课程实验报告

课程名称：并行编程原理与实践

院　　系：	计算机科学与技术
姓　　名：	徐锦慧
专业班级：	CS2011
学　　号：	U202011675
指导教师：	金海
报告日期：	2023.07.06

目　录

1 使用 OPENMP 进行并行矩阵乘法

1.1 实验目的与要求

1. 实验目的

掌握基本的 OpenMP 并行指令的用法，并能够正确调用相关的函数接口，对两个矩阵进行并行乘法运算。最终了解并掌握并行优化思路和可能存在的问题。

2. 实验要求

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) 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

1. 总体框架

阅读 main.cpp 文件可知，总代码的主要用途是计算两个矩阵的乘积并输出。main()函数的流程如下：

1) 输入矩阵的维度 dim
2) 根据 dim 动态分配矩阵 A、B 和 C 的内存空间，这 3 个矩阵的大小都是 dim×dim。
3) 输入矩阵 A 和 B 的值。
4) 调用矩阵乘法函数 matrix_multiply ()，并传递矩阵 A、B、C 和维度参数 dim。
5) 打印结果矩阵 C，并释放矩阵 A、B 和 C 的内存空间。

题目要求实现的即为 matrix_multiply ()函数。在本实验中，分别通过串行算法、并行算法、并行＋矩阵分块算法来实现 matrix_multiply ()，以下将分别介绍这 3 种算法。

2. 串行算法

【输入输出】

输入为矩阵 A、B、C 和矩阵维度 dim，无输出。

【算法流程】

1) 初始化结果矩阵 C 的所有元素为 0。
2) 使用两层嵌套循环遍历矩阵 C 的每个元素 C[i * dim + j]。
3) 对于 C 中的每个元素 C[i * dim + j]，使用第三层嵌套循环计算对应的乘法运算。内层循环中的变量 k 用于遍历矩阵 A 的行和矩阵 B 的列。在每次迭代中，计算 A[i][k] 和 B[k][j] 的乘积，并将结果累加到 C[i * dim + j] 上。

【时间复杂度】

O(n^3)。其中 n 是矩阵的维度。由于存在三层嵌套循环，每层循环的迭代次数均为 n，因此总的计算量为 n^3。

【空间复杂度】

O(1)。对于这个函数来说，只声明了变量 i、j、k，没有使用额外的数据结构来存储矩阵，矩阵的乘积结果直接存储在矩阵 C 中，因此空间复杂度为 O(1)。

核心代码如下：

```
void matrix_multiply(int A[], int B[], int C[], int dim) {
    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++) {
            C[i * dim + j] = 0;
            for (int k = 0; k < dim; k++)
                C[i * dim + j] += A[i * dim + k] * B[k * dim + j];
        }
    }
}
```

3. 并行算法

【输入输出】

输入为矩阵 A、B、C 和矩阵维度 dim，无输出。

【算法流程】

1) 使用 #pragma omp parallel for 指令将外层循环并行化，以利用多个线程并行计算。对于每个线程，它们将独立地计算矩阵 C 的一部分。
2) 在并行区域中，使用两层嵌套循环遍历矩阵 C 的每个元素。对于 C 中的每

个元素 C[i][j]，使用第三层嵌套循环计算对应的乘法运算，此部分和串行的逻辑类似。

【时间复杂度】

O(n^3)。该函数的时间复杂度与串行版本的矩阵乘法函数相同，其中 n 是矩阵的维度。由于并行化只是利用多个线程同时计算，没有改变计算量。

【空间复杂度】

O(1)。该函数的空间复杂度与串行版本的矩阵乘法函数相同。

核心代码如下：

```
void matrix_multiply (int A[], int B[], int C[], int dim) {
    #pragma omp parallel for
    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++) {
            C[i * dim + j] = 0;
            for (int k = 0; k < dim; k++)
                C[i * dim + j] += A[i * dim + k] * B[k * dim + j];
        }
    }
}
```

4. 并行+矩阵分块算法

【输入输出】

输入为矩阵 A、B、C 和矩阵维度 dim，无输出。

【算法流程】

1) 首先定义块大小 block_size，并计算块的数量 num_blocks，每个块的大小为 block_size * block_size。
2) 使用 #pragma omp parallel for collapse(2) 指令并行化嵌套循环，以利用多个线程并行计算。
3) 对于每个块索引 block_i 和 block_j，计算当前块的起始和结束索引。并使用两层嵌套循环遍历每个块内的元素。
4) 对于每个块内的元素 C[i * dim + j]，使用第三层嵌套循环计算对应的乘法运算。内层循环中的变量 k 用于遍历矩阵 A 的行和矩阵 B 的列。在每次迭代中，计算 A[i * dim + k] 和 B[k * dim + j] 的乘积，累加结果。

【时间复杂度】

O(n^3)。该函数的时间复杂度与串行版本的矩阵乘法函数相同，其中 n 是矩阵的维度。因为循环中没有改变计算量，只是将计算任务划分成了多个块。

【空间复杂度】

O(1)。对于这个函数来说，只声明了循环变量和临时变量，没有使用额外的数据结构来存储矩阵。

算法的流程图如图 1.1 所示。

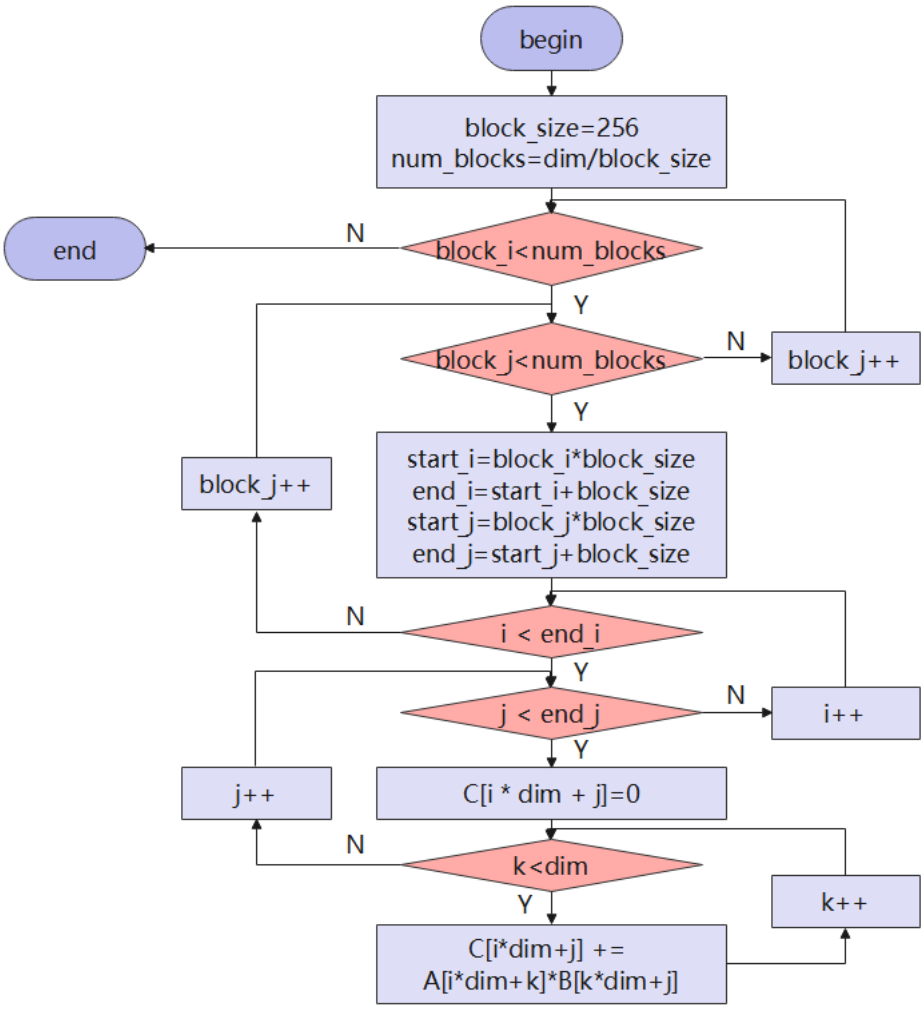

图 1.1 并行+矩阵分块算法流程图

核心代码如下：

```
void matrix_multiply (int A[], int B[], int C[], int dim) {
    int block_size = 256;                    // 块大小
    int num_blocks = dim / block_size;   // 计算块的数量

    #pragma omp parallel for collapse(2)
    for (int block_i = 0; block_i < num_blocks; block_i++) {
        for (int block_j = 0; block_j < num_blocks; block_j++) {
            // 计算块内元素的起始和结束索引
```

```
                int start_i = block_i * block_size;
                int end_i = start_i + block_size;
                int start_j = block_j * block_size;
                int end_j = start_j + block_size;

                // 对每个块进行乘法运算
                for (int i = start_i; i < end_i; i++) {
                    for (int j = start_j; j < end_j; j++) {
                        int sum = 0;
                        for (int k = 0; k < dim; k++)
                            sum += A[i * dim + k] * B[k * dim + j];
                        C[i * dim + j] = sum;
                    }
                }
            }
        }
    }
```

1.3 实验方案

1. 开发与运行环境

【主机机带】RAM 8.00 GB (7.75 GB 可用)

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

【主机版本】Windows 10 专业版

【虚拟机版本】 Ubuntu 20.04.4 LTS 64-bit

【实验平台】Visual Studio Code 1.63.0

2. 实验方案描述

本实验分别通过串行、直接并行、并行+矩阵分块这 3 种方法来实现函数 matrix_multiply()，并进行控制变量来对比其效率。通过不同大小的样例数据，来分析 dim 不同时，这 3 者的效率区别。效率主要通过运行时间和加速比来分析，运行时间通过 omp_get_wtime()获取，其核心代码如下：

```
double start_time = omp_get_wtime();
matrix_multiply_parallel(A, B, C, dim);
double end_time = omp_get_wtime();
double total_time = end_time - start_time;
cout << "Total time: " << total_time << " seconds\n";
```

在实现并行计算时，矩阵的数据划分和分配是通过 OpenMP 的并行循环指令 #pragma omp parallel for 自动处理的。该指令将外层循环进行划分和分配给不同的线程，以实现并行计算。每个线程将负责计算矩阵 C 的一部分。通过迭代循

环的索引变量，不同的线程被分配不同的行。这种自动的划分和分配方式可以充分利用并行计算的优势，将计算任务均匀地分布给不同的线程，从而加速整个矩阵乘法的运行。

在并行循环指令中， OpenMP 为每个线程创建一个私有的迭代索引变量，并根据并行度将循环的迭代分配给不同的线程。在循环迭代过程中，每个线程会独立地执行自己负责的迭代。当一个线程完成自己的迭代时，OpenMP 会隐式地进行同步，以确保所有线程完成自己的工作后再继续执行下一步操作。这种同步机制避免了竞态条件，确保了每个线程在访问和修改共享数据时的正确性。

3. 样例数据获取

本实验用了 6 组样例数据，dim 的大小分别为 500/1000/1500/2000/2500/3000，相邻组跨度为 500。样例数据的获取由 generate_input_file()函数实现：输入 dim，随机生成大小为 dim×dim 的矩阵 A 和矩阵 B，并将 dim、矩阵 A、矩阵 B 都写入文件中，以供下次实验使用。generate_input_file()的代码如下：

```
void generate_input_file(int dim) {
    std::ofstream file("input.txt");
    file << dim << "\n";
    // 生成随机矩阵 A
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> dis(1, 10);
    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++)    file << dis(gen) << " ";
        file << "\n";
    }
    // 生成随机矩阵 B
    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++)    file << dis(gen) << " ";
        file << "\n";
    }
    file.close();
}
```

1.4 实验结果与分析

由于 educoder 上的测试集较小，因此上述 3 种方法均可以通过测试。以下进行串行、并行、并行+矩阵分块 3 种算法的效率分析。测试集分别是 dim=500/1000/1500/2000/2500/3000 的情况。按照 1.3 中描述的方法生成的测试集

如图 1.2 所示。

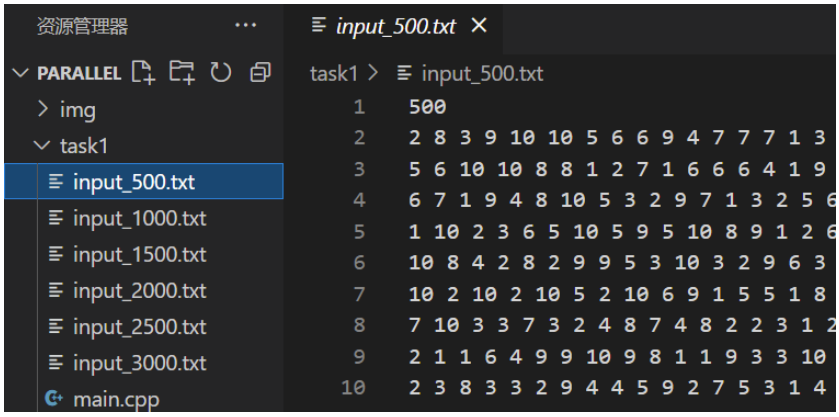

图 1.2 生成的测试集

当 dim=500 时，3 种方案运行结果如图 1.3 所示。

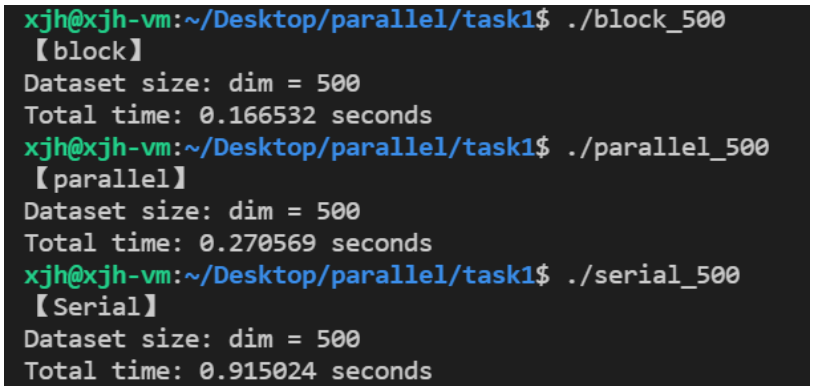

图 1.3 dim=500 的运行结果

serial_500、parallel_500、block_500 分别是 dim=500 时这 3 种方法的可执行文件。可以看出，并行算法的效率比串行的高很多，大约是其 3 倍~4 倍。加上矩阵分块后，并行算法的效率又略有提升，运行时间减为原来的一半。

当改变 dim 的大小，3 种方法花费的时间如表 1.1 所示：

表 1.1 不同方案时间对比表

方案	运行时间（单位：seconds）					
	dim=500	dim=1000	dim=1500	dim=2000	dim=2500	dim=3000
串行	0.915024	5.94302	26.2878	48.8584	107.19	226.567
并行	0.270569	1.1282	4.44981	11.2539	22.5924	46.5683
并行+分块	0.166532	0.82235	3.35585	9.646458	17.23222	46.34928

并行与并行+分块的加速比（和串行相比）如表 1.2 所示：

表 1.2 不同方案加速比对比表

方案	加速比（单位：无）					
	dim=500	dim=1000	dim=1500	dim=2000	dim=2500	dim=3000
并行	3.381851	5.267701	5.907623	4.341464	4.744516	4.865262
并行+分块	5.494584	7.226812	7.833412	5.064906	6.220323	4.888253

时间和加速比的折线图分别如图 1.4 和 1.5 所示。

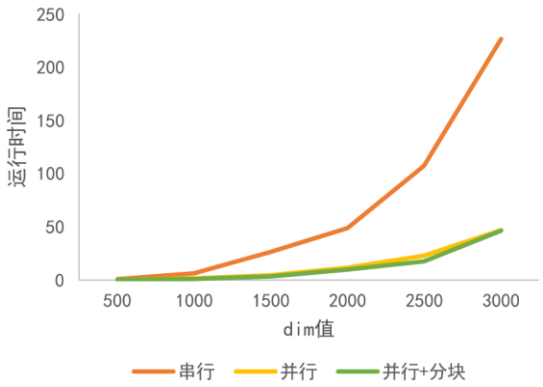

图 1.4 运行时间

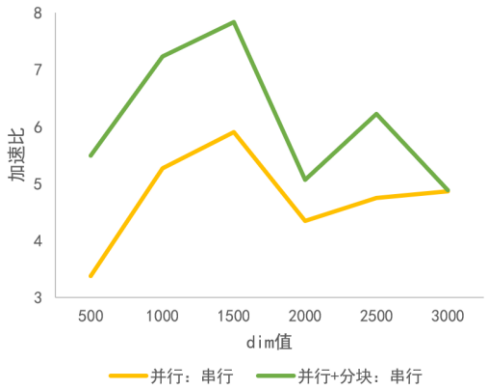

图 1.5 运行加速比

由折线图可以发现，并行执行的效率明显高于串行，加速比在 4~6 之间。当计算的矩阵较小时，区别不太明显，但当矩阵增长到一定规模，即 dim=1500 之后，二者区别十分明显。

为并行算法加入矩阵分块后，效率又略有提升，尤其是 dim<=2500 时，效果较明显。但当 dim=3000 时，结果和分块前类似，可能是由于分块的数目不太合适，若将其调整为合适的数值，效率也将提高。

2 使用 Pthreads 实现并行文本搜索

2.1 实验目的与要求

1. 实验目的

掌握 pthread 相关多线程函数的应用，并能够正确调用相关的函数接口，实现并行文本搜索运算。最终了解并掌握多线程并行编程的相关知识。

2. 实验要求

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) 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，需要合理地分配文件给不同的线程。
4) 确保程序是线程安全的，并正确处理多个线程之间的同步问题。

2.2 算法描述

1. 总体框架

阅读 main.cpp 文件可知，总代码的主要用途是在给定文件夹中查找目标字符串，并输出具体位置。main()函数的流程如下：

1) 定义文件夹路径变量 folder_path 和目标字符串变量 target_string。
2) 初始化互斥锁 output_mutex，用于保护对输出的访问。
3) 调用 search_files 函数，传递文件夹路径和目标字符串作为参数，开始执行文件搜索任务。
4) 调用 pthread_mutex_destroy 函数销毁互斥锁 output_mutex。

实验中，需要定义 ThreadData 结构，并补齐 search_file()和 search_files()函数。同时还需要在 main()函数中加上对互斥锁的调用，核心代码如下：

```
// 初始化互斥锁
pthread_mutex_init(&output_mutex, NULL);
search_files(folder_path, target_string);
// 销毁互斥锁
pthread_mutex_destroy(&output_mutex);
```

2. 数据结构

为了确保多个线程之间对输出的安全访问，避免竞态条件和数据混乱等线程同步问题，首先需要定义一个全局范围的互斥锁，代码如下：

```
// 互斥锁，用于保护对输出的访问
pthread_mutex_t output_mutex = PTHREAD_MUTEX_INITIALIZER;
```

接着补充对 ThreadData 结构体的定义，主要包含要搜索的文件名和目标字符串，二者均是 char*类型，代码如下：

```
typedef struct {
    const char* filename;          // 要搜索的文件名
    const char* target_string;   // 目标字符串
} ThreadData;
```

3. 函数 search_file()

【输入输出】

输入为参数 arg，即指向 ThreadData 结构体的指针，包含了要搜索的文件名和目标字符串；无输出。

【算法流程】

1) 将 arg 强制转换为 ThreadData*类型，以获取要搜索的文件名和目标字符串。
2) 打开指定的文件，若打开失败，则输出错误信息并调用 pthread_exit 函数终止当前线程。初始化行号为 1。
4) 循环读取文件中的每一行，使用 fgets 函数读取一行文本，如果读取成功，继续执行下面的步骤；否则，结束循环。
5) 在当前行中使用 strstr 函数搜索目标字符串，如果找到匹配的字符串，进入临界区。在临界区中，使用互斥锁 pthread_mutex_lock 锁住输出，以确保只有一个线程可以访问输出。使用 printf 打印匹配的行，包括文件名、行号和行内容。打印后使用 pthread_mutex_unlock 解锁互斥锁，允许其他线程访问输出。
6) 行号增加 1，继续下一行的搜索。
7) 关闭文件，并使用 pthread_exit 函数终止当前线程。

【时间复杂度】

假设文件中有 N 行文本，每行的平均长度为 M。fopen、fgets 和 fclose 的时间复杂度为 O(1)。在循环中，对于每一行文本，使用 strstr 函数搜索目标字符串

的时间复杂度为 O(M)。因此，整个循环的时间复杂度为 O(N*M)。

【空间复杂度】

函数中使用了固定大小的字符数组 line[MAX_LINE_LENGTH]，其他变量和指针的空间复杂度为 O(1)。因此空间复杂度为 O(MAX_LINE_LENGTH)。

关键代码如下：

```
// 执行文本搜索
while (fgets(line, MAX_LINE_LENGTH, file) != NULL){
        if (strstr(line, data->target_string) != NULL){
                // 对输出进行同步
                pthread_mutex_lock(&output_mutex);
                printf("%s:%d: %s", data->filename, line_number, line);
                pthread_mutex_unlock(&output_mutex);
        }
        line_number++;
}

// 文件关闭&线程退出
fclose(file);
pthread_exit(NULL);
```

4. 函数 search_files()

【输入输出】

输入为参数 folder_path 和 target_string，分别表示要搜索的文件夹路径和目标字符串；无输出。

【算法流程】

1) 声明一个指向 DIR 结构体的指针 directory 和一个指向 dirent 结构体的指针 entry。使用 opendir 函数打开指定的文件夹，如果打开失败，则输出错误信息并返回。
2) 声明一个线程数组 threads，一个 ThreadData 结构体数组 thread_data，以及一个整型变量 num_threads，用于记录创建的线程数。
3) 在一个循环中，使用 readdir 函数遍历文件夹中的每个文件。如果遍历到的条目类型是普通文件（DT_REG），执行（4）的步骤。
4) 构建完整的文件路径，将文件路径和目标字符串分配给一个 ThreadData 结构体。使用 strdup 函数为文件路径分配内存，并将分配的指针赋值给 thread_data[num_threads].filename 。将目标字符串赋值给

thread_data[num_threads].target_string。使用 pthread_create 函数创建一个线程，将线程函数设置为 search_file，并将相应的 ThreadData 结构体的地址作为参数传递给线程。递增 num_threads 以记录创建的线程数。

5) 使用 closedir()关闭文件夹。

6) 在一个循环中，使用 pthread_join 函数等待所有线程的完成。函数执行完毕，返回。

算法流程图如图 2.1 所示。

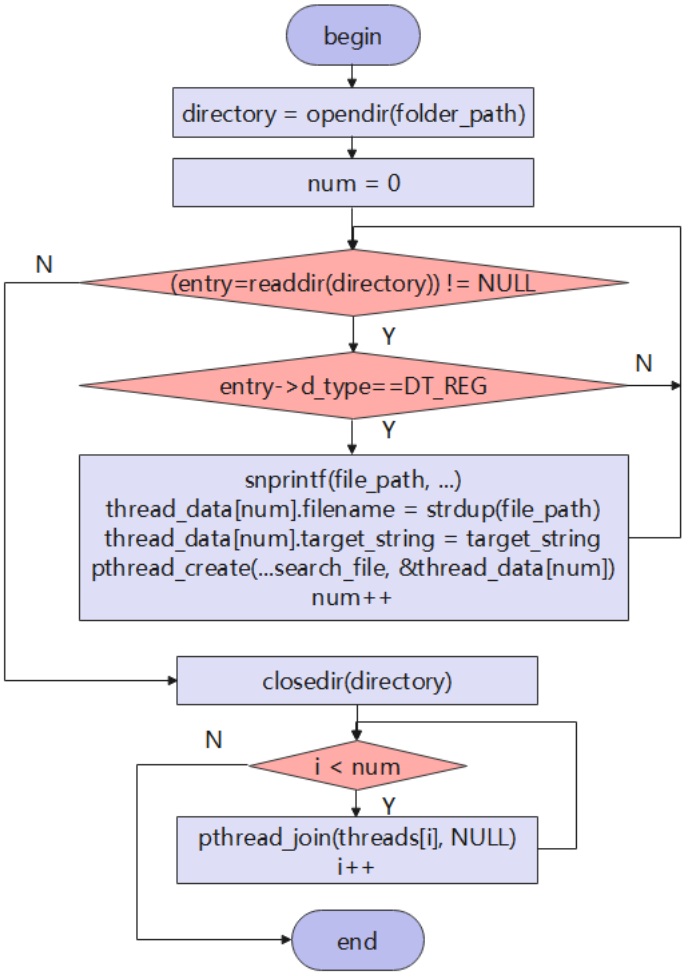

图 2.1 search_files()流程图

【时间复杂度】

假设文件夹下有 N 个文件。使用 opendir 和 closedir 的时间复杂度为 O(1)；使用 readdir 遍历文件夹的时间复杂度为 O(N)；在每个循环迭代中，都会创建一个线程，调用 pthread_create 函数的时间复杂度为 O(1)；最后使用 pthread_join 等待线程完成的时间复杂度为 O(N)；其他操作的时间复杂度为 O(1)。综上所述，

函数的时间复杂度为 O(N)。

【空间复杂度】

函数中使用了固定大小的线程数组 threads 和 thread_data 数组，它们的空间复杂度为 O(1)；使用 strdup 为每个文件路径分配内存，空间复杂度为 O(N)；其他变量和指针的空间复杂度为 O(1)。综上所述，函数的空间复杂度为 O(N)。

关键代码如下：

```
    // 执行搜索任务
    while ((entry = readdir(directory)) != NULL) {
        // 为每个线程分配任务
        if (entry->d_type == DT_REG){    // 只处理普通文件
            char file_path[256];
            snprintf(file_path, sizeof(file_path), "%s/%s", folder_path, entry->d_name);
            thread_data[num_threads].filename = strdup(file_path);
            thread_data[num_threads].target_string = target_string;
            pthread_create(&threads[num_threads], NULL, search_file,
&thread_data[num_threads]);
            num_threads++;
        }
    }
    closedir(directory);

    // 线程同步
    for (int i = 0; i < num_threads; i++)    pthread_join(threads[i], NULL);
```

2.3 实验方案

1. 开发与运行环境

【主机机带】RAM 8.00 GB (7.75 GB 可用)

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

【主机版本】Windows 10 专业版

【虚拟机版本】 Ubuntu 20.04.4 LTS 64-bit

【实验平台】Visual Studio Code 1.63.0

2. 实验方案描述

实验的主要步骤为添加互斥锁、定义 ThreadData 结构体、补充 main(), search_file(), search_files()函数。

在 search_file 函数中，使用 fgets 逐行读取文件内容，直到文件结束。对于每一行，使用 strstr 函数在该行中搜索目标字符串，如果找到匹配，则进入临界区。

在临界区中，使用互斥锁 pthread_mutex_lock 锁住输出，以确保只有一个线程可以访问输出。在此期间，打印匹配的行，包括文件名、行号和行内容。完成后，使用 pthread_mutex_unlock 解锁互斥锁，允许其他线程访问输出。

在 search_files 函数中，遍历文件夹中的每个文件，为每个普通文件创建一个线程。对于每个文件，构建完整的文件路径，并将文件路径和目标字符串分配给对应的 ThreadData 结构体。使用 pthread_create 创建线程，将线程函数设为 search_file，并传递对应的 ThreadData 结构体。

在 main 函数中，初始化互斥锁 pthread_mutex_init，以保护对输出的访问。在所有线程完成后，销毁互斥锁 pthread_mutex_destroy。

这样，每个线程会并行地搜索一个文件，并在找到匹配行时打印输出。通过使用互斥锁保护对输出的访问，确保了多个线程之间对输出的安全访问，避免了竞态条件和数据混乱等线程同步问题。

此外，在 search_files 函数中，通过调用 pthread_join 函数，主线程会等待所有线程的完成，确保所有线程都已完成任务后再继续执行后续代码。这样可以保证在主线程终止之前，所有线程都已经执行完毕，避免了线程之间的资源竞争和提前退出等问题。

2.4 实验结果与分析

根据以上方案实现 Pthreads 并行文本搜索，可以顺利通过 educoder 平台的测试。输入文件的目录“/data/workspace/myshixun/texts”和目标字符串“武汉”，可以从文件夹中多线程搜索，找出目标字符串的位置并正确输出“/data/workspace/myshixun/texts/cn.txt:386: 武汉”。运行结果如图 2.2 所示：

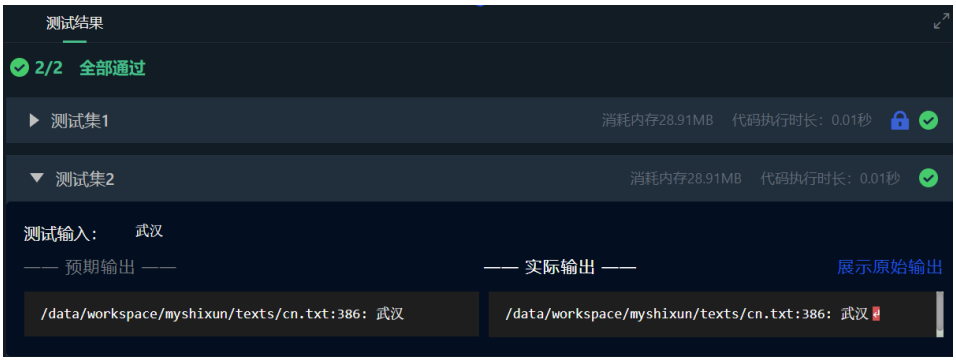

图 2.2 运行结果

3 实验小结

在本次实验中，我完成了两个任务，分别是设计使用 OpenMP 的并行程序进行矩阵乘法运算和设计多线程程序使用 Pthreads 进行并行文本搜索。通过这两个任务，我深入理解了并行编程的原理和实践，并有了一些自己的心得体会。

对于实验一，我使用 OpenMP 并行化了矩阵乘法运算，提高计算效率。在设计过程中，为了处理并行区域的同步，避免竞态条件和数据一致性问题，使用了 OpenMP 的并行循环指令，确保每个线程独立地计算乘法运算，并在结果累加时使用适当的同步机制。此外，我还利用了矩阵分块的思想，进一步提高了并行效率。通过这个任务，我深入了解了如何利用 OpenMP 实现并行计算，同时学会了处理并行化过程中的问题。

对于实验二，我设计了一个多线程程序，使用 Pthreads 实现并行文本搜索。在实现过程中，我首先补充了 ThreadData 数据结构的定义。然后，合理地将文件分配给不同的线程，确保每个线程处理一个文件。为了保证程序的线程安全性，我采用了适当的同步机制，例如互斥锁，以确保每个线程能够独立地搜索文件并打印匹配的行。通过这个任务，我学会了如何设计多线程程序并处理多个线程之间的同步问题，提高了程序的并发性和效率。

总的来说，通过完成这两个任务，我深入理解了并行编程的原理和实践，掌握了 OpenMP 和 Pthreads 的使用。我学会了将串行程序并行化，设计并实现并行算法，并解决并行化过程中可能遇到的问题。同时，我通过实验体会到并行编程在加速计算和提高程序性能方面的潜力。我相信这些知识和经验将对我未来的并行计算和优化工作有很大帮助。