并行计算课程
实 验 报 告

报告名称：	Pthread 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 3 月 29 日

2024 年制

一、实验内容概述

本实验旨在利用 Pthread 对矩阵乘法进行优化，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 Pthread 多线程技术，将结果矩阵中的每一个位置的计算交给不同的线程运算，提高运算效率与程序性能。并对不同线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。由于幂次需要用到之前矩阵计算所使用的数据，所以对于第一层循环，难以用 Pthread 进行优化。主要对于模拟计算单次矩阵乘法进行优化。

```
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

根据任务书提示，对结果矩阵进行分解，将结果矩阵不同位置的数据交给单独的线程来计算。根据线程 id 和线程总数来确定线程所需要计算的数据。利用 n 来定位线程所需要计算元素的位置。

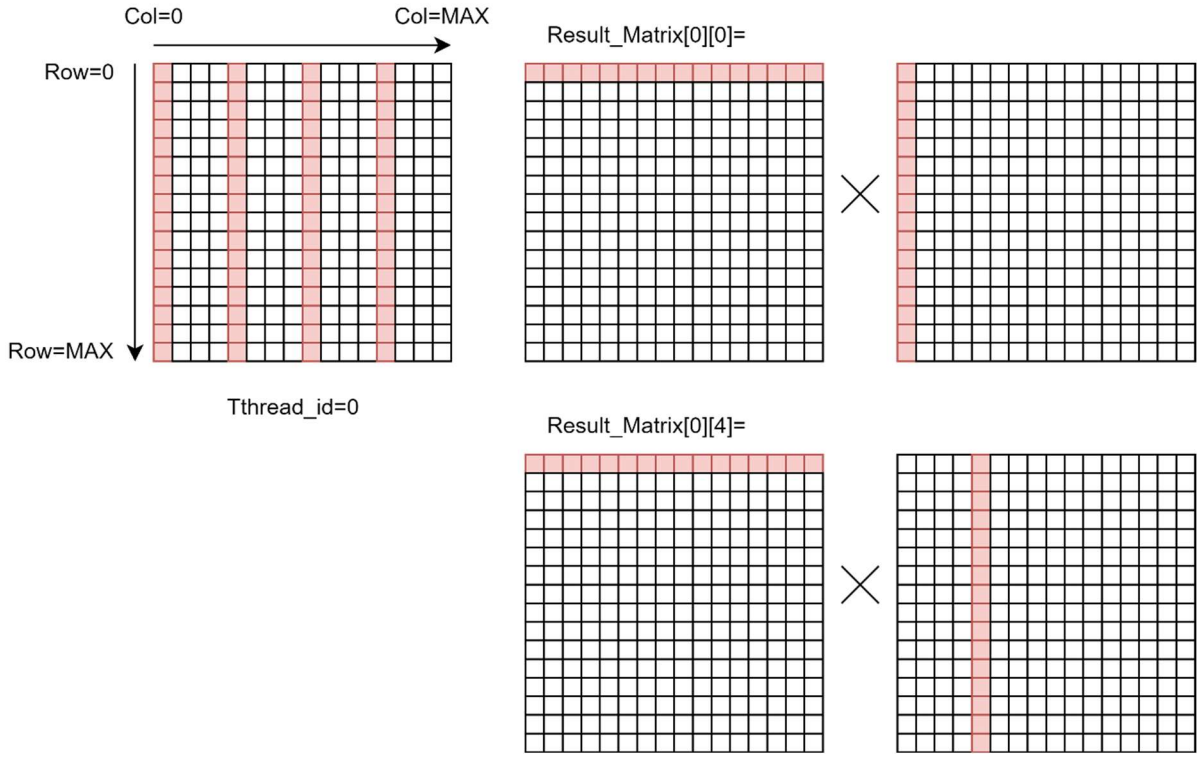
```
row = n / MATRIX_SIZE;
colum = n % MATRIX_SIZE;
```

接下来按照矩阵相乘的计算方法，将第一个矩阵第 row 行的数据与第二个矩阵第 col 列的元素对应相乘，将结果存放在结果矩阵中的[row,col]中。

```
for (int k = 0; k < MATRIX_SIZE; k++)
{
    sum += args->temp->data[row][k] * args->matrix->data[k][colum];
}
pthread_mutex_lock(args->mutex);
args->result->data[row][colum] = sum; // 将结果存入result矩阵
pthread_mutex_unlock(args->mutex);
```

同时需要注意，在存放计算结果时，需要使用互斥锁来避免竞态条件。否则会因为缓存数据不一致而造成计算结果丢失的问题。

下面是线程计算的示意图（以 4 线程，16×16 矩阵为例）：



线程的划分通过调用线程传入参数来实现。下面是具体任务划分的方法。

```
// 划分任务创建线程
int rows_per_thread = result->rows / num_threads;
for (int i = 0; i < num_threads; i++)
{
    thread_args[i].step = num_threads;
    thread_args[i].id = i;
    thread_args[i].result = result; // 专用于举证相乘的结果保存
    thread_args[i].matrix = matrix; // 为源矩阵，每次相乘中的一个
    thread_args[i].mutex = &mutex;
    thread_args[i].temp = temp; // 上一次矩阵相乘的结果，每次矩阵相乘中的一个
    pthread_create(&threads[i], NULL, multiply, (void *)&thread_args[i]);
}
```

程序每次会随机生成一个 350×350 的矩阵，然后通过 Pthread 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
# 定义要测试的核心数数组
cores=(1 2 4 8 16 32 64)
# 定义运行的次数
runs=5
n=150
# 对每个核心数进行循环
for core in "${cores[@]}"; do
    echo "Running with core=$core" >> 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 ./lab2.o $n $core; } 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


echo "All done."
```

3.2.3 对线程创建的优化以及解决伪共享

设计的算法会导致频繁地创建与销毁线程，所以需要对线程的创建消耗进行优化。采用线程池对线程进行统一管理，减小线程在创建时候的开销。以优化程序性能。

线程在使用源矩阵数据与将计算结果写入结果矩阵的过程中需要大量的访存以及修改缓存的操作，尤其是在对矩阵进行写入的时候，不仅要避免线程对于共享数据也就是 Result 矩阵的同时访问，也要注意伪共享问题导致的频繁的缓存换入换出。所以在定义矩阵结构体的时候使用数据填充。如下图所示：

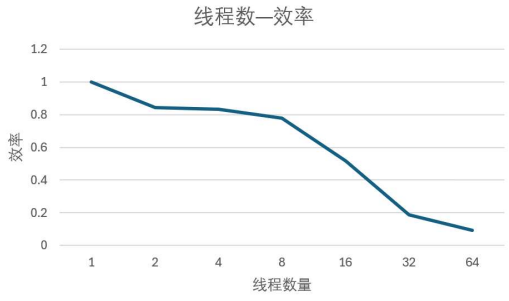
```c
#define CACHE_LINE_SIZE 64 // 缓存行大小为 64 字节
typedef struct
{
    int rows;
    int cols;
    double **data;
    char padding[CACHE_LINE_SIZE - sizeof(int) * 2 - sizeof(double **)]; // 填充数据
} Matrix;
```

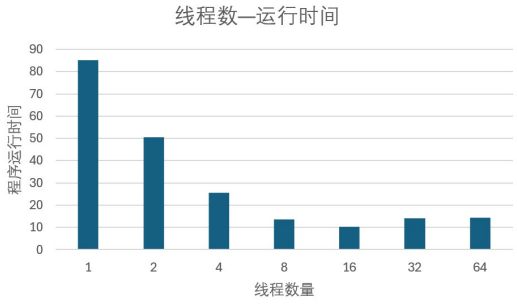
实验的结果如下图所示：

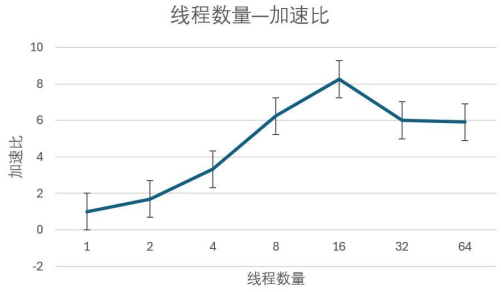
矩阵规模350×350幂次							
线程数量	1	2	4	8	16	32	64
	84.796	57.238	23.48	13.42	10.04	14.917	14.64
	84.99	50.393	23.737	13.404	10.703	13.01	14.623
	86.632	52.734	23.922	13.753	10.462	12.383	14.423
	84.835	44.793	32.933	14.033	10.313	17.602	14.232
	84.542	46.963	23.833	13.733	10.033	12.971	14.093
平均时间	85.159	50.4242	25.581	13.6686	10.3102	14.1766	14.4022

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。







在不断测试的过程中，发现并行计算使用与计算规模较大的情形，在计算规模较小的情况下，使用并行计算反而会由于线程创建的开销而拖慢程序的运行时间。在我得到最终实验数据之前，曾经使用过不同数据规模进行模拟，发现在总计算量较小的情况下，线程数量的增加反而会降低程序的运行时间，推测是由于线程创建的开销过大导致。随着我不断增加数据的规模，发现程序的运行时间也在不断增大但是线程数量的提升开始能够显著地降低程序运行时间，提高计算效率。由此得出一个结论，并型计算更加使用与计算规模更大的情形。

同时，更具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 6 左右的位置。最后没能突破 6，并且随着线程数量的增加，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

1/6，也就是说，串行部分在我的程序中只占了约 1/6，相比于上一个实验，有所提高，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-线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。每个线程的运行时间的降低速度也也就不断减少了。

四、实验总结

4.1 遇到的问题以及收获

（1)在使用并行计算对矩阵进行计算时，总是发现最后得到的结果不正确。经过手工模拟演算，发现第一次在计算矩阵的过程中，结果被保存在了原矩阵的存储空间中，导致原矩阵被改变，后续计算所使用的数据都是被修改过后的数据，最后在每次矩阵计算完成后，将作为乘数的矩阵与结果矩阵互换，最后得到了正确的结果。

（2）得出正确结果之后，使用 sh 脚本提交我的程序运行并且记录每次的运行时间，发现使用多线程的时候，程序的运行时间反而高于串行代码。联想到上课时老师提出的伪共享的问题，猜测是多个线程访问数据时候对缓存频繁的换入换出所导致的，在保存矩阵的结构体中添加了填充字节后，发现问题解决了。

（3）解决完伪共享问题之后，又发现串行代码和并行代码跑出的结果基本相同，甚至并行代码还会略微慢于串行代码。这里思考之后，猜测是数据量太小导致的，调大数据量之后，发现线程数量为 1 和 2 的时候运行时间有了明显区别，但是后续增加线程数量的程序运行时间与线程 2 又非常近似。遂不断增加数据规模，但是增加数据规模会导致程序运行时间疯狂增大，最后，还是得到了一个相对合理的结果，只是程序的运行时间已经来到了 80s 左右，耗费了一个下午对不同矩阵规模和幂次进行测试。

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。