并行计算课程
实 验 报 告

报告名称：	多级并行化计算矩阵幂
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 26 日

2024 年制

摘要

姓名 陈秋澄 学号 3022244290

一、实验名称与内容

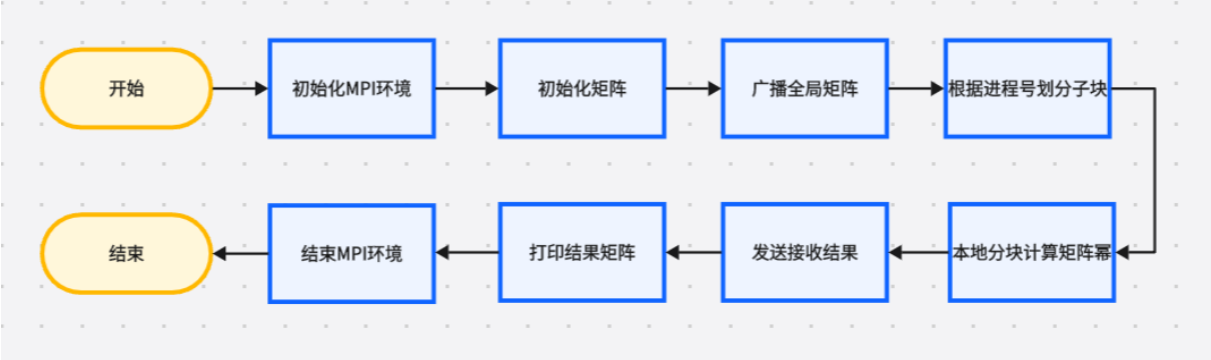
实验名称：多级并行化计算矩阵幂（MPI+OpenMP）

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+OpenMP 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。

根据执行代码时的进程数将原始矩阵划分成 n 个子块，每个进程处理自己相对应的子块。在每个子块对应的矩阵相乘函数中创建 omp 并行区域，每个线程共享本进程的数据，对卷积函数并行化处理，然后进程间交换临界数据，更新矩阵，再进行下一次运算。等到所有进程都处理完毕后，主进程收集各进程处理完的数据，输出最终结果。

五、结果分析

1. 小规模的矩阵计算，多线程能够提升程序性能，而多进程并不一定能够有效提升程序性能。
2. 多进程和多线程都能够提高程序的运算速率，但是对于矩阵规模较小的情况下，多线程的效果更好，对于矩阵规模较大的情况下，多进程的效果更好。
3. 在一定范围内，随着进程数和线程数的增加，加速比和效率都能够提高，但是进程数和线程数不是越多越好，需要根据具体的情况进行选择。
4. 对于计算密集型的程序，多进程的效率一般比多线程高。

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+OpenMP 编程模型实现矩阵幂计算。对每次矩阵相乘进行划分，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+OpenMP 实现并行化计算。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

我测试了不同进程数、不同线程数是程序的相关指标，分析程序性能。

关于“划分”：

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。接着通过将矩阵的行数除以进程数来计算每个进程应处理的行数。如有余数，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。

通过将矩阵的行分配给不同的 MPI 进程，实现了数据的划分。每个进程独立地计算其分配到的行的矩阵幂，最后将结果汇总到根进程中。

此外，在矩阵乘法函数中，您使用了#pragma omp parallel for 指令来并行化内层循环。这使得每个 MPI 进程中的多个线程可以同时执行矩阵乘法的一部分，从而加速计算。

我的程序通过 MPI 实现分布式内存并行，将大矩阵划分为多个子矩阵分配给不同的进程处理，同时利用 OpenMP 在每个进程中启用多线程并行计算子矩阵的乘法操作，从而达到双重并行化的目的。

二、并行算法分析设计

本次并行算法主要包括：

1. 数据划分：

使用 MPI，将矩阵的行均匀地分配给不同的进程。每个进程负责计算分配给它的矩阵行的矩阵幂。如果矩阵的行数不能被进程数整除，最后一个进程将处理剩余的行。

2. MPI 通信：

使用 MPI_Bcast 将初始矩阵广播到所有进程。

使用 MPI_Send 和 MPI_Recv 在根进程和其他进程之间传输局部矩阵幂的结果。

3. OpenMP 并行化：

在矩阵乘法函数中，使用#pragma omp parallel for 指令来并行化内层循环，从而利用多线程加速计算。

4．负载均衡：

通过均匀分配矩阵行到各个进程，尝试实现负载均衡。

5．内存管理：

动态分配和释放内存，以避免过多的内存占用。

本次实验的难点为在 MPI+OpenMP 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。此外，在矩阵乘法函数中，使用#pragma omp parallel for 指令来并行化内层循环，从而利用多线程加速计算。

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。

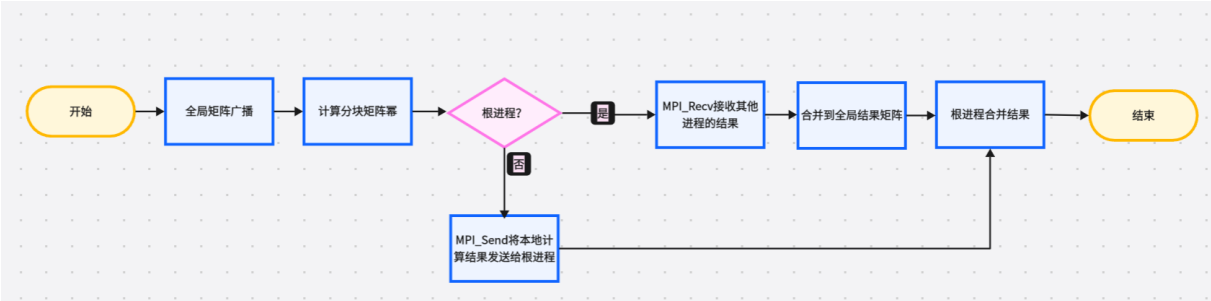
4) 结果合并：

根进程将接收到的结果矩阵合并到全局结果矩阵中。

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。

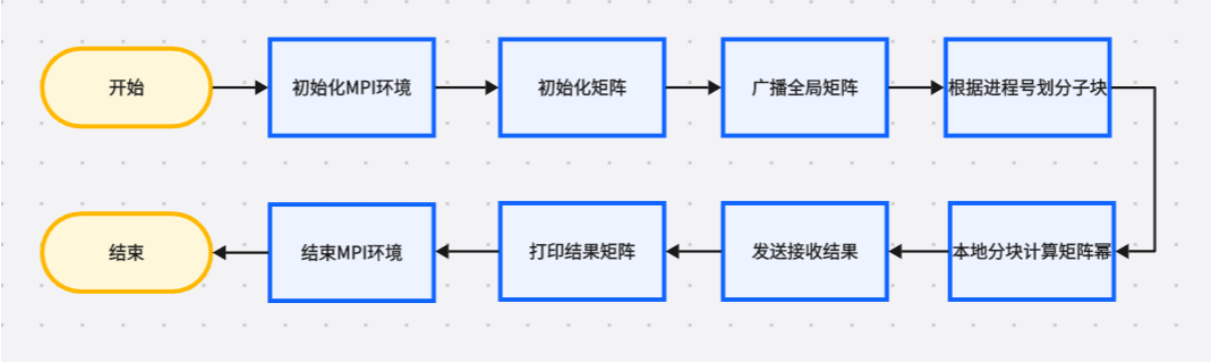
②定义矩阵大小和幂次数： 定义矩阵的大小 N 和计算矩阵幂的次数 power。

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。在矩阵乘法函数中，使用#pragma omp parallel for 指令来并行化内层循环，从而利用多线程加速计算。
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>
#include <omp.h> // 确保包含了 OpenMP 头文件

using namespace std;
int num_thread; // 每个进程的线程数
// 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) {
        #pragma omp parallel for num_threads(num_thread)
        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;
    // int num_thread; // 每个进程的线程数

    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_INT, 0, MPI_COMM_WORLD);

    // 设置每个进程的线程数，这里以 OpenMP 的最大线程数为准
    num_thread = omp_get_max_threads();

    // 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_INT, 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_INT, 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 16*16、32*32、50*50 矩阵进行 5 次幂运算的情景，将 1（串行）、2、4、8 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

（1）数据图表分析

相关图表如下：

第一，规模为 16*16*16*5 的性能分析（表格的行代表不同线程数，表格的列代表不同进程数）

不同进程、不同线程的运行时间(s)

n \ c	1	2	4	8
1	1.519	1.582	1.633	1.638
2	1.662	1.65	1.575	1.519
4	1.702	1.686	1.628	1.716
8	1.903	1.741	1.832	1.731

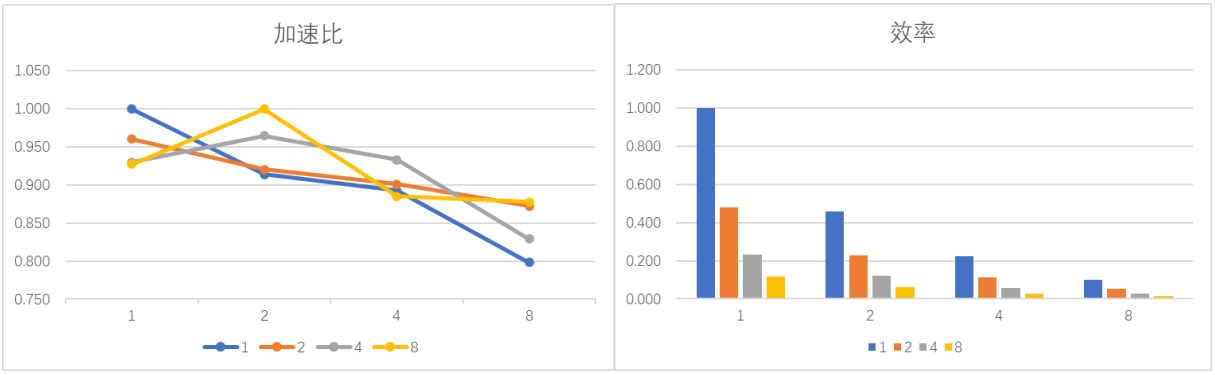
不同进程、不同线程的加速比

n \ c	1	2	4	8
1	1.000	0.960	0.930	0.927
2	0.914	0.921	0.964	1.000
4	0.892	0.901	0.933	0.885
8	0.798	0.872	0.829	0.878

不同进程、不同线程的效率

n \ c	1	2	4	8
1	1.000	0.480	0.233	0.116
2	0.457	0.230	0.121	0.063
4	0.223	0.113	0.058	0.028
8	0.100	0.055	0.026	0.014

加速比与效率图表如下：



注：图中横坐标代表进程数，不同颜色代表不同线程数，下面的图也是如此

分析：

理论上讲，矩阵规模较大时，进程数一定时，随着线程数增加，加速比应呈线性增长。但从实验结果总体上看，在规模一定的情况下，线程数越多，加速比甚至越低，但当进程数达到到一定数量时(如规模为 16*16*16*5，进程数为 8、线程数由 1 增加到 2 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。
总而言之，计算规模为 16*16*16*5 时，多进程对程序性能的提升并不明显，加速比甚至小于 1，效率下降得较快。多线程能够提升程序性能，随着线程数增加，加速比先随之增加，但是当进程数达到一定值之后，加速比会开始下降。当矩阵规模较小时，多进程并不能带来明显的性能提升，因为多进程需要进行进程间通信，而进程间通信的开销可能比较大，能会抵消掉并行化带来的好处，从而导致加速比小于 1，效率下降。而对于多线程，由于线程之间共享进程内存，线程间通信的开销相对较小，因此可以更有效地提高程序的性能。随着线程数的增加，可以充分利用计算机的多核心处理能力，进一步提高程序的性能，因此加速比可以有一定程度的提升。总体上看，在一定规模时，核数越多，效率越低。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。进程数一定时，效率随线程数的增加而减少。
第二，规模为 32*32*32*5 的性能分析（表格的行代表不同线程数，表格的列代表不同进程数）：

运行时间

n \ c	1	2	4	8
1	2.028	1.7	1.616	1.395
2	1.817	1.697	1.66	1.538
4	1.746	1.693	1.71	1.767
8	1.781	1.762	1.832	1.811

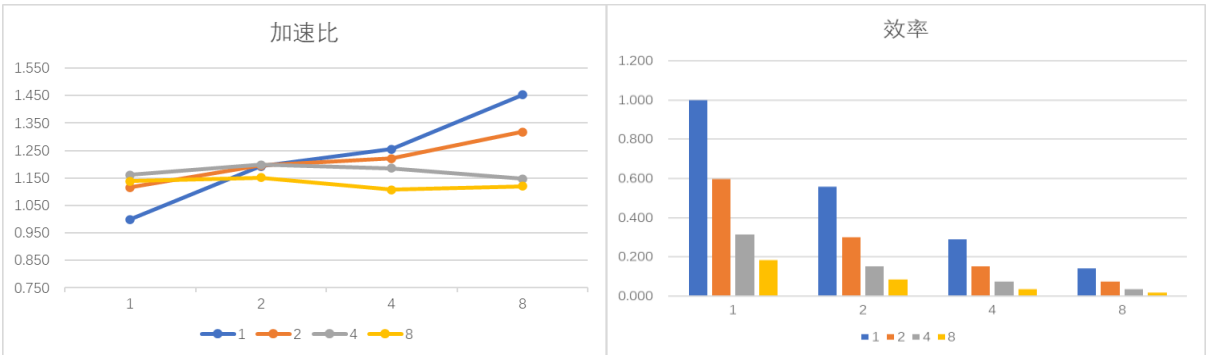
加速比

n \ c	1	2	4	8
1	1.000	1.193	1.255	1.454
2	1.116	1.195	1.222	1.319
4	1.162	1.198	1.186	1.148
8	1.139	1.151	1.107	1.120

效率

n \ c	1	2	4	8
1	1.000	0.596	0.314	0.182
2	0.558	0.299	0.153	0.082
4	0.290	0.150	0.074	0.036
8	0.142	0.072	0.035	0.017

加速比与效率图表如下：



分析：计算规模 32*32*32*5 时，随着进程数和线程数增加，加速比总体上不断增大，效率相对也有大幅度提升。并且多进程的效率略大于多线程。
这是因为当矩阵规模为 32*32*32*5 时，由于计算量较大，多线程在处理单个进程的任务时，线程之间可能会存在大量的竞争和等待，这会影响多线程的效率。而进程间通信的开销变得相对较小，因此多进程可以更好地利用计算资源，从而实现更好的效率。
第三，规模为 50*50*50*5 的性能分析（表格的行代表不同线程数，表格的列代表不同进程数）：

运行时间

n \ c	1	2	4	8
1	1.7	1.625	1.609	1.561
2	1.797	1.645	1.628	1.516
4	1.758	1.744	1.812	1.802
8	2.001	1.841	1.839	1.789

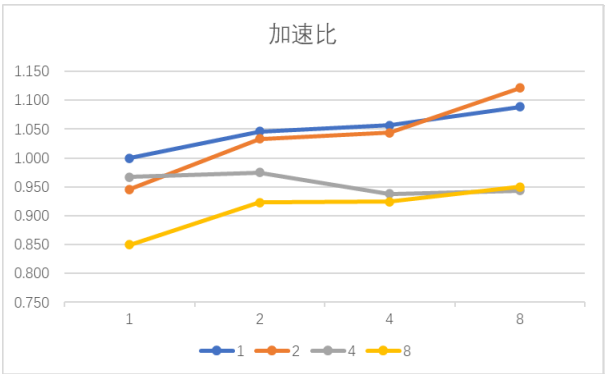
加速比

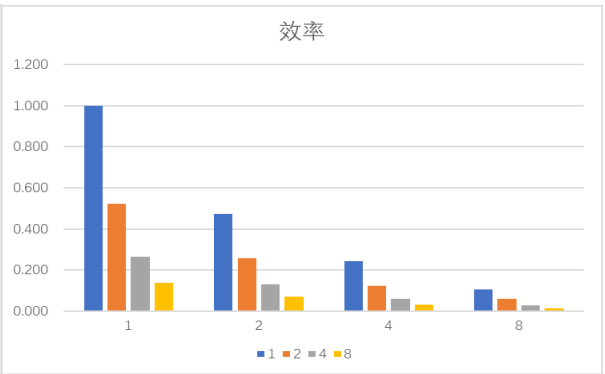
n \ c	1	2	4	8
1	1.000	1.046	1.057	1.089
2	0.946	1.033	1.044	1.121
4	0.967	0.975	0.938	0.943
8	0.850	0.923	0.924	0.950

效率

n \ c	1	2	4	8
1	1.000	0.523	0.264	0.136
2	0.473	0.258	0.131	0.070
4	0.242	0.122	0.059	0.029
8	0.106	0.058	0.029	0.015

加速比与效率图表如下：





矩阵规模 50*50*50*5 时，随着进程数和线程数增加，加速比总体上不断增大，多进程的效率略大于多线程。并且加速比在 2 进程 8 线程时达到最大，为 1.121。
说明在计算量较大时，并行化更能提升程序的运行效率。

（2）实验结论

A. 横向对比：

a) 当矩阵规模较小时，随着进程数增加，加速比并没有明显增加，甚至还会下降；随着线程数增加，加速比不断增加，但增加的幅度较低；多进程的效率值比多线程的低。
b) 当矩阵规模较大时，随着进程数增加，加速比呈线性增长；随着线程数增加，加速比也增加，但增加的幅度没有多进程大；多线程的效率比多进程的低。

B. 纵向对比：

a) 矩阵规模较小时，多线程的性能更好；矩阵规模增大时，多进程性能更好。
b) 并不是进程数和线程数越多越好。

C. 原因分析

a) 当矩阵规模较小时，加速比增长不明显或下降，这是因为矩阵规模较小，处理器之间的通信和同步开销较大，从而抵消了并行计算的优势。当矩阵规模较大时，由于计算量增加，处理器之间的通信和同步开销相对减少，从而提高了并行计算的效率。而加速比和效率的提高趋势，可以通过 Amdahl 定律进行解释。Amdahl 定律指出，系统的加速比取决于可并行化部分的比例，即串行部分在总执行时间中所占的比例。当可并行化部分较小时，增加处理器数目并不能显著提高加速比，因为串行部分的执行时间仍然占主导地位，从而抵消了并行计算的优势；而当可并行化部分较大时，增加处理器数目可以显著提高加速比。
b) 矩阵规模较小时，多线程的效果更好，这是因为在小规模问题中，进程的切换开销更大，所以多进程的效果不如多线程。但随着矩阵规模的增加，多进程的效果变得更好，这是因为多进程可以充分利用多个处理器，每个进程可以占用独立的处理器运行，从而避免了处理器之间的竞争和互斥，提高了并行效率。而多线程则受制于单个处理器的性能，其并行效率会随着线程数的增加而饱和甚至下降。

D. 实验结论

a) 小规模的矩阵计算，多线程能够提升程序性能，而多进程并不一定能够有效提升程序性能。
b) 随着矩阵规模的增加，多进程和多线程都能够提高运算速率，但多进程的优势更加明显。
c) 多进程和多线程都能够提高程序的运算速率，但是对于矩阵规模较小的情况下，多线程的效果更好，对于矩阵规模较大的情况下，多进程的效果更好。
d) 在一定范围内，随着进程数和线程数的增加，加速比能够提高，但是进程数和线程数不是越多越好，需要根据具体的情况进行选择。
e) 对于计算密集型的程序，多进程的效率一般比多线程高。

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

问题与解决：

（1） 刚开始，我在命令行里输入线程数，但在代码中没有指定，导致运行时间不符合预期。后来，我在代码中使用#pragma omp parallel for num_threads(num_thread) 语句给每个进程分配线程数，在命令行中输入 num_thread，这样就可以为每个进程分配指定的线程数。

（2） 我指定一个进程和一个线程，OpenMP 的运行时间比串行代码要短。后来通过查找资料，了解一些 OpenMP 的运行机制。OpenMP 本质上是一种并行编程模型，其主要目的是将并行计算的任务分配给多个处理器或核心，以提高程序的性能。在指定一个线程时，OpenMP 可以使用线程的并行性来更好地利用处理器或核心的能力。即使只有一个线程，OpenMP 仍然可以使用一些优化技术，如循环展开和向量化等，来提高程序的性能，从而得到比串行代码更快的运行时间。

收获与体会：

（1） 通过实验，我学会了用 OpenMP 并行化处理程序，并了解了 OpenMP 与 pthread 的区别。OpenMP 使用共享内存编程模型，即所有线程共享同一个地址空间，可以访问相同的变量和数据结构。程序员只需要在代码中添加一些指令，就可以让系统自动将指定的任务分配到不同的线程上执行。pthread 则使用多线程编程模型，它需要程序员显式地创建线程，并且需要自己负责线程间的同步和互斥。

（2） 学习了 MPI 和 OpenMP 的联合使用，通过对实验数据的分析，得到了进程数和线程数不是越多越好，需要不断进行实验，根据具体的矩阵规模和计算任务来选择合适的并行度。

（3） 将进程数和线程数进行对比，发现计算 I/O 操作密集型的代码多线程效率更高,因为线程创建要比进程创建开销少。但是计算密集型的代码多，那么进程操作更快,因为多进程可以应用多核技术，每个进程可以占用独立的处理器运行，从而避免了处理器之间的竞争和互斥，提高了并行效率。

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。

（3）学习将 MPI 和 OpenMP 的联合使用，实现多级并行化矩阵幂计算的程序实现。

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。