并行计算课程
实 验 报 告

报告名称：	多进程计算矩阵幂
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 4 月 16 日

2024 年制

摘要

姓名 陈秋澄 学号 3022244290

一、实验名称与内容

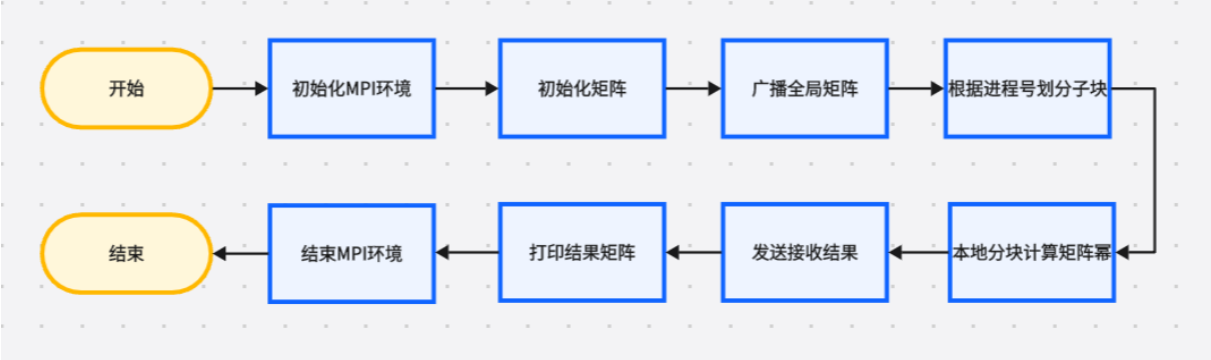
实验名称：多进程计算矩阵幂（MPI）

实验内容：本实验利用矩阵分块等思想，针对实验 2 的问题，采用 MPI 编程模型实现矩阵幂计算。

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

我们运用矩阵幂运算函数 (matrixPower):在给定的矩阵范围（由 start_row 和 end_row 指定）内，计算指定矩阵 matrix 的指定次幂 power，并将结果存入 local_result 数组。

函数首先初始化 local_result 为指定范围内的子矩阵，然后通过循环调用 matrixMultiply 函数逐步计算幂运算结果。

其中，主函数中：

我们定义广播全局矩阵：使用 MPI_Bcast 将根进程中的全局矩阵广播到所有进程，确保所有进程都具有相同的矩阵数据。

然后划分任务与分块计算：计算每个进程应处理的行数，并确定当前进程负责计算的矩阵行范围。

最后一个进程可能需要处理额外行以确保整个矩阵被覆盖，在本地计算矩阵幂。

五、结果分析

1. 随着矩阵规模的增加，加速比逐渐增加，效率逐渐下降。
2. 在数据规模较小时，随着进程数增加，加速比随之增加，但当进程数达一定值后，加速比开始下降。
3. 在数据规模较大时，加速比可以随着进程数增加而升高。

一、实验内容概述

1. 算法概述

本实验利用矩阵分块等思想，针对实验 2 的问题，采用 MPI 编程模型实现矩阵幂计算。对每次矩阵相乘进行划分，划分方法可参考课程中的 Jacobi 迭代，将结果矩阵划分成 p（进程数）个子块，每个线程处理一个子块，再同步计算结果，以这种方式进行 N 次矩阵相乘，计算中每一个进程都要向其它进程发送数据，同时从其它进程接收数据。

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 MPI 并行编程技术进行并行化计算。并行计算环境设置参数为两个服务器，8 个内核。

3. 数据分析要求及并行化方法

关于“划分”：

我们首先确定进程数和矩阵大小：矩阵的大小决定了可以划分的块数，而进程数决定了每个块应该被分配给哪个进程。

接着计算每个进程应处理的行数（通常是通过将矩阵的行数除以进程数来实现的）。如果有余数，则需要将额外的行数分配给最后一个进程。

其次，分配矩阵块给进程：根据每个进程应处理的行数，将矩阵分成多个块。

第一个进程处理从第 0 行到第 rows_per_process - 1 行的块，第二个进程处理从第 rows_per_process 行到第 2 * rows_per_process - 1 行的块，依此类推。最后一个进程处理从第(size - 1) * rows_per_process 行到第 N - 1 行的块。

我们在本地进程中计算矩阵块，并合并结果。

并行化方法是将矩阵按行分块，每个进程负责计算一个块的矩阵幂，然后使用 MPI 通信将结果合并，充分利用多进程的优势，提高计算效率。

二、并行算法分析设计

本次实验的难点为在 MPI 环境中实现进程间通信。此部分我的代码实现主要包括：

1. 全局矩阵广播：

在根进程中，使用 MPI_Bcast 函数将全局矩阵广播到所有进程。

每个进程接收到全局矩阵后，可以在本地内存中创建一个与全局矩阵大小相同的矩阵副本。

2. 计算分块矩阵幂：

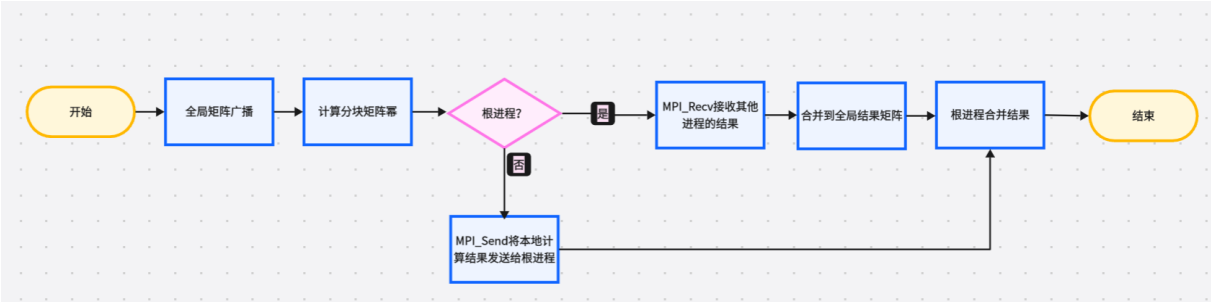
每个进程使用其本地矩阵副本计算分块矩阵的幂。

代码定义了 start_row 和 end_row，分别表示每个进程计算的矩阵块的起始和结束行。

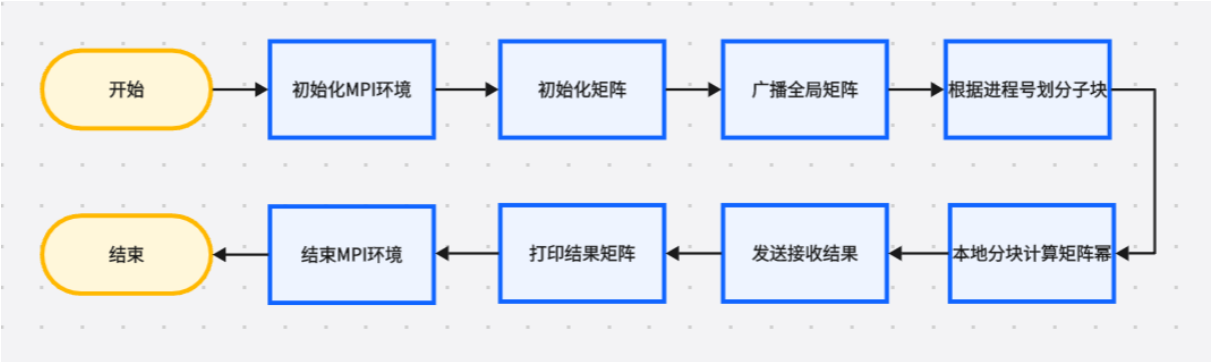
每个进程使用 matrixPower 函数计算其分块矩阵的幂，并将结果存储在局部变量 local_result 中。

3. 结果发送和接收：

如果进程不是根进程（rank 不为 0），则使用 MPI_Send 将本地计算结果发送给根进程。
如果进程是根进程，则使用 MPI_Recv 接收其他进程的结果，并将其合并到全局结果矩阵中。
发送和接收操作需要指定目标进程 ID、数据类型、数据大小和 MPI 通信域。
4. 结果合并：
根进程将接收到的结果矩阵合并到全局结果矩阵中。
合并时，需要确保结果矩阵的行数和列数与全局矩阵一致。
5. 打印结果：
如果进程是根进程，则打印合并后的全局结果矩阵。
进程通信的流程图如下：



我编写代码的思路（伪代码）为：
①初始化 MPI 环境： 使用 MPI_Init 函数初始化 MPI 环境，并使用 MPI_Comm_rank 和 MPI_Comm_size 获取当前进程的 ID 和总进程数。
②定义矩阵大小和幂次数： 定义矩阵的大小 N 和计算矩阵幂的次数 power。
③在根进程中初始化全局矩阵： 在根进程（rank 为 0）中，使用随机数生成一个 N 阶的矩阵，并打印出来。
④广播全局矩阵： 使用 MPI_Bcast 函数将全局矩阵广播到所有进程。
⑤分块计算矩阵幂： 根据进程数 size 将矩阵按行分块，每个进程负责计算一部分矩阵的幂。
⑥在本地计算分块矩阵的幂： 在本地进程中，使用 matrixPower 函数计算分块矩阵的幂。
⑦发送和接收结果：如果进程不是根进程，则使用 MPI_Send 将本地计算结果发送给根进程。如果进程是根进程，则使用 MPI_Recv 接收其他进程的结果，并合并到全局结果矩阵中。
⑧打印结果矩阵： 如果进程是根进程，则打印合并后的全局结果矩阵。
⑨结束 MPI 环境： 使用 MPI_Finalize 结束 MPI 环境。
⑩结束程序： 返回 0 以结束主函数。
下面是代码的流程图：



实验代码如下：

```
#include <iostream>
#include <cmath>
#include <mpi.h>

using namespace std;

// 矩阵乘法
void matrixMultiply(const int* A, const int* B, int* result, int m, int n, int p) {
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            result[i * n + j] = 0;
            for (int k = 0; k < p; ++k) {
                result[i * n + j] += A[i * p + k] * B[k * n + j];
            }
        }
    }
}

// 矩阵幂运算
void matrixPower(const int start_row, const int end_row, const int* matrix, const int N,
const int power, int* local_result) {
    // 开始进行矩阵幂运算
    for(int i = 0; i < (end_row - start_row); i++){
        for (int j = 0; j < N; j++) {
            local_result[i * N + j] = matrix[(start_row + i) * N + j];
        }
    }
    int* temp_result = new int[(end_row - start_row) * N];

    for(int i = 1; i < power; i++){
        matrixMultiply(local_result, matrix, temp_result, (end_row - start_row), N, N);
        // 将临时结果复制到 local_result 中
        for (int j = 0; j < (end_row - start_row) * N; j++) {
            local_result[j] = temp_result[j];
```

```
        }
    }

    delete[] temp_result;
}

int main(int argc, char *argv[]) {
    // 初始化 MPI 环境
    MPI_Init(&argc, &argv);

    // 获取当前的进程编号与进程总数
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // 本实验设定测试数据，设定矩阵规模为 16*16
    const int N = 16;
    // 定义计算矩阵幂的次数为 5
    const int power = 5;
    // 定义本程序的全局矩阵，且仅能在根进程中被初始化
    int matrix[N][N];

    // 在根进程中初始化全局矩阵，生成随机矩阵
    if (rank == 0) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                matrix[i][j] = rand() % 10 + 1;
                cout<<matrix[i][j]<<" ";
            }
            cout<<endl;
        }
    }
    // 将全局矩阵广播到所有进程
    MPI_Bcast(&matrix[0][0], N * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // 将矩阵按行分配给各个进程，实现分块计算以提高运行效率
    // 首先定义好每个进程分得的矩阵的起始与终点行数
    int rows_per_process = N / size;
    int start_row = rank * rows_per_process;
    int end_row = (rank + 1) * rows_per_process;
    if (rank == size - 1)  end_row = N; // 最后一个进程需要单独定义

    // 在本地计算分块后矩阵的矩阵幂
    int local_result[(end_row - start_row) * N];
```

```
    matrixPower(start_row, end_row, &matrix[0][0], N, power, local_result);

    // 发送每个进程的局部矩阵幂计算结果到根进程
    if (rank != 0) {
        MPI_Send(&local_result[0], (end_row - start_row) * N, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
    }
    else {
        // 根进程接收每个进程的局部结果，并存储在结果矩阵中
        int result[N][N];
        for (int i = 0; i < (end_row - start_row); i++) {
            for (int j = 0; j < N; ++j) {
                result[start_row + i][j] = local_result[i * N + j];
            }
        }

        // 接收其他进程的局部结果
        for (int source = 1; source < size; ++source) {
            int start_row1 = source * rows_per_process;
            int end_row1 = (source + 1) * rows_per_process;
            if (source == size - 1)  end_row1 = N;

            int recv_buffer[(end_row1 - start_row1) * N];
            MPI_Recv(&recv_buffer[0], (end_row1 - start_row1) * N, MPI_DOUBLE, source, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for (int i = 0; i < (end_row1 - start_row1); ++i) {
                for (int j = 0; j < N; ++j) {
                    result[start_row1 + i][j] = recv_buffer[i * N + j];
                }
            }
        }

        // 打印结果矩阵
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j) {
                cout << result[i][j] << " ";
            }
            cout << endl;
        }
    }

    // 注意要结束 MPI 环境
    MPI_Finalize();
    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 4.1.1

（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 8*8、16*16、32*32 矩阵进行 5 次幂运算的情景，将 1（串行）、2、4、8、16 进程情况下的运行时间对比，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：

```
tjucic100@ln0:~$ module load openmpi/4.1.4-mpi-x-gcc9.3.0
tjucic100@ln0:~$ mpic++ -o test3.o test3.cpp
tjucic100@ln0:~$ time yhrun -p thcp1 -N 2 -n 4 test3.o &>run.log

real    0m1.677s
user    0m0.307s
sys     0m0.091s
```

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：

运行时间

计算规模	1	2	4	8	16
8*8*8*5	1.475	1.509	1.647	1.731	1.753
16*16*16*5	1.510	1.571	1.687	1.764	1.704
32*32*32*5	1.679	1.765	1.798	1.876	1.986

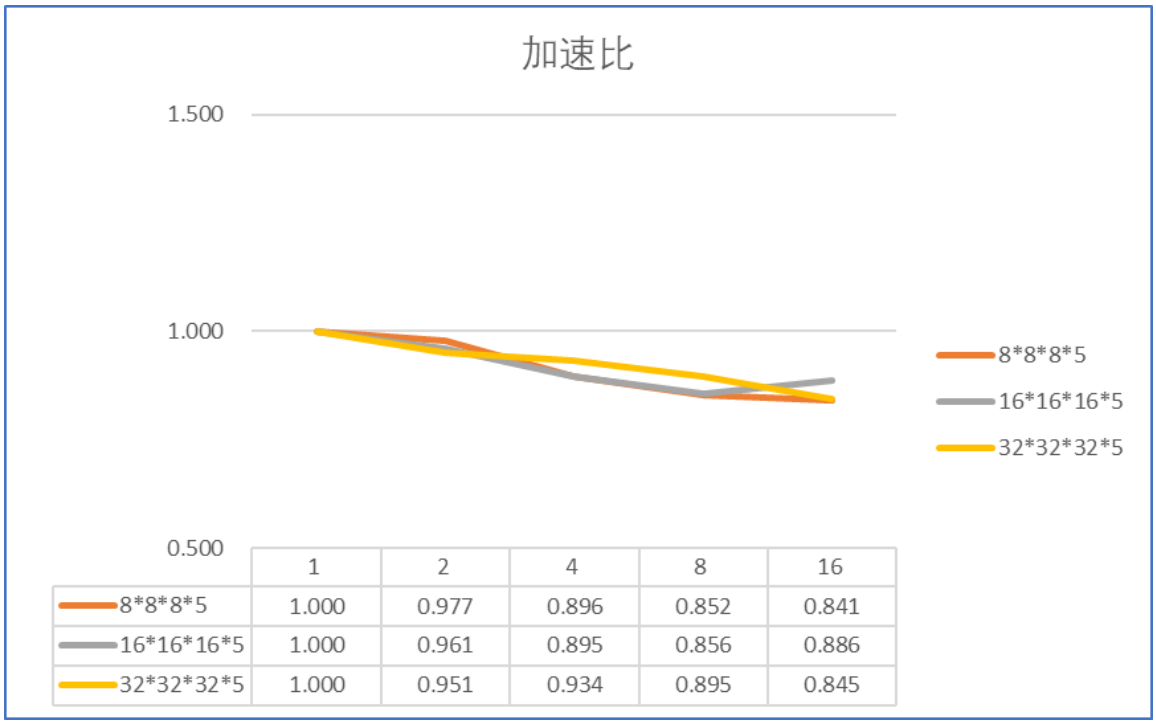
加速比

进程数	1	2	4	8	16
8*8*8*5	1.000	0.977	0.896	0.852	0.841
16*16*16*5	1.000	0.961	0.895	0.856	0.886
32*32*32*5	1.000	0.951	0.934	0.895	0.845

效率

进程数	1	2	4	8	16
8*8*8*5	1.000	0.489	0.224	0.107	0.053
16*16*16*5	1.000	0.481	0.224	0.107	0.055
32*32*32*5	1.000	0.476	0.233	0.112	0.053

加速比图表如下：



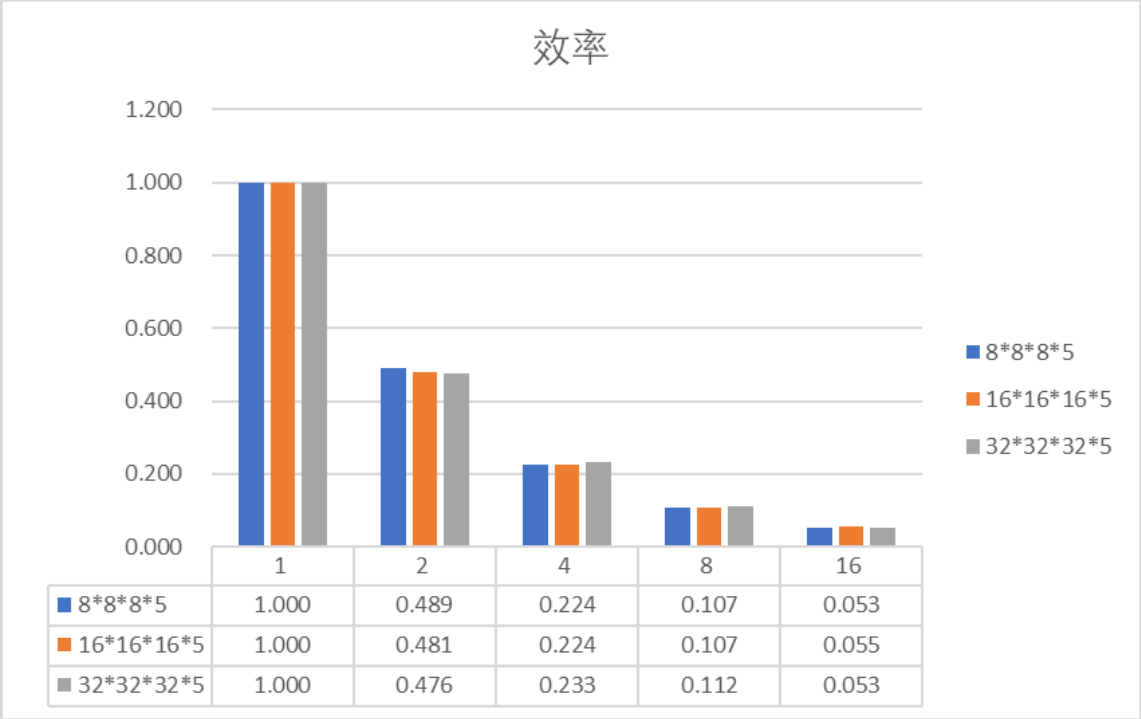
分析：

理论上讲，矩阵规模较大时，随着进程数增加，加速比呈线性增长。

但从实验结果总体上看，在规模一定的情况下，核数越多，加速比甚至越低，但当核数增加到一定数量时(如规模为 16*16*16*5，核数由 8 增加到 16 时)，加速比甚至会回升，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，我们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，总体上加速比越高；在规模相对较小时，加速比的提升较小甚至近似相等。

效率图表如下：



	1	2	4	8	16
8*8*8*5	1.000	0.489	0.224	0.107	0.053
16*16*16*5	1.000	0.481	0.224	0.107	0.055
32*32*32*5	1.000	0.476	0.233	0.112	0.053

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 16*16*16*5 时，并行程序的性能并没有串行程序好,加速比并没有明显提升，甚至有的数据集小于 1。
当数据规模为 8*8*8*5 时，随着进程数的增加，加速比没有大幅度的增加，甚至总体上低于其他规模条。随着进程数的增加，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

（1）随着矩阵规模的增加，加速比逐渐增加，效率逐渐下降。
（2）矩阵规模较小时，随着进程数的增加，加速比会随之增加，但是当进程数达到一定值之后，加速比会开始下降。
（3）矩阵规模较大时，随着进程数的增加，加速比会随之增加，且增加的幅度比较大。
（4）矩阵规模越大，进程数越少，效率越高。
（5）对于不同的矩阵规模，进程数达到最优的值是不同的。本次实验，矩阵规模 8*8，进程数为 2 时，加速比最高，达到 0.977；矩阵规模 16*16，进程数为 2 时，效率值最高，达到 0.961。
综上，通过这些实验可以得出使用 MPI 进行并行计算时，需要考虑到计算规模、进程数和通信开销的关系，以及进程数达到一定程度后效率下降的问题。此外，还可以通过实验数据来优化程序，选择合适的进程数和计算规模，从而获得更好的性能。

四、实验总结

之前对多进程并行程序的认识只是理论层面，通过本次实验，我亲手实现了相关代码，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

对于 MPI 进程并行化编程中主要使用到函数是 MPI_Init()、MPI_Finalize()用于初始化和终止并行环境，MPI_Comm_size()用于获取进程数量，MPI_Comm_rank()用于获取当前进程的 id，以及 MPI_Send()和 MPI_Recv()用于执行进程间的通信。这里遇到的一个比较奇怪的问题是，各个进程开始位置，理论上因该从 MPI_Init()初始化之后，才是真正并行化区域，但是实际过程中，在初始化之前的输出也会被每个进程都输出一遍，即每个进程的执行是整个程序，类似于 fork()的进程创建过程，这个问题在调试了很多遍依旧没有得到解决，最后我将输出的信息和计时放在了 0 进程中，才能达到只在开始时执行一遍地效果。

其次，MPI_Send()和 MPI_Recv()通信函数的参数要求为传送缓冲区的起始地址和指定类型的长度，其实质是传输了一个一维连续的空间到另一个进程，而矩阵的幂计算是在二维空间的操作，因此对子域的传输需要我们进行一维和二维之间的转换。其中，在计算分块矩阵幂部分时，每个进程使用其本地矩阵副本计算分块矩阵的幂。代码定义了 start_row 和 end_row，分别表示每个进程计算的矩阵块的起始和结束行。

每个进程使用 matrixPower 函数计算其分块矩阵的幂，并将结果存储在局部变量 local_result 中。我们将矩阵按行分块，每个进程负责计算一个块的矩阵幂。这种方法可以充分利用多进程的优势，提高计算效率。

最后，是未使用的进程忙等待的问题，是在矩阵划分中发现的问题，由于该方法对进程数量有完全平方数的要求，因此当有多余进程无法被用到时，它会一直阻塞在 MPI_Recv()等待接收信息，而导致程序无法终止，不能完成计时，最初我采用了给其他进程发送长度为 0 的空消息试图解决，但是对于进程数 7、12 等较大数可以正常截止，对于进程数 3、4 依旧无法结束，最后修改了脚本文件仅仅测量合适进程数。

2. 并行计算方式的理解与分析

（1）对并行计算的理解更加深入：在实验中，我深入了解了进程之间的通信机制，以及如何避免并行计算中出现的竞争和死锁等问题。

（2）了解了 MPI 编程的基本技巧：在实验中，我学会了使用 MPI 库中的基本函数，如 MPI_Send、MPI_Recv 等。使用这两个函数时要格外仔细，应该先把数据发出去，然后再接收，如果顺序反过来，进程没有收到数据会一直处于等待状态，有可能造成死循环。并且发送和接收方的进程号，发送的数据量，数据类型也要一一对应。

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	多进程处理子域进行相关计算	MPI 并行编程技术	主要使用到函数 MPI_Init(&argc, &argv) 与 MPI_Finalize()进行并行化环境的初始化与终止，MPI_Send()和 MPI_Recv()进行进程间点对点通信过程（传递和收集各个进程需要进行计算的子域），注重这些库函数的原型以及要求的参数类型，进行正确传参和调用。
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。