华中科技大学

课程实验报告

课程名称：并行编程原理与实践

专　　业：　计算机科学与技术

班　　级：　CS2008

学　　号：　U202015526

姓　　名：　谢林桦

指导老师：　陆枫

报告日期：　2023.06.15

独创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签名： 

日期：2023 年　6 月　15 日

综合成绩	
教师签名	

1 使用 OpenMP 进行并行矩阵乘法

1.1 实验目的与要求

设计一个使用 OpenMP 的并行程序，用于对两个矩阵进行乘法运算。具体要求如下：

（1）你的程序应该能够接受两个矩阵作为输入，并计算它们的乘积。

（2）使用 OpenMP 将矩阵乘法操作并行化，以加快计算速度。

（3）考虑如何将矩阵数据进行划分和分配给不同的线程，以实现并行计算。

（4）考虑如何处理并行区域的同步，以避免竞态条件和数据一致性问题。

（5）考虑如何利用 OpenMP 的并行循环和矩阵计算指令，以进一步提高并行效率。

1.2 算法描述

矩阵乘法采用最基础的算法即矩阵乘法的计算公式，具体实现为三重 for 循环，在此基础上使用 OpenMP 的并行优化指令，进行计算优化。

```
#pragma omp parallel for schedule(static, 1) num_threads(6)
#pragma omp parallel for reduction(+:sum)
```

对于基础的矩阵乘法，假定其维度为 n，以二维数组$A[n][n]$的形式表示，很容易证得其时间复杂度为$O(n^3)$，其空间复杂度为$O(n^2)$。使用 OpenMP 进行并行优化后，加速比的获得在后续说明。下面以一个例子来说明算法的时间复杂度与空间复杂度：

$$\begin{matrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{matrix} \times \begin{matrix}9 & 8 & 7\\ 6 & 5 & 4\\ 3 & 2 & 1\end{matrix} = \begin{matrix}30 & 24 & 18\\ 84 & 69 & 54\\ 138 & 114 & 90\end{matrix}$$

根据矩阵乘法计算公式：

$$C_{ij} = \sum_{1\le k\le 3} A_{ik}B_{kj}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} = 1\times 9 + 2\times 6 + 3\times 3 = 30$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} = 1\times 8 + 2\times 5 + 3\times 2 = 24$$

$$C_{13} = A_{11}B_{13} + A_{12}B_{23} + A_{13}B_{33} = 1\times 7 + 2\times 4 + 3\times 1 = 18$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31} = 4\times 9 + 5\times 6 + 6\times 3 = 84$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} = 4 \times 8 + 5 \times 5 + 6 \times 2 = 54$$

$$C_{23} = A_{21}B_{13} + A_{22}B_{23} + A_{23}B_{33} = 4 \times 7 + 5 \times 4 + 6 \times 1 = 54$$

$$C_{31} = A_{31}B_{11} + A_{32}B_{21} + A_{33}B_{31} = 7 \times 9 + 8 \times 6 + 9 \times 3 = 138$$

$$C_{32} = A_{31}B_{12} + A_{32}B_{22} + A_{33}B_{32} = 7 \times 8 + 8 \times 5 + 9 \times 2 = 114$$

$$C_{33} = A_{31}B_{13} + A_{32}B_{23} + A_{33}B_{33} = 7 \times 7 + 8 \times 4 + 9 \times 1 = 90$$

以三行三列的矩阵为例，每计算结果的一个元素需要进行 3 次乘法运算，2 次加法运算，总共的计算量为 27 次乘法运算，18 次加法运算，我们假定理想情况下 CPU 做一次乘法运算与做一次加法运算的时间相同，均为 T，因此计算3×3矩阵的乘法所需要的时间开销为$3 \times 3 \times 3 + 2 \times 3 \times 3 = 45$次，同理，我们可以计算出任意阶矩阵的时间开销为$T_n = n^3 + (n-1)n^2 = (2n^3 - n^2)T$，因此，矩阵乘法的时间复杂度为$O(n^3)$。在此例中，我们需要 A，B 两个数组来保存两个相乘矩阵，数组 C 来存储结果，数组大小均为3×3，推广到 n 维矩阵的情况，可得该算法的空间复杂度为$O(n^2)$。

1.3 实验方案

1.3.1 开发环境

VS 2019 professional

PyCharm Community 2022，Python 3.10

1.3.2 测试环境

educoder 平台

1.3.3 具体过程

（1）为获得使用 OpenMP 并行优化指令后的加速比，设置对照实验（使用 OpenMP 加速，不使用 OpenMP 加速），并获得两种条件下，计算 N 阶矩阵所用时间（单位 ns），以使用 OpenMP 加速为例，如下：

```
// 获取当前时间
auto t1 = std::chrono::high_resolution_clock::now();
// 进行矩阵乘法
matrix_multiply(A, B, C, dim);
// 获取当前时间
auto t2 = std::chrono::high_resolution_clock::now();
```

```
// 计算函数运行时间，单位为 ns
auto durationM =
std::chrono::duration_cast<std::chrono::nanoseconds>(t2 - t1);
```

（2）之后，取$dim \in [2,1000]$,重复上述过程，将计算 dim 阶矩阵所需要的时间存储在.txt 中，待后续处理。

（3）用 Python 的 matplotlib 库处理数据，绘制加速比随 dim 变化曲线。

1.4 实验结果与分析

1.4.1 静态调度方案

```
#pragma omp parallel for schedule(static, 1) num_threads(6)
#pragma omp parallel for reduction(+:sum)
```

实验结果如下图 1.1 所示

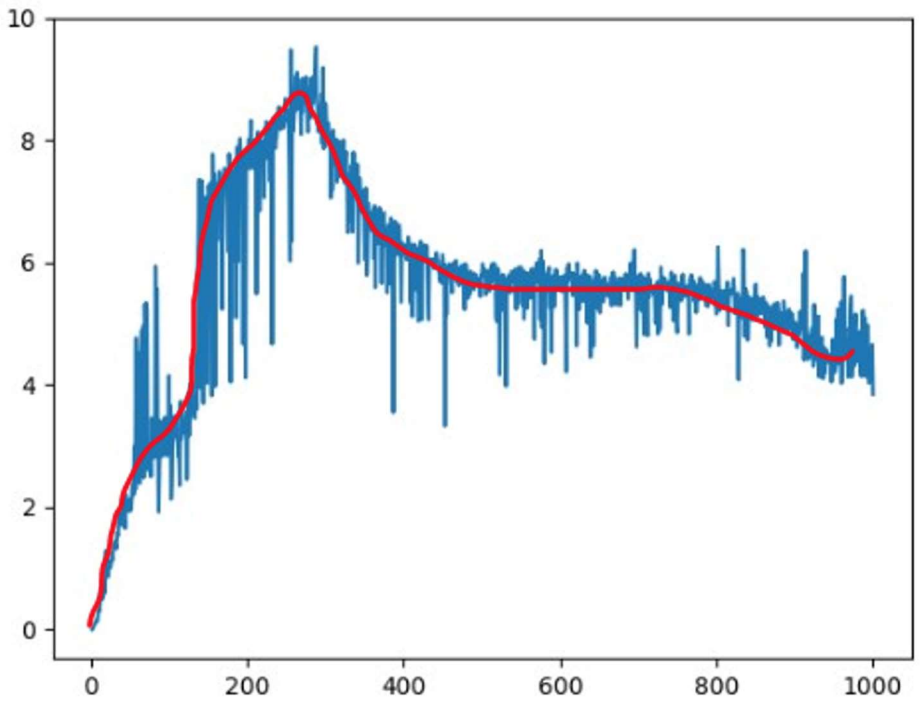

图 1.1 S 随 dim 变化曲线

分析实验结果可知：

（1）当矩阵阶数在[2, 30]区间之内时，使用 OpenMP 并行优化指令计算矩阵乘法所需要的时间要大于直接串行运算的时间，主要原因是矩阵乘法运算过程中存在较多的内存访问和数据依赖，这导致并行计算时需要频繁地进行线程间的同步和通信，在小规模问题上这种开销可能会抵消甚至超过了并行计算的速度优势，

另外还有并行化的启动和管理开销，因此，对于小规模的运算，直接串行计算可能会更有效率，详细可参考下图 1.2，当使用并行优化指令计算 2 阶矩阵的乘积时，所需要的时间开销为 2191600ns，对比之下，直接串行计算只需要 300ns，这部分时间开销包括了并行计算的启动与管理以及进程调度与同步，可以看到，并行化启动后，时间开销直接减少了10^2个数量级。

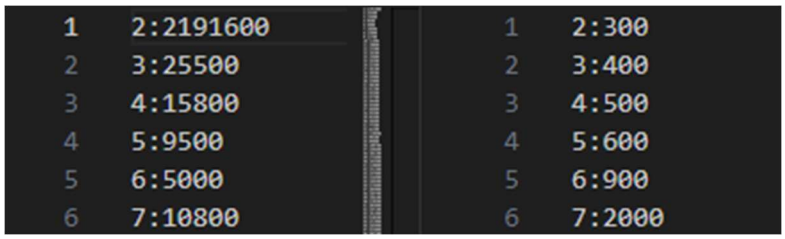

图 1.2 矩阵维度与运算时间对应关系

（2）当矩阵阶数在[200, 400]区间时，加速比达到峰值，之后随着矩阵阶数的增长，加速比呈现下降趋势，主要原因是 OpenMP 并行优化指令存在线程开销和负载不均衡等问题；当矩阵阶数较小时，使用 OpenMP 并行优化指令可以有效地将计算任务划分给多个线程处理，使得计算速度得到显著提升，从而达到加速比峰值。但随着矩阵阶数的继续增加，线程数也需要相应增加，这会带来额外的线程开销，如线程创建和销毁、上下文切换等，从而导致加速比逐渐下降。另外，随着矩阵阶数的增加，矩阵乘法中的计算量也会大幅增加，如果任务划分不均衡，可能会出现某些线程计算任务过重，而其他线程处于空闲状态的情况，从而浪费了 CPU 资源，导致加速比进一步下降。

1.4.2 动态调度方案

```
#pragma omp parallel for schedule(dynamic, 1) num_threads(6)
#pragma omp parallel for reduction(+:sum)
```

实验结果如下所示：

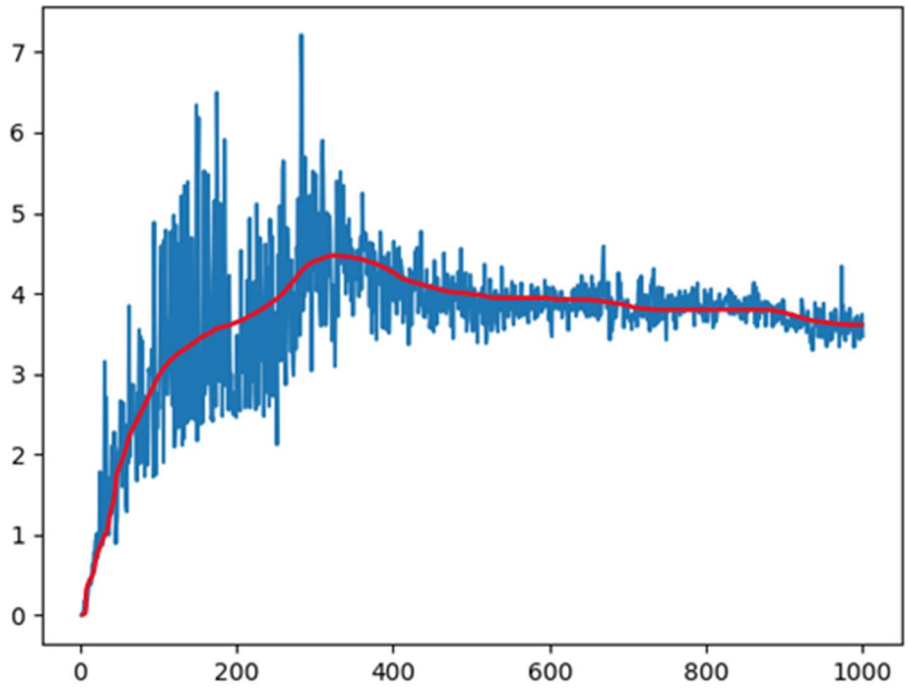

图 1. 3 S 随 dim 变化曲线

分析结果可知：

（1）当矩阵运算规模较小时，与静态调度方案类似，用于管理并行化的时间开销远远大于其计算的速度优势。

（2）当运算规模开始增加，动态调度方案的加速比增长较为平缓，且存在巨大的不确定性，但当规模增加到一定程度后（[500, 1000]）加速比稳定在 4 左右。

1.4.3 两种方案的对比

（1）当矩阵运算的规模较小时[0, 200]，发现加速比均随矩阵阶数 dim 的增大而增大，区别在于当矩阵规模在[200, 1000)区间时，静态调度方案的加速比呈现较为明显的下降趋势，最终下降到 4.0 左右，而动态调度方案的加速比则是稳定在 4.0，并无太大变化。

（2）静态调度依然按照固定的任务块大小进行任务分配，但此时每个任务块中的计算量可能较小，导致一些线程的计算任务更快完成，而其他线程仍在进行计算。这样会造成一些线程提前完成工作，而其他线程还在忙碌，从而造成了一些线程的空闲时间增加。相比之下，动态调度方案能够更加均衡地将任务分配给不同的线程。动态调度在运行时根据实际的线程负载情况动态地将任务分配给空闲

线程，每个线程处理完一个任务块后再获取下一个任务块，因此能够更好地利用线程资源。即使在矩阵规模较小的情况下，动态调度仍然可以保持较好的加速比，因为它能够及时地将任务分配给空闲线程，减少了线程的空闲时间。

（3）针对这一具体问题，明显可以看出，尽管静态调度时，存在明显的下加速比下降趋势，但其总体性能要优于动态调度方案。

（4）综上所述，静态调度方案在矩阵规模较小时由于任务分配不均衡而导致部分线程的空闲时间增加，而动态调度方案能够更好地均衡任务分配，因此在各种矩阵规模下都能保持较为稳定的加速比。具体采用哪种方案还要视情况而定，具体问题具体分析。

2 使用 Pthread 进行并行文本搜索

2.1 实验目的与要求

设计一个多线程程序，使用 Pthreads 来实现并行文本搜索。具体要求如下：

（1）你需要实现一个函数，该函数接受一个目标字符串和一个包含多个文本文件的文件夹路径。

（2）程序应该并行地搜索每个文本文件，查找包含目标字符串的行，并将匹配的行打印出来。

（3）每个线程应该处理一个文件，你需要合理地分配文件给不同的线程。

（4）确保你的程序是线程安全的，并正确处理多个线程之间的同步问题。

2.2 算法描述

首先打开指定的文件夹，并遍历其中的所有文件和子文件夹。对于每一个普通文件，即不是文件夹或隐藏文件，该算法创建一个新线程进行文件内容的搜索操作。每个线程都会打开对应的文件，并按行读取文件内容。如果找到了包含目标字符串的行，线程就会将匹配结果输出到控制台上。需要注意的是，在输出语句之前，线程会通过一个互斥锁锁住共享的输出资源，以避免多个线程同时输出产生冲突。当所有线程完成搜索任务后，会释放线程占用的资源，并关闭文件夹。下面以一个实例，来说明算法流程：

在工程目录下新建文件夹"text"，text 目录下包含两个文件 cn.txt 和 us.txt。在 search_files 函数标记断点，进行跟踪调试。如下图 2. 1 所示：

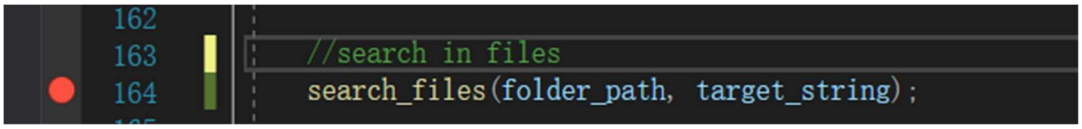

图 2. 1 在 search_files 处断点

首先打开目录，file_folder 为 "../PThreadTest/text"，如下图 2. 2 opendir：

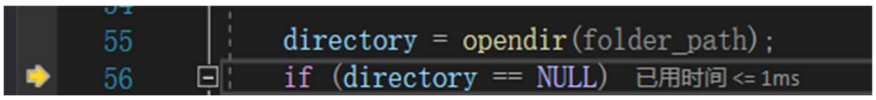

图 2. 2 opendir

接着，读取该目录下的每一个文件夹与文件，如下图 2.3 readdir，entry!=NULL 说明仍存在文件，而 if (entry->d_type == DT_REG) 则可以过滤掉当前目录与父级目录；

图 2. 3 readdir

当读取到文本文件.txt 时，创建一个新的进程，通过该进程对.txt 进行文本搜索，如下图 2. 4 thread_create：

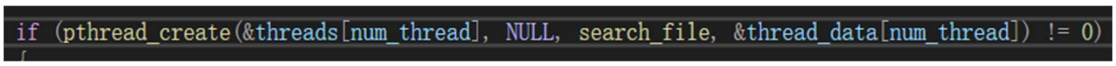

图 2. 4 thread_create

跟踪进入 search_file 函数，首先调用 fopen 函数以只读模式打开对应的 txt 文件，如下图 2. 5 open file：

图 2. 5 open file

最后，每次在文本文件中读取一行文本，并调用 string.h 中的函数对每一个 line 和 target_string 进行匹配，若匹配成功，则将结果输出，在输出语句之前，线程会通过一个互斥锁锁住共享的输出资源，以避免多个线程同时输出产生冲突。如下图 2. 6 GetLine and Match：

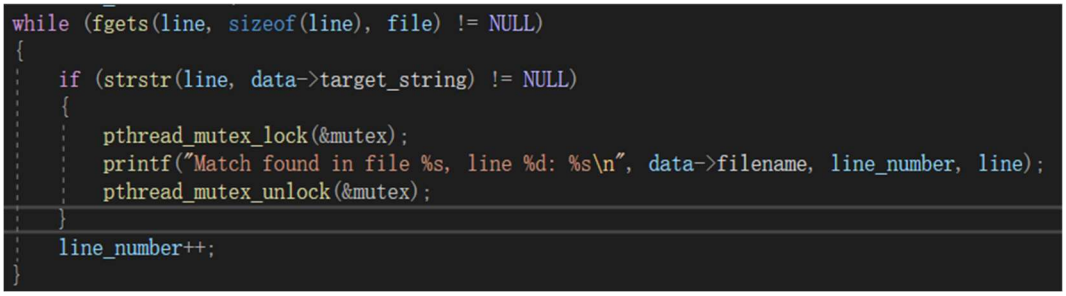

图 2. 6 GetLine and Match

2.3 实验方案

2.3.1 开发环境

处理器 Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz

VS 2019 professional

PyCharm Community 2022，Python 3.10

2.3.2 测试环境

educoder 平台

2.3.3 具体过程

由于样本数量太少，并行文本搜索的速度优势带来的增益远远小于创建管理线程带来的开销。因此并行搜索的时间大于串行搜索的时间，在上述案例中，并行搜索的时间大约是串行搜索的时间的 2.5 倍。所以，实验设计成在大批量的文本中搜索指定文本，同时比较并行与串行的时间，计算加速比。具体过程如下：

（1）编写 Python 脚本，批量生成.txt 文档，文档数量为 100，500；每个 txt 文档中均含有 100 条数据，每条数据包含”ABCDEFG”中的任意 10 个字母；

（2）搜索指定文本，记录并行和串行分别需要的时间（多次记录取平均值）；

2.4 实验结果与分析

（1）文档数量为 100

表 2.1 文档数量为 100

	并行/ms	串行/ms	加速比
1	13350.00	18970.00	1.42
2	12150.00	18332.00	1.51
3	12542.00	16659.00	1.33
4	11525.00	20162.00	1.75
5	11633.00	20534.00	1.77
6	12396.00	18473.00	1.49
7	11637.00	18478.00	1.59

8	12329.00	19059.00	1.55
9	11251.00	16191.00	1.44
10	12694.00	17748.00	1.40
平均值	12150.7	18460.6	1.52

（2）文档数量为 500

表 2.2 **文档数量为** 500

	并行/ms	串行/ms	加速比
1	143572.00	183286.00	1.28
2	142855.00	171019.00	1.20
3	148854.00	181513.00	1.22
4	149836.00	181185.00	1.21
5	147459.00	184738.00	1.25
6	142594.00	177024.00	1.24
7	140311.00	174266.00	1.24
8	143500.00	174039.00	1.21
9	141202.00	172745.00	1.22
10	148460.00	170493.00	1.15
平均值	144864.3	177030.8	1.22

（3）观察实验结果发现，随着文本数量增加，加速比下降，原因可能是当搜索的文件数量增多时，同时开启的线程数量也相应地增加，但是线程数量过多会导致上下文切换的开销变得非常大，从而影响并行搜索的效率。此时，由于线程之间的竞争和调度等因素，加速比可能会降低。另外，在第一种情况下，可用 CPU 资源较为充足，因此可以同时开启更多的线程进行搜索，从而提高了并行搜索的效率。而在第二种情况下，如果可用的 CPU 资源不足，则不能同时开启大量的线程进行搜索，因此加速比可能会降低。

3 实验小结

通过这次实验，学习了 OpenMP 的相关并行优化指令，以及 PThreads 对多线程的相关知识，包括进程的创建与管理。

对于小规模的问题，在这次实验中体现为低阶矩阵的乘法和文本数量较少时，并行化需要的维护进程创建与管理所带来的时间开销远大于其并行速度优势带来的时间增益，在测试时表现为并行运行时间大于串行运行时间，因此，对于小规模问题，串行方式执行性能更加优越；当问题的规模增加时，比如矩阵阶数增加到 1000 甚至更大，文本数量达到 500 甚至更多，并行执行的优势就体现出来了，对于高阶矩阵的乘法，使用 OpenMP 进行优化的加速比通常稳定在 4 左右，使用 PThread 搜索文本的加速比并没有矩阵计算那样显著，通常在 1.2-1.5 之间。另外，随着问题规模增加，开启的线程数量也相应地增加，但是线程数量过多会导致线程管理的开销变得非常大，从而影响并行的效率，这在文本搜索中体现的比较明显;此外,调度方式也会在很大程度上影响并行的效率,静态调度时，按照固定的任务块大小进行任务分配，但此时每个任务块中的计算量可能较小，导致一些线程的计算任务更快完成，而其他线程仍在进行计算。这样会造成一些线程提前完成工作,而其他线程还在忙碌,从而造成了一些线程的空闲时间增加。