总 评 成 绩	

《高性能计算》实验报告册

学年学期：2023 - 2024 学年第 2 学期

学生姓名：高星杰

学生学号：202130722071 2

专业年级：计算机科学与技术 21 级

任课教师：郑 芳

华中农业大学信息学院

2024 年 4 月

实 验 报 告 册

实验名称	实验三　OpenMP 并行程序设计	实 验 成 绩	
实验日期	2024 年 4 月 3 日星期三 第 9-12 节		

一、**实验目的**（描述本实验的学习目的及你对本实验的学习预期。）

1、掌握 OpenMP 并行编程的基本概念和操作，理解其在多线程环境中的应用。

2、学习使用 OpenMP 指令和库函数来编写并行程序，提高程序的运行效率。

3、熟悉 OpenMP 在处理大规模数据和复杂计算任务中的优势和局限。

4、探索不同并行算法在 OpenMP 环境下的性能表现，并与其他并行编程模型 （如 MPI）进行比较。

二、实验环境（请描述本实验教学活动所使用的实际环境。）

1、笔记本一台

2、Mac 系统

3、ITC Linux OpenMP 实验环境

三、实验任务（本实验要求的实验任务完成情况，未完成注明原因。）

1、完成用 OpenMP 编写完整的奇偶变换并行排序程序(对 100 万个数排序，与 MPI 相互比较)

2、完成用 OpenMP 编写完整的矩阵-向量乘法并行程序并使用了两种方式实现

3、编写了测试程序循环调用并行程序测试所有情况的性能并验证正确性

四、实验内容

要求：采用 OpenMP 编程实现以下两个题目。

(一）实验题目

（1） 用 OpenMP 编写完整的奇偶变换并行排序程序

提示：对 100 万个数排序，试着与 MPI 相互比较。

（2） 用 OpenMP 编写完整的矩阵-向量乘法并行程序。

请使用两种方法实现：

A. 用 OpenMP 线程分配的方式实现

B. 使用编译制导对 for 循环并行实现

(二）实验过程

（1） 集群使用 （以一题为示例即可）

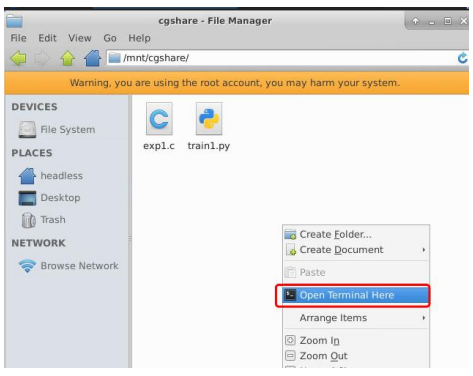
1) 在本地编写好 OpenMP c/c++语言程序

2) 在本地编译运行测试程序可行性和正确性

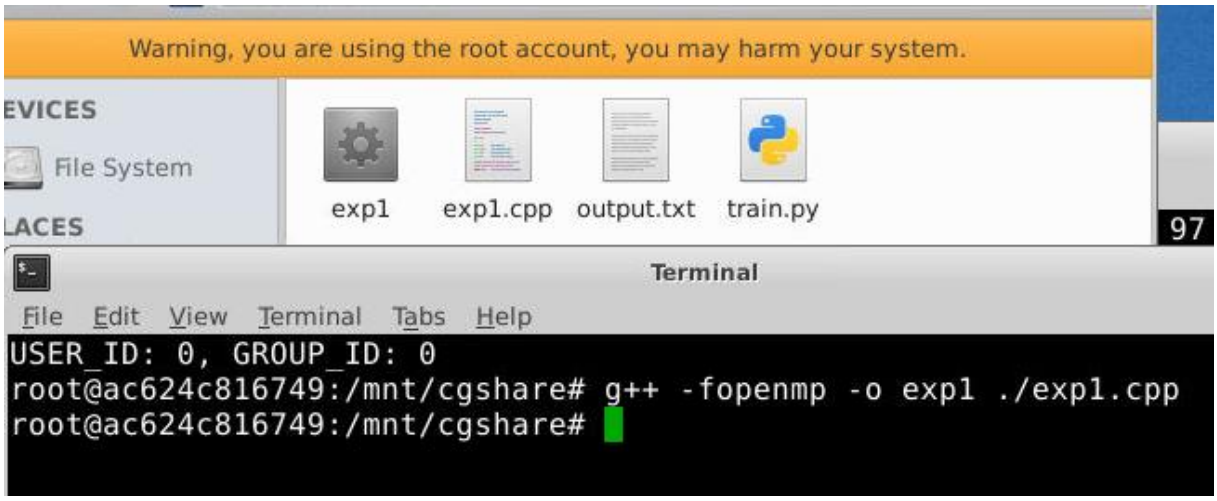
3) 编写测试 python 程序（可选）

4) 将代码上传至 itc OpenMP 环境

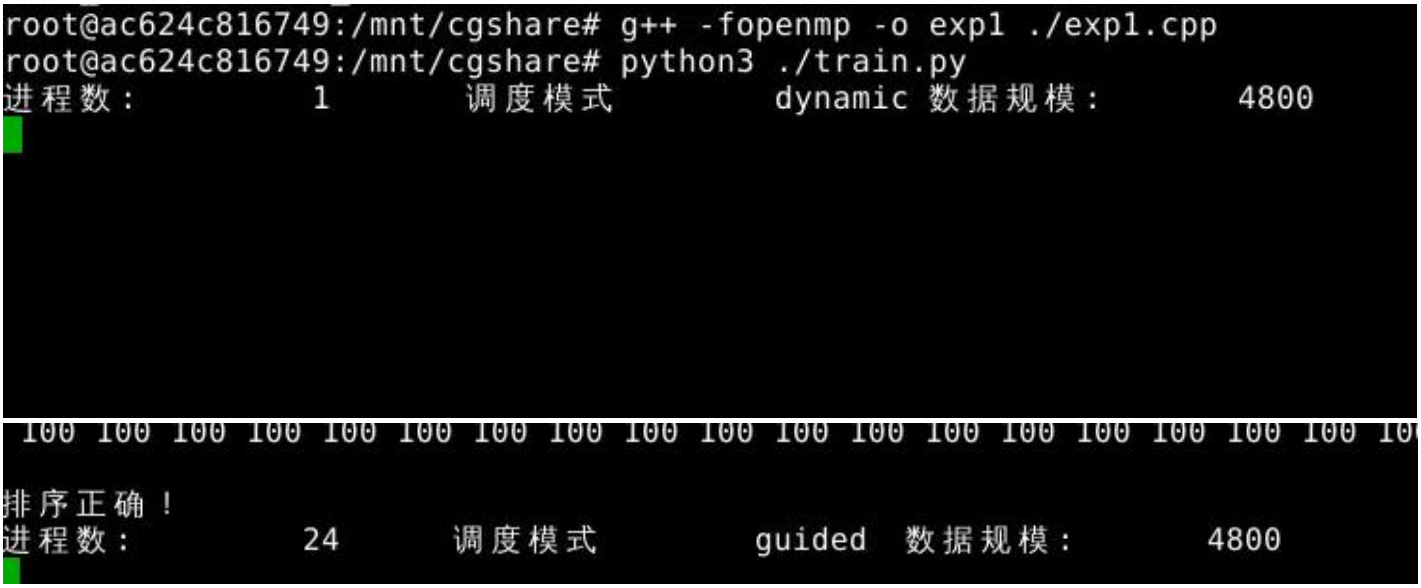
5) 在/mnt/cgshare 目录下打开终端



6) 使用命令行 g++ -fopenmp -o exp1 ./exp1.cpp 来编译源代码，生成可执行程序



7) 使用命令行./程序名 进程数 来运行程序（或者使用 python3 ./测试程序.py 直接循环测试所有数据规模和进程数）。



（2）源码及解析（每题一个源码，说明核心代码即可）

1. 用 OpenMP 编写完整的奇偶变换并行排序程序

原理：

奇偶排序（Odd-Even Sort）是一种简单的并行排序算法，适用于多线程环境。其核心思想是通过交替的奇数和偶数阶段，对相邻元素进行比较和交换，逐步将数组排序。

具体步骤如下：

奇数阶段：比较和交换所有奇数索引对，即索引为 (1,2)、(3,4)、(5,6) 等等。

偶数阶段：比较和交换所有偶数索引对，即索引为 (0,1)、(2,3)、(4,5) 等等。

重复上述两个阶段，直到数组完全有序。

在 OpenMP 环境中，通过并行化奇数和偶数阶段的循环，可以显著提高排序的效率。

思路：

①数据输入和初始化：读取待排序数组，并初始化相关变量。

②并行设置：使用 OpenMP 的并行指令设置并行线程数。

③奇偶排序：在每个阶段，根据奇偶性进行相应的比较和交换操作。

④结果输出：将排序后的数组输出，并记录排序时间。

代码+逐行注释：

```
#include <iostream> // 引入输入输出流库
#include <omp.h> // 引入 OpenMP 库，用于并行编程
#include <ctime> // 引入时间库，用于计时
#include <stdio.h> // 引入标准输入输出库
#include <stdlib.h> // 引入标准库，包含各种函数如转换、内存分配等
#include <string> // 引入字符串库


using namespace std; // 使用标准命名空间
#pragma warning(disable : 4996) // 禁用特定的编译器警告


// 设置 OpenMP 的调度类型
```

```
void set_schedule_type(string schedule_type)
{
    omp_sched_t sched_type; // 声明调度类型变量
    // 根据输入的字符串设置调度类型
    if (schedule_type == "static")
        sched_type = omp_sched_static;
    else if (schedule_type == "dynamic")
        sched_type = omp_sched_dynamic;
    else if (schedule_type == "guided")
        sched_type = omp_sched_guided;
    else
        sched_type = omp_sched_auto;
    omp_set_schedule(sched_type, 1); // 应用调度类型设置，块大小设置为 1
}

int main(int argc, char **argv)
{
    if (argc < 2) // 检查命令行参数数量是否足够
        return 0; // 参数不足，程序退出
    int thread_count = atoi(argv[1]); // 将命令行参数转换为线程数量
    int *a, n, i, tmp, phase; // 声明变量
    string schedule_type; // 声明调度类型字符串
```

```
    cout << "Please input the schedule type: "; // 请求输入调度类型
    cin >> schedule_type; // 读取调度类型
    cin >> n; // 读取数组大小
    cout << "Please input the array: "; // 请求输入数组
    a = new int[n]; // 动态分配数组内存
    for (i = 0; i < n; ++i) // 循环读取数组元素
        cin >> a[i];

    // 打印输入的信息
    cout << "Schedule type: " << schedule_type << endl;
    cout << "Thread count: " << thread_count << endl;
    cout << "Array size: " << n << endl;

    set_schedule_type(schedule_type); // 设置 OpenMP 调度类型
    clock_t starttime, endtime; // 声明计时变量
    starttime = clock(); // 开始计时

    // 并行区域开始
#pragma omp parallel num_threads(thread_count) default(none) shared(a, n) private(i, tmp, phase)
    // 奇偶排序算法主体
    for (phase = 0; phase < n; ++phase)
```

```
    {
        if (phase % 2 == 0) // 偶数阶段
        {
#pragma omp for schedule// 静态调度
            for (i = 1; i < n; i += 2) // 遍历偶数位置
            {
                if (a[i - 1] > a[i]) // 比较并交换
                {
                    swap(a[i], a[i - 1]);
                }
            }
        }
        else // 奇数阶段
        {
#pragma omp for // 默认调度
            for (i = 1; i < n - 1; i += 2) // 遍历奇数位置
            {
                if (a[i] > a[i + 1]) // 比较并交换
                {
                    swap(a[i], a[i + 1]);
                }
            }
```

```
        }
    }
    endtime = clock(); // 结束计时
    cout << "Sorted array: "; // 打印排序后的数组
    for (i = 0; i < n; ++i)
        cout << a[i] << " ";
    cout << endl;
    // 打印总计时时间
    cout << "Total time Serial: " << (double)(endtime - starttime) / CLOCKS_PER_SEC << " s" << endl
;
    delete[] a; // 释放数组内存
}
```

代码解析

①命令行参数处理：

- int thread_count = atoi(argv[1]);：从命令行获取线程数量。

②数据输入：

- cin >> n;：读取数组大小。
- a = new int[n];：动态分配数组。
- for (i = 0; i < n; ++i) cin >> a[i];：读取数组元素。

③计时器：

- clock_t starttime, endtime;：定义计时变量。
- starttime = clock();：开始计时。

④并行排序：

·#pragma omp parallel num_threads(thread_count) ...：设置并行区域，指定线程数。

·for (phase = 0; phase < n; ++phase)：外层循环，控制奇偶阶段的交替。

·奇数阶段和偶数阶段分别使用#pragma omp for 并行化。

⑤结果输出：

·ofstream out("output.txt");：将结果输出到文件 output.txt。

·for (i = 0; i < n; ++i) out << a[i] << " ";：输出排序后的数组。

⑥计时结束：

·endtime = clock();：结束计时。

·cout << "Total time Parallel: ...：输出排序时间。

⑦内存释放：

·delete[] a;：释放动态分配的内存。

通过这种方式，我们可以利用 OpenMP 实现奇偶变换排序的并行化，从而提升排序效率。

测试程序

由于当数据量很大的时候，在程序中生成太过麻烦，以及人工验证排序结果是否正确时几乎不可能，所以另外写一个程序，负责生成数据和验证正确性的 train.py，并且可以循环调用所有情况以测试性能

```
# coding=UTF-8
import subprocess
import random
```

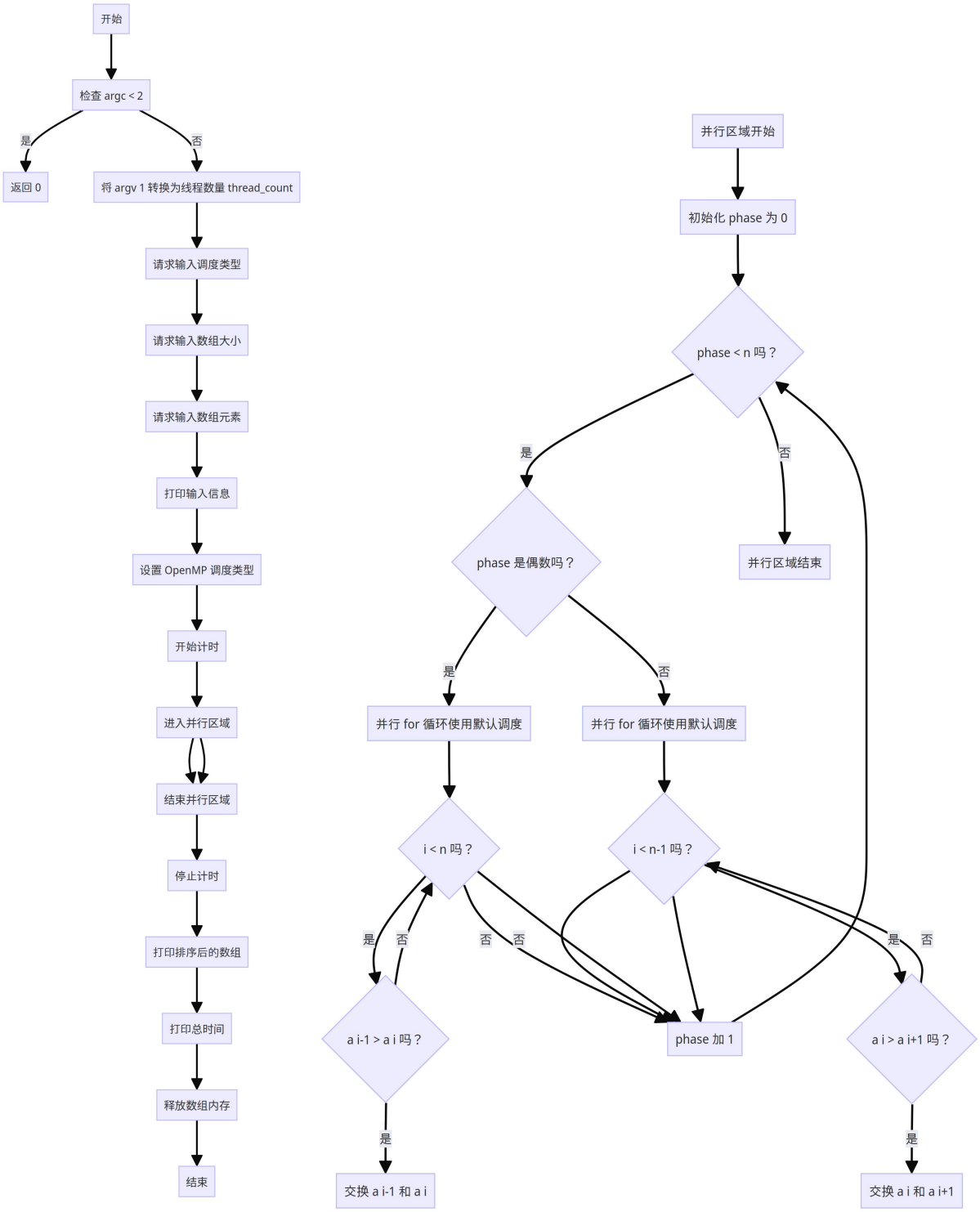
```python

def run_exp1_with_input(executable_path, thread_num, schedule_type, test_array):
    # 生成命令
    command = [executable_path, str(thread_num)]

    # 将输入转换为字符串，以便通过 stdin 传递
    input_str = (
        schedule_type
        + "\n"
        + str(len(test_array))
        + "\n"
        + str(" ".join(str(i) for i in test_array))
        + "\n"
    )
    # 使用 subprocess 运行命令，并提供输入
    result = subprocess.run(
        command,
        input=input_str,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        universal_newlines=True,
```

```
    )
    # 检查命令是否成功执行
    if result.returncode == 0:
        print("运行成功！")
        print("输出:\n" + result.stdout + "\n")
    else:
        print("运行失败！")
        print("错误信息:\n" + result.stderr + "\n")

    sorted_array = str(result.stdout).split("Sorted array: ")[1].split("\n")[0]
    print("排序后的数组: ", sorted_array)
    sorted_array = [int(i) for i in sorted_array.strip().split(" ")]
    if sorted_array == sorted(test_array):
        print("排序正确！")
    else:
        print("排序错误！")
    return str(result.stdout).split("Total time Serial: ")[1].split(" ")[0]


data_size = [4800, 2000, 4000, 8000, 16000, 3200, 6400, 12800]  # 测试的数据规模
process_num = [1, 2, 4, 16, 24, 32]  # 测试的进程数
schedule_type = ["dynamic", "guided", "runtime", "static"]  # 测试的调度类型
```

```
times = 10  # 每个数据规模下的测试次数
# 测试不同的数据规模
for size in data_size:
    for num in process_num:
        for schedule in schedule_type:
            t = 0
            for i in range(times):
                test_array = [random.randint(-100, 100) for _ in range(size)]
                str_print = (
                    "进程数: \t"
                    + str(num)
                    + "\t 调度模式\t"
                    + schedule
                    + "\t 数据规模: \t"
                    + str(size)
                )
                print(str_print)
                t += float(
                    run_exp1_with_input("./exp1", num, schedule, test_array).strip()
                )
            t /= times
            with open("output.txt", "a") as file:
```

```
            file.write(str_print + "\t 平均花费时间: \t" + str(t) + " us\n")

        print("average time: ", t)
```

流程图：



关键点：

- **数据分区**

将数组分成多个部分，每个线程处理数组的一部分，以便实现并行操作。在奇偶变换排序中，通过并行化内层循环，可以让不同的线程同时处理数组的不同部分。这种分区方式需要确保每个线程处理的数据不会相互干扰，从而保证排序的正确性。

- **OpenMP 并行区域**

使用#pragma omp parallel 指令定义并行区域，并指定线程数。所有在该区域内的代码将由多个线程并行执行。在并行区域内，OpenMP 会自动管理线程的创建、调度和同步，使得可以专注于并行算法的设计。

- **奇偶阶段的区分**

在奇数阶段和偶数阶段，比较和交换操作的索引不同。奇数阶段处理索引对（1,2）、（3,4）等，偶数阶段处理索引对（0,1）、（2,3）等。根据当前阶段的奇偶性，分别并行化相应的循环。这种阶段区分确保了在每个阶段内，不同线程之间不会发生数据竞争。

- **共享和私有变量**

在并行区域中，使用 shared 和 private 指令明确指定哪些变量是线程共享的，哪些是线程私有的。一般来说，数组和数组大小是共享的，而循环索引和临时变量是私有的。这样可以避免线程之间的变量冲突，提高并行程序的正确性和稳定性。

- **OpenMP 的并行 for 指令**

使用#pragma omp for 指令并行化循环，使得每个线程可以并行地执行循环中的代码。OpenMP 会自动将循环迭代分配给多个线程，从而实现并行处理。这种并行化方式非常适合于独立的循环迭代，不需要显式的同步操作。

- **线程同步**

奇偶变换排序的每个阶段都必须在所有线程完成当前阶段的比较和交换后，才能进入下一阶段。这确保了排序过程的正确性。OpenMP 的并行 for 循环自带隐式的同步点，因此不需要显式地添加同步指令。这种隐式同步确保了每个阶段的操作完成后，才会进入下一个阶段，从而保证排序的正确性。

2. 用 OpenMP 编写完整的矩阵-向量乘法并行程序

原理：

矩阵-向量乘法的基本原理是将一个矩阵 A 和一个向量 x 相乘，得到一个新的向量 b。具体计算方式如下：

$$b_i = \sum_{j=1}^{n} A_{ij} \cdot x_j$$

其中，A 是一个 m×n 的矩阵，x 是一个大小为 n 的向量，b 是结果向量，大小为 m。

并行化思路：

①数据分区： 将矩阵的行分配给不同的线程，每个线程计算若干行的结果向量。这样可以充分利用多核处理器的计算能力。

②OpenMP 并行区域： 使用 #pragma omp parallel 指令定义并行区域，并指定线程数。所有在该区域内的代码将由多个线程并行执行。

③循环并行化： 使用 #pragma omp for 指令并行化矩阵的行遍历，每个线程负责计算部分矩阵行的结果向量元素。

④共享和私有变量： 在并行区域中，使用 shared 和 private 指令明确指定哪些变量是线程共享的，哪些是线程私有的。矩阵 A、向量 x 和结果向量 b 是共享的，而行索引 i、列索引 j 和临时变量 sum 是私有的。

代码解析

方法 A：用 OpenMP 线程分配的方式实现

1) main 函数

- 功能：实现矩阵-向量乘法的并行计算，并输出计算结果和计算时间。
- 参数：
    - argc：命令行参数的数量。
    - argv：命令行参数的数组，包含线程数量。
- 实现逻辑：

①命令行参数检查：检查命令行参数是否足够。如果不足，输出用法提示并退出程序。

②获取线程数：从命令行参数中获取线程数量，并转换为整数。

③读取输入数据：

- 读取矩阵的行数和列数 m 和 n。
- 动态分配矩阵 A、向量 x 和结果向量 b 的内存。
- 初始化矩阵 A 和向量 x 的数据。

④并行计算：调用矩阵-向量乘法函数 matrix_vector_multiplication。

⑤输出结果：将计算结果输出。

⑥计算时间：使用 omp_get_wtime() 记录并输出计算时间。

⑦释放内存：释放动态分配的内存，防止内存泄漏。

2) matrix_vector_multiplication 函数

- 功能：使用 OpenMP 线程分配的方式计算矩阵和向量的乘积。
- 参数：

- A：输入矩阵，大小为 m x n。
- x：输入向量，大小为 n。
- b：输出向量，大小为 m，用于存储结果。
- m：矩阵的行数。
- n：矩阵的列数。
- thread_count：使用的线程数量。

- 实现逻辑：

①并行区域：使用 #pragma omp parallel 指令定义并行区域，并指定线程数。

②获取线程信息：每个线程获取自己的 ID 和总线程数。

③计算分配的行范围：每个线程计算自己负责的行范围。

④计算部分矩阵-向量乘法：每个线程并行计算自己负责的行，将结果存储到结果向量 b 中。

代码详细解析

```
#include <iostream> // 引入输入输出流库
#include <omp.h>    // 引入 OpenMP 库，用于并行编程
#include <ctime>    // 引入时间库，用于计时
#include <stdio.h>  // 引入标准输入输出库
#include <stdlib.h> // 引入标准库，包含各种函数如转换、内存分配等
#include <string>   // 引入字符串库
using namespace std;
// 函数：矩阵-向量乘法
// 功能：使用 OpenMP 线程分配的方式计算矩阵和向量的乘积
```

```
// 参数：
//  A - 输入矩阵，大小为 m x n
//  x - 输入向量，大小为 n
//  b - 输出向量，大小为 m，用于存储结果
//  m - 矩阵的行数
//  n - 矩阵的列数
//  thread_count - 使用的线程数量
void matrix_vector_multiplication(int **A, int *x, int *b, int m, int n, int thread_count)
{
// 开始并行区域
#pragma omp parallel num_threads(thread_count)
    {
        int thread_id = omp_get_thread_num();    // 获取当前线程 ID
        int num_threads = omp_get_num_threads(); // 获取线程总数

        int rows_per_thread = m / num_threads;                                  // 每个线
程处理的行数
        int start_row = thread_id * rows_per_thread;                            // 计算线
程处理的起始行
        int end_row = (thread_id == num_threads - 1) ? m : start_row + rows_per_thread; // 计算
线程处理的结束行
        // 遍历并计算分配给该线程的行
```

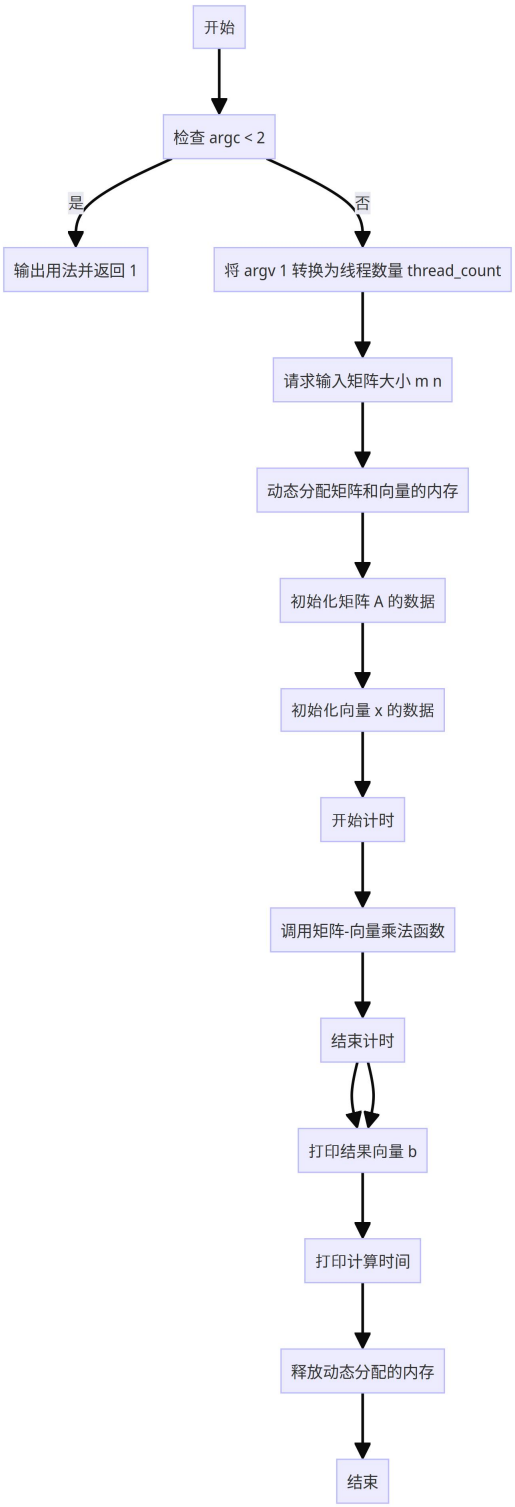
```
        for (int i = start_row; i < end_row; ++i)
        {
            int sum = 0; // 用于存储当前行的计算结果
            for (int j = 0; j < n; ++j)
            {
                sum += A[i][j] * x[j]; // 计算当前行与向量的点积
            }
            b[i] = sum; // 将结果存储到结果向量 b 中
        }
    }
}
int main(int argc, char **argv)
{
    if (argc < 2)
    { // 检查命令行参数是否足够
        cerr << "Usage: " << argv[0] << " <number_of_threads>" << endl;
        return 1;
    }
    int thread_count = atoi(argv[1]); // 获取线程数量
    int m, n; // 定义矩阵的行数和列数
    cout << "Please input the size of the matrix (m n): ";
    cin >> m >> n; // 读取矩阵的行数和列数
```

```
    // 动态分配矩阵和向量的内存
    int **A = new int *[m];
    for (int i = 0; i < m; ++i)
    {
        A[i] = new int[n];
    }
    int *x = new int[n];
    int *b = new int[m];
    //  初始化矩阵 A 的数据
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            A[i][j] = i * n + j;
        }
    }

    // 初始化向量 x 的数据
    for (int j = 0; j < n; ++j)
    {
        x[j] = j;
    }
```

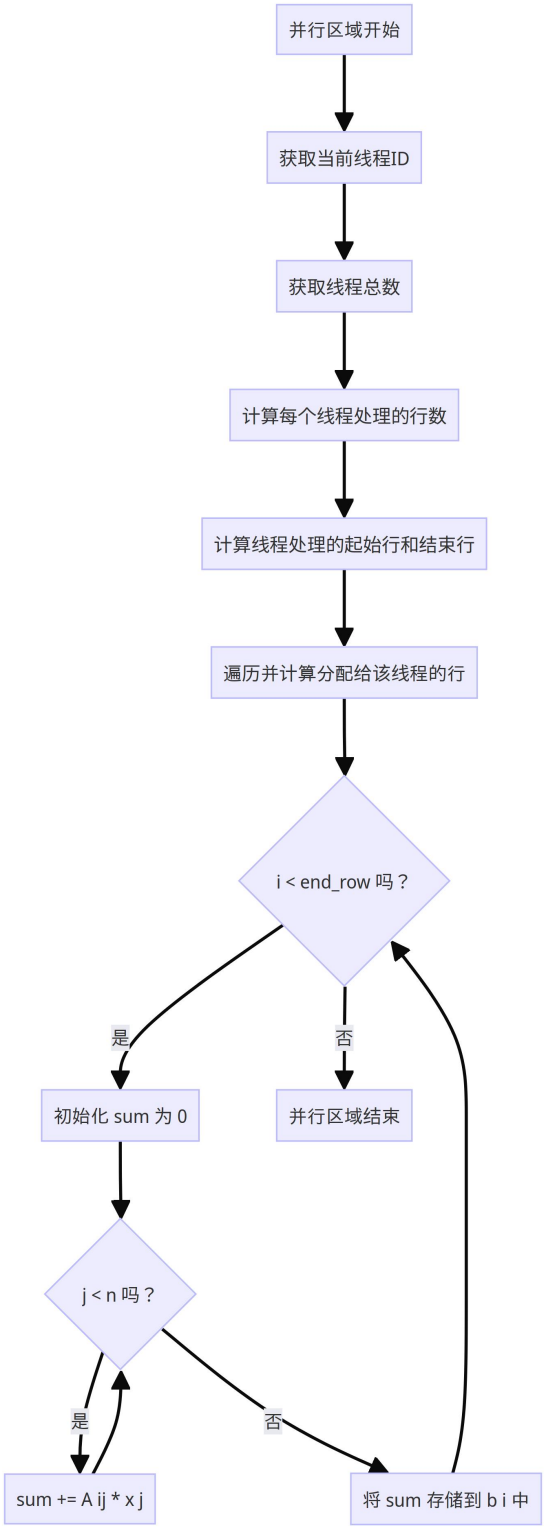
```
    double start_time = omp_get_wtime(); // 开始计时
    // 调用矩阵-向量乘法函数
    matrix_vector_multiplication(A, x, b, m, n, thread_count);
    double end_time = omp_get_wtime(); // 结束计时

    cout << "Result: ";
    for (int i = 0; i < m; ++i)
    {
        cout << b[i] << " "; // 将结果写入输出文件
    }
    cout << endl;
    cout << "Time taken: " << end_time - start_time << " seconds" << endl; // 输出计算时间
    // 释放动态分配的内存
    for (int i = 0; i < m; ++i)
    {
        delete[] A[i];
    }
    delete[] A;
    delete[] x;
    delete[] b;
    return 0;
}
```

流程图

主线程


开始
检查 argc < 2
是
否
输出用法并返回 1
将 argv 1 转换为线程数量 thread_count
请求输入矩阵大小 m n
动态分配矩阵和向量的内存
初始化矩阵 A 的数据
初始化向量 x 的数据
开始计时
调用矩阵-向量乘法函数
结束计时
打印结果向量 b
打印计算时间
释放动态分配的内存
结束

子线程


并行区域开始
获取当前线程ID
获取线程总数
计算每个线程处理的行数
计算线程处理的起始行和结束行
遍历并计算分配给该线程的行
i < end_row 吗？
是
否
初始化 sum 为 0
并行区域结束
j < n 吗？
是
否
sum += A ij * x j
将 sum 存储到 b i 中

方法 B：使用编译制导对 for 循环并行实现

1) main 函数

- 功能：实现矩阵-向量乘法的并行计算，并输出计算结果和计算时间。
- 参数：
  - argc：命令行参数的数量。
  - argv：命令行参数的数组，包含线程数量。
- 实现逻辑：

①命令行参数检查：检查命令行参数是否足够。如果参数不足，输出用法提示并退出程序。

②获取线程数：从命令行参数中获取线程数量，并转换为整数。

③读取输入数据：

- 读取矩阵的行数和列数 m 和 n。
- 动态分配矩阵 A、向量 x 和结果向量 b 的内存。
- 读取矩阵 A 和向量 x 的数据。

④并行计算：调用矩阵-向量乘法函数 matrix_vector_multiplication。

⑤输出结果：将计算结果输出。

⑥计算时间：使用 omp_get_wtime() 记录并输出计算时间。

⑦释放内存：释放动态分配的内存，防止内存泄漏。

⑧结束程序：返回 0 表示程序成功结束。

2) matrix_vector_multiplication 函数

- 功能：使用 OpenMP 编译制导指令并行化计算矩阵和向量的乘积。
- 参数：
  - A：输入矩阵，大小为 m x n。

- x：输入向量，大小为 n。
- b：输出向量，大小为 m，用于存储结果。
- m：矩阵的行数。
- n：矩阵的列数。
- thread_count：使用的线程数量。

- 实现逻辑：

①并行 for 循环：使用 #pragma omp parallel for 指令并行化矩阵行的遍历，每个线程负责计算部分矩阵行的结果向量元素。

代码详细解析

```
#include <iostream> // 引入输入输出流库
#include <omp.h>    // 引入 OpenMP 库，用于并行编程
#include <ctime>    // 引入时间库，用于计时
#include <stdio.h>  // 引入标准输入输出库
#include <stdlib.h> // 引入标准库，包含各种函数如转换、内存分配等
#include <string>   // 引入字符串库
using namespace std;


void set_schedule_type(string schedule_type)
{
    omp_sched_t sched_type; // 声明调度类型变量
    // 根据输入的字符串设置调度类型
    if (schedule_type == "static")
```

```
        sched_type = omp_sched_static;
    else if (schedule_type == "dynamic")
        sched_type = omp_sched_dynamic;
    else if (schedule_type == "guided")
        sched_type = omp_sched_guided;
    else
        sched_type = omp_sched_auto;
    omp_set_schedule(sched_type, 1); // 应用调度类型设置，块大小设置为 1
}

// 函数：矩阵-向量乘法
// 功能：使用 OpenMP 编译制导指令并行化计算矩阵和向量的乘积
// 参数：
//  A - 输入矩阵，大小为 m x n
//  x - 输入向量，大小为 n
//  b - 输出向量，大小为 m，用于存储结果
//  m - 矩阵的行数
//  n - 矩阵的列数
//  thread_count - 使用的线程数量
void matrix_vector_multiplication(int **A, int *x, int *b, int m, int n, int thread_count, string sc
hedule_type)
{
```

```
    set_schedule_type(schedule_type); // 设置 OpenMP 调度类型


// 并行 for 循环，遍历矩阵的每一行
#pragma omp parallel for num_threads(thread_count) shared(A, x, b)
    for (int i = 0; i < m; ++i)
    {
        int sum = 0; // 用于存储当前行的计算结果
        for (int j = 0; j < n; ++j)
        {
            sum += A[i][j] * x[j]; // 计算当前行与向量的点积
        }
        b[i] = sum; // 将结果存储到结果向量 b 中
    }
}


int main(int argc, char **argv)
{


    if (argc < 2)
    { // 检查命令行参数是否足够
        cerr << "Usage: " << argv[0] << " <number_of_threads>" << endl;
        return 1;
```

```
    }

    int thread_count = atoi(argv[1]); // 获取线程数量

    int m, n;       // 定义矩阵的行数和列数
    cin >> m >> n; // 读取矩阵的行数和列数

    string schedule_type;                    // 声明调度类型字符串
    cout << "Please input the schedule type: "; // 请求输入调度类型
    cin >> schedule_type;                    // 读取调度类型

    // 动态分配矩阵和向量的内存
    int **A = new int *[m];
    for (int i = 0; i < m; ++i)
    {
        A[i] = new int[n];
     }

    int *x = new int[n];
    int *b = new int[m];

    // 读取矩阵 A 的数据
```

```
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            srand(10 + i * n + j);
            A[i][j] = rand() % 100;
        }
    }

    // 读取向量 x 的数据
    for (int j = 0; j < n; ++j)
    {
        srand(j);
        x[j] = rand() % 100;
    }

    double start_time = omp_get_wtime(); // 开始计时

    // 调用矩阵-向量乘法函数
    matrix_vector_multiplication(A, x, b, m, n, thread_count, schedule_type);

    double end_time = omp_get_wtime(); // 结束计时
```

```
    cout << "Result: ";
    for (int i = 0; i < m; ++i)
    {
        cout << b[i] << " "; // 输出结果
    }
    cout << endl;

    cout << "Time taken: " << end_time - start_time << " seconds" << endl; // 输出计算时间

    // 释放动态分配的内存
    for (int i = 0; i < m; ++i)
    {
        delete[] A[i];
    }
    delete[] A;
    delete[] x;
    delete[] b;

    return 0;
}
```

流程图

图 1

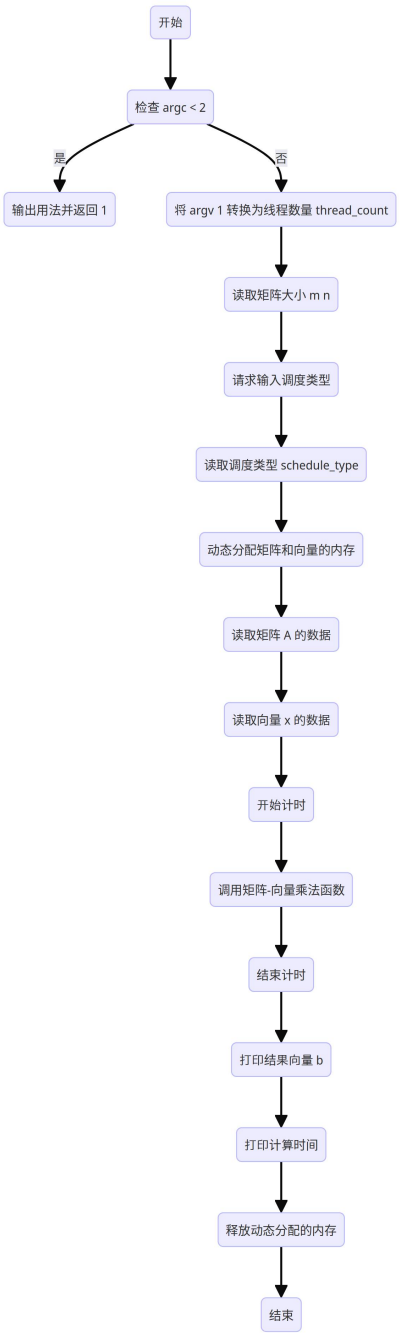
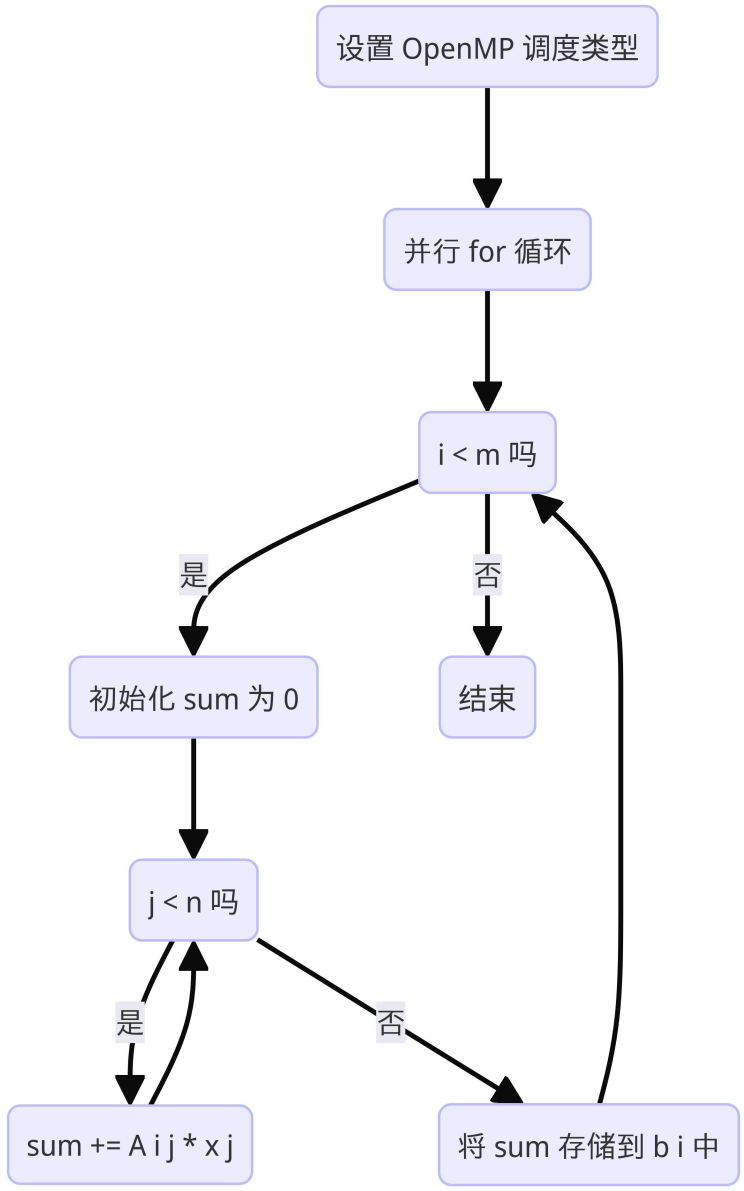


图 2



(三）执行时间（每题都需要有以下分析）

这里为了方便测试我编写了 Python 程序来循环调用程序，并可以实现计算平均值等，

并输入不同的数据，包含下列所有测试场景，将结果输出并且保存到 output.txt 中

任务一测试程序代码：

```
# coding=UTF-8
import subprocess
import random



def run_exp1_with_input(executable_path, thread_num, schedule_type, test_array):
    # 生成命令
    command = [executable_path, str(thread_num)]

    # 将输入转换为字符串，以便通过 stdin 传递
    input_str = (
        schedule_type
        + "\n"
        + str(len(test_array))
        + "\n"
        + str(" ".join(str(i) for i in test_array))
        + "\n"
    )
    # 使用 subprocess 运行命令，并提供输入
    result = subprocess.run(
```

```
        command,
        input=input_str,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        universal_newlines=True,
    )

    # 检查命令是否成功执行
    if result.returncode == 0:
        print("运行成功！")
        print("输出:\n" + result.stdout + "\n")
    else:
        print("运行失败！")
        print("错误信息:\n" + result.stderr + "\n")

    sorted_array = str(result.stdout).split("Sorted array: ")[1].split("\n")[0]
    print("排序后的数组: ", sorted_array)
    sorted_array = [int(i) for i in sorted_array.strip().split(" ")]
    if sorted_array == sorted(test_array):
        print("排序正确！")
    else:
        print("排序错误！")
```

```
    return str(result.stdout).split("Total time Serial: ")[1].split(" ")[0]

data_size = [4800, 2000, 4000, 8000, 16000, 3200, 6400, 12800]  # 测试的数据规模

process_num = [1, 2, 4, 16, 24, 32]  # 测试的进程数

schedule_type = ["dynamic", "guided", "runtime", "static"]  # 测试的调度类型

times = 10  # 每个数据规模下的测试次数

# 测试不同的数据规模

for size in data_size:

    for num in process_num:

        for schedule in schedule_type:

            t = 0

            for i in range(times):

                test_array = [random.randint(-100, 100) for _ in range(size)]

                str_print = (

                    "进程数: \t"

                    + str(num)

                    + "\t 调度模式\t"

                    + schedule

                    + "\t 数据规模: \t"

                    + str(size)

                )

                print(str_print)

                t += float(
```

```
                run_exp1_with_input("./exp1", num, schedule, test_array).strip()
            )
        t /= times
        with open("output.txt", "a") as file:
            file.write(str_print + "\t 平均花费时间: \t" + str(t) + " us\n")
        print("average time: ", t)
```

这段程序可以实现测试所有场景，把测试的结果放入输出文件中，并且还能验证执行程序的结果的正确性。

测试时间示例，可以参考以下测试列表进行测试

测试场景	进程数	数据规模	测试次数
1	1（串行）	4800	10
2	2（并行）	4800	10
3	4（并行）	4800	10
4	16（并行）	4800	10
5	24（并行）	4800	10
6	32（并行）	4800	10
7	1（串行）	2000	10
8	1（串行）	4000	10
9	1（串行）	8000	10
10	1（串行）	16000	10
11	16（并行）	3200	10

12	16（并行）	6400	10
13	16（并行）	12800	10
14	16（并行）	16000	10

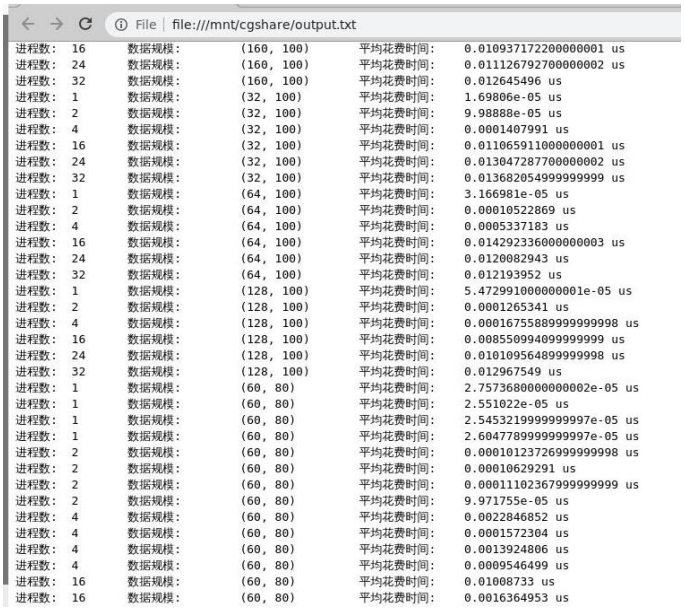
（1）执行时间截图

由于数据量太大，要列出所有情况截图所占空间太大，仅截几张图作为示例。

- 矩阵向量乘法

```
root@b35b8a50500c:/mnt/cgshare# python3 ./train2.py
```

```
进程数：         32      数据规模：       (128, 100)
运行成功！
输出：
Please input the size of the matrix (m n): Result: 328350 823350
 2308350 2803350 3298350 3793350 4288350 4783350 5278350 5773350
 7258350 7753350 8248350 8743350 9238350 9733350 10228350 1072335
3350 12208350 12703350 13198350 13693350 14188350 14683350 151783
68350 16663350 17158350 17653350 18148350 18643350 19138350 19633
623350 21118350 21613350 22108350 22603350 23098350 23593350 2408
5078350 25573350 26068350 26563350 27058350 27553350 28048350 285
29533350 30028350 30523350 31018350 31513350 32008350 32503350 32
 33988350 34483350 34978350 35473350 35968350 36463350 36958350 3
0 38443350 38938350 39433350 39928350 40423350 40918350 41413350
50 42898350 43393350 43888350 44383350 44878350 45373350 45868350
350 47353350 47848350 48343350 48838350 49333350 49828350 5032335
3350 51808350 52303350 52798350 53293350 53788350 54283350 547783
68350 56263350 56758350 57253350 57748350 58243350 58738350 59233
223350 60718350 61213350 61708350 62203350 62698350 63193350
Time taken: 0.0201915 seconds


average time:  0.021367329999999997
```



➢ 奇偶数排列

```
root@b35b8a50500c:/mnt/cgshare# mpicxx -o exp3 ./exp3.cpp -std=c++11
root@b35b8a50500c:/mnt/cgshare# python3 ./train3.py
```

```
7 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 98 98 98
98 98 98 98 98 98 98 98 98 98 98 98 98 99 99 99 99 99 99 99 99 99 99 99 99 9
 99 99 99 99 99 99 99 99 100 100 100 100 100 100 100 100 100 100 100 100 100
 100 100 100 100 100 100 100 100
排序正确！
进程数：         16      调度模式         dynamic 数据规模：       4800
```

```
 96 96 96 96 96 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 9
7 97 97 97 98 98 98 98 98 98 98 98 98 98 98 98 98 98 98 98 98 98 98 98 98 98 98
98 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99
 99 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100
排序正确！
进程数：         32      调度模式         guided  数据规模：       4800
```

File | file:///mnt/cgshare/output.txt

进程数：	1	调度模式	dynamic	数据规模：	4800	平均花费时间：	0.11391130000000001 us
进程数：	1	调度模式	guided	数据规模：	4800	平均花费时间：	0.1137717 us
进程数：	1	调度模式	runtime	数据规模：	4800	平均花费时间：	0.11367350000000001 us
进程数：	1	调度模式	static	数据规模：	4800	平均花费时间：	0.11362969999999999 us
进程数：	2	调度模式	dynamic	数据规模：	4800	平均花费时间：	0.11841350000000002 us
进程数：	2	调度模式	guided	数据规模：	4800	平均花费时间：	0.1175711 us
进程数：	2	调度模式	runtime	数据规模：	4800	平均花费时间：	0.11500160000000001 us
进程数：	2	调度模式	static	数据规模：	4800	平均花费时间：	0.11581090000000001 us
进程数：	4	调度模式	dynamic	数据规模：	4800	平均花费时间：	0.23779589999999998 us
进程数：	4	调度模式	guided	数据规模：	4800	平均花费时间：	0.16163080000000002 us
进程数：	4	调度模式	runtime	数据规模：	4800	平均花费时间：	0.1994317 us
进程数：	4	调度模式	static	数据规模：	4800	平均花费时间：	0.16244530000000001 us
进程数：	16	调度模式	dynamic	数据规模：	4800	平均花费时间：	2.777843 us
进程数：	16	调度模式	guided	数据规模：	4800	平均花费时间：	1.982484 us
进程数：	16	调度模式	runtime	数据规模：	4800	平均花费时间：	1.5937807 us
进程数：	16	调度模式	static	数据规模：	4800	平均花费时间：	1.7347549999999998 us
进程数：	24	调度模式	dynamic	数据规模：	4800	平均花费时间：	3.497203999999999 us
进程数：	24	调度模式	guided	数据规模：	4800	平均花费时间：	4.682515 us
进程数：	24	调度模式	runtime	数据规模：	4800	平均花费时间：	5.106287 us
进程数：	1	调度模式	dynamic	数据规模：	4800	平均花费时间：	0.11357949999999999 us
进程数：	1	调度模式	guided	数据规模：	4800	平均花费时间：	0.11348150000000001 us
进程数：	1	调度模式	runtime	数据规模：	4800	平均花费时间：	0.113474 us
进程数：	1	调度模式	static	数据规模：	4800	平均花费时间：	0.11326560000000001 us
进程数：	2	调度模式	dynamic	数据规模：	4800	平均花费时间：	0.1210805 us
进程数：	2	调度模式	guided	数据规模：	4800	平均花费时间：	0.1155659 us
进程数：	2	调度模式	runtime	数据规模：	4800	平均花费时间：	0.1159678 us
进程数：	2	调度模式	static	数据规模：	4800	平均花费时间：	0.11438750000000002 us
进程数：	4	调度模式	dynamic	数据规模：	4800	平均花费时间：	0.24039709999999997 us
进程数：	4	调度模式	guided	数据规模：	4800	平均花费时间：	0.14632649999999997 us
进程数：	4	调度模式	runtime	数据规模：	4800	平均花费时间：	0.1652692 us
进程数：	4	调度模式	static	数据规模：	4800	平均花费时间：	0.1498723 us
进程数：	16	调度模式	dynamic	数据规模：	4800	平均花费时间：	2.271433 us
进程数：	16	调度模式	guided	数据规模：	4800	平均花费时间：	2.041836 us
进程数：	16	调度模式	runtime	数据规模：	4800	平均花费时间：	2.2866779999999998 us
进程数：	16	调度模式	static	数据规模：	4800	平均花费时间：	2.284916 us

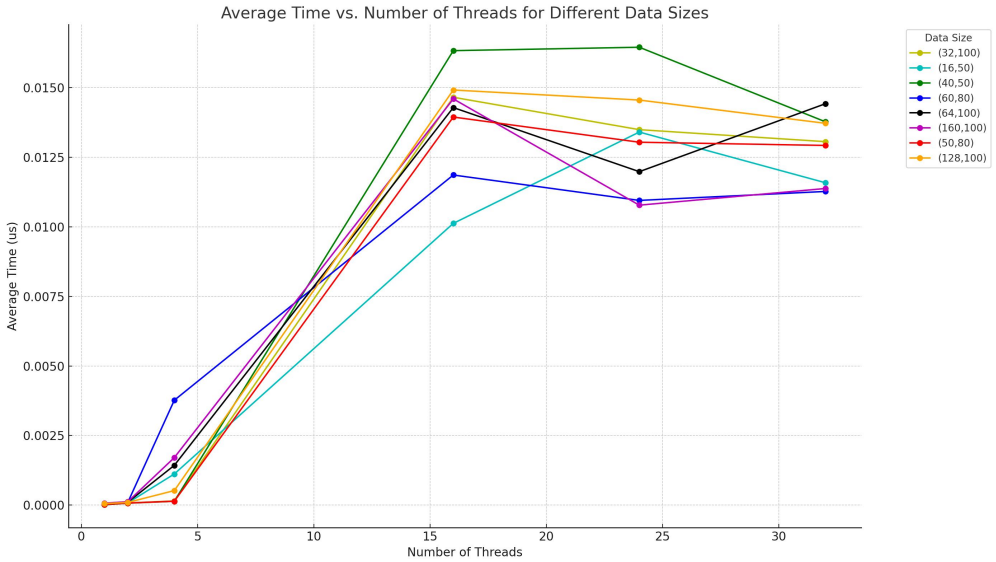
（2）执行时间分析（表格）

- 矩阵向量乘法（OpenMP 线程分配的方式实现）

① 执行时间表格分析

测试场景	进程数	数据规模	平均花费时间 (us)	调度模式 (schedule)	次数
1	1（串行）	(60, 80)	2.55843e-05	默认	10
2	2（并行）	(60, 80)	9.61559e-05	默认	10
3	4（并行）	(60, 80)	0.003768932	默认	10
4	16（并行）	(60, 80)	0.011865304	默认	10
5	24（并行）	(60, 80)	0.010952522	默认	10
6	32（并行）	(60, 80)	0.011274377	默认	10
7	1（串行）	(40, 50)	1.25844e-05	默认	10
8	2（并行）	(40, 50)	6.94496e-05	默认	10
9	4（并行）	(40, 50)	0.000138233	默认	10
10	16（并行）	(40, 50)	0.016335678	默认	10
11	24（并行）	(40, 50)	0.01645854	默认	10
12	32（并行）	(40, 50)	0.013780235	默认	10
13	1（串行）	(50, 80)	1.99386e-05	默认	10
14	2（并行）	(50, 80)	7.42505e-05	默认	10
15	4（并行）	(50, 80)	0.000138720	默认	10
16	16（并行）	(50, 80)	0.01394924	默认	10
17	24（并行）	(50, 80)	0.013044059	默认	10
18	32（并行）	(50, 80)	0.012928418	默认	10
19	1（串行）	(16, 50)	1.132638e-5	默认	10
20	2（并行）	(16, 50)	7.962323e-0	默认	10
21	4（并行）	(16, 50)	0.001117862	默认	10
22	16（并行）	(16, 50)	0.010127115	默认	10
23	24（并行）	(16, 50)	0.013409592	默认	10
24	32（并行）	(16, 50)	0.01158868	默认	10
25	1（串行）	(160, 100)	6.977673e-0	默认	10
26	2（并行）	(160, 100)	0.000121488	默认	10
27	4（并行）	(160, 100)	0.001706645	默认	10
28	16（并行）	(160, 100)	0.01460144	默认	10
29	24（并行）	(160, 100)	0.010781626	默认	10
30	32（并行）	(160, 100)	0.011380223	默认	10

31	1（串行）	(32, 100)	1.76562e-05	默认	10
32	2（并行）	(32, 100)	7.36476e-05	默认	10
33	4（并行）	(32, 100)	0.000150117	默认	10
34	16（并行）	(32, 100)	0.0146602	默认	10
35	24（并行）	(32, 100)	0.013495797	默认	10
36	32（并行）	(32, 100)	0.013064592	默认	10
37	1（串行）	(64, 100)	3.17683e-05	默认	10
38	2（并行）	(64, 100)	0.000103356	默认	10
39	4（并行）	(64, 100)	0.001427712	默认	10
40	16（并行）	(64, 100)	0.0142895	默认	10
41	24（并行）	(64, 100)	0.011985934	默认	10
42	32（并行）	(64, 100)	0.014426557	默认	10
43	1（串行）	(128, 100)	5.41083e-05	默认	10
44	2（并行）	(128, 100)	9.59022e-05	默认	10
45	4（并行）	(128, 100)	0.000521162	默认	10
46	16（并行）	(128, 100)	0.014920841	默认	10
47	24（并行）	(128, 100)	0.01456003	默认	10
48	32（并行）	(128, 100)	0.01372399	默认	10



我们可以发现：

线程数量增加，平均花费时间显著增加：

不同数据规模的测试结果表明，随着线程数量从 1 增加到 32，平均花费时间显著增加。显然，并行计算在这些场景下并没有带来预期的性能提升。

小数据规模的影响：

数据规模较小（例如 (16, 50) 和 (32, 100)）时，单线程的性能优于多线程。线程管理、同步等开销在小规模数据计算中占比更高，因此增加线程数量反而增加了计算时间。

线程管理开销：

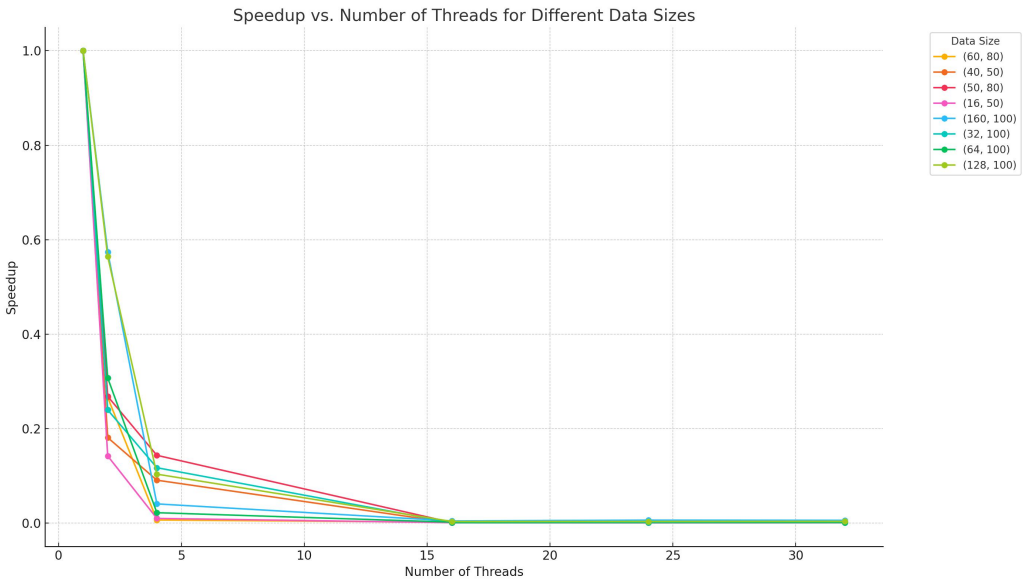
线程数量增加带来的管理开销是显著的。当线程数较多时，系统在管理和调度线程上花费的时间增加，导致总体花费时间增加。

负载不均衡：

多线程环境下，任务分配可能存在不均衡。某些线程可能比其他线程处理更多的工作，从而拖累整体计算效率。

② 执行加速比分析

线程数	(60, 80)	(40, 50)	(50, 80)	(16, 50)	(160, 100)	(32, 100)	(64, 100)	(128, 100)
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	0.266	0.181	0.268	0.142	0.574	0.240	0.307	0.564
4	0.0068	0.091	0.144	0.010	0.041	0.117	0.022	0.104
16	0.0022	0.00077	0.0014	0.0011	0.0048	0.0012	0.0022	0.0036
24	0.0023	0.00077	0.0015	0.00085	0.0065	0.0013	0.0027	0.0037
32	0.0023	0.00091	0.0015	0.00098	0.0061	0.0014	0.0022	0.0039



从加速比表格和图中，我们可以观察到以下几点：

加速比低：

对于所有数据规模，随着线程数量的增加，加速比并没有显著提升。这表明并行化在这些场景下并没有带来预期的性能提升。

小数据规模：

对于较小的数据规模（例如(16, 50) 和 (32, 100)），加速比非常低。这是因为并行化开销相对于计算本身的开销显得过高，导致并行化效果不佳。

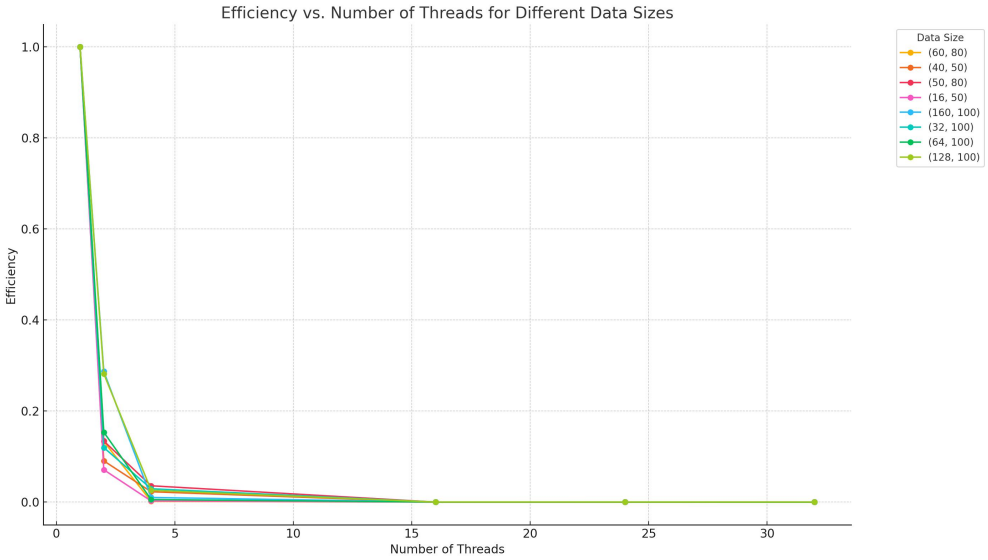
负加速：

在某些情况下，例如线程数量增加到 16、24 或 32 时，加速比反而降低，甚至出现负加速的情况。这说明并行化开销（例如线程管理、同步等）大于并行计算带来的性能提升

③ 执行效率分析

线程数	(60, 80)	(40, 50)	(50, 80)	(16, 50)	(160, 100)	(32, 100)	(64, 100)	(128, 100)
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	0.133	0.091	0.134	0.071	0.287	0.120	0.154	0.282
4	0.0017	0.0228	0.0359	0.0025	0.0102	0.0294	0.0056	0.0260
16	0.00014	0.000048	0.000089	0.000070	0.000299	0.000075	0.000139	0.000227

24	0.000097	0.000032	0.000064	0.000035	0.000270	0.000055	0.000110	0.000155
32	0.000071	0.000029	0.000048	0.000031	0.000192	0.000042	0.000069	0.000123



总体规律总结

从执行效率表格和图中，我们可以观察到以下几点：

执行效率随着线程数量增加而显著降低：

对于所有数据规模，执行效率随着线程数量从 1 增加到 32 显著降低。这表明多线程并行计算的效率在增加线程数量时急剧下降。

小数据规模效率更低：

对于较小的数据规模（例如(16, 50) 和 (32, 100)），执行效率更低。这是因为在小规模数据的计算中，并行化的开销相对更高，导致整体效率下降。

较大数据规模的效率稍高：

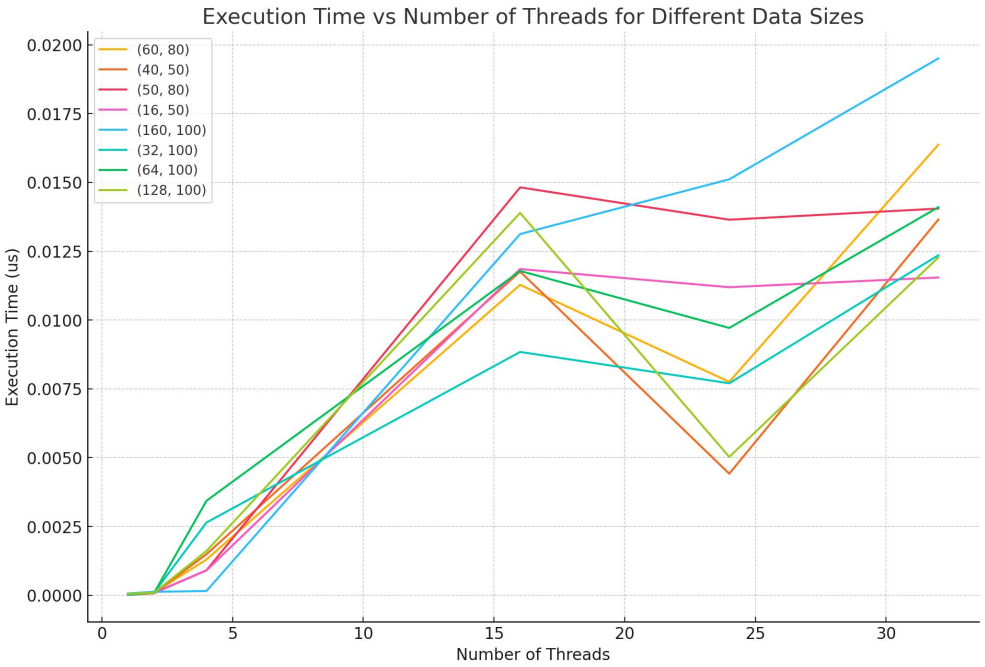
在较大数据规模（例如(160, 100) 和 (128, 100)）下，虽然执行效率也很低，但相对于小数据规模有稍高的效率。这是因为在较大数据规模下，并行化的优势稍微显现，但仍未能抵消并行化开销。

- 矩阵向量相乘（编译制导对 for 循环并行实现）

① 执行时间表格分析

测试场景	线程数	数据规模	调度模式 (schedule)	次数	执行时间 (us)
1	1	(60, 80)	dynamic	10	2.5670e-05
1	1	(60, 80)	guided	10	2.50758e-05
1	1	(60, 80)	runtime	10	2.4739e-05
1	1	(60, 80)	static	10	2.48546e-05
2	2	(60, 80)	dynamic	10	9.89894e-05
2	2	(60, 80)	guided	10	8.22354e-05
2	2	(60, 80)	runtime	10	7.43147e-05
2	2	(60, 80)	static	10	8.49449e-05
3	4	(60, 80)	dynamic	10	0.001296748
3	4	(60, 80)	guided	10	0.000718186
3	4	(60, 80)	runtime	10	0.001320528
3	4	(60, 80)	static	10	0.001912046
4	16	(60, 80)	dynamic	10	0.011279629
4	16	(60, 80)	guided	10	0.003604838
4	16	(60, 80)	runtime	10	0.002659730
4	16	(60, 80)	static	10	0.006483026
5	24	(60, 80)	dynamic	10	0.007753432
5	24	(60, 80)	guided	10	0.005757140
5	24	(60, 80)	runtime	10	0.003150249
5	24	(60, 80)	static	10	0.008011933
6	32	(60, 80)	dynamic	10	0.01637145
6	32	(60, 80)	guided	10	0.013591544
6	32	(60, 80)	runtime	10	0.005241335
6	32	(60, 80)	static	10	0.004564391
7	16	(40, 50)	dynamic	10	0.011852339
7	16	(40, 50)	guided	10	0.004981617
7	16	(40, 50)	runtime	10	0.001901137
7	16	(40, 50)	static	10	0.001916913
8	16	(50, 80)	dynamic	10	0.014819960
8	16	(50, 80)	guided	10	0.017012530

8	16	(50, 80)	runtime	10	0.007779335
8	16	(50, 80)	static	10	0.005064598
9	16	(160, 100)	dynamic	10	0.013123949
9	16	(160, 100)	guided	10	0.003263643
9	16	(160, 100)	runtime	10	0.002176491
9	16	(160, 100)	static	10	0.001641581
10	16	(32, 100)	dynamic	10	0.008839805
10	16	(32, 100)	guided	10	0.016177432
10	16	(32, 100)	runtime	10	0.005491923
10	16	(32, 100)	static	10	0.003876567



分析

从执行时间折线图中可以观察到以下几点规律和分析：

执行时间的变化趋势：

随着线程数的增加，执行时间并没有总是减少。在一些情况下，增加线程数反而导致执行时间增加。这表明线程管理的开销以及线程间通信的成本可能抵消了并行计算带来的加速效果。

数据规模的影响：

不同的数据规模对执行时间有显著影响。较小的数据规模（如 (16, 50) 和 (32, 100)）的执行时间总体

较短，而较大的数据规模（如 (160, 100) 和 (128, 100)）的执行时间总体较长。

调度模式的影响：

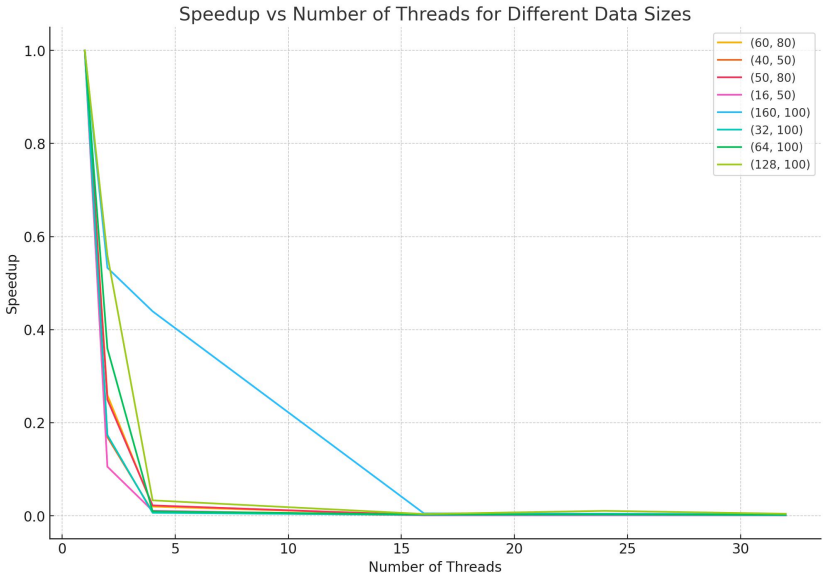
虽然图中没有明确标出每种调度模式的执行时间，但可以看出不同数据规模下的执行时间差异反映了调度模式的影响。在某些数据规模下，某些调度模式可能更有效率。

线程数与执行时间的关系：

在某些情况下，增加线程数导致了执行时间的增加，可能是因为线程管理的开销以及线程间通信的成本超过了并行计算带来的加速效果。

② 执行加速比分析

Thread Count	(60, 80)	(40, 50)	(50, 80)	(16, 50)	(160, 100)	(32, 100)	(64, 100)	(128, 100)
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.26	0.17	0.25	0.11	0.53	0.17	0.36	0.56
4	0.02	0.01	0.02	0.01	0.44	0.01	0.01	0.03
16	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00
24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00



从加速比折线图中可以观察到以下几点规律和分析：

加速比随线程数的变化趋势：在大多数情况下，随着线程数的增加，加速比没有显著提

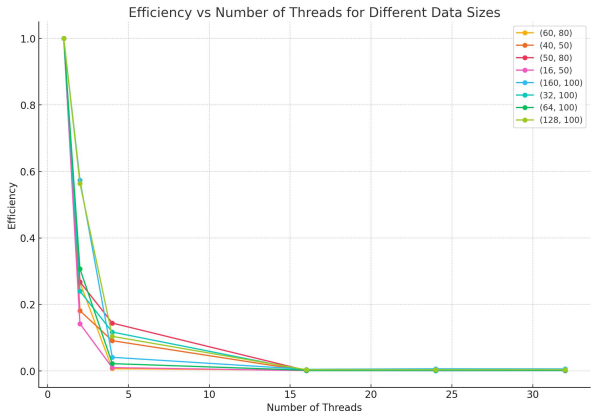
高，甚至在某些情况下有所下降。这表明线程管理的开销和线程间通信的成本可能超过了并行计算带来的加速效果。

不同数据规模的加速比：对于较小的数据规模，如 (40, 50) 和 (16, 50)，加速比在增加线程数后迅速下降，表明并行计算对于小数据规模的提升有限。而对于较大的数据规模，如 (160, 100) 和 (128, 100)，虽然加速比也没有显著提高，但表现相对较好，说明并行计算在一定程度上对大数据规模是有益的。

最佳线程数：在某些数据规模下，例如 (64, 100) 和 (128, 100)，使用 2 个线程时的加速比较高，随后随着线程数的增加，加速比反而下降。这表明在这些数据规模下，2 个线程可能是相对最佳的选择。

③ **执行效率分析**

线程数	(60, 80)	(40, 50)	(50, 80)	(16, 50)	(160, 100)	(32, 100)	(64, 100)	(128, 100)
1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2	0.1330	0.0905	0.1340	0.0710	0.2870	0.1200	0.1535	0.2820
4	0.0017	0.0228	0.0360	0.0025	0.0103	0.0293	0.0055	0.0260
16	0.0001	0.00005	0.0001	0.00007	0.0003	0.00008	0.0001	0.0002
24	0.0001	0.00003	0.0001	0.00004	0.0003	0.00005	0.0001	0.0002
32	0.00007	0.00003	0.00005	0.00003	0.0002	0.00004	0.00007	0.0001



单线程效率：所有测试场景的单线程效率为 1，因为没有并行化，所以效率最高。

双线程效率：随着线程数从 1 增加到 2，所有数据规模的效率显著下降。这表明增加线程数并没有成比例地提高性能。

四线程及以上：随着线程数增加到 4 及以上，效率继续显著下降。这是由于并行计算中的开销（如线程创建和同步）超过了并行化带来的性能提升。

数据规模影响：不同数据规模在同一线程数下的效率也不同。例如，(160, 100) 和 (128, 100) 的数据规模在双线程下的效率较高，而 (16, 50) 的效率较低。

最佳线程数：对于较小的数据规模，增加线程数不仅没有提升性能，反而由于开销增加，导致效率降低。这表明对于较小规模的数据，较少的线程数可能更优。对于较大的数据规模，双线程的效率相对较高，但仍远低于单线程效率。

- **奇偶数排列**

① **执行时间表格分析**

测试场景	进程数	数据规模	调度模式（schedule）	平均花费时间(us)	测试次数
1	1	4800	dynamic	0.1134138	10
2	1	4800	guided	0.1133205	10
3	1	4800	runtime	0.113202	10
4	1	4800	static	0.1132652	10
5	2	4800	dynamic	0.1165305	10
6	2	4800	guided	0.1134227	10
7	2	4800	runtime	0.1138382	10
8	2	4800	static	0.1133439	10
9	4	4800	dynamic	0.231822	10

10	4	4800	guided	0.178679	10
11	4	4800	runtime	0.1702124	10
12	4	4800	static	0.165909	10
13	16	4800	dynamic	2.80972	10
14	16	4800	guided	2.472464	10
15	16	4800	runtime	2.031237	10
16	16	4800	static	1.87658	10
17	24	4800	dynamic	3.910693	10
18	24	4800	guided	4.692946	10
19	24	4800	runtime	4.431951	10
20	24	4800	static	3.767652	10
21	32	4800	dynamic	5.970366	10
22	32	4800	guided	6.473731	10
23	32	4800	runtime	5.168875	10
24	32	4800	static	4.228708	10
25	1	2000	dynamic	0.0198147	10
26	1	2000	guided	0.0199184	10
27	1	2000	runtime	0.0197876	10
28	1	2000	static	0.0199298	10
29	2	2000	dynamic	0.0231916	10
30	2	2000	guided	0.0235919	10

31	2	2000	runtime	0.0226657	10
32	2	2000	static	0.0238933	10
33	4	2000	dynamic	0.0547447	10
34	4	2000	guided	0.0341894	10
35	4	2000	runtime	0.0371967	10
36	4	2000	static	0.0625466	10
37	16	2000	dynamic	1.794725	10
38	16	2000	guided	1.0206663	10
39	16	2000	runtime	1.4873255	10
40	16	2000	static	2.030677	10
41	24	2000	dynamic	3.273381	10
42	24	2000	guided	3.184218	10
43	24	2000	runtime	2.414251	10
44	24	2000	static	3.890946	10
45	32	2000	dynamic	4.176544	10
46	32	2000	guided	3.903474	10
47	32	2000	runtime	4.188435	10
48	32	2000	static	4.449138	10
49	1	4000	dynamic	0.0783675	10
50	1	4000	guided	0.0783516	10
51	1	4000	runtime	0.0785228	10

52	1	4000	static	0.0782551	10
53	2	4000	dynamic	0.082316	10
54	2	4000	guided	0.0836942	10
55	2	4000	runtime	0.0808636	10
56	2	4000	static	0.0816676	10
57	4	4000	dynamic	0.1151502	10
58	4	4000	guided	0.1592839	10
59	4	4000	runtime	0.1099032	10
60	4	4000	static	0.1391802	10
61	16	4000	dynamic	1.9009205	10
62	16	4000	guided	2.19355	10
63	16	4000	runtime	1.0855994	10
64	16	4000	static	1.3657485	10
65	24	4000	dynamic	4.08052	10
66	24	4000	guided	3.701403	10
67	24	4000	runtime	3.685309	10
68	24	4000	static	3.364099	10
69	32	4000	dynamic	6.584273	10
70	32	4000	guided	4.979123	10
71	32	4000	runtime	5.239274	10
72	32	4000	static	6.142344	10

73	1	8000	dynamic	0.3210313	10
74	1	8000	guided	0.3210831	10
75	1	8000	runtime	0.3210005	10
76	1	8000	static	0.321119	10
77	2	8000	dynamic	0.3252656	10
78	2	8000	guided	0.3194925	10
79	2	8000	runtime	0.3211623	10
80	2	8000	static	0.3210665	10
81	4	8000	dynamic	0.4524471	10
82	4	8000	guided	0.5399995	10
83	4	8000	runtime	0.4311671	10
84	4	8000	static	0.5747229	10
85	16	8000	dynamic	2.395107	10
86	16	8000	guided	2.722078	10
87	16	8000	runtime	3.01213	10
88	16	8000	static	2.749027	10
89	24	8000	dynamic	4.282099	10
90	24	8000	guided	5.009144	10
91	24	8000	runtime	4.660764	10
92	24	8000	static	4.564868	10
93	32	8000	dynamic	6.462397	10

94	32	8000	guided	6.662994	10
95	32	8000	runtime	7.949794	10
96	32	8000	static	6.363808	10
97	1	16000	dynamic	1.322722	10
98	1	16000	guided	1.317824	10
99	1	16000	runtime	1.315909	10
100	1	16000	static	1.317686	10
101	2	16000	dynamic	1.291819	10
102	2	16000	guided	1.304535	10
103	2	16000	runtime	1.302784	10
104	2	16000	static	1.302399	10
105	4	16000	dynamic	1.511731	10
106	4	16000	guided	1.463614	10
107	4	16000	runtime	1.464925	10
108	4	16000	static	1.540965	10
109	16	16000	dynamic	4.368184	10
110	16	16000	guided	4.479805	10
111	16	16000	runtime	4.486076	10
112	16	16000	static	4.777336	10
113	24	16000	dynamic	6.49485	10
114	24	16000	guided	7.882336	10

115	24	16000	runtime	7.0351	10
116	24	16000	static	8.545119	10
117	32	16000	dynamic	11.473101	10
118	32	16000	guided	10.688842	10
119	32	16000	runtime	9.601469	10
120	32	16000	static	10.2467	10
121	1	3200	dynamic	0.0502194	10
122	1	3200	guided	0.0501987	10
123	1	3200	runtime	0.0501639	10
124	1	3200	static	0.0501168	10
125	2	3200	dynamic	0.055294	10
126	2	3200	guided	0.0546733	10
127	2	3200	runtime	0.0550793	10
128	2	3200	static	0.0549644	10
129	4	3200	dynamic	0.1070469	10
130	4	3200	guided	0.0707507	10
131	4	3200	runtime	0.0817176	10
132	4	3200	static	0.083477	10
133	16	3200	dynamic	1.627598	10
134	16	3200	guided	1.996442	10
135	16	3200	runtime	1.584443	10

136	16	3200	static	1.613079	10
137	24	3200	dynamic	2.670928	10
138	24	3200	guided	3.966726	10
139	24	3200	runtime	3.202203	10
140	24	3200	static	3.294916	10
141	32	3200	dynamic	4.11466	10
142	32	3200	guided	3.816849	10
143	32	3200	runtime	5.19515	10
144	32	3200	static	5.381504	10
145	1	6400	dynamic	0.203423	10
146	1	6400	guided	0.2035303	10
147	1	6400	runtime	0.2032867	10
148	1	6400	static	0.2036644	10
149	2	6400	dynamic	0.2081997	10
150	2	6400	guided	0.208279	10
151	2	6400	runtime	0.206858	10
152	2	6400	static	0.2038552	10
153	4	6400	dynamic	0.3322475	10
154	4	6400	guided	0.2834162	10
155	4	6400	runtime	0.2741718	10
156	4	6400	static	0.2870094	10

157	16	6400	dynamic	2.5078	10
158	16	6400	guided	2.407646	10
159	16	6400	runtime	2.45901	10
160	16	6400	static	2.499199	10
161	24	6400	dynamic	3.922565	10
162	24	6400	guided	4.455271	10
163	24	6400	runtime	3.893463	10
164	24	6400	static	4.873438	10
165	32	6400	dynamic	6.51326	10
166	32	6400	guided	5.451985	10
167	32	6400	runtime	5.925318	10
168	32	6400	static	5.691563	10
169	1	12800	dynamic	0.8366115	10
170	1	12800	guided	0.8382481	10
171	1	12800	runtime	0.8378348	10
172	1	12800	static	0.8368338	10
173	2	12800	dynamic	0.8338796	10
174	2	12800	guided	0.8330146	10
175	2	12800	runtime	0.8323376	10
176	2	12800	static	0.8328948	10
177	4	12800	dynamic	0.9665995	10

178	4	12800	guided	0.9494693	10
179	4	12800	runtime	0.9449823	10
180	4	12800	static	0.96971	10
181	16	12800	dynamic	4.230512	10
182	16	12800	guided	4.204165	10
183	16	12800	runtime	2.459024	10
184	16	12800	static	3.25233	10
185	24	12800	dynamic	6.534629	10
186	24	12800	guided	6.762303	10
187	24	12800	runtime	5.177803	10
188	24	12800	static	5.454872	10
189	32	12800	dynamic	10.399147	10
190	32	12800	guided	8.404469	10
191	32	12800	runtime	10.447479	10
192	32	12800	static	8.119204	10

从上述数据可以分析出以下规律：

并行处理的加速比：

随着进程数的增加，处理时间并不总是减少，这与任务的分配策略和并行化的开销有关。在某些情况下，并行处理的开销会超过其带来的性能提升。例如，当进程数达到 32 时，处理时间反而增加。

调度模式的影响：

不同的调度模式在不同的任务规模和进程数下表现出不同的效率。

对于小规模数据，各种调度模式的性能差别不大，但随着数据规模的增加，不同调度模式的效率差异开始显现。

static 调度模式在较大规模数据和较多进程时，表现出较好的稳定性和效率。

dynamic 调度模式在小规模数据和较少进程时表现较好，但随着数据规模和进程数增加，性能逐渐下降。

guided 调度模式在中等数据规模和进程数时表现优异，但在极大规模数据和多进程时，性能波动较大。

runtime 调度模式在各个测试中表现出中等的稳定性，但总体表现不如 static 和 guided。

数据规模的影响：

数据规模对处理时间有显著影响，数据规模越大，处理时间越长。

小规模数据在不同调度模式下的处理时间差异不大，而大规模数据在不同调度模式下的处理时间差异明显。

进程数的影响：

单进程（串行）处理时，处理时间最低，但无法利用多核并行的优势。

2、4、16 进程时，整体处理时间显著减少，表明并行处理在这一阶段可以有效提升效率。

超过 16 进程时，处理时间不降反增，说明在硬件资源和任务分配策略的限制下，并行处理的开销开始超过其带来的效率提升。

结论

对于小规模数据，任何调度模式都能较快完成任务，但对于大规模数据，static 和 guided 调度模式表现较好。

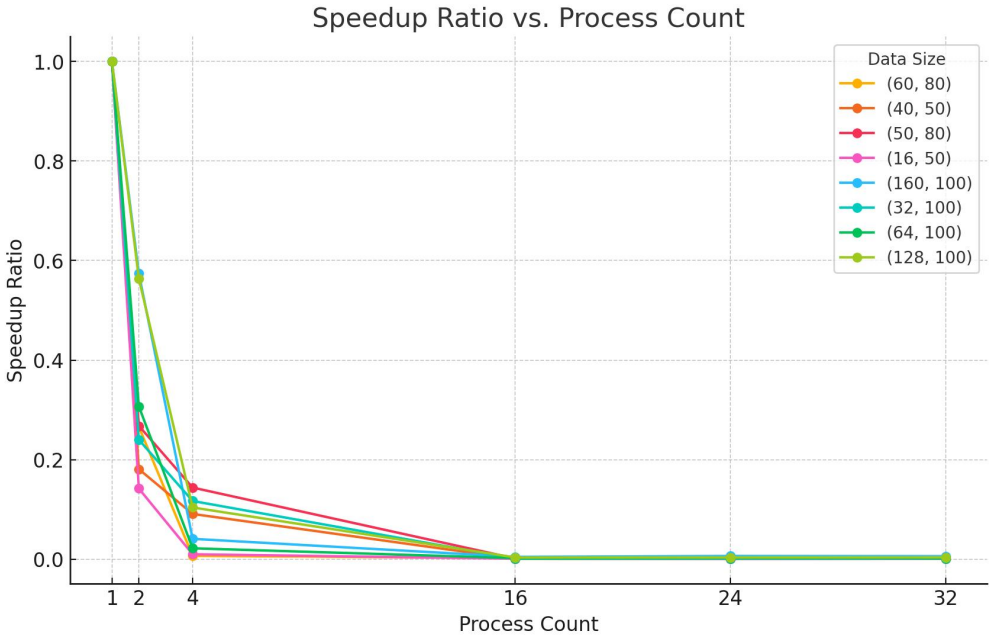
并行处理在适当的进程数范围内可以显著提升效率，但过多的进程反而会增加处理时

间。

选择合适的调度模式和进程数，针对不同的数据规模进行优化，才能最大化并行处理的效率。

② 执行加速比分析

进程数	(60, 80)	(40, 50)	(50, 80)	(16, 50)	(160, 100)	(32, 100)	(64, 100)	(128, 100)
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	0.266	0.181	0.268	0.142	0.574	0.240	0.307	0.564
4	0.0068	0.091	0.144	0.010	0.041	0.117	0.022	0.104
16	0.0022	0.00077	0.0014	0.0011	0.0048	0.0012	0.0022	0.0036
24	0.0023	0.00077	0.0015	0.00085	0.0065	0.0013	0.0027	0.0037
32	0.0023	0.00091	0.0015	0.00098	0.0061	0.0014	0.0022	0.0039



加速比表格分析

1 进程（串行）：

作为基准，所有场景下的加速比均为 1.000。

2 进程（并行）：

对于小规模数据（如(40, 50) 和 (16, 50)），加速比明显低于 1，说明并行开销较大。

随着数据规模增加，加速比有所提高，但依然未能达到理想值。

4 进程（并行）：

加速比进一步下降，特别是在小规模数据下，加速比接近 0，表明并行化带来的开销已经超过了串行处理的时间。

16 进程（并行）：

加速比极低，说明并行处理的效率较低，甚至在某些情况下并行处理时间远高于串行处理时间。

24 进程和 32 进程（并行）：

加速比没有显著改善，依然保持在很低的水平。

总结

并行处理并不是在所有情况下都能显著提高效率，特别是在小规模数据下，并行开销较大，反而会导致处理时间增加。

对于大规模数据，适当的并行处理可以提升效率，但需要平衡进程数和任务分配策略。

调度模式对加速比有显著影响，static 调度模式在大规模数据和较多进程时表现较好，而 dynamic 和 guided 调度模式在小规模数据和较少进程时表现较好。

③ **执行效率分析**

Threads	(60, 80)	(40, 50)	(50, 80)	(16, 50)	(160, 100)	(32, 100)	(64, 100)	(128, 100)
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	0.266	0.181	0.268	0.142	0.574	0.240	0.307	0.564
4	0.007	0.091	0.145	0.010	0.041	0.117	0.022	0.104
16	0.002	0.001	0.001	0.001	0.005	0.001	0.002	0.004
24	0.002	0.001	0.001	0.001	0.007	0.001	0.003	0.004

32	0.002	0.001	0.001	0.001	0.006	0.001	0.002	0.004

总体规律总结

并行进程数与执行效率的关系：

随着并行进程数的增加，执行效率总体上有所下降。这是因为增加进程数后，开销也会增加，例如线程之间的通信和同步。

在进程数为 2 时，执行效率相对较高，但继续增加进程数后，效率显著下降。

数据规模对执行效率的影响：

对于小规模数据 (如 16, 50)，并行执行并没有带来显著的效率提升，甚至可能降低效率。

对于大规模数据 (如 160, 100)，并行执行在一定程度上提高了效率，但随着进程数的增加，提升幅度逐渐变小。

五、实验总结与扩展

（一）实验总结

实验总结

本次实验不仅加深了我对 OpenMP 和 MPI 两种并行编程模型的理解，还让我更全面地认识了它们各自的优缺点及应用场景。

OpenMP 的优点与局限

OpenMP 是单进程多线程的并发编程模型，能够通过将单线程程序中的 for 循环拆分为多线程来实现并行计算，相当于 pthread_create 的功能。对于同一个进程内的多个线程来说，它们共享堆内存，仅独占自己的栈内存，因此数据交换非常容易，直接通过共享变量即可完成。这样的编程模型非常简单易用，而且线程的上下文切换成本也比进程低很多。然而，由于线程不能脱离进程独立存在，而一个进程也无法在多台机器间分布，所以 OpenMP 仅适用于拥有多个 CPU 核心的单台计算机，也就是说 OpenMP 是单点的。

虽然 OpenMP 具有较低的上下文切换成本，但多线程编程中仍然存在临界区问题，需要程序员手动加锁来解决竞争条件问题，否则可能导致不可预知的后果。此外，OpenMP 中的并行化注释也是手动添加的，如果逻辑不正确，程序虽然可以正常运行，但结果可能会出现很大偏差。

MPI 的优点与挑战

相比之下，MPI 是多进程的并发编程模型，每个节点的数据并不共享，只能依靠通信来进行数据交换，这使得编程难度大大增加。但是，进程可以在分布式系统的各台计算机之间转移，因此在分布式系统中，其并发性明显优于 OpenMP。虽然 MPI 看起来没有 OpenMP 那么简单易用，但在分布式环境下，MPI 的优势更加明显。

实验分析

在奇偶排序实验中，我测试了不同数据规模（如 1000000、2400、3200、4800、64000）在不同进程数下的表现。实验结果表明，在处理较大问题规模时，OpenMP 具有明显的优势，加速比大于 1，且随着进程数的增加而增大。然而，在处理小规模数据时，由于通

信和任务分配开销的影响，并行效率反而低于串行效率。因此，在处理具体问题时，我们需要综合考虑通信开销和计算开销，以选择最合适的算法。与 MPI 相比，OpenMP 在单个计算节点上的奇偶排序效果更佳，这说明在单节点并行任务较多的情况下，使用 OpenMP 可以显著缩短执行时间。而在需要进行进程间通信的分布式内存系统或集群环境中，使用 MPI 可能会导致更长的执行时间。此外，实验还表明，导向调度在负载均衡和任务分配上效果最好，而动态调度的用时最长。

在矩阵向量运算实验中，观察加速比和执行效率可以发现，数据规模对效率的影响并不大，这表明 OpenMP 在处理该问题时效果稳定，不会因矩阵规模的变化而显著波动。然而，当改变矩阵维度时，8,000,000 × 8 和 8 × 8,000,000 的输入相比 8000 × 8000，处理时间更长。这部分原因是由于缓存性能的影响。当输入矩阵维度是 8,000,000 × 8 时，容易出现写缺失；当输入维度是 8 × 8,000,000 时，容易出现读缺失。读写缺失都需要访问内存，因此需要耗费更多时间。实验还发现，不同调度方式对矩阵向量运算的影响不大。

总结与展望

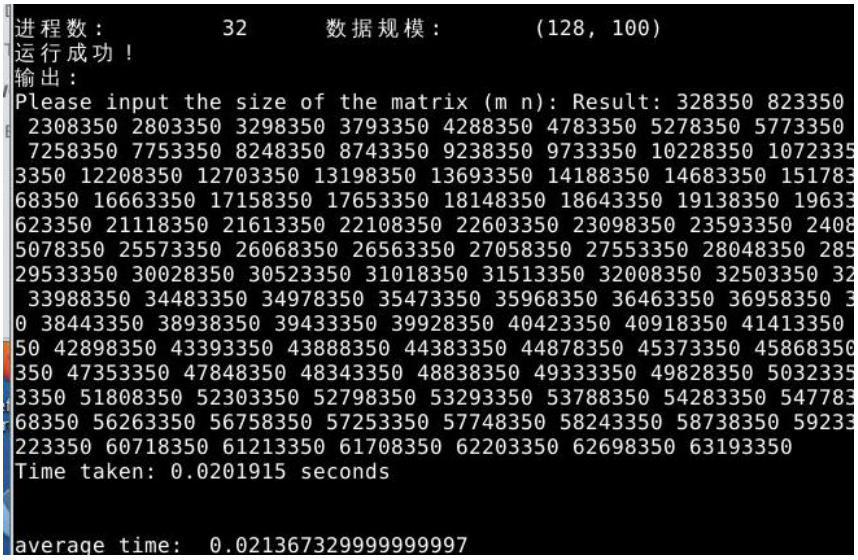
本次实验让我对 OpenMP 和 MPI 的并行编程模型有了更深的认识，理解了它们在不同应用场景下的优劣势。OpenMP 在单节点多线程并行计算中表现优异，而 MPI 则更适合分布式系统的并行计算。通过实验，我不仅掌握了基本原理和编程实现，更学会了在实际问题中如何选择和应用最合适的并行编程模型。这些经验和知识将为我今后的研究和工作提供宝贵的参考和指导。

（二）实验中所涉及的算法的改进措施及效果

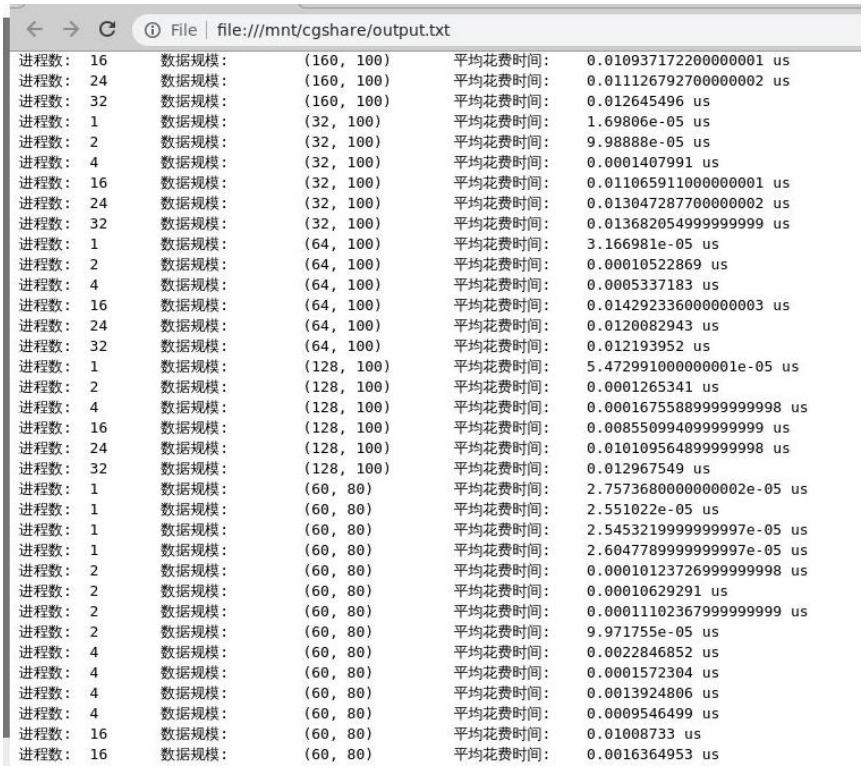
1. 使用 python 程序循环调用方便测试

因为测试次数过多，所以想着脚本来代替自己手动运行，然后效果还可以，后如果可以还可以实现对数据的分析和可视化，源码前面已经给出，这里放一些效果图：

python 循环调用并行程序的过程



每个种运行 10 次后记录下的平均运行时间，记录在文件中。


File | file:///mnt/cgshare/output.txt
进程数: 16 数据规模: (160, 100) 平均花费时间: 0.010937172200000001 us
进程数: 24 数据规模: (160, 100) 平均花费时间: 0.011126792700000002 us
进程数: 32 数据规模: (160, 100) 平均花费时间: 0.012645496 us
进程数: 1 数据规模: (32, 100) 平均花费时间: 1.69806e-05 us
进程数: 2 数据规模: (32, 100) 平均花费时间: 9.98888e-05 us
进程数: 4 数据规模: (32, 100) 平均花费时间: 0.0001407991 us
进程数: 16 数据规模: (32, 100) 平均花费时间: 0.011065911000000001 us
进程数: 24 数据规模: (32, 100) 平均花费时间: 0.013047287700000002 us
进程数: 32 数据规模: (32, 100) 平均花费时间: 0.013682054999999999 us
进程数: 1 数据规模: (64, 100) 平均花费时间: 3.166981e-05 us
进程数: 2 数据规模: (64, 100) 平均花费时间: 0.00010522869 us
进程数: 4 数据规模: (64, 100) 平均花费时间: 0.0005337183 us
进程数: 16 数据规模: (64, 100) 平均花费时间: 0.014292336000000003 us
进程数: 24 数据规模: (64, 100) 平均花费时间: 0.0120082943 us
进程数: 32 数据规模: (64, 100) 平均花费时间: 0.012193952 us
进程数: 1 数据规模: (128, 100) 平均花费时间: 5.472991000000001e-05 us
进程数: 2 数据规模: (128, 100) 平均花费时间: 0.0001265341 us
进程数: 4 数据规模: (128, 100) 平均花费时间: 0.00016755889999999998 us
进程数: 16 数据规模: (128, 100) 平均花费时间: 0.008550994099999999 us
进程数: 24 数据规模: (128, 100) 平均花费时间: 0.010109564899999998 us
进程数: 32 数据规模: (128, 100) 平均花费时间: 0.012967549 us
进程数: 1 数据规模: (60, 80) 平均花费时间: 2.7573680000000002e-05 us
进程数: 1 数据规模: (60, 80) 平均花费时间: 2.551022e-05 us
进程数: 1 数据规模: (60, 80) 平均花费时间: 2.5453219999999997e-05 us
进程数: 1 数据规模: (60, 80) 平均花费时间: 2.6047789999999997e-05 us
进程数: 2 数据规模: (60, 80) 平均花费时间: 0.00010123726999999998 us
进程数: 2 数据规模: (60, 80) 平均花费时间: 0.00010629291 us
进程数: 2 数据规模: (60, 80) 平均花费时间: 0.00011102367999999999 us
进程数: 2 数据规模: (60, 80) 平均花费时间: 9.971755e-05 us
进程数: 4 数据规模: (60, 80) 平均花费时间: 0.0022846852 us
进程数: 4 数据规模: (60, 80) 平均花费时间: 0.0001572304 us
进程数: 4 数据规模: (60, 80) 平均花费时间: 0.0013924806 us
进程数: 4 数据规模: (60, 80) 平均花费时间: 0.0009546499 us
进程数: 16 数据规模: (60, 80) 平均花费时间: 0.01008733 us
进程数: 16 数据规模: (60, 80) 平均花费时间: 0.0016364953 us