《并行计算》

结课报告

姓名： 陈秋澄

学号： 3022244290

时间： 2024年5月6日

成绩：

评分：

审核：

天津大学 智能与计算学部

2024年

1. **实验内容概述**

**1.1 并行算法**

我们一共有4个实验，实验一为多线程计算sin值，利用正弦函数的泰勒级数展开式的计算结果，用PThread并行化实现对sin x的求值。后3个实验均为矩阵幂的计算，但实现方式不同，其中：

实验二中我们利用矩阵分块等思想，用 PThread 并行化实现矩阵幂的多线程并行计算。我们将PThread线程库的基本使用方法，包括线程的创建、同步和互斥等知识融入到实验中。在此基础上，我们设计合理的算法，将矩阵幂的计算过程划分为多个线程并行执行的任务，分块计算以实现矩阵幂的快速计算。

实验三中我们利用矩阵分块等思想，使用MPI库实现矩阵幂的多进程并行计算。我们需要熟悉MPI库的基本使用方法，包括进程的创建、数据传输和进程同步等。我们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，每个进程处理一个子块，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。

实验四要求我们使用MPI+OpenMP库实现矩阵幂的多级并行计算。对于代码部分，我在实验三代码的基础上进行了简单的修改。包括但不限于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。

**1.2 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。其中我们的计算集群是由国家超级计算天津中心提供的国产飞腾处理器。

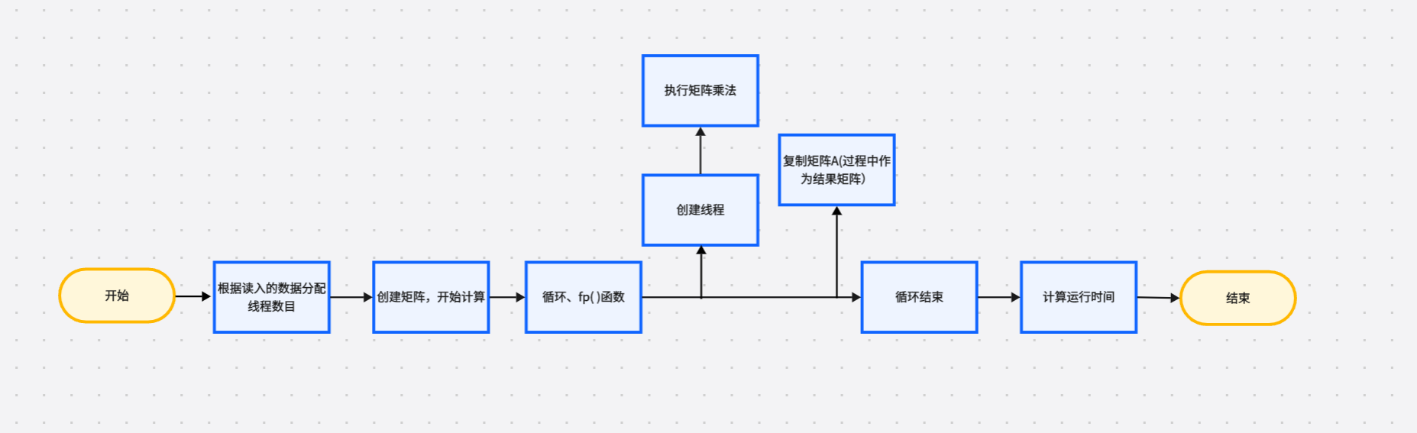
* 1. **数据分析要求**

根据题目要求，实验2、3、4中我们分别测试了不同线程数或进程数、计算规模下的运行时间，计算得到加速比与效率，绘制图表、进行分析。

**二、并行算法分析设计**

**2.1 PThread多线程算法设计**

实验2主函数流程图：



主要函数伪代码如下：

void \*func(void \*arg)// 子线程执行函数\*func

    开始行 = 参数指向的整数

    结束行 = 开始行 + L

    for i 从 开始行 到 结束行-1 执行

        for j 从 0 到 N-1 执行

            for k 从 0 到 N-1 执行

                结果矩阵[i][j] += 矩阵A[i][k] \* 矩阵B[k][j]

    返回 空指针

void fp()// 多线程并行计算过程fp()

    索引数组[线程数量]

    L = N / 线程数量

    for i 从 0 起步，每次增加 L 执行

        索引数组[j] = i

        如果 创建线程(线程标识符[j], NULL, func, 索引数组+j)失败 那么

            报错并退出

        j++

    for 每个创建的线程 执行

        等待线程结束

int main(int argc,char \*argv[])// 主函数逻辑

    调用 生成随机矩阵\_A()

    开始计时

    for 次方次数 i 从 1 到 Power-2 执行

        调用 fp()

        将 结果矩阵 的值复制回 矩阵A，重置 结果矩阵

    再次调用 fp()

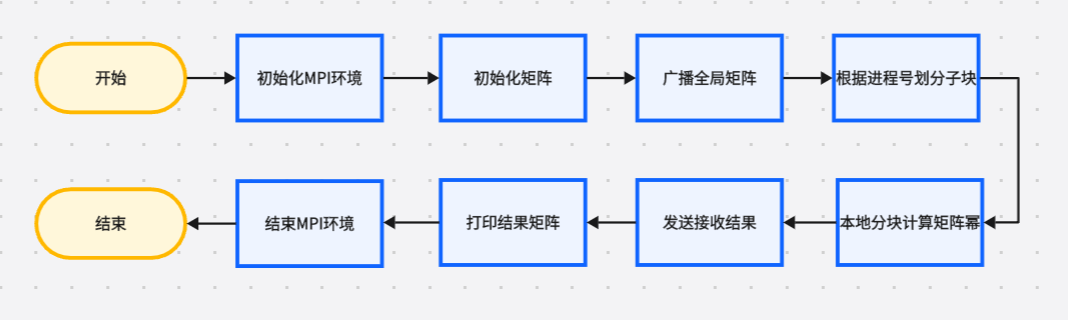
    计时结束，输出运行时间

    输出 结果矩阵的内容

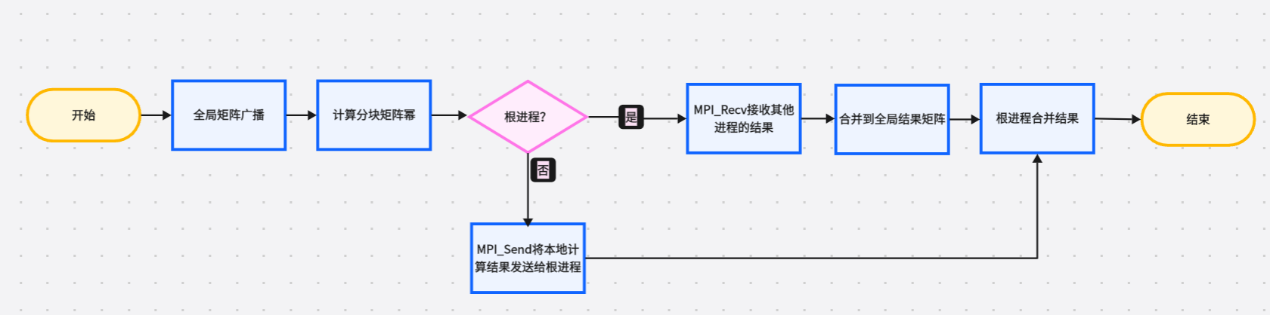
    退出线程

**2.2 MPI并行算法设计**

主函数流程图：



进程通信的流程图如下：



思路及伪代码如下：

①初始化MPI环境。

②获取进程编号和进程总数。

③在根进程中，如果进程编号为0，则初始化一个16x16的随机矩阵，并输出该矩阵。

④使用MPI的广播功能，将根进程中的矩阵发送到所有其他进程。

⑤计算每个进程需要处理的矩阵行的范围。最后一个进程处理剩余的行。

⑥在每一个进程中，调用`matrixPower`函数计算本地矩阵的幂。

⑦如果进程编号不是0，则将本地计算的结果发送到根进程。

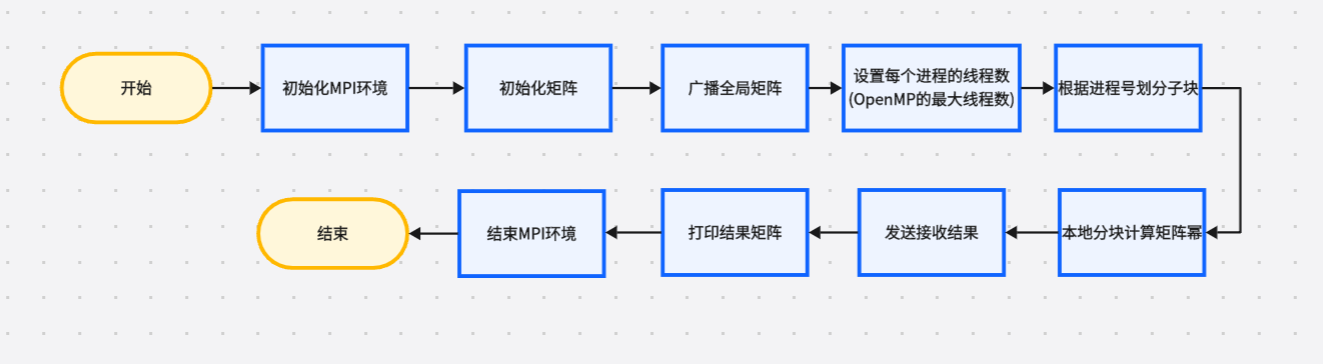
如果进程编号是0，则接收其他进程发送的局部结果，并合并成最终结果矩阵。

⑧打印出最终结果矩阵。

⑨结束MPI环境

**2.3 MPI+OpenMP混合并行算法设计**

主函数流程图：



思路及伪代码如下：

1. 初始化MPI环境
2. 获取当前进程编号(rank)和进程总数(size)
3. 如果 rank 等于 0，则初始化 matrix 为随机值，并打印矩阵
4. 使用MPI广播将 matrix 分发给所有进程
5. 设置每个进程的线程数 num\_thread 为OpenMP最大线程数
6. 计算每个进程负责的矩阵行范围：

rows\_per\_process 表示每进程处理的行数

start\_row 和 end\_row 定义了当前进程处理的矩阵行区间

1. 定义函数 matrixMultiply，用于执行矩阵乘法，内部使用OpenMP指令并行化循环
2. 定义函数 matrixPower，计算矩阵的幂
3. 在每个进程中：

为本地计算分配内存 local\_result

调用 matrixPower 计算指定行范围内的矩阵幂

1. 非根进程向根进程发送local\_result
2. 如果rank等于0（根进程）：

收集所有进程的 local\_result 并合并至 result

打印最终计算得到的 result 矩阵

1. 终止MPI环境

**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。

1. 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

1. 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

1. 编译环境

GCC 9.3.0; OpenMPI 4.1.1

1. 作业管理系统

SLURM 20.11.9

**3.2 实验数据综合分析**

1. 图表分析

三种并行实现方式的运行时间、加速比、效率与并行度之间的表格如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 运行时间 |  |  |  |  |  |
| **n type** | **1** | **2** | **4** | **8** | **16** |
| PThread | 0.941 | 0.91 | 0.89 | 0.883 | 0.9 |
| MPI | 1.51 | 1.548 | 1.687 | 1.764 | 1.704 |
| MPI+OpenMP | 1.519 | 1.582 | 1.633 | 1.638 | 1.633 |
| 加速比 |  |  |  |  |  |
| **n type** | **1** | **2** | **4** | **8** | **16** |
| PThread | 1.000 | 1.034 | 1.057 | 1.066 | 1.046 |
| MPI | 1.000 | 0.975 | 0.895 | 0.856 | 0.886 |
| MPI+OpenMP | 1.000 | 0.960 | 0.930 | 0.927 | 0.930 |
| 效率 |  |  |  |  |  |
| **n type** | **1** | **2** | **4** | **8** | **16** |
| PThread | 1.000 | 0.517 | 0.264 | 0.133 | 0.065 |
| MPI | 1.000 | 0.488 | 0.224 | 0.107 | 0.055 |
| MPI+OpenMP | 1.000 | 0.480 | 0.233 | 0.116 | 0.058 |

相关图表如下：

根据加速比与并行度的折线图，分析得出：

PThread: 随着并行度的增加，使用PThread时，加速比起初有所增长，在并行度为4时达到最大值1.066，之后开始下降，这表明在低并行度时，线程之间的负载均衡和通信开销相对较小，因此能较好地利用多核资源。然而，当并行度进一步增加，上下文切换、内存访问冲突以及同步开销可能导致了加速比的降低。特别是当并行度达到16时，加速比几乎接近1，意味着增加更多线程不再带来明显的性能提升，甚至可能因为过度的并行开销而造成性能损失。

MPI: MPI的加速比在并行度从1增加到2时略有下降，之后持续下降至并行度16时的0.886。这种模式说明虽然MPI能够有效利用多进程，但在高并行度下效率受到一定影响，可能是由于进程间通信成本增加。

MPI+OpenMP: MPI+OpenMP混合模式的加速比整体保持相对稳定，虽然随并行度增加有轻微波动，但变化不大，保持在0.927到0.930之间，这表明该模式在高并行度下相对较好地控制了通信开销和负载平衡。此外，总结上机实验及相关资料，OpenMP在单个节点上提供了细粒度的并行，而MPI负责跨节点的通信。这种策略在一定程度上缓解了通信瓶颈，尤其是在并行度较高时，效率下降相比纯MPI更为平缓。然而，即便如此，效率仍然随着并行度增加而下降，表明即使是在优化的混合模式下，也无法完全避免并行计算固有的开销问题。

根据效率与并行度的折线图，分析得出：

PThread: PThread的效率随并行度增加而迅速下降，特别是在并行度为16时效率仅为0.065，这说明线程数增加后，资源争抢和管理开销显著降低了每个线程的工作效率。

MPI: MPI的效率也呈现下降趋势，且在并行度16时达到最低点0.055，表明虽然MPI擅长处理分布式任务，但在高度并行化时，尽管总的计算工作被更多进程分担，但每个进程的实际工作量减少，加上通信开销，导致效率降低。

MPI+OpenMP: 相比之下，MPI+OpenMP的效率虽然也随并行度增加而下降，但下降幅度较前两者平缓，保持了相对较高的效率水平，即使在并行度为16时仍有0.058的效率，显示出较好的混合并行策略效果。

1. 总结

PThread: PThread的效率随着并行度增加而快速下降，说明增加线程并未能线性地提高计算能力，反而因线程管理成本增加导致整体效率降低。特别是在高并行度下，效率极低，意味着大部分线程在等待资源或进行无谓的通信。

MPI擅长处理跨节点的并行计算，初始的下降可能是因为初始化通信开销的分摊以及任务划分的不均。随着并行度的增加，网络通信成为瓶颈，特别是在高并行度时，大量的进程间通信导致了效率的急剧下降。尽管MPI能够分配任务到不同的节点上，但过高的并行度使得通信成本超过了计算节省的时间。

MPI+OpenMP的加速比也随着并行度的增加而下降，但下降的速率相对较缓。这表明结合了MPI和OpenMP的混合并行方式在一定程度上能够平衡进程间通信和线程级并行的开销。其效率同样随着并行度的增加而降低，但相对于单独使用MPI或PThread，其在高并行度下的效率略有优势。

1. 三种并行方式适用问题分析

PThread：

PThread适用于共享内存模型的并行计算，适合解决那些可以划分为多个独立任务，且任务间通信较少的问题。它特别适合于多核处理器的计算机，因为线程可以在同一个物理核心的不同超线程上运行，从而减少线程间的通信开销。

因此，在数据集较小，计算密度高，且不需要大量数据交换的情况下，PThread可以提供较好的性能。

MPI：

MPI适用于分布式内存模型的并行计算，适合解决那些可以划分为多个独立任务，且任务间通信较多的问题。它通常用于多处理器计算机集群或超级计算机。

因此，在需要处理大量数据，且数据分布在不同的计算节点上时，MPI可以通过高效的进程间通信来优化性能。例如，大规模的科学模拟和计算就非常适合使用MPI。

MPI+OpenMP：

MPI+OpenMP结合了MPI和OpenMP的优势，适用于既需要处理大量数据，又需要在单个计算节点上进行多线程并行的情况。它适合于那些可以同时在任务间和任务内部进行并行化的问题。

因此，在大规模并行计算中，尤其是在多核节点的集群上，MPI+OpenMP可以同时利用节点内的多核处理能力和节点间的并行计算能力，提高整体的计算效率。例如复杂的大规模物理模拟和数据分析任务可以通过这种方式获得优异的性能。

**四、实验总结**

**4.1 问题、解决与收获**

1. 问题与解决：

实验二中遇到的知识型错误：

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。

实验三中遇到的理论与实际“不符”类问题：

每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的效果。

实验四中遇到的知识型错误：

（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） 我锁定一个进程和一个线程，发现OpenMP的运行时间比串行代码要短。后来通过查找资料，明白了在指定一个线程时，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。

1. 收获与体会：

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

**4.2 对不同类型并行计算方式的理解与分析**

将实验4不同进程数和线程数的各参数进行对比，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。

总结三个实验，PThread适用于共享内存模型的并行计算，适合解决那些可以划分为多个独立任务，且任务间通信较少的问题。它特别适合于多核处理器的计算机，因为线程可以在同一个物理核心的不同超线程上运行，从而减少线程间的通信开销。因此，在数据集较小，计算密度高，且不需要大量数据交换的情况下，PThread可以提供较好的性能。

MPI适用于分布式内存模型的并行计算，适合解决那些可以划分为多个独立任务，且任务间通信较多的问题。它通常用于多处理器计算机集群或超级计算机。因此，在需要处理大量数据，且数据分布在不同的计算节点上时，MPI可以通过高效的进程间通信来优化性能。

MPI+OpenMP结合了MPI和OpenMP的优势，适用于既需要处理大量数据，又需要在单个计算节点上进行多线程并行的情况。因此，在大规模并行计算中，尤其是在多核节点的集群上，MPI+OpenMP可以同时利用节点内的多核处理能力和节点间的并行计算能力，提高整体的计算效率。

**五、课程总结**

**5.1 本学期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。

平时的12次作业内容包括对课上重点知识的回顾与讨论，有助于我们进行学习与复习，整理课上所学，查阅相关资料，对课程有更深入而全面的理解。

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。此外，老师们在平时以及上机课上对我们的问题进行答疑，为我们答疑解惑，有助于我们对知识的理解与实现。

结课大作业让我们又一次对课程有一个全面的回顾与思考，对实验进行横向对比总结；实验报告及PPT的编写让我回顾了知识，视频的录制让我对许多知识加深了印象，也提高了表达能力。

**5.2 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，最新的算法介绍有时跟不上，会使得在真正应用于实际时这种落后效应更加明显，希望老师们可以及时更新授课内容，更换新的课件与材料。

此外，部分实验的难度较大（比如实验3），而且计算机科学技术专业这学期的课程数量多难度大、多门课程都要完成上机实验，因此课业压力较大，在老师给定时间内完成上机实验作业有一定的压力，希望老师们可以适当放宽作业期限，给我们较为充足的时间去实现代码、掌握知识。

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 | 矩阵计算中的乘法（PThread并行化实现） | 向量化编程、循环展开、线程同步 | 将矩阵元素的计算任务分散到多个线程中，利用向量化编程和循环展开来提高计算效率，同时通过线程同步机制来处理共享资源（如矩阵元素）的访问，避免竞态条件。 |
| 3 | 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。 |
| 4 | 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。 |
| 5 | 矩阵幂计算任务分解 | 矩阵分块 | 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。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。 |
| 6 | 多进程计算矩阵幂（MPI） | MPI 中的六个基本调用及用途 | 学习了MPI 中的六个基本调用及用途。MPI\_Init：MPI 程序的初始化；  MPI\_Finalize：MPI 程序的终止；  MPI\_Comm\_size：获取通信器中的进程数；MPI\_Comm\_rank：获取进程在通信子中的标识号；  MPI\_Send：发送消息；MPI\_Recv：接收消息。 |
| 7 | 多进程计算矩阵幂（MPI） | MPI环境中线程间通信常用函数 | 1）点对点通信：MPI中最基本的通信方式。通过可以将一个线程中的数据发送到另一个线程或进程的MPI\_Send和用于接收这些数据的MPI\_Recv等函数，线程可以在同一进程内的不同线程之间，或者在不同进程之间进行通信。  2）广播：MPI\_Bcast函数允许线程广播数据到同一进程内的所有线程，或者广播数据到其他进程中的所有线程。 |
| 8 | 多级并行化计算矩阵幂 | MPI+OpenMP 环境中实现进程间通信 | 可以使用#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。 |
| 9 | 多级并行化计算矩阵幂(MPI+OpenMP) | MPI+OpenMP的协同工作 | MPI是一种消息传递接口，适用于分布式内存计算，而OpenMP适用于共享内存计算。在这次的后三个实验中，这两种接口可以结合使用，其中MPI可以帮助将大量的数据分发到不同的节点上，而OpenMP可以在每个节点上对这些数据进行并行处理；使用MPI来进行全局数据分布和节点间的通信，同时使用OpenMP来进行局部的共享内存并行计算。 |
| 10 | 多级并行化计算矩阵幂与多线程计算矩阵幂 | OpenMP与Pthread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。 |
| 11 | 多级并行化计算矩阵幂(MPI+OpenMP) | OpenMP在共享内存上的并行化 | 在共享内存模型中，多个处理器核心共享同一块内存空间，这使得数据共享变得容易，但也要求我们必须处理数据竞争和竞态条件等问题。OpenMP提供了数据并行、任务并行以及工作组（simd）等多种并行化机制。 |
| 12 | 不同方式实现矩阵幂计算 | 并行计算基本概念 | 在代码实现中体会任务并行性、数据并行性、分布式计算、共享内存、消息传递、并行算法等基础概念，对其有更全面的理解。 |
| 13 | 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。 |
| 14 | 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。 |