并行计算课程
实 验 报 告

报告名称：	MPI+OpenMP 计算矩阵幂
姓　　名：	陆子毅
学　　号：	3022206045
联系电话：	15262559069
电子邮箱：	3022206045@tju.edu.cn
填写日期：	2024 年 4 月 26 日

2024 年制

一、实验内容概述

本实验旨在利用 MPI+OpenMP 计算矩阵幂，以加深对并行算法的理解和应用。通过采用 MPI+OpenMP 多级并行技术，将规模为 n*n 的数据分配给多个进程进行计算，以提高计算效率。实验输入包括进程数量和幂次数，要求采用 MPI+OpenMP 并行化实现。

在实验中，我们将设计并实现一个并行算法，利用 MPI+OpenMP 计算矩阵幂。该算法将数据划分为多个子任务，并由不同的进程并行执行。通过合理的任务划分和线程管理，我们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。

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

（1）分析程序可并行化部分

可并行化的地方为独立且互不依赖的部分，也就是每一次矩阵乘法中各个位置值的计算。比如 result[0][0]的值就可以和 result[1][0]的值同时计算。接下来考虑最大并行化程度，由于矩阵幂次的计算需要依赖到上一次的计算结果，所以第 i 次幂和第 i+1 次幂就不能再并行计算了。这里还有一种并行化方法但是没有实现：将 N 次幂除以进程数量，假设进程数量为 process_num，每个进程计算矩阵的 N/process_num 次幂，最后由 0 号进程接着计算 N 次幂，但是这样子做容易造成进程之间的任务分配不均匀，而且实现较为繁琐，故没有做。

（2）确定通信需求

进程之间通信主要是每一乘法计算完成之后，需要将结果作为乘数进行下一轮乘法。但是由于矩阵乘法的一些特殊性质，矩阵行与行之间的计算是互不依赖的，独立的，所以可以避免在每次矩阵乘法完成以后进行通信，减少了不必要的通信开销，最后的通信只是在所有进程都计算完成之后，以行为单位将结果汇总给 0 号进程。

三、实验数据分析

3.1 实验环境（CPU 型号与参数、内存容量与带宽、互联网络参数等）

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

通过编写 sh 脚本自动重复运行程序并且过滤出数据自动计算平均值。

```
#!/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++ -fopenmp -o lab4.o lab4.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 $size -n $size ./lab4.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
```

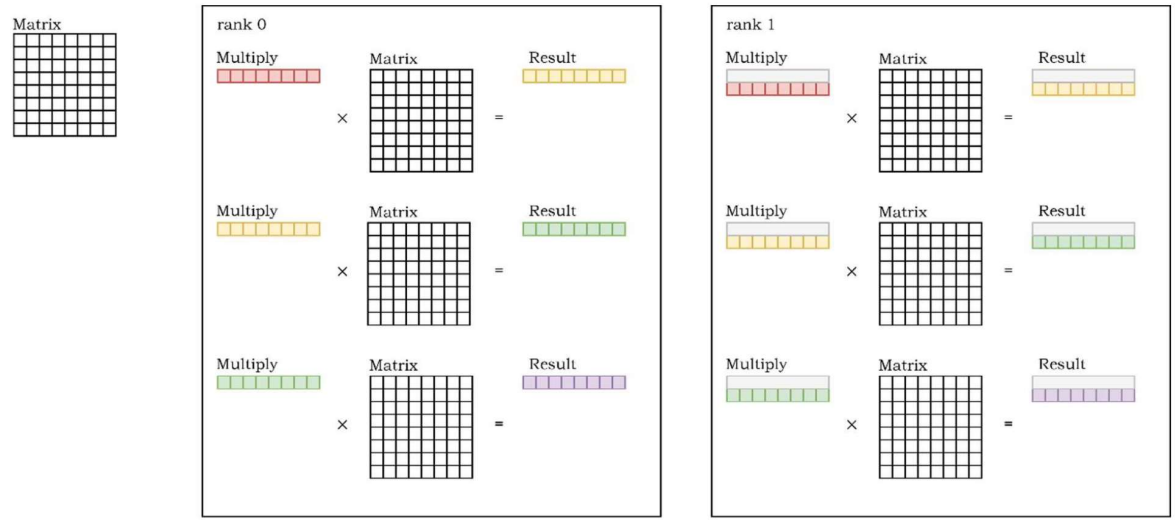
3.2.2 OpenMP+MPI 多级并行计算的设计

在实验三原有的程序框架上，加入 OpenMP 线程并行的编译制导语句，让每个进程自动进行线程并行化计算。

```
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++)
    {
#pragma omp parallel for
        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;
        }
#pragma omp barrier
        for (int i = 0; i < MATRIX_SIZE; i++)
        {
            multiply->data[row][i] = temp[i];
            result->data[row][i] = temp[i];
        }
    }
    free(temp);
}
```

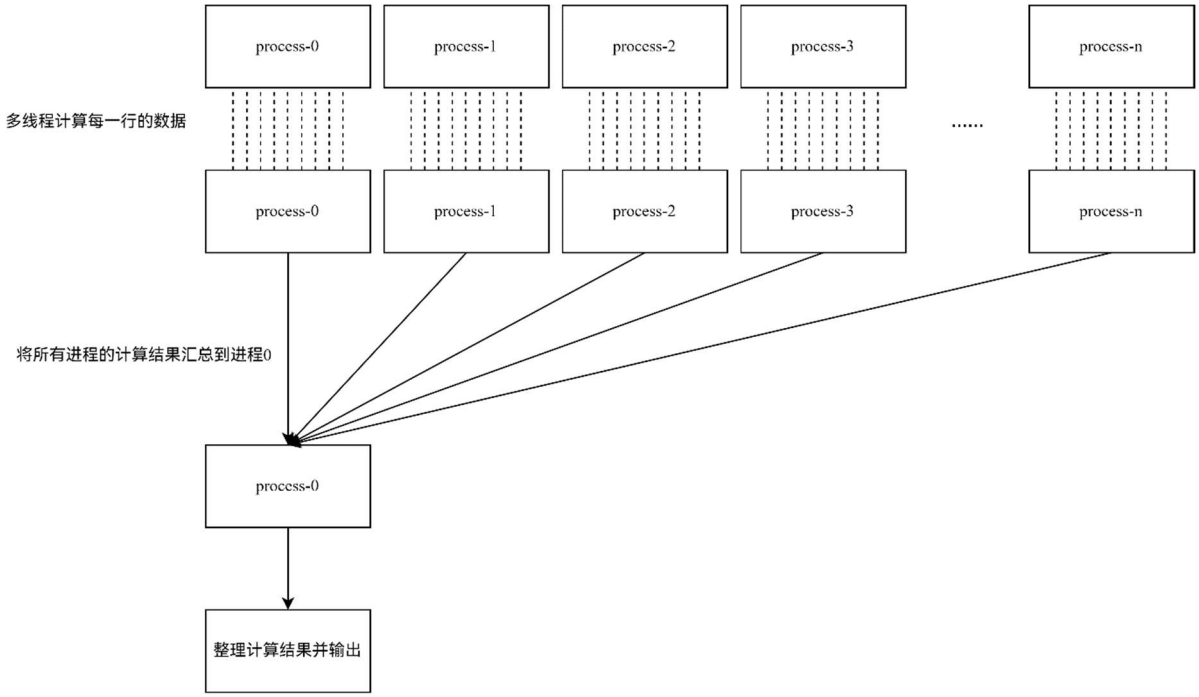
可以多级并行的部分为每一行每个元素的计算可以并行执行，遂直接在 for 循环之前加上编译制导语句，来让下面这一个 for 循环利用多线程并行执行，但是在每次计算之后需要设置 barrier 来使每个线程同步，完成一行的计算，因为下一次计算依赖于每个线程计算完成的最终结果。

每个进程所被分配的任务和实验三一样，如下图所示：



程序其他部分的大致结构和实验三的相同

对于 OpenMP 的多线程，根据设计的多级并行计算算法，每行的数据由多个线程计算。

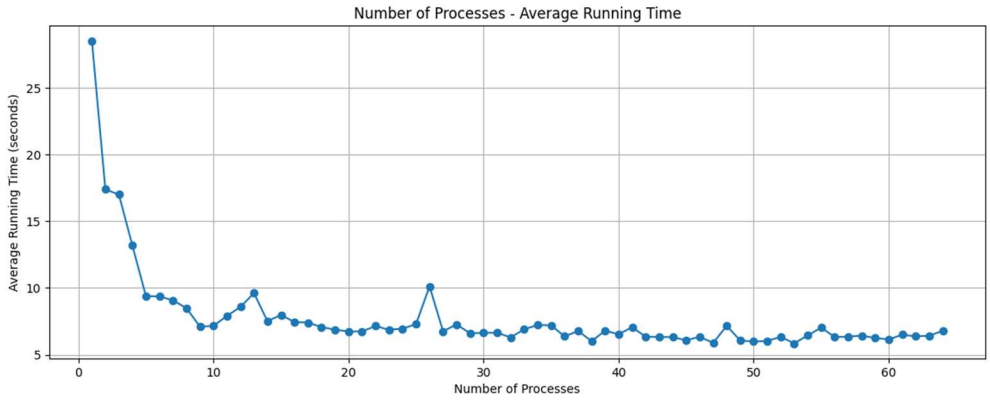


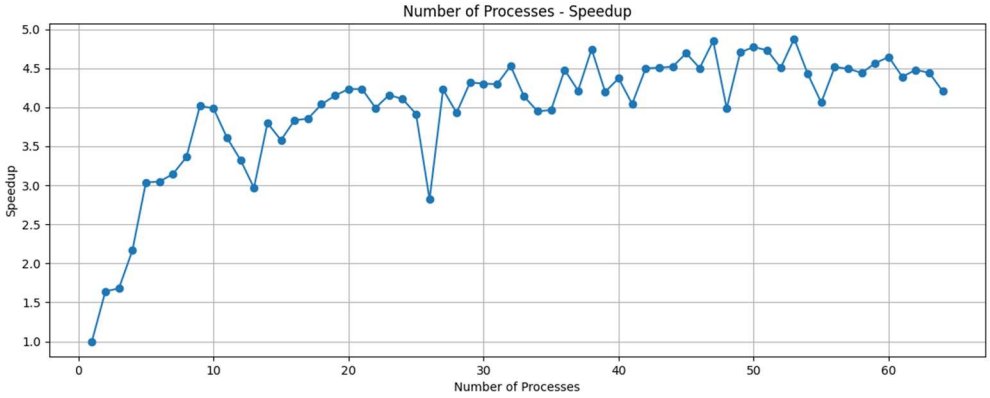
实验四的整体代码和实验三相差无几，最主要的就是在每一行的计算过程中添加了编译制导语句，实现了多级并行。

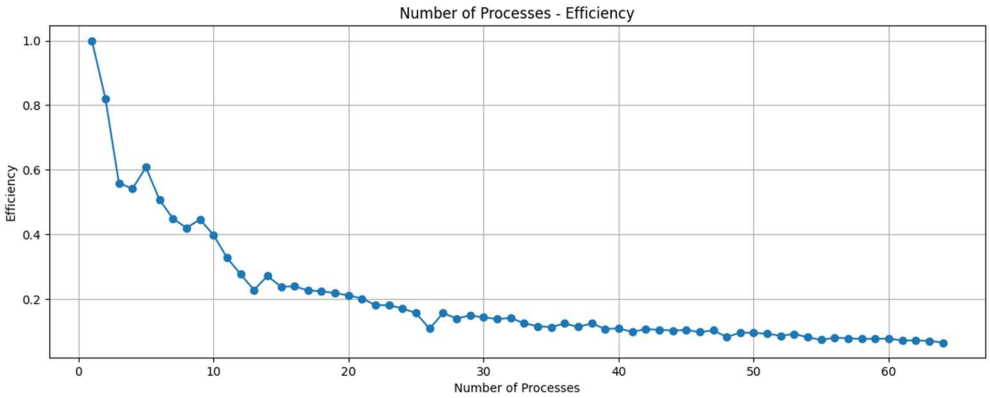
但是实验的结果出乎预料，对比实验三，实验四所采用的多级并行带来的性能优化非常明显。

3.3 OpenMP+MPI 多级并行计算实验结果与数据分析

下面是实验数据以及进程数量-运行时间，进程数量-加速比，以及进程数量，效率的图像。







首先是与未使用 OpenMP 多级并行的实验三进行比较，在相同数据规模（1000x1000）以及相同幂次（15）的情况下，仅使用 MPI 的计算时间约为 428s，使用 MPI+OpenMP 的运行时间约为 28s。综合其他数据分析，OpenMP 将原有的运行时间降至 6%~50%之间，优化的效果波动较大，但是也是具有规律的，随着进程数量的增加，OpenMP 的优化效果也逐渐下降。分析原因，当进程数量较少的时候，一个进程所要运算的数据量比较大，使用线程并行能够带来较大的提升，随着进程数量的增加，每个进程所需要运算的数据减少，此时多线程带来的优化就不那么明显了。

其次观察加速比，最后稳定在大约 4.5 的位置，说明程序的串行部分和并行部分的比重是 0.8：0.2，串行程序占了大约占了整个程序的 2/9，大概能反映出程序的一个基本情况。

最后，根据实验结果综合分析，在进程数量在 1-10 的时候，MPI+OpenMP 对程序运行时间的优化很明显，但是随着进程数量的增加，所带来的提升逐渐减少。这可能与数据规模由一定关系，但是放到具体实践中，使用 10 个进程就可以获得较好的效果，可以避免资源的浪费。

四、实验总结

4.1 遇到的问题以及收获

（1）第一个问题就是 OpenMP 的使用，一开始觉得这是一个非常复杂的东西，但是具体使用的时候发现，相较于 Pthread，OpenMP 带来了极大的便利，仅仅使用一些编译制导语句就可以实现需要很多 pthread 函数才能实现的多线程。但是，OpenMP 的主要作用还是体现在将串行程序并行化，也就是说，要先在串行程序中找到可以并行化的部分，然后再使用 OpenMP，如果是存在先后顺序的任务划分，使用 OpenMP 来处理可能会有一些困难。

（2）和实验三的问题类似，由于 OpenMP 对程序的性能提升比较大，所以在数据规模较小的情况下，无法明显观测到 OpenMP 的加速效果，所以又一次提高了数据规模。

4.2 对不同类型并行计算方式的理解与分析

（1）Pthread

线程并行，在一个节点上运行，只有一个进程，依靠线程的并行来实现并行计算，优点是线程之间共享内存，对于内存的访问比较方便，但是同时也要注意伪共享的数据一致性问题，总体来说是一个比较基础的并行方式，在并行计算之外的领域也有引用。

（2）MPI

进程并行，这种并行方式通俗来说是一段程序交给不同的进程去执行，由于在操作系统中，进程是独立分配资源的单位，所以，各个进程之间需要通过总线或者网络来交换数据，不像 Pthread 一样可以直接访问内存来共享和同步数据。使用进程并行，需要使用 MPI 的一系列进程通信函数，函数提供了各种数据交换的接口，可扩展性很强，课上只是介绍了基本的 6 个函数，但是还有其他功能更加复杂的函数，可以实现更加多样的功能。

（3）MPI+OpenMP

多级并行计算，也就是在进程并行的基础上，对于每一个进程又使用多线程的技术，这样一来并行的效率又进一步提高，而且是叠加型提高而不是线性提高。实现较为容易，而且根据实验结果来看，最后的优化效果也很好。

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 。

应该允许学生有一次重新提交作业或者实验的机会，学生提交了错误的报告，或者报告里面忘记加入一些文件，或者想要丰富一下自己的报告，作为老师应该给一个机会，可以有一次重交是合理的，而不是连交空白报告都不允许重交，我认为，这并不是一个原则性错误，就算犯了错，应该也需要给学生一个纠正的机会，谁能不犯错呢，社会应该是包容的而不是死板的，并不是说犯了错就不能再去纠正，可以给一次重新提交的机会，但是分数上可以适当减少，这样是比较合理的。

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		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
16		MapReduce	MapReduce 适于处理大规模数据的原因： 分布式处理：将任务分解为多个小任务，利用多台计算机的计算能力进行并行处理。 容错机制：自动处理节点故障，重新调度任务，保证高可靠性。 数据局部性：计算任务在数据存储节点附近执行，减少数据传输，提高效率。 可扩展性：通过增加计算节点线性扩展处理能力。 MapReduce 采用 key-value 数据结构的原因： 简化并行计算：通过键对数据进行分组，便于并行处理和聚合。 数据分片：将数据按键分割，分配到不同节点独立处理。 适用性：key-value 结构也适用于串行编程，如哈希表或关联数组。 Hadoop 封装的 MapReduce 并行计算功能： 分布式存储：使用 HDFS 存储大规模数据集，支持分布式存储和处理。 资源管理：YARN 管理集群资源分配和调度，提高资源利用率。 任务调度：JobTracker 和 TaskTracker 管理作业调度和任务执行。 容错性：处理节点故障，重新分配任务，确保作业连续性。 数据复制和备份：HDFS 数据块复制，提高数据可靠性。
17		并行计算的关键概念和设计方法	计算密集型与数据密集型算法的并行计算环境要求： 计算密集型：需要强大的计算资源，如多核处

			理器或 GPU，以及有效的任务调度和负载均衡机制。 数据密集型：需要高速的数据传输能力和大容量的存储空间，以及优化的数据分布和访问策略。 并行求前缀和算法与数组求和算法的关系： 数组求和：将数组分割，多处理器独立计算子数组和，再合并结果。 前缀和算法：计算每个元素的累积和，涉及处理器间的通信和数据交换。 并行程序设计方法： 直接并行化：将串行算法改造为并行算法，如数组求和算法的并行化。 从头设计：根据问题属性设计并行算法，如求前缀和算法。 借用算法：利用已知并行算法解决新问题，如 3PCF 问题通过矩阵乘法解决。
18		PCAM 与并行计算任务调度	并行化（Parallelization）、通信（Communication）、调度（Scheduling）和映射（Mapping）。 任务的划分与分配、数据的分布与管理、通信开销、负载均衡、调度策略、资源管理、容错和异常处理、性能优化、任务调度算法的设计、动态环境下的调度、任务的优先级和公平性、能耗考虑
19	Lab4 实验代码编写		熟悉 OpenMP 的基础函数，在实验三代码的基础上分析并加入编译制导语句。
20	Lab4 数据分析以及是要报告编写		运行代码，选择合适的数据规模，绘制图像，分析实验结果。