并行计算课程
实 验 报 告

报告名称：	多线程计算矩阵幂
姓　　名：	陈秋澄
学　　号：	3022244290
联系电话：	15041259366
电子邮箱：	3133242711@qq.com
填写日期：	2024 年 3 月 30 日

2024 年制

摘要

姓名 陈秋澄 学号 3022244290

一、实验名称与内容

实验名称：多线程计算矩阵幂

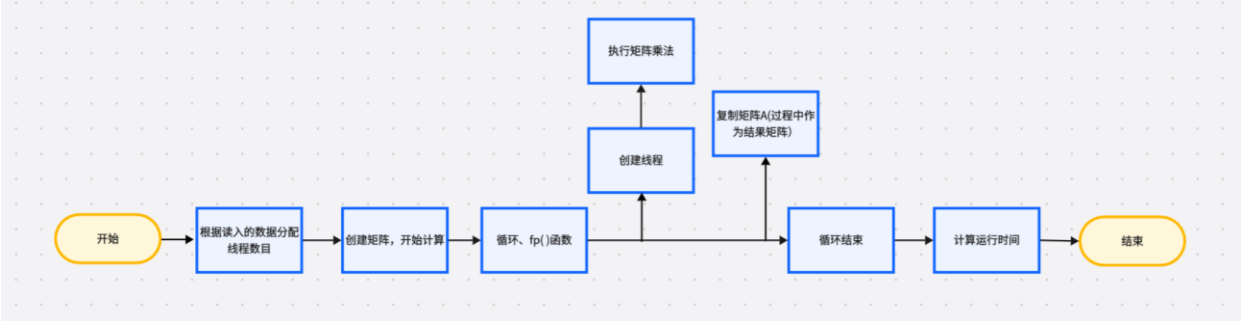
实验内容：本实验利用矩阵分块等思想，用 Pthread 并行化实现对矩阵幂的求解

二、实验环境的配置参数

1. 计算集群
- 国家超级计算天津中心提供
- 国产飞腾处理器

2. CPU 型号：国产自主 FT2000+@2.30GHz 56cores
3. 节点数：5000 个
4. 内存：128GB
5. 网络：天河自主高速互连网络：400Gb/s

三、方案设计



四、实现方法

利用多线程求解 N*N 矩阵的 power 次幂，将其分解为多个矩阵相乘，每两个矩阵相乘时将第一个矩阵按行分为 n 块，每块用一个线程进行计算，再将每块计算出的结果都存入同一结果矩阵之中，再将此结果矩阵作为前一个矩阵，利用随机数生成的下一个矩阵，再进行两个矩阵的相乘，如此循环 power-2 次，即可求得矩阵的幂。

其中，多线程计算阶段：

我们先定义一个名为 func 的线程函数，用于执行矩阵乘法的一部分计算。在 fp() 函数中，首先计算每个线程需要处理的行数（即“块大小”L）。遍历矩阵行数，每次分配 L 行给一个新线程，传递起始行索引作为参数。使用 pthread_create 创建线程，每个线程调用 func 函数处理其分配的子矩阵的乘法。使用 pthread_join 等待所有线程完成计算。

五、结果分析

1. 多线程计算能够提高程序的性能，加速比随着线程数的增加而增加，但效率随着线程数的增加而下降。
2. 在数据规模较小时，多线程计算的性能优势不明显，反而可能会降低程序的运行效率。因为线程创建、线程同步等开销会导致程序效率下降。
3. 在数据规模较大时，多线程计算的加速比相对较高，可以有效地提高程序的运行效率。

一、实验内容概述

1. 算法概述

本实验利用矩阵分块等思想，用 Pthread 并行化实现对矩阵幂的求解。

2. 并行计算环境

通过远程登录方式链接集群，由客户端传输文件到集群文件夹运行。
Linux 系统下采用 pthread 多线程进行并行化计算。
并行计算环境设置参数为一个服务器，16 个内核。

3. 数据分析及并行化方法

利用多线程求解 N*N 矩阵的 power 次幂，将其分解为多个矩阵相乘，每两个矩阵相乘时将第一个矩阵按行分为 n 块，每块用一个线程进行计算，再将每块计算出的结果都存入同一结果矩阵之中，再将此结果矩阵作为前一个矩阵，利用随机数生成的下一个矩阵，再进行两个矩阵的相乘，如此循环 power-2 次，即可求得矩阵的幂。

其中，多线程计算阶段：

我们先定义一个名为 func 的线程函数，用于执行矩阵乘法的一部分计算。在 fp()函数中，首先计算每个线程需要处理的行数（即“块大小”L）。遍历矩阵行数，每次分配 L 行给一个新线程，传递起始行索引作为参数。使用 pthread_create 创建线程，每个线程调用 func 函数处理其分配的子矩阵的乘法。使用 pthread_join 等待所有线程完成计算。

二、并行算法分析设计

第一，在初始化阶段：
首先，读取命令行参数（Power 代表矩阵 A 被乘幂次数，N 代表矩阵的维度，NUM_THREADS 代表线程数量）。
其次，动态分配内存来存储三个二维矩阵（matrixA、matrixB 和 result）。
然后，调用 makeRandomMatrix_A()函数生成一个随机的 N×N 矩阵 A。

第二，多线程计算阶段：
首先，定义一个名为 func 的线程函数，用于执行矩阵乘法的一部分计算。
其次，在 fp()函数中，首先计算每个线程需要处理的行数（即“块大小”L）。
再次，遍历矩阵行数，每次分配 L 行给一个新线程，传递起始行索引作为参数。
然后，使用 pthread_create 创建线程，每个线程调用 func 函数处理其分配的子矩阵的乘法。
最后，使用 pthread_join 等待所有线程完成计算。
核心代码如下：

```
//子线程函数
void *func(void *arg)
{
    int s=*(int *)arg;      //接收传入的参数（此线程从哪一行开始计算）
```

```
    int t=s+L;              //线程算到哪一行为止
    for(int i=s;i<t;i++)
        for(int j=0;j<N;j++)
            for(int k=0;k<N;k++)
                result[i][j]+=matrixA[i][k]*matrixB[k][j];

}

//串多线程函数
void fp(){
    int i;
    int j = 0;
    int t[NUM_THREADS];   //传参索引
    L = N / NUM_THREADS;  //按设置的线程数分配工作块（单个线程所要计算的行数L）

    for(i=0;i<N;i+=L)
    {
        t[j] = i;
        if (pthread_create(&tids[j], NULL, func, (void *)&(t[j]))) //产生线程，去完成矩阵相乘的部分工作量
        {
            perror("pthread_create");
            exit(1);
        }
        j++;
    }

    for(i=0;i<NUM_THREADS;i++)
        pthread_join(tids[i],NULL); //等所有的子线程计算结束
}
```

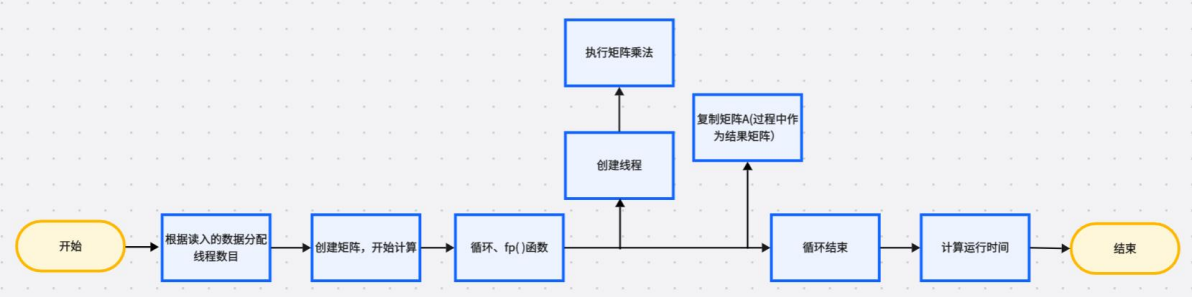
第三，矩阵乘幂迭代：
对于幂次计算，主循环在 Power-2 次内不断重复上述多线程计算过程，并在每次迭代后将上一步的结果矩阵替换到矩阵 A 的位置，以便下一次乘法运算。
核心代码如下：

```
for(int i=1;i<=Power-2;i++)
    {
        fp();    //多线程
        for(int j=0;j<N;j++)
        {
            for(int k=0;k<N;k++)
            {
                matrixA[j][k]=result[j][k];
```

```
                result[j][k]=0;
            }

        }
    }
    fp();
```

下面是主函数的流程图：



实验代码如下：

```
#include<stdio.h>
#include<time.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include<memory.h>
#include<iostream>
using namespace std;
int Power;
int N;
int **matrixA,**matrixB,**result;
void *func(void *arg);

int NUM_THREADS; //线程数
pthread_t *tids;  //线程
int L;                      //每个线程计算的块大小

void makeRandomMatrix_A()  //生成矩阵
{
    srand(time(NULL));
    int i, j;
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            matrixA[i][j] = matrixB[i][j] = rand() % 10 + 1;
            cout<<matrixA[i][j]<<" ";
```

```
        }
        cout<<endl;
    }
}
//子线程函数
void *func(void *arg)
{
    int s=*(int *)arg;      //接收传入的参数（此线程从哪一行开始计算）
    int t=s+L;              //线程算到哪一行为止
    for(int i=s;i<t;i++)
        for(int j=0;j<N;j++)
            for(int k=0;k<N;k++)
                result[i][j]+=matrixA[i][k]*matrixB[k][j];
}

//串多线程函数
void fp(){
    int i;
    int j = 0;
    int t[NUM_THREADS];   //传参索引
    L = N / NUM_THREADS;  //按设置的线程数分配工作块（单个线程所要计算的行数L）

    for(i=0;i<N;i+=L)
    {
        t[j] = i;
        if (pthread_create(&tids[j], NULL, func, (void *)&(t[j]))) //产生线程，去完成矩阵相乘的部分工作量
        {
            perror("pthread_create");
            exit(1);
        }
        j++;
    }

    for(i=0;i<NUM_THREADS;i++)
        pthread_join(tids[i],NULL); //等所有的子线程计算结束


}
int main(int argc,char *argv[])
{
    Power=atoi(argv[1]);//第一个参数传幂次
    N=atoi(argv[2]);//第二个参数传矩阵行数和列数
```

```
    NUM_THREADS=atoi(argv[3]);//第三个参数传线程数
    cout<<"Power="<<Power<<endl;
    printf("MN(%d*%d)...\n",N,N);
    //动态分配
    matrixA=new int* [N];
    matrixB=new int* [N];
    result=new int* [N];
    for(int i=0;i<N;i++)
    {
        matrixA[i]=new int[N];
        matrixB[i]=new int[N];
        result[i]=new int[N];
    }
    tids=new pthread_t [NUM_THREADS];
    makeRandomMatrix_A();       //用随机数产生待相乘的矩阵，并存入文件中
    //从文件中读出数据赋给 matrixA
    printf("Makeing matrix(%d*%d)...\n",N,N);


    //多线程计算
    clock_t start1=clock();     //开始计时

    for(int i=1;i<=Power-2;i++)
    {
        fp();    //多线程
        for(int j=0;j<N;j++)
        {
            for(int k=0;k<N;k++)
            {
                matrixA[j][k]=result[j][k];
                result[j][k]=0;
            }

        }

    }
    fp();
    clock_t finish1=clock();    //结束计算
    printf("%d threads --- Running time=%f s\n",
NUM_THREADS,(double)(finish1 - start1) / CLOCKS_PER_SEC);

    for(int i=0;i<N;i++)
    {
        for(int j=0;j<N;j++)
        {
```

```
            cout<<result[i][j]<<" ";
        }
        cout<<endl;
    }
    pthread_exit(NULL);
    return 0;
}
```

三、实验数据分析

1. 实验环境

（1） 计算集群

由国家超级计算天津中心提供的国产飞腾处理器。

（2） 计算节点配置

CPU 型号：国产自主 FT2000+@2.30GHz 56cores

节点数：5000 个

内存：128GB

（3） 互联网络参数

天河自主高速互连网络：400Gb/s

单核理论性能（双精度）：9.2GFlops

单节点理论性能（双精度）：588.8GFlops

（4） 编译环境

GCC 9.3.0; OpenMPI 4.1

（5） 作业管理系统

SLURM 20.11.9

2. 实验数据综合分析

我分别测试了线程数为 1（串行）、2、4、8、16 情况下的运行时间，计算得到加速比与效率。

计算公式：

a) 加速比 = 串行运行时间/并行运行时间

b) 效率 = 加速比/并行处理器数

我们运用样例测试程序：

例如：

```
time yhrun -p thcp1 -n 1 -c 8 try.o 2 16 8&>>run2.log
time yhrun -p thcp1 -n 1 -c 16 try.o 2 16 16&>>run2.log
```

我们要明晰以下三个概念：

real：指的是从开始到结束所花费的时间。比如进程在等待 I/O 完成，这个阻塞时间也会被计算在内。

user：指的是进程在用户态（User Mode）所花费的时间，只统计本进程所使用的时间，注意是指多核。

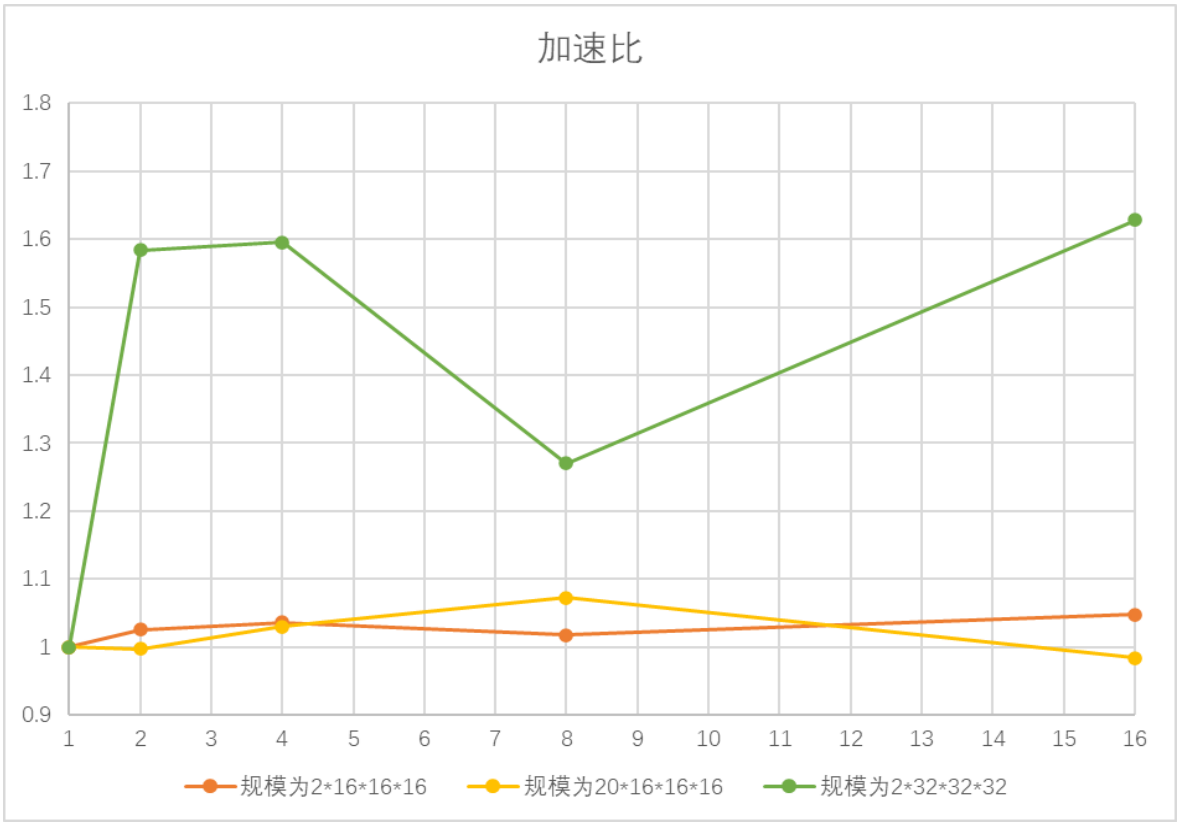
sys：指的是进程在核心态（Kernel Mode）花费的 CPU 时间量，指的是内核中的系统调用所花费的时间，只统计本进程所使用的时间。

（1）数据图表分析

相关图表如下：

核数	线程数	规模	real/s	user/s	sys/s	加速比	效率
1	1	2*16*16*16	0.974	0.311	0.051	1	1
2	2	2*16*16*16	0.949	0.29	0.068	1.02634	0.51317
4	4	2*16*16*16	0.94	0.313	0.045	1.03617	0.25904
8	8	2*16*16*16	0.957	0.305	0.058	1.01776	0.12722
16	16	2*16*16*16	0.929	0.291	0.07	1.04844	0.06553
1	1	20*16*16*16	0.957	0.292	0.079	1	1
2	2	20*16*16*16	0.959	0.301	0.066	0.99791	0.49896
4	4	20*16*16*16	0.929	0.306	0.052	1.03014	0.25753
8	8	20*16*16*16	0.892	0.283	0.08	1.07287	0.13411
16	16	20*16*16*16	0.972	0.281	0.077	0.98457	0.06154
1	1	2*32*32*32	1.498	0.322	0.049	1	1
2	2	2*32*32*32	0.946	0.299	0.063	1.58351	0.79175
4	4	2*32*32*32	0.939	0.309	0.053	1.59531	0.39883
8	8	2*32*32*32	1.179	0.311	0.048	1.27057	0.15882
16	16	2*32*32*32	0.92	0.295	0.072	1.62826	0.10177

加速比图表如下：

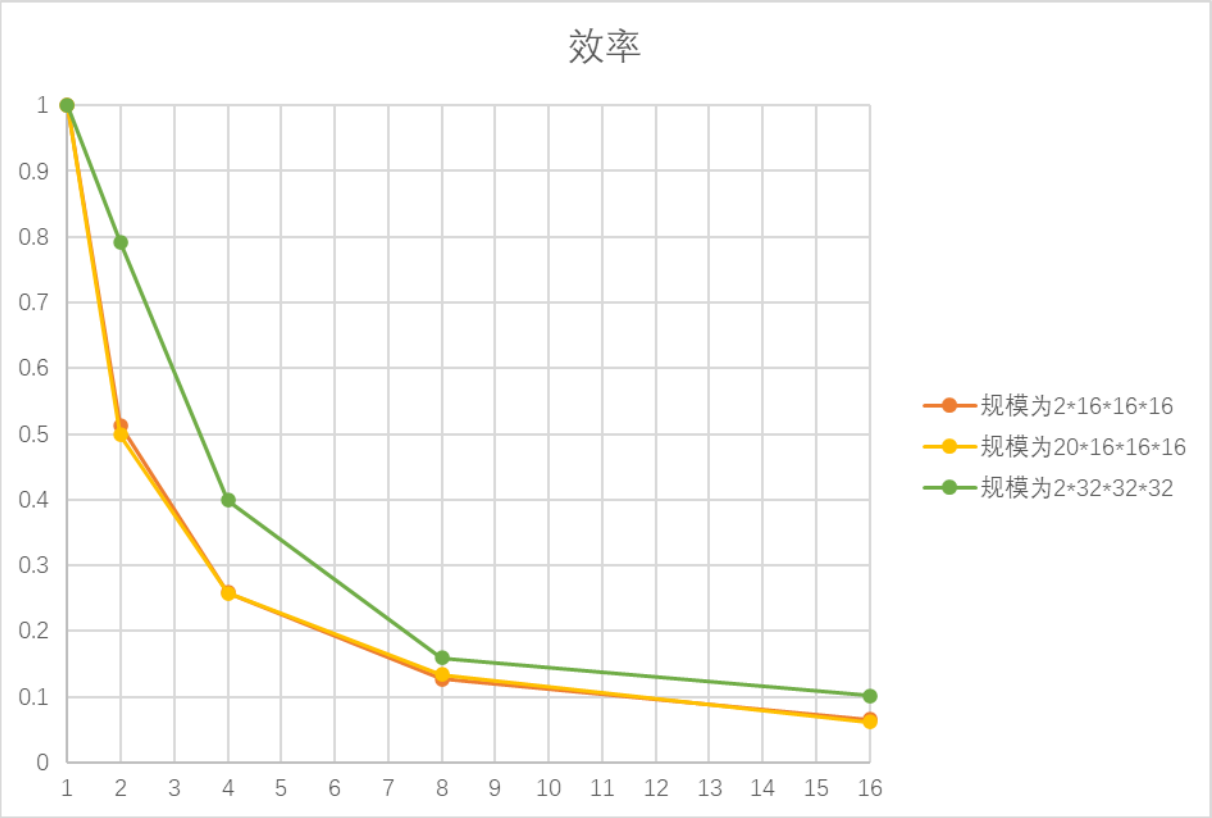


分析：

总体上看，在规模一定的情况下，核数越多，加速比通常越高，但当核数增加到一定数

量时(如规模为 20*16*16*16，核数由 8 增加到 16 时)，加速比会饱和甚至下降，因为增加核数也会带来一些额外的开销,例如通信开销、同步开销等。因此，在在进行并行计算时，需要仔细评估问题规模、核数和并行算法的复杂度，以确定最优的并行计算策略，以获得最佳的加速比和效率。
在核数相同的情况下,规模越大,加速比越高 在规模相对较小时,加速比的提升较小甚至近似相等，但当规模足够大时，核数越多，加速比会有显著提升。

效率图表如下：



分析：总体上看，在一定规模时，核数越多，效率越低。
在核数相同时，规模越大，效率近似相等或稍稍升高。
从上述两图综合分析得出，当数据规模为 2*16*16*16 时，并行程序的性能有时并没有串行程序好,加速比并没有明显提升，甚至有的数据集小于 1。
当数据规模一定时，随着线程数的增加，加速比没有大幅度的增加，甚至有时上低于其他规模条。随着线程数的增加，程序运行的效率依然下降，但下降的幅度有所减缓。

（2）实验结论

a) 多线程计算能够提高程序的性能，加速比总体上随着线程数的增加而增加，但当线程数超过一定值后，加速比反而随着线程数的增加而总体上低于低线程数的对应值。
b) 在数据规模较小时，多线程计算的性能优势不明显，反而可能会降低程序的运行效率。因为线程创建、线程同步等开销会导致程序效率下降。
c) 在数据规模较大时，多线程计算的加速比相对较高，可以有效地提高程序的运行效率。
d) 随着线程数的增加，程序的效率下降，下降幅度随着矩阵规模的增加而减缓。这是由于多线程计算会增加线程切换的开销，同时线程之间的竞争和同步也会对程序的

运行效率造成影响。

四、实验总结

之前 pthread 并行程序认识只是理论层面，通过本次实验，我亲手实现了 pthread 程序，以及像计时的操作，让我对之前学到的理论有了更深刻的理解。

1. 问题、解决与收获

（1） 对于多线程 pthread 的操作主要使用到函数 pthread_create()和 pthread_join()，pthread_create()函数进行一个线程的创建，如果在运行时单独使用，可能线程开始运行的 位置不确定，也可能主线程直接使整个进程直接结束而无法开始运行子线程。pthread_join() 函数的作用是以阻塞的方式等待指定的线程运行结束，再返回上一个线程，将已创建的线程通 过该函数进行阻塞运行，可以将线程在需要的位置使用，该函数放在创建之后的 for 循环中，可以确保每一个线程完整运行，对于域分解并行化计算，可确保每一个域的正确计算。

（2） 其次遇到的问题是 rand()函数，该函数用于生成一个随机数生成器，且所有的线程共用这个随机数生成器，每次调用 rand()都会去修改该生成器的一些参数，因此 rand()需要多线程互斥访问，线程将频繁地对临界段进行上锁和解锁，容易造成其他线程阻塞，本质上这些线程无法并行，从而使得概率法的串行时间低于并行时间。我完善程序结构及执行顺序，解决了这一问题，这提醒我们，在设计并行化程序的时候，我们对于函数的引用要确保其可并行性，保证并行程序中的任意一个步骤同其他程序都不互斥。

（3） 由于对竞争条件理解的缺失，在开始的程序实现中每个线程每个数据处理之后都直接加到了总的 result 当中，这样会导致不同线程同时修改 result，导致结果出现如溢出、计算值不匹配等错误。可以通过完善程序结构来解决这个问题。

（4） 由于传入参数的线程数和矩阵的规模不匹配，因为矩阵是按行数被划分成块，每块由不同的线程进行执行，所以传入的矩阵的 N 必须可以被线程数整除，否则可能会报段错误。

2. 并行计算方式的理解与分析

我将 Power 次幂划分为块进行计算，思路和第一次实验相同，但通过仔细阅读实验要求可知，需将矩阵内部进行划分，再将同样的步骤重复 Power 次。在对 result 矩阵进行输出测试时一度发现输出的结果为零矩阵：解决方法是在循环结束后 result 矩阵清零，即还需再加一次 fp（）函数。

在实际中并行线程数量增大时，切换线程的代价将越来越大，os 的调度算法也会逐渐复杂，线程个数增加到一定数量，反而会使得加速比下降，无法在靠近理想值的位置保持稳定。

假设现在执行 T 线程并发，我们可以理解为对于每一个线程的数据规模为 N/T，根据这些数据可以计算出当前进程的正弦值，之后将他们相加求平均值消除个别误差，实际是和理解为对 N 规模的数据域分解并发求值的结果相同。

五、课程总结

1. 课程授课方式有助于提升学习质量的方面

老师将 PPT 上传至智慧树平台，有助于我们在学习时可以进行反复观看，巩固知识点。上机实验的内容也是主要侧重于对数据的分析，对并行计算原理的理解方面，其所要求的编程任务确实很有意义。通过此次实验，我认识到了并行程序在数据规模较大算法复杂度较高的时候相较于串行程序而言效率的提高，通过作图直观的认识到了在不同环境下并行程序和串行程序性能的变化，对并行编程有了更深入的理解。

2. 不合理之处及建议

首先，课上授课的侧重点主要在于理论知识，有些概念对于初学者而言是宏观且略有模糊的，例如对于并行计算的发展历史，硬件环境演变历史及现状，或者各种高级算法的框架等，对于这些知识的理解往往比较片面，也不够深入，且在今后的学习工作中使用的几率也不大，因此只做了解即可。这部分内容可以适当减小课堂占比，空出宝贵的课堂时间，将授课的重心更偏向于应用与实践，引导养成并行计算思维方式。我也真切地希望老师可以从更基础的点出发，特别是上机实验时，可以将实验参考书编写得更详细一些，增加一些原理或可能遇到的函数等的介绍，考虑到编程基础相对薄弱的同学，让我们由浅入深地理解知识并将其付诸实践。

其次，课堂的知识内容相比于算法或性能发展现状略有滞后，例如对于一些最新技术的介绍，其实质推出或发行日期是在两年以前，授课内容日期略有延迟，最新的算法介绍跟不上，会使得在真正应用于实际时这种落后效应更加明显，希望可以及时更新授课内容，更换新的课件与材料。

附：上机实验与课程知识点分析

序号	上机实验内容	理论知识点	分析总结
1	矩阵幂计算任务分解	并行化方法：域分解	域分解时将计算域划分成不同的子域，再将各个子域分给不同的处理器，各个处理器完成对子计算域的计算，求解出子域的子结果再根据所有子结果综合得到最终结果，通过各个处理器并发执行实现并行化计算。
2	多线程程序编写	Pthread 多线程并行程序设计	主要使用到函数 pthread_create()、pthread_join()、pthread_exit()进行简单线程编写，要注重这些库函数的原型及参数类型，进行正确调用。

3	实验数据分析	并行计算的性能	并行程序个数安排不合理，对同一资源互斥访问，假共享破坏 cache 高速缓存的一致性原则使缓存失败，可能会导致并行程序比串行程序更慢。这要求我们指定合理的并行策略，优化算法逻辑，提高并行性能，达到良好的加速效果。
4	加速比、效率分析	加速比定律	影响加速比的因素包括：计算总负载量、处理器个数、处理器执行速度、任务可并行化部分和不可并行化部分的占比及其分别耗时、并行额外开销。基于 Amdahl 定律的加速比有上限，但基于 Gustafson 定律的加速比考虑数据精度可以无限增大。