)（MIMD），而常用的串行机也称为[单指令流单数据流](https://zh.wikipedia.org/wiki/%E5%96%AE%E6%8C%87%E4%BB%A4%E6%B5%81%E5%96%AE%E6%95%B8%E6%93%9A%E6%B5%81" \o "单指令流单数据流)（SISD）。MIMD类的机器又可分为常见的五类：[并行向量处理机](https://zh.wikipedia.org/wiki/%E5%B9%B6%E8%A1%8C%E5%90%91%E9%87%8F%E5%A4%84%E7%90%86%E6%9C%BA" \o "并行向量处理机)（PVP）、[对称多处理机](https://zh.wikipedia.org/wiki/%E5%AF%B9%E7%A7%B0%E5%A4%9A%E5%A4%84%E7%90%86%E6%9C%BA" \o "对称多处理机)（SMP）、[大规模并行处理机](https://zh.wikipedia.org/wiki/%E5%A4%A7%E8%A7%84%E6%A8%A1%E5%B9%B6%E8%A1%8C%E5%A4%84%E7%90%86%E6%9C%BA" \o "大规模并行处理机)（MPP）、[工作站机群](https://zh.wikipedia.org/wiki/%E5%B7%A5%E4%BD%9C%E7%AB%99%E6%9C%BA%E7%BE%A4" \o "工作站机群)（COW）、[分布式共享存储处理机](https://zh.wikipedia.org/wiki/%E5%88%86%E5%B8%83%E5%BC%8F%E5%85%B1%E4%BA%AB%E5%AD%98%E5%82%A8%E5%A4%84%E7%90%86%E6%9C%BA" \o "分布式共享存储处理机)（DSM）。

### 发展

40 年代开始的现代计算机发展历程可以分为两个明显的发展时代：串行计算时代、并行计算时代。每一个计算时代都从体系结构发展开始，接着是系统软件、应用软件，最后随着问题求解环境的发展而达到顶峰。创建和使用并行计算机的主要原因是因为并行计算机是解决单处理器速度瓶颈的最好方法之一。多核计算平台的普及化使得并行程序设计成为一种编程技术主流。其实并行计算的软件技术早已存在了几十年，然而其原来主要服务于高性能计算一类的应用，所以并行化编程一直也都为阳春白雪的光环笼罩。任何新事物，都是在旧事物的基础上发展演变出来的，新的并行编程模型的产生也是一样，不会完全摆脱现在流行的并行编程模型的影响。目前流行的并行编程模型主要有以下几种：

（1）SMP编程模型

（2）消息传送编程模型

（3）数据并行编程模型

（4）面向对象等其他编程模型

预测模型的构造和模拟、工程设计和自动化、能源勘探、医学、军事、机械制造、计算数学、石油资源模拟问题，常常涉及到现代的复杂数学问题和计算方法，具有很强的实用性。这些领域的许多现象的描述都是各种数学物理方程，从数学物理方程的求解方法导出大型稀疏线性方程组，而并行计算在求解大型稀疏线性方程组方面具有巨大的优势。到目前为止，并行程序已经在勘探地球物理、机械制造、计算数学、人工智能、图像处理等领域得到了广泛的应用。

## 二、GPU编程实践的作业、

### 2.1 项目要求

编写一个矩阵乘法的GPU并行程序，并且与对应规模的串行程序进行运行时间的比对（n = 10 、100、200、500、1000、1500、2000）

矩阵A(n,n) 矩阵B(n,n) C = A X B

### 2.2串行程序

#### 2.2.1基本思路和简单介绍

为了和以前的知识串联起来，我使用了c++的类来巩固一下知识，使用了SerialMatrixMultiply类封装了矩阵乘法的操作。包括矩阵初始化，空间申请，空间释放。这样的操作能够极大简化main程序的代码行数，如果要测试多个数据的时候可以很方便的进行测试，代码冗余小。

而且我还会读取函数执行时的参数，可以在不改变源代码的情况下更改矩阵规模，不用每次都进行编译。方便了数据的测试。

基本的矩阵相乘很简单，就是三次for循环就可以解决，时间复杂度是o(n^3)

#### 2.2.2 源代码展示

##### SerialMatrixMultiply.h 文件

#include "stdio.h"

/\*\*

 \* @Author: yhn

 \* @Date: 2020/9/16 15:03

 \* @Description: 本 .h 文件用于串行执行矩阵相乘的任务

 \*\*/

class SerialMatrixMultiply{

private:

    long size;

    long \*\*longArrayA; //要申请的二级指针 会调用myMalloc()函数申请一个 size \* size 的数组

    long \*\*longArrayB; //要申请的二级指针 会调用myMalloc()函数申请一个 size \* size 的数组

    long \*\*longArrayC; //要申请的二级指针 会调用myMalloc()函数申请一个 size \* size 的数组

    /\*\*

     \* @brief 主要用于动态申请(malloc) size \* size 的二维数组空间

     \* @param longArrayX 传入引用的指针 否则就会出现断错误 因为二级指针也是一个变量 占四个字节32位空间

     \*/

    void  myMalloc(long  \*\* & longArrayX);

    /\*\*

     \* @brief 主要用于给传入的值动态申请空间 并初始化值 每行的数据 从1 - size

     \* @param longArrayX  传入引用的指针 否则就会出现断错误 因为二级指针也是一个变量 占四个字节32位空间

     \*/

    void setArray(long \*\* & longArrayX);

    /\*\*

     \* @brief  setZero函数主要用于给一个二维数组赋初值为0

     \* @param longArrayX 需要全部置零的数组二级指针

     \*/

    void setZero(long \*\* & longArrayX);

    /\*\*

     \* @brief  打印传入的二维数组的值 用于调试

     \* @param longArrayX  需要打印的数组指针

     \*/

    void print(long \*\* longArrayX);

public:

    /\*\*

     \* @brief  构造调用 setarray函数开辟空间

     \* @param ssize  MatrixMultiply类的构造函数

     \*/

    SerialMatrixMultiply(long ssize);

    /\*\*

     \* @brief 执行矩阵相乘的命令

     \*/

    void multiply();

    /\*\*

     \* @brief 打印经过计算后的矩阵C 用于测试

     \*/

    void printCArray();

};

##### SerialMatrixMultiply.cpp 文件

#include "SerialMatrixMultiply.h"

#include <malloc.h>

#include <iostream>

using  namespace std;

/\*\*

 \* @Author: yhn

 \* @Date: 2020/9/16 15:03

 \* @Description: 本.cpp文件用于串行执行矩阵相乘的任务

 \*\*/

void  SerialMatrixMultiply::myMalloc(long  \*\* & longArrayX){

    //申请空间   size \* size 个空间

    longArrayX = new long\*[size];

    for(long i= 0;i<size;i++){

        longArrayX[i] = new long[size];

    }

}

void SerialMatrixMultiply::setArray(long \*\* &longArrayX) {

     myMalloc(longArrayX);

    for(int i=0;i<size;i++){

        for(int j=0;j<size;j++){

            longArrayX[i][j] = i+j;

        }

    }

}

void SerialMatrixMultiply::setZero(long \*\* & longArrayX) {

    myMalloc(longArrayX);

    for(int i=0;i<size;i++){

        for(int j=0;j<size;j++){

            longArrayX[i][j] = 0;

        }

    }

}

SerialMatrixMultiply::SerialMatrixMultiply(long ssize):size(ssize){

    //第一步初始化数组 动态申请存储空间  其中A 和 B 数组初始化数字 其他

     setArray(longArrayA);

     setArray(longArrayB);

     setZero(longArrayC);

}

**// 矩阵相乘的核心代码 3层for循环**

**void SerialMatrixMultiply::multiply() {**

**for(long i=0;i<size;i++){**

**for(long j=0;j<size;j++){**

**for(long k=0;k<size;k++){**

**longArrayC[i][j] += longArrayA[i][k] \* longArrayB[k][j];**

**}**

**}**

**}**

**}**

void SerialMatrixMultiply::print(long \*\* longArrayX) {

    for(long i=0;i<size;i++){

        for(long j=0;j<size;j++){

            cout<<longArrayX[i][j]<<" ";

        }

        cout<<endl;

    }

}

void SerialMatrixMultiply::printCArray() {

    print(longArrayC);

}

##### Main.cpp 文件

#include "SerialMatrixMultiply.h"

#include <time.h>

#include <iostream>

int a = 500;

using namespace std;

int main(int argc,char \* argv[]){

    if(argc >1){

        int a = atoi(argv[1]); //读取执行时参数 并把它转换为int值 这个值代表矩阵大小 size \* size 大小的两个矩阵相乘

        cout<<"已输入参数 矩阵规模为 "<<a<<"\*"<<a<<endl;   // 把size打印出来

    }else{

        cout<<"未输入参数!!!!!!,默认规模是 "<<a<<"\*"<<a<<endl;

    }

    double seconds;  //定义double类型的秒数  用于串行记录执行矩阵相乘前后的时间差

    SerialMatrixMultiply \*m = new SerialMatrixMultiply(a);  //新建一个类 将a 也就是矩阵的规模填进去

    clock\_t begin\_time = clock();   //记录开始的时间

    m->multiply();  //执行矩阵的乘法

    clock\_t end\_time = clock();  //记录结束的时间

    seconds = ((double)end\_time - begin\_time) / CLOCKS\_PER\_SEC; //这个CLOCKS\_PER\_SEC 在不同的操作系统的值不一样 最终算出来的单位是秒

    cout<<"cost time "<<seconds<<" seconds"<<endl; //将最终的消耗时间进行打印

}

### 2.3 并行程序

#### 2.3.1基本思路和简单介绍

将结果矩阵的每一个元素当做并行程序执行的一个基本单元，取出A矩阵的一行和B矩阵的一列，将其中对应位置的元素相乘之后相加，得到的值就是结果矩阵的对应元素的值。

#### 2.3.2 源代码展示

#include <cstdio>

#include <cstdlib>

#include <cuda\_runtime.h>

#include <iostream>

int arrayScale = 2000; //设置矩阵规模 全局变量

int arrayScale\_square = arrayScale \* arrayScale;  //算出矩阵规模的平方 之后的程序会用到

#define size 2  //这个值是更改一个块中有多少个线程的  我设置的是二维的thread排布 10 \*10 为100 < 1024 因为老师给的数据都是10的倍数 所以设置10很合适

using namespace std;

\_\_global\_\_ void MatMul(int\* M, int\* N, int\* P, int scale)  //真正的核心函数 传入显存中的A B数组 result数组 和 数组规模

{

    //其实并行程序设计的目的就是同时计算 如果你的数组规模是10 \* 10  那么需要用到100个线程计算result矩阵的每一个值

    //所以并行程序的核心是定位到这100个线程 (多维降维到二维或者一维) 然后把计算后的信息存入到显存中

    int Col = blockIdx.x \* blockDim.x + threadIdx.x; // cloumn 这里是将4维 降维到 2维  去除block的边框就做到了 这一行是定位到那一列

    int Row = blockIdx.y \* blockDim.y + threadIdx.y; // row   这一行是定位到哪一个行

    float elem1 = 0.0, elem2 = 0.0, value = 0.0;

    for (int i = 0; i < scale; i++)

    {

        elem1 = M[Row \* scale + i];//取M矩阵的一行

        elem2 = N[i \* scale + Col];//取N矩阵的一列

        value += elem1 \* elem2;//求和

    }

    P[ Row \* scale + Col] = value;

}

int main(int argc,char \* argv[])

{

    if(argc > 1){

        int hhh = atoi(argv[1]); //读取执行时参数 并把它转换为int值 这个值代表矩阵大小 size \* size 大小的两个矩阵相乘

        arrayScale = hhh;

        arrayScale\_square = arrayScale \* arrayScale;

        cout<<"已输入参数， 矩阵规模为"<<arrayScale<<" \* "<<arrayScale<<endl;

    }else{

        cout<<"未输入参数！！！ 默认矩阵规模为"<<arrayScale<<" \* "<<arrayScale<<endl;

    }

    int \*intArrayA = new int[arrayScale\_square];   // A矩阵

    int \*intArrayB = new int[arrayScale\_square];   // B矩阵

    int \*intArrayResult = new int[arrayScale\_square];  // 结果矩阵

    int \*gpuMappingIntArrayA,\*gpuMappingIntArrayB,\*gpuMappingIntArrayResult;  //显存映射矩阵

    dim3 blocksPerGrid(arrayScale/size,arrayScale/size);  // grid中block排布方式

    dim3 threadsPerBock(size,size);  // block中thread的排布方式

    cudaEvent\_t start,stop;  // 记录cuda的运行时间

    float elapsedTime = 0;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    //cuda中申请矩阵A B和结果矩阵的空间

    cudaMalloc((void\*\*)&gpuMappingIntArrayA,arrayScale\_square \* sizeof(int));

    cudaMalloc((void\*\*)&gpuMappingIntArrayB,arrayScale\_square \* sizeof(int));

    cudaMalloc((void\*\*)&gpuMappingIntArrayResult,arrayScale\_square \* sizeof(int));

    //初始化 A B数组

    for(int i = 0;i < arrayScale;i++)

    {

        for(int j = 0;j < arrayScale;j++)

        {

            intArrayA[i\*arrayScale + j] = 1;

            intArrayB[i\*arrayScale + j] = 1;

        }

    }

    //数据拷贝，主机到设备  将内存中的 A B 数组数据拷贝到 显存中的A B数组中去

    cudaMemcpy(gpuMappingIntArrayA,intArrayA,arrayScale\_square \* sizeof(int),cudaMemcpyHostToDevice);

    cudaMemcpy(gpuMappingIntArrayB,intArrayB,arrayScale\_square \* sizeof(int),cudaMemcpyHostToDevice);

    cudaEventRecord(start,0);

    // 执行核函数 计算结果数组的每一个值

    MatMul<<<blocksPerGrid,threadsPerBock>>>(gpuMappingIntArrayA,gpuMappingIntArrayB,gpuMappingIntArrayResult,arrayScale);//调用核函数

    cudaThreadSynchronize();

    cudaEventRecord(stop,0);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime,start,stop);

    // 将结果数组的每一个值拷贝回内存

    cudaMemcpy(intArrayResult,gpuMappingIntArrayResult,arrayScale\_square \* sizeof(int),cudaMemcpyDeviceToHost);

    // 输出执行cuda执行时间

    printf("cost time : %f ms $$$$ %f s \n",elapsedTime,elapsedTime/1000);

    //释放设备内存

    cudaFree(gpuMappingIntArrayA);

    cudaFree(gpuMappingIntArrayB);

    cudaFree(gpuMappingIntArrayResult);

    free(intArrayA);

    free(intArrayB);

    free(intArrayResult);

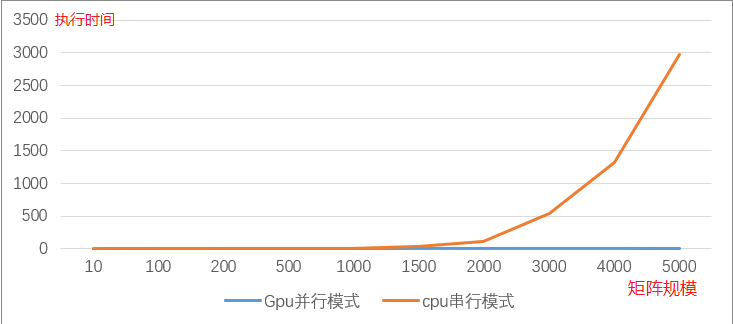
    return 0;

}

### 2.4 串行和并行程序的时间对比

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 矩阵规模 | 10 | 100 | 200 | 500 | 1000 | 1500 | 2000 | 3000 | 4000 | 5000 |
| 串行方式 | 0 | 0.01 | 0.1 | 1.32 | 9.81 | 33.6 | 115.48 | 542.6 | 1330.1 | 2972.4 |
| 并行模式 | 0.000238 | 0.000405 | 0.003258 | 0.016569 | 0.111984 | 0.338095 | 0.600478 | 2.022536 | 6.780954 | 9.633363 |

**图表数据**



**由上方数据和图表可以看出，cuda项目对于矩阵相乘这个项目的优化效果是效果拔群的，尤其是当矩阵规模很大的时候，对于串行程序来说不可能完成的事情，并行程序却能很短时间内解决。**

## 三、MPICH编程实践的作业

### 3.1 项目要求

### 3.2积分方法方法求PI （第一种方法）

#### 3.2.1 openMP

#include <iostream>

#include <cstdio>

#include <time.h>

#include <omp.h>

#include <stdlib.h>

int threadNum =56;   // 全局变量 记录并行的线程数目

using namespace std;

int main(int argc,char \* argv[]){

    if(argc >1){

        int threadNum = atoi(argv[1]);

        cout<<"already input value the thread num is "<<threadNum<<endl;

    }else{

        cout<<"no input value found!!!!!!,the default "<<threadNum<<endl;

    }

    double begin\_time = omp\_get\_wtime();   //记录开始的时间

    double seconds;  // 记录花费时间

    double dx = 10e-8;  // 积分中 dx的值

    double sum = 0;   // 积分中每个小矩形面积的累加值

    int n = 1/dx;   // 将0 到 1 分成多少份

    sum = 0;

    omp\_set\_num\_threads(threadNum);  // 设置并行的线程数

    #pragma omp  parallel  for reduction(+:sum)  // 告诉编译器要进行并行了 将每个i平均分配到合适的核心上边

    // reduction 是防止 sum值这个临界资源发生数据丢失

    for(int i=0;i<n;i++){

        double xx = i\*dx ;

        sum +=  1/(1 + xx\*xx) \* dx; // 将每个小矩阵的面积累加起来得到最终的值就是积分的值

    }

    double end\_time = omp\_get\_wtime();  //记录结束的时间

    seconds = end\_time - begin\_time;

    cout<<"cost time "<<seconds<<" seconds"<<endl; //将最终的消耗时间进行打印

    printf("%.16lf\n",4\*sum);  // 输出pi的值

}

#### 3.2.2 MPI

#include<stdio.h>

#include<mpi.h>

#include <iostream>

using namespace  std;

int main(int argc,char \*argv[]){

    int rank,num\_procs;  // rank 是当前运行线程编号  proc\_len 是进程名字长度  num\_procs 是进程数量

    double startTime = 0,endTime = 0; //记录时间

    double dx = 10e-8;

    double n = 1/dx;

    double pi = 0;

    MPI\_Init(&argc,&argv);

    //用来获取正在调用进程的通信子中的进程号的函数

    MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

    //用来得到通信子的进程数的函数

    MPI\_Comm\_size(MPI\_COMM\_WORLD,&num\_procs);

    //mpi接口获取进程名 int MPI\_Get\_processor\_name(char \*name,int \*resultlen)

//    printf("current process is %d,total %d processes\n",rank,num\_procs);

    if(rank == 0){  // 开始计时

        startTime = MPI\_Wtime();

        printf("total %d processes\n",num\_procs);

    }

    double partSum = 0;

    double partPi = 0;

    // 每一个进程 比如 有10个核  那么 0 10 20 30 .. 的sum归 0 号进程所算

    for(int i = rank;i<n;i+=num\_procs){

        double xx =dx \* i;

        partSum += 4.0/(1.0+xx\*xx) ;

    }

    partPi = partSum\*dx;   // 每个进程计算一部分pi的值

    // 最终使用 reduce的sum函数 将各个进程中算出的值 加起来 得到最终的pi值

    MPI\_Reduce(&partPi,&pi,1,MPI\_DOUBLE,MPI\_SUM,0,MPI\_COMM\_WORLD);

    if(rank == 0){ // 打印pi 然后计算时间 输出执行时间

        printf("PI is %.20f\n",pi);

        endTime = MPI\_Wtime();

        printf("Time cost :%f s \n",endTime-startTime);

    }

    MPI\_Finalize();

    return 0;

}

#### 3.2.3 openMP + MPI

#include<stdio.h>

#include<mpi.h>

#include <iostream>

#include <omp.h>

int threadNum = 2;

using namespace  std;

int main(int argc,char \*argv[]){

    int rank,num\_procs;  // rank 是当前运行线程编号  proc\_len 是进程名字长度  num\_procs 是进程数量

    double startTime = 0,endTime = 0; //记录时间

    double dx = 10e-8;

    double n = 1/dx;

    int nn = n;

    int i ;

    double pi = 0;

    MPI\_Init(&argc,&argv);

    //用来获取正在调用进程的通信子中的进程号的函数

    MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

    //用来得到通信子的进程数的函数

    MPI\_Comm\_size(MPI\_COMM\_WORLD,&num\_procs);

    //mpi接口获取进程名 int MPI\_Get\_processor\_name(char \*name,int \*resultlen)

//    printf("current process is %d,total %d processes\n",rank,num\_procs);

    if(rank == 0){  // 开始计时

        startTime = MPI\_Wtime();

        printf("total %d processes\n",num\_procs);

    }

    double partSum = 0;

    double partPi = 0;

    // 每一个进程 比如 有10个核  那么 0 10 20 30 .. 的sum归 0 号进程所算

    omp\_set\_num\_threads(threadNum); // 设置一个进程并行的线程数

    // 一个进程中中线程再次进行并行 两者相乘是占用的所有内核数

    #pragma omp  parallel  for  reduction(+:partSum)

    for(i = rank;i<nn;i+=num\_procs){

        double xx =dx \* i;

        partSum  += 4.0/(1.0+xx\*xx) ;

    }

    partPi = partSum\*dx;

    // 最终使用 reduce的sum函数 将各个进程中算出的值 加起来 得到最终的pi值

    MPI\_Reduce(&partPi,&pi,1,MPI\_DOUBLE,MPI\_SUM,0,MPI\_COMM\_WORLD);

    if(rank == 0){ // 打印pi 然后计算时间 输出执行时间

        printf("PI is %.20f\n",pi);

        endTime = MPI\_Wtime();

        printf("Time cost :%f s \n",endTime-startTime);

    }

    MPI\_Finalize();

    return 0;

}

### 3.3 蒙特卡洛方法求pi (第二种方法)

#### 3.3.1 openMP

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <omp.h>

#include <cstdio>

using namespace std;

int threadNum = 6;

int TestNum = 100000000;

int main(int argc,char \* argv[]){

    if(argc >1){

        int threadNum = atoi(argv[1]);

        cout<<"already input value the thread num is "<<threadNum<<endl;

    }else{

        cout<<"no input value found!!!!!!,the default "<<threadNum<<endl;

    }

    double begin\_time = omp\_get\_wtime();   //记录开始的时间

    double seconds;

    srand(time(0));

    int inside = 0;

    omp\_set\_num\_threads(threadNum);    // 设置并行的线程数有 threadNum 个

    #pragma omp  parallel  for reduction(+:inside)

    for(int i = 0; i < TestNum; ++i) {

        // 随机获得 x 和 y 的值 均小于1

        double x = (double) rand() / RAND\_MAX;

        double y = (double) rand() / RAND\_MAX;

        // 如果距离原点距离小于=1 那么这个点就算在在圆中

        if(x \* x + y \* y <= 1.0) ++inside;

    }

    double pi = 4.0 \* inside / TestNum;

    printf("test cases is %d\n",TestNum);

    printf("PI = %.10lf\n",pi);

    double end\_time = omp\_get\_wtime();  //记录结束的时间

    seconds = end\_time - begin\_time;

    cout<<"cost time "<<seconds<<" seconds"<<endl; //将最终的消耗时间进行打印

    return 0;

}

#### 3.3.2 MPI

#include<cstdio>

#include <ctime>

#include<mpi.h>

#include <iostream>

#include <cstdlib>

using namespace  std;

int main(int argc,char \*argv[]){

    int testNum = 100000000;  //TestNum 是测试数量

    int insideCount = 0;   // 在圆中的次数

    int rank,num\_procs;  // rank 是当前运行线程编号  proc\_len 是进程名字长度  num\_procs 是进程数量

    double startTime = 0,endTime = 0; //记录时间

    srand(time(0));

    MPI\_Init(&argc,&argv);

    //用来获取正在调用进程的通信子中的进程号的函数

    MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

    //用来得到通信子的进程数的函数

    MPI\_Comm\_size(MPI\_COMM\_WORLD,&num\_procs);

    //mpi接口获取进程名 int MPI\_Get\_processor\_name(char \*name,int \*resultlen)

//    printf("current process is %d,total %d processes\n",rank,num\_procs);

    if(rank == 0){  // 开始计时

        startTime = MPI\_Wtime();

        printf("total %d processes\n",num\_procs);

    }

    int partInsideCount = 0;

    // 将数据平均分配给每一个进程 然后用每一个进程只执行自己的投点实验

    // 比如有 1亿数据  10个进程 那么一个进程就分到 一千万个随机点

    for(int i = rank; i < testNum; i+=num\_procs) {

        double x = (double) rand() / RAND\_MAX;

        double y = (double) rand() / RAND\_MAX;

        if(x \* x + y \* y <= 1.0) ++partInsideCount;

    }

    // 将每个进程算的数 汇总起来 得到 所有落在圆中的点的个数

    MPI\_Reduce(&partInsideCount,&insideCount,1,MPI\_INT,MPI\_SUM,0,MPI\_COMM\_WORLD);

    if(rank == 0){ // 打印pi 然后计算时间 输出执行时间

        printf("all test cases is %.d\n",testNum);

        printf("insideCount is %.d\n",insideCount);

        printf("pi is %lf\n", 4.0 \* insideCount / testNum);

        endTime = MPI\_Wtime();

        printf("Time cost :%f s \n",endTime-startTime);

    }

    MPI\_Finalize();

    return 0;

}

#### 3.3.3 openMP + MPI

#include<cstdio>

#include <ctime>

#include <iostream>

#include <cstdlib>

#include <omp.h>

#include <mpi.h>

using namespace  std;

int main(int argc,char \*argv[]){

    long long testNum = 100000000;  //TestNum 是测试数量

    long long insideCount = 0;   // 在圆中的次数

    int rank,num\_procs;  // rank 是当前运行线程编号  proc\_len 是进程名字长度  num\_procs 是进程数量

    double startTime = 0,endTime = 0; //记录时间

    srand(time(0));

    MPI\_Init(&argc,&argv);

    //用来获取正在调用进程的通信子中的进程号的函数

    MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

    //用来得到通信子的进程数的函数

    MPI\_Comm\_size(MPI\_COMM\_WORLD,&num\_procs);

    //mpi接口获取进程名 int MPI\_Get\_processor\_name(char \*name,int \*resultlen)

//    printf("current process is %d,total %d processes\n",rank,num\_procs);

    if(rank == 0){  // 开始计时

        startTime = MPI\_Wtime();

        printf("total %d processes\n",num\_procs);

    }

    long long partInsideCount = 0;

    omp\_set\_num\_threads(3); // 设置并行的线程数有 3 个

    #pragma omp  parallel  for reduction(+:partInsideCount) // 告诉编译器要进行并行了 将每个i平均分配到合适的核心上边

    for(long long i = rank; i < testNum; i+=num\_procs) {

        double x = (double) rand() / RAND\_MAX;

        double y = (double) rand() / RAND\_MAX;

    //        printf("rank = %d , x = %llf  y = %llf",rank,x,y);

        if(x \* x + y \* y <= 1.0) ++partInsideCount;

    }

    // 将每个进程算的数 汇总起来 得到 所有落在圆中的点的个数

    MPI\_Reduce(&partInsideCount,&insideCount,1,MPI\_LONG\_LONG,MPI\_SUM,0,MPI\_COMM\_WORLD);

    if(rank == 0){ // 打印pi 然后计算时间 输出执行时间

        printf("all test cases is %lld\n",testNum);

        printf("insideCount is %.lld\n",insideCount);

        printf("pi is %lf\n", 4.0 \* insideCount / testNum);

        endTime = MPI\_Wtime();

        printf("Time cost :%f s \n",endTime-startTime);

    }

    MPI\_Finalize();

    return 0;

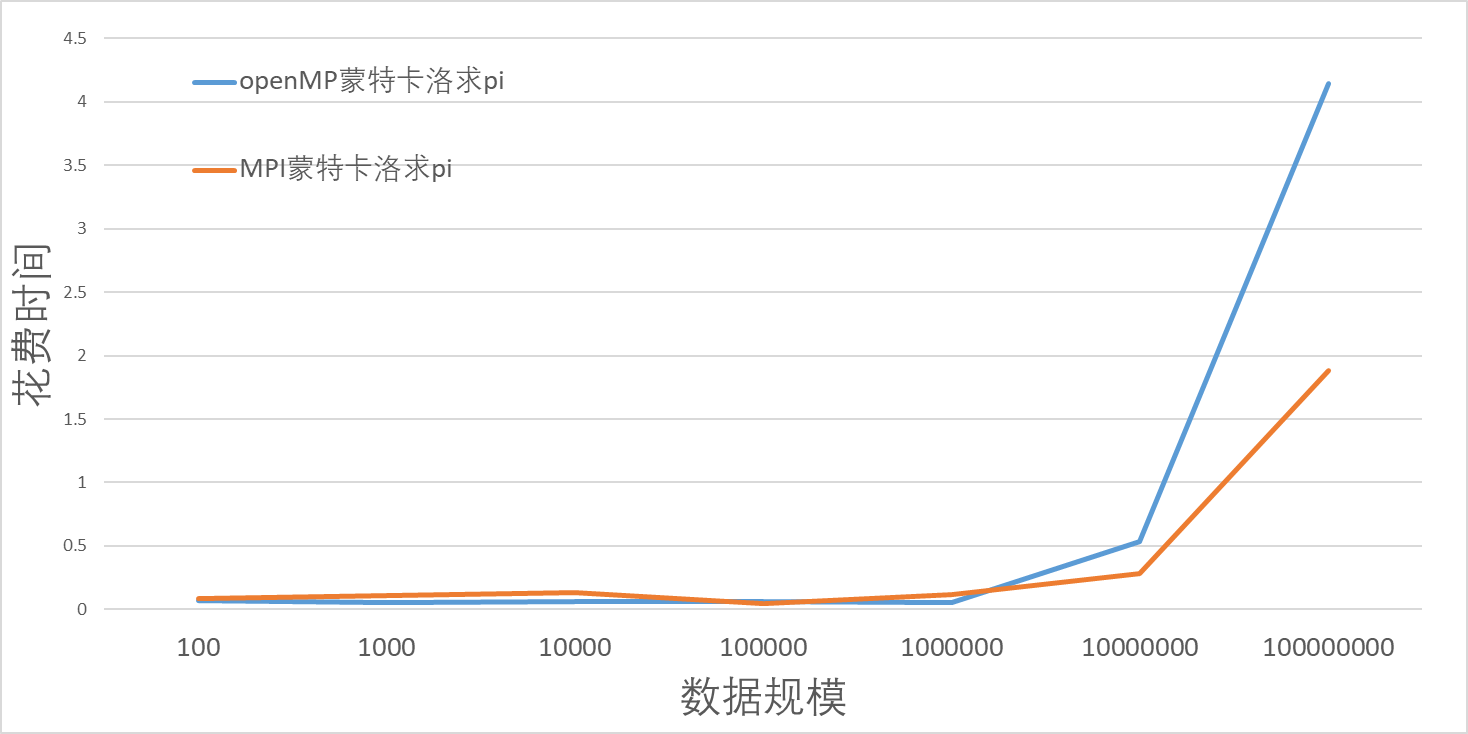
}

### 3.4数据展示

#### 不同数据规模测试

##### 积分法

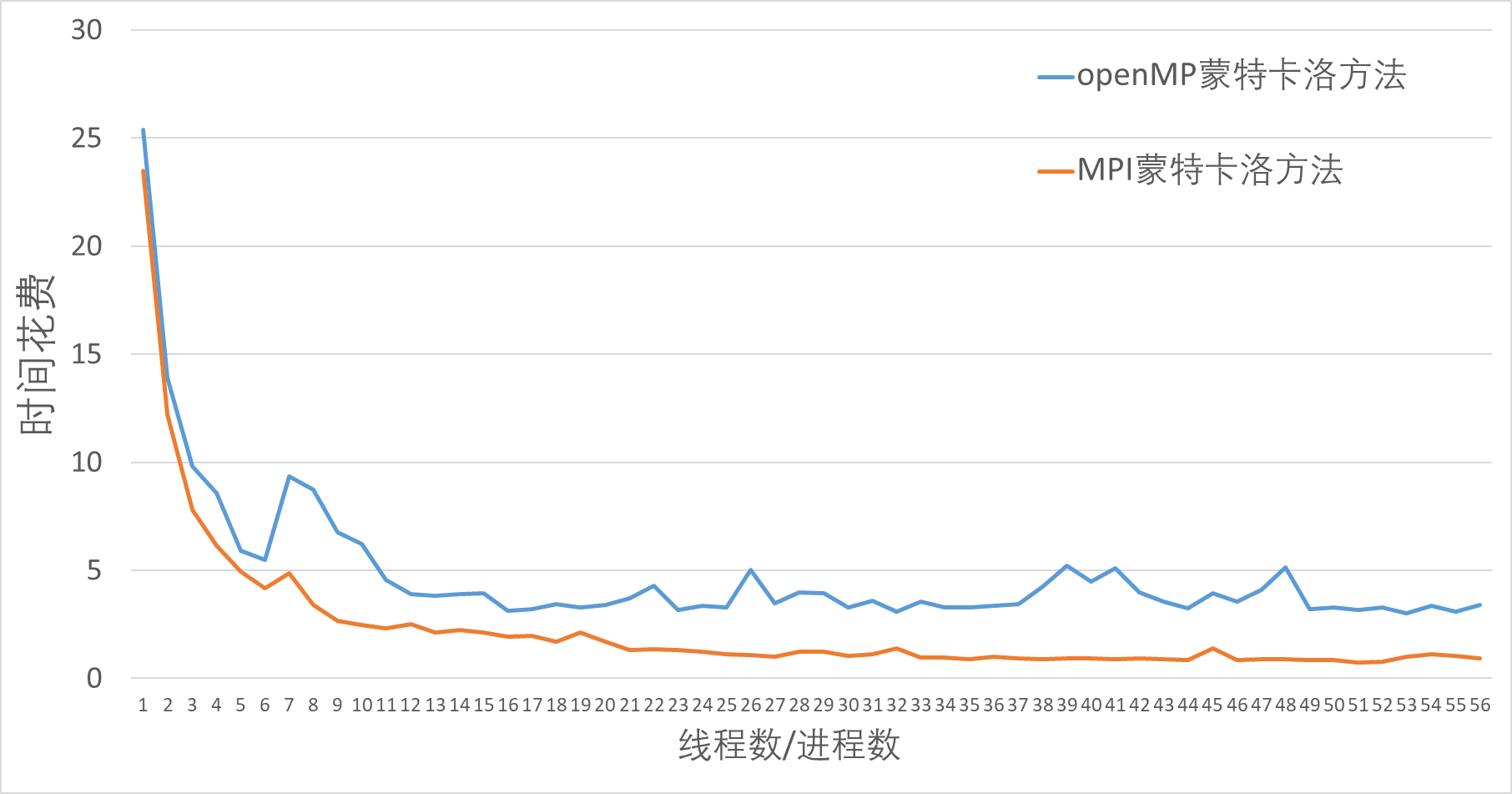
##### 蒙特卡洛法



#### 不同数据分配方式的运行时间

##### 积分法

##### 蒙特卡洛法



##### 混合方法

默认都是两个线程 进程数从1到28

