并行计算课程
实 验 报 告

报告名称：	MPI 计算矩阵幂
姓　　名：	陆子毅
学　　号：	3022206045
联系电话：	15262559069
电子邮箱：	tommygong@tju.edu.cn
填写日期：	2024 年 4 月 16 日

2024 年制

一、实验内容概述

本实验旨在采取 MPI 通信的方式进行矩阵幂次的计算，在此基础上完成矩阵幂次的计算，以加深对并行算法的理解和应用。

通过采用 MPI 进程通信技术，将结果矩阵中的每一行的计算交给不同的进程运算，提高运算效率与程序性能。并对不同进程数量的运行结果进行性能分析。

在实验中，我们将设计并实现一个并行算法，将矩阵相乘的不同区块进行划分，根据 result 矩阵中元素的位置来判断原矩阵参与运算的行和列，通过合理的任务划分和进程管理，将尽可能地提高计算效率，并在实验结束后对加速比进行分析。所有进程最后将运算结果汇集到 0 号进程当中。

并行计算环境：

内存 128GB；硬盘 11PB；CPU 数量 64；

单核理论性能（双精度）9.2 GFlops；单节点理论性能（双精度）588.8 GFlops

国家超级计算天津中心定制操作系统、使用国产飞腾处理器、天河自主高速互联网络

二、并行算法分析设计

2.1 串行计算

四重循环计算矩阵幂次。第一层循环控制矩阵乘法的次数，第二三四层采用线性代数的方法计算矩阵乘法。

```
Begin
  output = 单位矩阵
  for n in 1..N
    for i in 1..M (matrix_size)
      for j in 1..M (matrix_size)
        for k in 1..M (matrix_size)
          temp(i, j) += output(i, k) * matrix(k, j)
        endfor
        output = temp
      endfor
    endfor
  endfor
```

2.2 并行计算

为了降低通信次数，避免通信时间占比过多而影响程序性能，程序采用行划分的方式对矩阵进行计算。

以行为单位计算结果，最后将所有计算结果汇总到 0 号进程当中。由于过多的通信会导致程序性能下载，所以，每个线程会计算一行的 n 次幂，当所有进程结束以后，进行 MatrixSize 次数的通信，将不同进程的计算结果汇集。

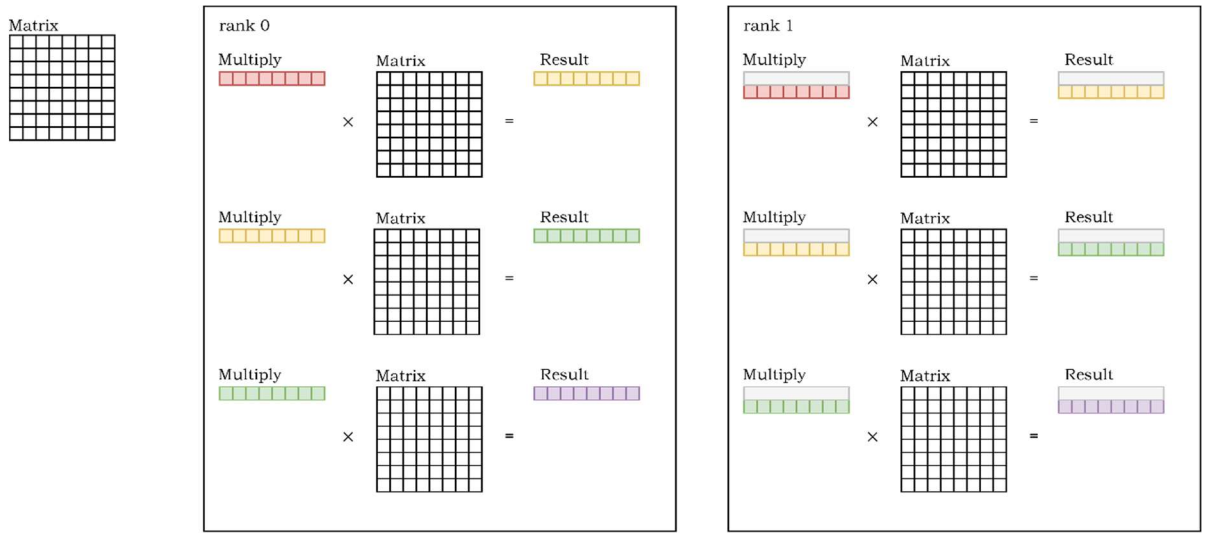
```
void matrixMultiply(int n, int row, Matrix *multiply, Matrix *matrix, Matrix *result){
    double *temp = (double *)malloc(MATRIX_SIZE * sizeof(double));
    for (int power = 0; power < n; power++){
        for (int i = 0; i < MATRIX_SIZE; i++){
            double sum = 0;
            for (int j = 0; j < MATRIX_SIZE; j++){
                sum += multiply->data[row][j] * matrix->data[j][i];
            }
            temp[i] = sum;
        }
        for (int i = 0; i < MATRIX_SIZE; i++){
            multiply->data[row][i] = temp[i];
            result->data[row][i] = temp[i];
        }
    }
    free(temp);
}
```

如上图代码所示，以 multiply 矩阵第 0 行为例，将第 0 行右乘 matrix 矩阵，得到一行大小为矩阵列数的数据，再将这行数据写回 multiply 的第 0 行，将上述操作反复 n 次就完成了矩阵中第 0 行的幂次计算。

```
void matrixPower(Matrix *matrix, Matrix *multiply, Matrix *result, int n, int size, int rank){
    MPI_Bcast(&matrix->data[0][0], MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD); // 广播矩阵
    MPI_Barrier(MPI_COMM_WORLD);                                                               // 同步所有进程
    int row = rank;
    while (row < MATRIX_SIZE){
        matrixMultiply(n, row, multiply, matrix, result);
        row += size;
    }
}
```

现在实现了矩阵中一行的计算，为了得到最终的结果，我们需要将矩阵的不同行进行分配，交给不同的线程去完成，这里使用 rank+n*size 的方法来分配，rank 表示当前进程号，size 表示总进程数量。由于进程之间拥有独立的内存空间，所以相较于 Pthread 编程，不需要考虑互斥锁的问题。进程的划分通过 rank 来控制。

下面是进程计算以及数据划分的示意图：



程序每次会随机生成一个 350×350 的矩阵，然后通过 MPI 并行计算该矩阵的 150 次幂。最后输出结果。

为了方便调试与检查代码错误，定义了一个矩阵输出函数。

三、实验数据分析

3.1 实验环境

环境：国家计算天津中心天河超算
CPU：飞腾@2.3GHz
CPU 数量：64
运行内存容量：128GB
存储容量：>11PB
并行数据传输：1PB
内存带宽：204.8GB/s
每个核心的线程数：1
每个插槽的核心数：64
NUMA 节点数：8
制造商 ID：0x70
型号：2
步进：0x1
BogoMIPS：100.00
L1 数据缓存：2 MiB
L1 指令缓存：2 MiB

3.2 实验数据综合分析

3.2.1 实验数据获取以及处理方式

采用 sh 脚本循环运行编译后的程序，递增线程数量，获取 time 指令得到的 real 值，计算 5 次取平均值。

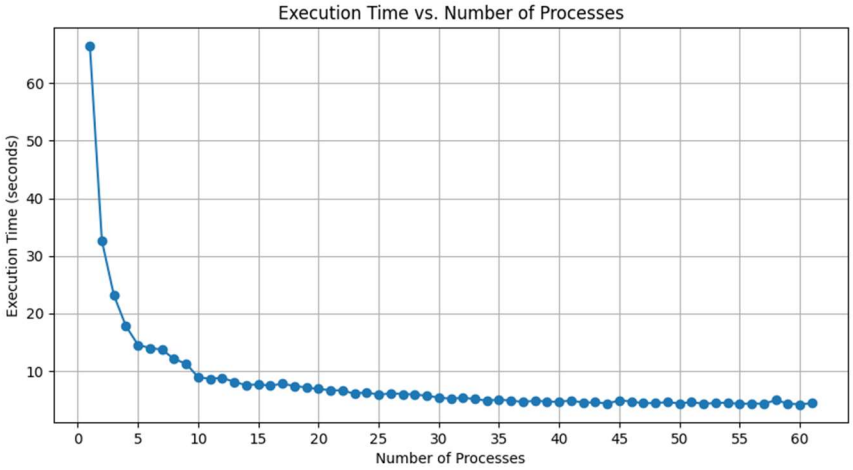
```
#!/bin/bash
> run.log
power=15
runs=2
module unload openmpi/mpi-x-gcc9.3.0
module load openmpi/4.1.4-mpi-x-gcc9.3.0
mpic++ -o lab3.o lab3.c


for size in {1..64}; do
    echo "Running with core=$size" >> run.log
    total_time=0
    # 多次运行以获取平均值
    for ((i=1; i<=runs; i++)); do
        # 使用time命令并通过grep提取real时间
        run_time=$( { time yhrun -p thcp1 -N 4 -n $size ./lab3.o $power ; } 2>&1 | grep "real" | awk '{print $2}')
        # 将real时间转换为秒
        # 假设run_time的格式为minutes:seconds，比如0m1.234s
        min=$(echo $run_time | cut -d'm' -f1)
        sec=$(echo $run_time | cut -d'm' -f2 | sed 's/s//')
        # 计算总秒数
        total_sec=$(echo "$min * 60 + $sec" | bc)
        total_time=$(echo "$total_time + $total_sec" | bc)
        # 输出每次运行的时间到日志
        echo "Run $i: $total_sec sec" >> run.log
    done
    # 计算平均运行时间
    avg_time=$(echo "scale=3; $total_time / $runs" | bc)
    echo "Average time for core $core: $avg_time sec" >> run.log
    echo "" >> run.log
done
```

3.2.3 MPI 线程通信以及解决栈溢出

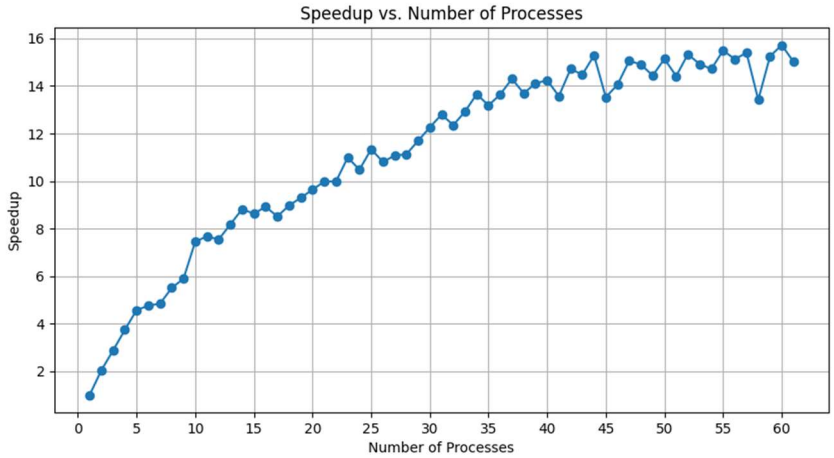
MPI 通信的使用不是很熟练，需要编写一些小程序来确保函数的使用能够达到预期效果。程序的优化这次主要体现在算法上面，开始时候的想法是每计算完一次乘法之后，都将结果汇总至 0 号进程，然后有 0 号进程再将数据 Bcast 给其他进程。但是这样子做会导致频繁的数据通信，降低程序的效率，为了减少通信的次数，数据的划分就非常重要，划分给每一个进程的数据块要相对独立，计算的过程和结果不依赖于其他进程的计算结果，遂采用行划分的方法，每一行加上源矩阵可以相对独立地完成计算，得到最后的结果。

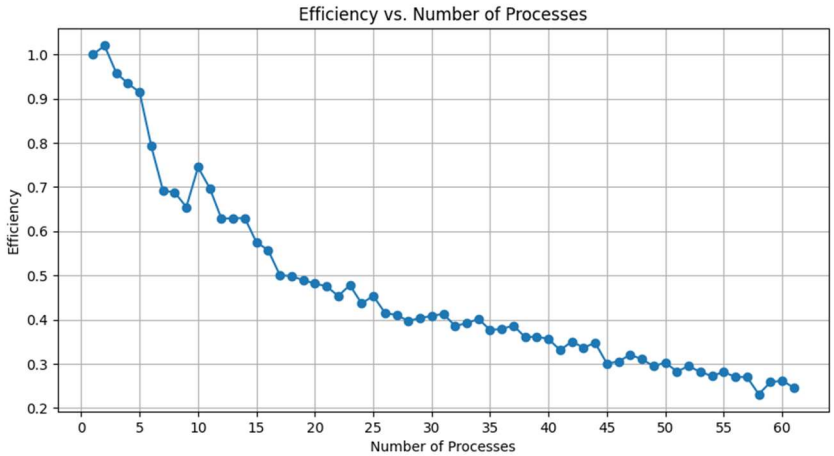
实验的结果如下图所示（采用随机数生成矩阵，实验原始数据在 zip 中）：



3.2.3 加速比以及效率的分析

根据实验结果数据绘制图像。





从实验结果可以看出，在 16 个进程以内的时候，进程数量的增加对于程序运行时间的降低有非常显著的作用，但是后面效果开始逐渐下降，实际应用的时候，为了合理使用资源，对于节点数

量的申请尽量控制在 1-16 的范围内比较合适。

和实验二一样，在数据量和计算规模都较小的情况下，增加进程数量带来的收益并不是很明显，程序的运行时间几乎没有什么变化，反而会因为进程之间的通信，导致额外的时间开销而降低程序的性能。

同时，根据实验所得到的加速比，可以大致推断出程序中串行部分的占比。根据图像可以看到，加速比随着线程数量的提高不断提高，但是最后稳定在了 15 左右的位置。最后没能突破 15，并且随着进程数量的增加，程序的运行时间并没有持续下降，最后反而有略微的上升。

根据加速比的公式以及阿姆达尔定律推断，我所采用的并行计算的方法不可以并行的部分大约占整个程序的 1/15，也就是说，串行部分在我的程序中只占了约 1/15，相比于实验二，有所降低，同时也是由于这个实验的实验内容程序主要的运行时间都在计算矩阵上。

而效率-进程数图则反应了当问题规模一定时，不断增加线程数量会降低每个计算核心的利用率，问题规模不变，线程增加，每个线程所分配到的任务减少，最开始任务量会减少 1/2，1/4，到后来任务量的减少量不断减小。每个进程的运行时间的降低速度也就不断减少了。

四、实验总结

4.1 遇到的问题以及收获

（1）相比较于 Pthread，MPI 的编程比较容易，所考虑的事情也比较少，可以让我专注于目标问题的划分上。但是个人猜测 MPI 的效率会低于 Pthread 的效率，采用相同方式的情况下，由于 MPI 需要在进程之间通信，相比于内存共享会消耗更多的时间。

（2）由于实验所要求的矩阵大小较大，如果使用程序的栈空间容易造成栈溢出，会报错导致程序无法正常运行，所以采取 malloc 使用堆空间来解决，但是这又导致一个问题，malloc 的使用需要指针来实现，容易造成野指针的出现导致程序访问系统中禁止访问的位置。

（3）MPI 函数的使用非常方便，尤其是 Bcast 这个函数，可以快速将 0 号进程特定位置的数据拷贝到其他进程中。还有 MPI_Send 和 Recv 函数，实现数据交换也很容易。

（4）在思考如何减少进程间数据通信的时候，发现了这个实验所采取的更好的方法。打算利用周末时间，将实验二的算法更改一下，比较两种并行方式的优劣。

4.2 对不同类型并行计算方式的理解与分析

并行计算的主要问题是如何把一个串行程序并行化，书本上给出了求前缀和的一个例子，我觉得那个优化方式真的非常巧妙，这种应该是比较困难一点的并行化思路。

这一次的实验是计算矩阵的幂次，考虑过类似于用加法的方式，比如先计算矩阵的 10 次幂，然后再通过剩下的线程将结果合并，反复执行这个过程最后得到目标结果。但是实现上面较为困难，首先是控制幂次，采用这个方法计算，每一级线程所计算的矩阵幂次将会是指数级增长，为每个线程分配任务比较苦难，而且对存储空间的要求也比较高。

任务分解在这次的实验中不是非常适用，所有的线程执行的都是矩阵相乘的算法，只是线程之间的所使用的数据不同。不同的问题所适合的并行计算方式有所不同。本实验就比较适合使用域分解的方式来解决。

并行计算的作用其中之一就是将循环展开，这次实验给出的伪代码中包含着四重循环，第一层控制幂次，不是很好展开，而后面三层用于计算每一次矩阵相乘，可以使用多线程，每一个线程计算结果中的一个数据，从而将三重循环展开成一重循环，而降低计算时间，提升计算效率。

五、课程总结

5.1 课程授课方式有助于提升学习质量

课程的安排非常清晰，每堂课都有预习内容，理论课和实验课相结合，使得学习更加高效。在理论课上，老师会系统地讲解相关知识，而每节课后都有一次作业，有助于巩固理论知识，也为后续实验提供了复习的机会。相比于先学理论再进行实验的课程安排，这种交替进行的方式更合理，避免了在实验时忘记理论知识的尴尬情况。

实验课的安排也十分贴合理论课的内容，能够让我更加直观地感受到理论知识在实际操作中的应用。通过实验，我不仅加深了对理论的理解，还提高了解决问题和动手能力。这种理论和实践相结合的教学方式，让我学到的知识更加系统和全面，也让学习过程更加生动有趣。

课程的安排十分合理，理论与实践相辅相成，正反馈及时到位。这种教学方式不仅提高了学习效率，也增强了我对所学知识的理解和记忆。

5.2 不合理之处及建议

对于实验报告的内容需要进行调整，可以考虑两种方式：要么只给出大纲，要么将细节写清楚。这样可以让学生更清晰地了解实验报告的要求和结构，同时也有助于他们更好地理解实验内容和目的。如果只提供大纲，学生可以更自由地根据大纲的指引来完成报告，但需要确保大纲涵盖了实验报告所需的所有内容。另一方面，如果提供了详细的内容要求，学生则可以更加系统地编写实验报告，确保每个部分都得到了充分的阐述和解释。

实验可以定期查收，这有助于避免学生最后一刻的赶工。定期查收实验报告可以促使学生按时完成实验，并在过程中及时发现和纠正问题，从而避免最后一刻的赶工和不完整的报告。通过定期查收，老师可以及时给予学生反馈，指导他们在实验报告的撰写过程中改进和提高，从而提高学生的学习效果和报告质量。同时，定期查收还有助于监督学生的学习进度，确保实验课程的顺利进行。

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	熟悉天河环境，初步修改代码	并行程序的编译运行，Pthread 库	利用智慧树上的代码，在天河实验环境下编译运行，熟悉 terminal 指令。结合课程上对于 Pthread 库的讲解理解程序并进行初步修改
2	编写 sh 脚本，测试程序	sh 脚本语法，并行计算设计	利用 sh 进行程序运行，可以高效得到实验数据。结合并行计算性能分析对实验结果进行分析。
3		并行计算类型	时间并行的流水线方式简单易实现，但提升计算速度有限。 空间并行的域分解和任务分解能够处理复杂问题，但需要考虑优化和任务分配等问题。
4		SIMD，双核与超线程，存储访问模式	特性要求：数据结构：连续存储、数据对齐、元素大小相同。算法：可并行性、各部分相互独立、负载均衡。 双核：真实的两个核心；超线程：单核心模拟两个核心，可能出现性能问题。 SMP：UMA 均匀存储访问。 MPP：NUMA 非均匀存储访问。 Cluster：分布式存储访问。
5		线程与进程区别，竞态条件	线程是系统资源分配的基本单位，进程是 CPU 调度的基本单位。线程不独立拥有资源，但可以访问隶属进程的资源；进程拥有系统资源。创建线程消耗资源较少，多线程更轻量化。线程共享资源和打开文件，而进程相对独立，无

			法互相访问。线程切换不会引起进程切换，节省系统开销。 竞态条件出现情况：多个线程同时读写共享数据。
6		Amdahl 定律与 Gustafson 定律	对于并行计算性能的大致分析与预测，但是两种算法都具有局限性。只是用来大致的预测与验证，以及解释一些实验现象。
7		OpenMP	通过“编译制导语句”将一个串行程序快速地转变为并行程序。可以看作是一个工具而不是一种独立的语言。
8		GPU 编程与 CUDA	线程，线程块，以及网格构成了 GPU 的三层结构，通过 CPU 控制，实现异构计算。可以让 GPU 中的多个线程块同时运算。
9	矩阵乘法		熟悉矩阵相乘的计算机实现方法，确定矩阵幂次计算的划分方式，以及具体如何划分，根据任务书提示编写程序。
10	Lab2 算法性能分析与优化		不断地对算法进行优化，针对伪共享以及线程开销进行优化，测试并得到实验结果。 体会到数据规模（任务量），以及线程数量之间的关系。
11		MPI、集群、与作业管理系统	集群与 MPP： 集群：通用操作系统连接节点，独立机器组成。 MPP：定制组件，高速网络连接处理器，共享操作系统。 线程跨节点限制： 进程在节点内运行，线程共享进程资源，不能跨节点。 作业管理系统： 管理资源、调度任务、监控运行、提高利用率。 MPI 六调用： Init/Finalize：初始化/释放 MPI 环境。 Comm_size/Rank：获取进程数/线程 ID。 Send/Recv：发送/接收消息。
12		MPI 通信进阶	非阻塞通信的优缺点： 优点：避免性能资源浪费，不需要等待通信完成。 缺点：后续依赖通信消息的计算需要确保通信成功，并需要额外判断通信完成的语句。 组通信操作及场景： 一对多（广播）：MPI_Bcast。场景：节点 0 有大量数据需要发送给其他所有节点。 多对一（归约）：MPI_Reduce。场景：每个节点有局部计算结果，需要汇总成一个全局结果。

			同步：MPI_Barrier。场景：需要等待所有进程计算完成后才进行下一步操作，如矩阵乘法。 MPI 消息中使用标签的原因： 区分不同进程发送的消息，避免消息混乱，确保正确处理每个消息。 MPI 消息传递中可能出现死锁的情况及避免方法： 当数据发送和接收都采用阻塞方式时，可能因为互相等待而产生死锁。 避免死锁的方法是将发送或接收操作中的其中一个改为非阻塞方式，或者确保发送和接收顺序的一致性。
13		MPI 分析比较	注意事项：捆绑发送接收操作需考虑消息大小、内存使用、通信重叠和错误处理等。 自定义数据结构：MPI 支持自定义数据结构，提高灵活性和通信效率。 MPI 与多线程：MPI 消息传递，多线程共享内存；MPI 通信复杂，多线程简单；MPI 可跨节点，多线程局限于单节点。 多层次并行架构：提高性能、缓解功耗问题、适应多样性任务需求。 MPI+ 多线程通信：MPI_THREAD_FUNNELED、MPI_THREAD_SERIALIZED、MPI_THREAD_MUTIPLE。
14	Lab3 实验代码编写调试		Source ~/.bashrc //更新环境变量 Module 加载 openmpi 尝试利用 lab2 的代码修改，无果。 尝试 MPI 函数。 周末完成代码编写
15	Lab3 实验报告撰写		收集实验数据，分析并给出报告