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 3 月

实 验 报 告 册

实验名称	实验一　MPI 并行程序设计	实 验 成 绩	
实验日期	2024 年 3 月 20 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 MPI 程序的设计思想、设计原则和设计方法。

2、掌握 MPI 并行程序的编程方法。

3、掌握并行程序的评估方法

二、实验环境（请描述本实验教学活动所使用的实际环境。）

1、电脑主机一台

2、Linux 系统

3、MPI 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。）

采用 MPI 编程方式实现以下三个题目，串并行加速比越高越好。

(1）梯形求积分

(2) 矩阵向量乘法

(3）奇偶排序

四、实验内容

要求：采用 MPI 编程方式实现以下三个题目，串并行加速比越高越好

(一) 实验题目

(1) 梯形求积分

(2) 矩阵向量乘法

(3) 奇偶排序

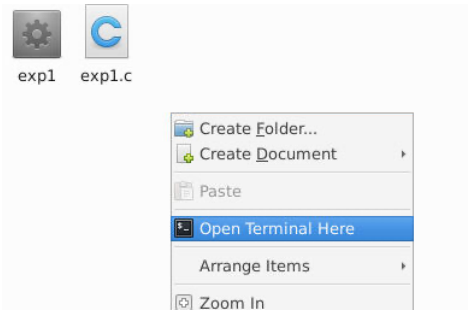
(二) 实验过程

(1) 集群使用 （以一题为示例即可）

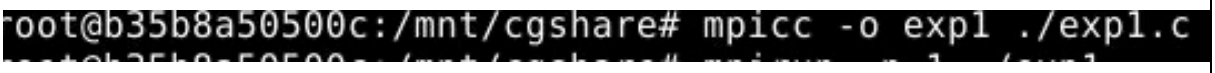
① 打开 MPI 实验环境

② 将本地写好的 c 代码上传至 itc （本来想使用本机的 MPI 环境编译好在上传的但是发现服务器运行不了本机编译的，可能因为编译器版本不同吧）

③ 在/mnt/cgshare 目录下打开终端



④ 使用命令行 mpicc -o 程序名 XXX.c 来编译源代码（或者 mpicxx 编译 c++），生成可执行程序



⑤ 使用命令行 mpirun -n 4 ./程序名 来运行程序。

```
root@b35b8a50500c:/mnt/cgshare# mpirun -n 1 ./exp1
input: 1 5 10
With n=10 trapezoids, our estimate of the area is 38.320000 from 1.000000 t
00000
time = 1.454353e-05 second
root@b35b8a50500c:/mnt/cgshare#
```

⑥ 使用测试程序批量运行 获得结果

```
root@b35b8a50500c:/mnt/cgshare# python3 ./train.py
进程数：          32      数据规模：        12800
运行成功！
输出：
input: With n=12800 trapezoids, our estimate of the area is 332506.107712 from 0
.000000 to 100.000000
time = 1.094341e-04 second


进程数：          32      数据规模：        12800
运行成功！
输出：
input: With n=12800 trapezoids, our estimate of the area is 332506.107712 from 0
.000000 to 100.000000
time = 8.773804e-05 second


进程数：          32      数据规模：        12800
运行成功！
输出：
input: With n=12800 trapezoids, our estimate of the area is 332506.107712 from 0
.000000 to 100.000000
time = 8.845329e-05 second
```

（2）源码及解析（每题一个源码，说明核心代码即可）

1. 梯形求积分

原理：将$[a,b]$区间划分为 n 个等长部分，求 n 个梯形的面积求和就是最后积分面积的近似值。

令函数为 $f(x) = x^2$.

[x1, x2] 之间的梯形面积是：$\frac{|x_1-x_2|*(f(x_1)+f(x_2))}{2}$

每个区间的长度为：$h = \frac{(b-a)}{2}$

所以推倒后的求和公式为：$S = h[\frac{f(x_1)}{2} + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2}]$

思路：我们可以将**整个区间问题规约成多个小区间**，并且每个区间的计算是互不干扰的**所以就可以进行并行计算**，最后使用 MPI 的通信机制计算出总的结果。

核心计算代码：

```
/**
 * Calculates the square of a given number.
 *
 * This function takes a double value as input and returns the square of that value.
 *
 * @param x The input value.
 * @return The square of the input value.
 */
double f(double x)
{
    return x * x; // 函数f(x) = x^2
}
/**
 * Calculates the value within a given range.
 *
 * This function calculates the value within the range defined by `local_a` and `local_b`,
 * using the specified step size `h` and the number of intervals `local_n`.
 *
 * @param local_a The lower bound of the range.
 * @param local_b The upper bound of the range.
 * @param local_n The number of intervals.
 * @param h The step size.
```

```
 * @return The calculated value within the range.
 */
double calculate_in_range(double local_a, double local_b, int local_n,
double h)
{
    // 当前部分长度是 local_n，当前部分的两侧端点是 [local_a, local_b]
    // 惊醒梯形求积分的计算
    double local_area, x;
    local_area = (f(local_a) + f(local_b)) / 2 * h;
    for (int i = 1; i < local_n; i++)
    {
        x = local_a + i * h;
        local_area = local_area + f(x);
    }
    local_area = local_area * h; // 按照推导公式进行面积计算
    return local_area;          // 返回这一部分的面积
}
```

说明：这段代码使用用来解决子问题的，当一整个区间划分为小区间后调用此函数计算

初始化进程需要的参数：

```
/**
 * Initializes the range of values for each process.
 *
 * @param my_rank The rank of the current process.
 * @param comm_sz The total number of processes.
 * @param a_p     Pointer to the variable 'a'.
 * @param b_p     Pointer to the variable 'b'.
 * @param n_p     Pointer to the variable 'n'.
 */
void init_range(
    int my_rank, // 进程号
    int comm_sz, // 进程总数
```

```
    double *a_p, // 指向 a 的指针
    double *b_p, // 指向 b 的指针
    int *n_p)    // 指向 n 的指针
{
    int i;

    if (my_rank == 0) // 0 号进程读取数据
    {
        printf("input: ");
        scanf("%lf%lf%d", a_p, b_p, n_p); // 读取数据
    }

    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD); // 将读取到的数据发送给其他进程
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD); // 将读取到的数据发送给其他进程
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);    // 将读取到的数据发送给其他进程
    // 只有 0 号进程会发送数据，其他进程会在这里接收数据
}
```

主函数：

```
int main(int argc, char *argv[])
{
    int comm_sz, my_rank, source; // 定义进程总数、当前进程的排名和消息来源
    double start, end, t, t_sum;  // 定义开始时间、结束时间、单个进程时间和总时间
    // 初始化 MPI 环境
    MPI_Init(&argc, &argv);
    // 获取进程总数
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    // 获取当前进程的排名
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
    int n;                                  // 定义梯形的总数
    double a, b;                            // 定义积分的起始点和终点
    init_range(my_rank, comm_sz, &a, &b, &n); // 初始化每个进程的积分范围

    // 同步所有进程，确保所有进程都完成初始化后再继续
    MPI_Barrier(MPI_COMM_WORLD);

    start = MPI_Wtime(); // 开始计时

    double h = (b - a) / n;                 // 计算每个梯形的高度
    int local_n;                            // 定义每个进程处理的梯形数量
    double local_a, local_b, local_area, total_area; // 定义每个进程的积分范围和面积

    local_n = n / comm_sz;                  // 计算每个进程处理的梯形数量
    local_a = a + my_rank * local_n * h;    // 计算每个进程的积分起始点
    local_b = local_a + local_n * h;        // 计算每个进程的积分终点
    local_area = calculate_in_range(local_a, local_b, local_n, h); // 计算每个进程的积分面积

    // 将所有进程的积分面积累加，结果存储在 0 号进程的 total_area 变量中
    MPI_Reduce(&local_area, &total_area, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    end = MPI_Wtime(); // 结束计时

    t = end - start; // 计算本进程的执行时间

    // 找出所有进程中的最大执行时间，并将结果存储在 0 号进程的 t_sum 变量中
```

```
    MPI_Reduce(&t, &t_sum, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    if (my_rank == 0) // 仅0号进程执行
    {
        // 输出总面积
        printf("With n=%d trapezoids, our estimate of the area is %lf from %lf
to %lf\n", n, total_area, a, b);
        // 输出最大执行时间
        printf("time = %e second\n", t_sum);
    }

    MPI_Finalize(); // 结束MPI环境
    return 0;
}
```

流程图：

图 1

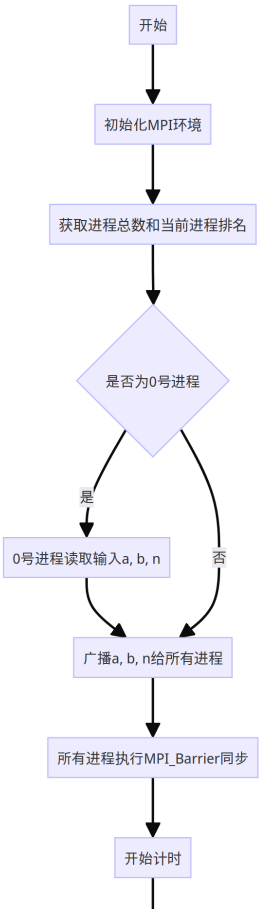
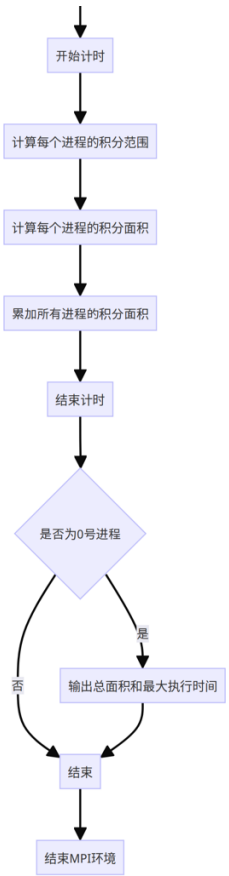


图 2



关键点：

- **并行程序的 IO 处理之输入输出**

虽然 MPI 标准没有指定那些进程可以访问哪些 I/O 设备，但是几乎所有的 MPI 实现都允许 MPI_COMM_WORLD 里的所有进程都能访问标准输出(stdout)、标准输入(stdin)和标准错误输出(stderr)，所以，大部分的 MPI 实现都允许所有进程执行 printf、scanf、fprintf 和 fscanf，

但是，大部分的 MPI 实现并不提供对这些 I/O 设备访问的自动调度，也就是说，如果多个进程试图写标准输出 stdout，那么这些进程的输出顺序是无法预测的，甚至会发生一个进程的输出被另一个进出的输出打断，输入也是如此。

所以我采用了只让 0 号进程作为一个入口和出口，只有他可以输入输出，在程序中体现就是每次输入输出的时候都要判断 rank 是不是 0。

- **并行进程之间的通信**

广播时使用了 MPI_Bcast 函数而不是循环调用 MPI_Send 函数有以下几点：

1) **数据共享：**`MPI_Bcast`用于从一个进程（通常是根进程，这里是 0 号进程）广播数据到所有其他进程。在这个程序中，0 号进程读取输入数据（`a`, `b`, `n`），这些数据对所有进程都是必需的，以便它们可以独立地计算自己负责的积分范围的面积。使用`MPI_Bcast`可以确保所有进程都接收到一致的输入数据，这是并行计算中的一个常见需求。
2) **效率：**与使用多个`MPI_Send`调用从一个进程向所有其他进程发送相同数据相比，`MPI_Bcast`更高效。`MPI_Bcast`是专门为广播操作优化的，能够利用底层硬件和网络拓扑的特性，如多播或广播能力，从而减少通信开销和提高整体性能。
3) **简化代码**：使用`MPI_Bcast`可以减少编程复杂性。如果使用`MPI_Send`和`MPI_Recv`来实现广播，每个进程都需要写额外的代码来处理发送或接收操作。而`MPI_Bcast`

只需要一行代码，所有进程都调用同一个函数，无论是发送方还是接收方，这使得代码更简洁、易于理解和维护。

4) **集体通信的一致性：**在 MPI 中，集体通信操作要求所有参与的进程都调用该操作。这有助于保持程序的一致性和同步。`MPI_Bcast`作为集体通信操作，自然地保证了所有进程在广播操作完成前都在等待，这有助于同步进程状态。

- **并行程序计时问题**

在开始计时之前使用了 **MPI_Barrier(MPI_COMM_WORLD);**同步所有进程，确保所有进程都完成初始化后再继续，这样计时更准确。

- **并行程序规约求合问题**

使用 MPI_Reduce 规约求合（求时间和、求面积和）；

1) **性能优化**：MPI_Reduce 利用了底层通信网络的特性，如树形归约等算法，以优化归约操作的性能，特别是在大规模并行计算中。

2) 易用性：MPI_Reduce 简化了并行程序中的归约操作实现。开发者只需指定归约操作的类型和根进程，无需编写复杂的数据传输和同步代码。

3) 通用性：MPI_Reduce 可以用于各种数据类型的归约操作，并且支持自定义的归约操作，这使得它在多种并行计算场景中都非常有用。

2. 矩阵向量乘法

代码解析

init_data 函数:

- 功能: 获取输入的矩阵维度，并将维度广播到其他进程。
- 参数:

- ✧ `my_rank`：进程号。
- ✧ `comm_sz`：进程总数。
- ✧ `n_p`：指向矩阵维度的指针。

➢ 实现细节：如果是 0 号进程，则从标准输入读取矩阵的维度 N，并使用`MPI_Bcast`函数将 N 广播给所有进程。

main 函数：

➢ 功能：主函数，执行矩阵向量乘法的并行计算。

➢ 实现细节：

- ✧ 初始化 MPI 环境。
- ✧ 调用`init_data`函数获取矩阵维度 N，并广播给所有进程。
- ✧ 根据进程数计算每个进程处理的行数。
- ✧ 为每个进程分配存储矩阵、向量和结果向量的空间。
- ✧ 0 号进程构建矩阵和向量，并使用`MPI_Scatter`分发矩阵的一部分给每个进程。
- ✧ 每个进程计算自己负责的矩阵部分与向量的乘积，并将结果存储在结果向量中。
- ✧ 使用`MPI_Gather`将所有进程的结果向量聚集到 0 号进程。
- ✧ 0 号进程计算总的运行时间并输出。
- ✧ 释放分配的内存资源。
- ✧ 结束 MPI 环境。

代码详细解析：

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
```

```c
// time_t start,end;//
double start, end;
void init_data(  // 获取输入的矩阵维度，并将维度广播到其他进程
    int my_rank, // 进程号
    int comm_sz, // 进程总数
    int *n_p)
{
    int i, j;
    int n;
    if (my_rank == 0)
    {
        printf("Enter N:\n");
        scanf("%d", n_p);
    }
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
int main()
{
    int N;

    int *vec = NULL;        // 列向量
    double *mat = NULL;     // 自己进程的那部分矩阵
    int my_rank;           // 自己进程的进程号
    int comm_sz;           // 总的进程数目
    int my_row;            // 本进程处理的行数
    int i, j;              // 通用游标
    double *result = NULL;  // 用来存本进程计算出的结果向量
    double *all_rst = NULL; // 只给 0 号进程存总的结果
```

```
// 初始化
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
init_data(my_rank, comm_sz, &N);
if (my_rank == 0)
{
    // start=time(NULL);
    start = MPI_Wtime();
}
my_row = N / comm_sz; // 本进程处理的行数就是总阶数/进程数
// 为每个进程都申请空间
mat = malloc(N * my_row * sizeof(double)); // my_row 行的小矩阵
vec = malloc(N * sizeof(int));            // 每个进程各自读入列向量
result = malloc(my_row * sizeof(double));  // 每个进程各自的结果向量
double *a = NULL;
if (my_rank == 0)
{
    // 开辟存储空间
    all_rst = malloc(N * sizeof(double));
    a = malloc(N * N * sizeof(double));
    // 构建矩阵
    for (j = 0; j < N; j++)
    {
        for (i = 0; i < N; i++)
            a[i * N + j] = j;
        vec[j] = 1;
    }
```

```
        // 分发给其他进程
        MPI_Scatter(a, N * my_row, MPI_DOUBLE, mat, N * my_row, MPI_DOUBLE,
0, MPI_COMM_WORLD); // 块划分分发矩阵
        // 计算
        for (i = 0; i < my_row; i++)
        {
            result[i] = mat[i * N] * vec[0];
            for (j = 1; j < N; j++)
                result[i] += mat[i * N + j] * vec[j];
        }
        // 聚集给 0 号进程
        MPI_Gather(
            result,        /*发送内容的地址*/
            my_row,        /*发送的长度*/
            MPI_DOUBLE,    /*发送的数据类型*/
            all_rst,       /*接收内容的地址*/
            my_row,        /*接收的长度*/
            MPI_DOUBLE,    /*接收的数据类型*/
            0,            /*接收至哪个进程*/
            MPI_COMM_WORLD /*通信域*/
        );
    }
    else
    {
        // 接收矩阵
        MPI_Scatter(a, N * my_row, MPI_DOUBLE, mat, N * my_row, MPI_DOUBLE,
0, MPI_COMM_WORLD);
        // 计算
        for (i = 0; i < my_row; i++)
```

```
    {
        result[i] = mat[i * N] * vec[0];
        for (j = 1; j < N; j++)
            result[i] += mat[i * N + j] * vec[j];
    }
    // 聚集给0号进程
    MPI_Gather(
        result,
        my_row,
        MPI_DOUBLE,
        all_rst,
        my_row,
        MPI_DOUBLE,
        0,
        MPI_COMM_WORLD);
}
// 0号进程负责输出
if (my_rank == 0)
{
    end = MPI_Wtime();
    printf("time=%e\n", end - start);
}

MPI_Finalize();
/**********************/
// 最终,free应无遗漏
free(all_rst);
free(mat);
free(vec);
```

```
    free(result);

    return 0;
}
```

流程图

图 1

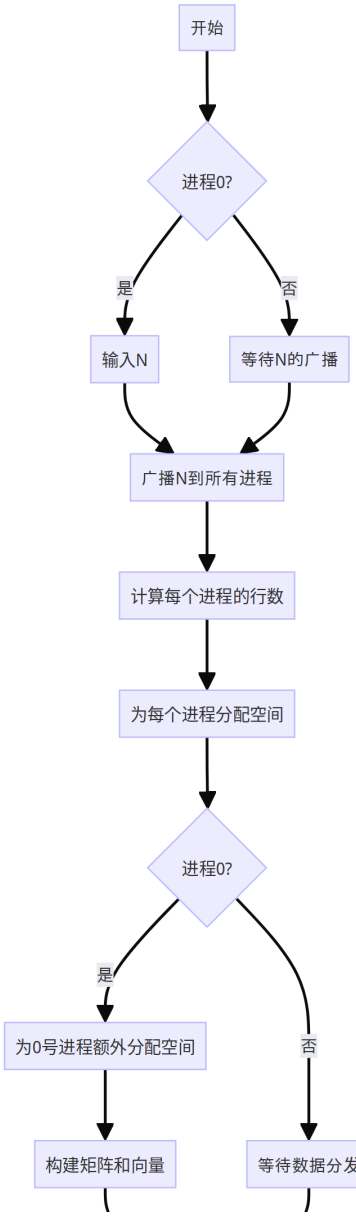
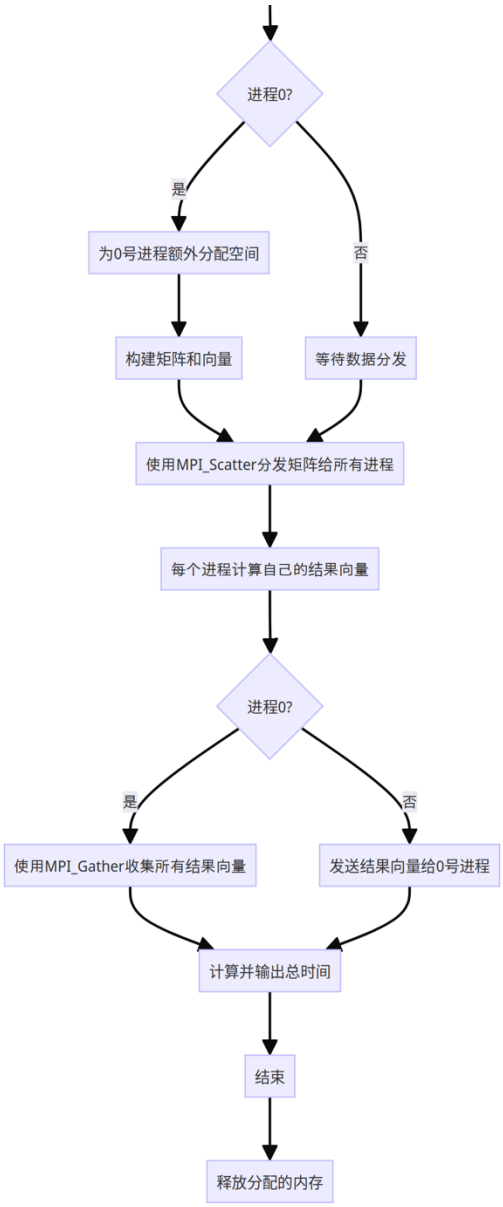


图 2



3. 奇偶数排列

原理：

传统的排序算法因为串行度很高难以实现并行化，所以需要一种新的方法可以使得排序算法并行。

奇偶交换排序：

- 奇数阶段：按 (a[0], a[1]), (a[2], a[3]), a[4], ... 分组比较大小，并交换位置
- 偶数阶段：按 a[0], (a[1], a[2]), （a[3], a[4]), ... 分组比较大小，并交换位置

定理：对于 n 个元素的序列，作为奇偶交换排序的输入，那么至多经过 n 个阶段后，该序列一定能排好序

每个阶段的比较和交换操作是可以同时进行的，所以奇偶交换排序适合并行。

所以如果将上述的 a[0], a[1] 看成不同进程所拥有的那一部分数据，那么此时需要做的不是简单地交换两个数据，而是将小的数据放在 a[0] 部分，大的数据放在 a[1] 部分。如果两部分的数据就是有序的，那么可以更高效更简单的完成大小数据的分配。

所以可以在进程内部使用快速排序使得数据有序，再在相邻进程之间使用归并排序完成大小数据分配。

代码解析

1) init_data 函数

- 功能：初始化数据。如果是主进程（0 号进程），则从标准输入读取一个整数 N，并将这个值通过 MPI 广播到所有进程。
- 参数：
    - my_rank：当前进程的编号。

- comm_sz：进程总数。
- n_p：指向整数的指针，用于存储输入的 N 值。

- 实现逻辑：如果当前进程是 0 号进程，提示用户输入 N 的值并读取。然后，使用 MPI_Bcast 函数将 N 的值广播给所有进程。

2) get_partner 函数

- 功能：根据当前的通信阶段和进程编号，计算出当前进程的通信伙伴。
- 参数：
  - my_rank：当前进程的编号。
  - phase：当前的通信阶段。
- 实现逻辑：根据奇偶排序算法的要求，进程在不同的阶段与不同的伙伴进行通信。偶数阶段时，偶数编号的进程与前一个进程通信，奇数编号的进程与后一个进程通信；奇数阶段时，情况相反。

3) random_input 函数

- 功能：在 0 号进程中生成 N 个随机数，并通过 MPI 分发到所有进程中。
- 参数：
  - A：整型数组，用于存储每个进程接收到的随机数。
  - local_n：每个进程应接收的随机数数量。
  - my_rank：当前进程的编号。
- 实现逻辑：0 号进程生成 N 个随机数并存储在数组中，然后使用 MPI_Scatter 函数将这些随机数平均分配给所有进程。

4) merge_to_get_low 和 merge_to_get_high 函数

- 功能：这两个函数用于合并两个进程的数据，并分别取出合并后的较小一半或较大

一半数据。

- 参数：
    - A 和 B：两个整型数组，分别存储当前进程的数据和通信伙伴的数据。
    - local_n：每个进程的数据量。
- 实现逻辑：通过比较 A 和 B 中的元素，选择较小（或较大）的元素存储到临时数组中，然后将临时数组的内容复制回 A 数组。

5) output_vector 函数

- 功能：输出排序后的数组。0 号进程收集所有进程的数据，并输出。
- 参数：
    - A：整型数组，存储当前进程的部分数据。
    - local_n：每个进程的数据量。
    - my_rank：当前进程的编号。
- 实现逻辑：使用 MPI_Gather 函数将所有进程的数据收集到 0 号进程，然后由 0 号进程输出所有数据。

6) main 函数

- 功能：程序的入口点。执行 MPI 环境的初始化，数据的初始化和分发，奇偶排序算法，计时，以及最终的数据输出。
- 实现逻辑：
    - 初始化 MPI 环境，获取进程总数和当前进程编号。
    - 初始化数据，计算每个进程的数据量，分配内存。
    - 产生并分发随机数据，对每个进程的数据进行本地排序。
    - 执行奇偶排序算法的主循环，包括数据的交换和合并。

- 计时并在 0 号进程中输出总时间。
- 输出排序后的数组。
- 结束 MPI 环境。

代码详细解析

```
#include <iostream>  // 引入标准输入输出流库
#include <algorithm> // 引入算法库，用于排序等
#include <random>    // 引入随机数生成库
#include <time.h>    // 引入时间库，用于随机数种子生成
#include <mpi.h>     // 引入MPI 库，用于并行计算

// 使用命名空间std，简化代码
using namespace std;

int N; // 定义全局变量N，表示待排序的整型数量

// 初始化数据函数，用于获取输入的矩阵维度，并将维度广播到其他进程
void init_data(
    int my_rank, // 当前进程编号
    int comm_sz, // 进程总数
    int *n_p)    // 指向整型的指针，用于存储输入的N 值
{
    if (my_rank == 0) // 如果是0 号进程（主进程）
    {
        printf("Enter N:\n"); // 提示输入N
        scanf("%d", n_p);     // 读取输入的N
    }
    // 广播N 的值到所有进程
```

```
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}

// 获取通信伙伴的函数
int get_partner(int my_rank, int phase) // 当前进程编号和通信阶段
{
    // 偶数阶段
    if (phase % 2 == 0)
    {
        if (my_rank % 2 == 0)
        {
            return my_rank - 1; // 偶数进程与前一个进程通信
        }
        else
        {
            return my_rank + 1; // 奇数进程与后一个进程通信
        }
    }
    // 奇数阶段
    else
    {
        if (my_rank % 2 == 0)
        {
            return my_rank + 1; // 偶数进程与后一个进程通信
        }
        else
        {
            return my_rank - 1; // 奇数进程与前一个进程通信
        }
```

```
    }
}

// 产生随机数并分发至各个进程的函数
void random_input(int A[], int local_n, int my_rank) // 数组 A，每个进程的数据量，当前进程编号
{
    int *a = NULL;    // 主进程用于存储所有随机数的数组
    if (my_rank == 0) // 如果是 0 号进程
    {
        a = new int[N];      // 动态分配 N 个整型的空间
        srand((int)time(0)); // 设置随机数种子
        for (int i = 0; i < N; i++)
        {
            a[i] = rand() % 1000; // 生成随机数并存储
        }
        // 分发随机数到各个进程
        MPI_Scatter(a, local_n, MPI_INT, A, local_n, MPI_INT, 0, MPI_COMM_WORLD);
        delete[] a; // 释放动态分配的内存
    }
    else // 其他进程
    {
        // 接收随机数
        MPI_Scatter(a, local_n, MPI_INT, A, local_n, MPI_INT, 0, MPI_COMM_WORLD);
    }
}
```

```
// 合并两个进程的数据，并取较小的一半数据
void merge_to_get_low(int A[], int B[], int local_n) // 两个数组和每个进程的数据量
{
    int *a = new int[local_n];   // 临时数组，用于存储合并后的较小数据
    int p_a = 0, p_b = 0, i = 0; // A、B 和a 的索引

    while (i < local_n) // 合并数据直到填满临时数组
    {
        if (A[p_a] < B[p_b])
        {
            a[i++] = A[p_a++];
        }
        else
        {
            a[i++] = B[p_b++];
        }
    }
    // 将合并后的数据复制回A 数组
    for (i = 0; i < local_n; i++)
    {
        A[i] = a[i];
    }
    delete[] a; // 释放临时数组的内存
}

// 合并两个进程的数据，并取较大的一半数据
void merge_to_get_high(int A[], int B[], int local_n) // 两个数组和每个进程的数据量
```

```
{
    int p_a = local_n - 1, p_b = local_n - 1, i = local_n - 1; // 从数组末尾开始合并
    int *a = new int[local_n];                                  // 临时数组，用于存储合并后的较大数据

    while (i >= 0) // 从后向前合并数据
    {
        if (A[p_a] > B[p_b])
        {
            a[i--] = A[p_a--];
        }
        else
        {
            a[i--] = B[p_b--];
        }
    }
    // 将合并后的数据复制回A 数组
    for (i = 0; i < local_n; i++)
    {
        A[i] = a[i];
    }
    delete[] a; // 释放临时数组的内存
}

// 输出排序后的数组
void output_vector(int A[], int local_n, int my_rank) // 数组A，每个进程的数据量，当前进程编号
{
```

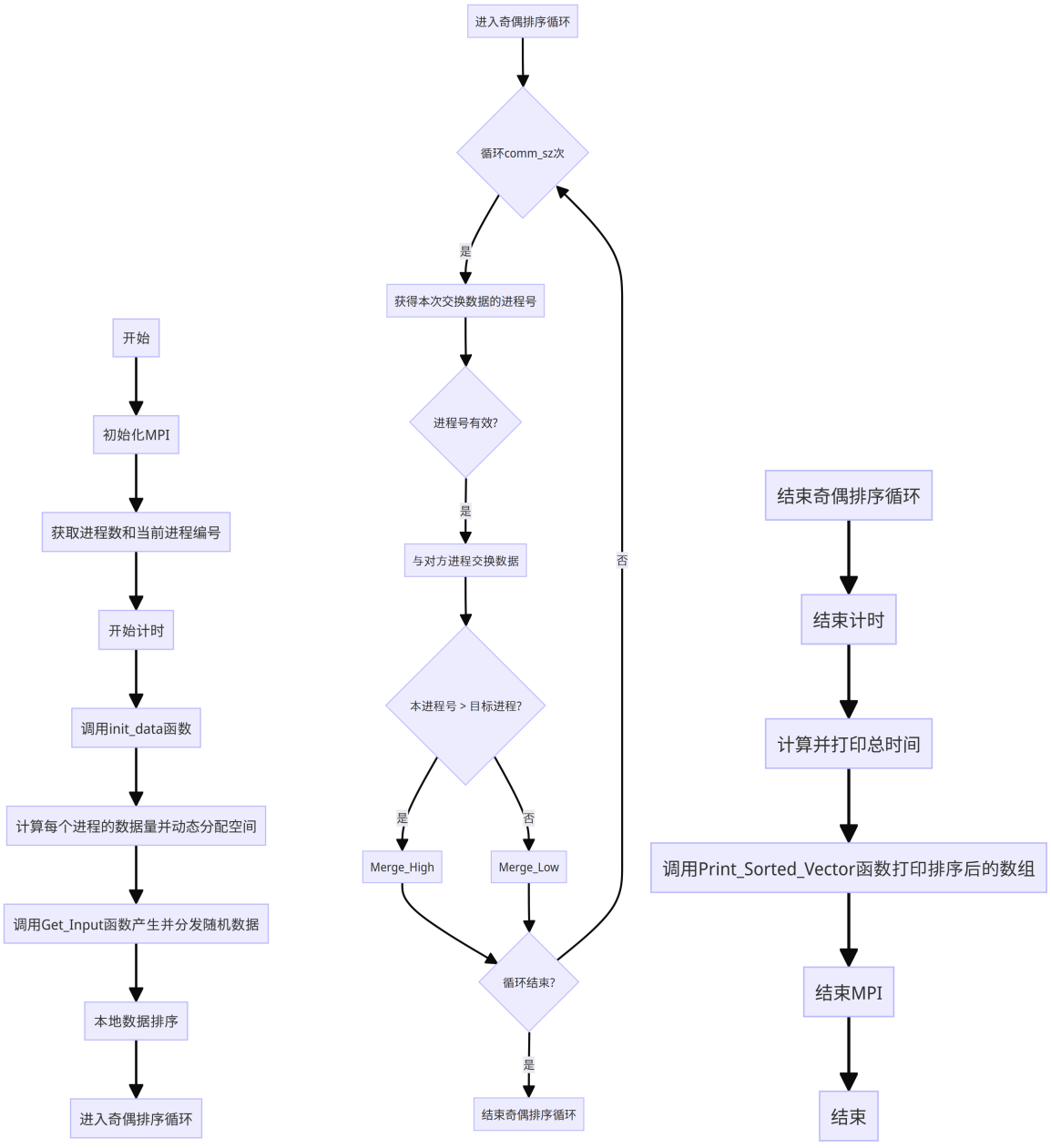
```
    int *a = NULL;    // 0 号进程用于接收所有进程的数据
    if (my_rank == 0) // 如果是 0 号进程
    {
        a = new int[N]; // 动态分配空间以接收所有数据
        // 收集各个进程的数据
        MPI_Gather(A,  local_n,  MPI_INT,  a,  local_n,  MPI_INT,  0,
MPI_COMM_WORLD);
        // 输出数据
        for (int i = 0; i < N; i++)
        {
            cout << a[i] << "\t";
            if (i % 4 == 3) // 每 4 个数据换一行
            {
                cout << endl;
            }
        }
        cout << endl;
        delete[] a; // 释放动态分配的内存
    }
    else // 其他进程
    {
        // 将数据发送给 0 号进程
        MPI_Gather(A,  local_n,  MPI_INT,  a,  local_n,  MPI_INT,  0,
MPI_COMM_WORLD);
    }
}

int main()
{
```

```
int local_n;                // 各个进程中数组的大小
int *A, *B;                 // A 为进程中保存的数据，B 为进程通信中获得的数据
int comm_sz, my_rank;       // 进程总数和当前进程编号
double start, end, t, t_sum; // 计时相关变量

MPI_Init(NULL, NULL);                   // 初始化MPI 环境
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); // 获取进程总数
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // 获取当前进程编号

start = MPI_Wtime(); // 记录开始时间

init_data(my_rank, comm_sz, &N); // 初始化数据

local_n = N / comm_sz; // 计算每个进程的数据量
A = new int[local_n];  // 动态分配空间给A
B = new int[local_n];  // 动态分配空间给B

random_input(A, local_n, my_rank); // 产生并分发随机数据

sort(A, A + local_n); // 对本地数据进行排序

// 奇偶排序主循环
for (int i = 0; i < comm_sz; i++)
{
    int partner = get_partner(my_rank, i);   // 获取通信伙伴
    if (partner != -1 && partner != comm_sz) // 如果通信伙伴有效
    {
        // 交换数据
        MPI_Sendrecv(A, local_n, MPI_INT, partner, 0, B, local_n,
```

```
MPI_INT, partner, 0, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
            if (my_rank > partner) // 根据进程编号决定合并方式
            {
                merge_to_get_high(A, B, local_n);
            }
            else
            {
                merge_to_get_low(A, B, local_n);
            }
        }
    }

    end = MPI_Wtime();                                          // 记录结束时间
    t = end - start;                                            // 计算总时间
    MPI_Reduce(&t, &t_sum, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD); // 计算所有进程中的最大时间
    if (my_rank == 0)                                           // 如果是0号进程，则输出总时间
    {
        printf("time=%e second\n", t_sum);
    }

    output_vector(A, local_n, my_rank); // 输出排序后的数组

    MPI_Finalize(); // 结束MPI环境
    return 0;       // 程序结束
}
```

流程图

初始化和数据准备　　奇偶排序循环　　结束和结果输出



(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

测试程序代码：

```
#coding=UTF-8
import subprocess

def run_exp1_with_input(num_processes, executable_path, a, b, n,
output_file):
    """
    使用 mpirun 运行 exp1 程序，并提供输入 a, b, n。将输出保存到文件中。
    参数:
    num_processes: 要使用的进程数。
    executable_path: exp1 可执行文件的路径。
    a, b: 积分的起始点和终点。
    n: 梯形的总数。
    output_file: 输出文件的路径。
    返回:
    无
    """
    # 构建 mpirun 命令
    command = ["mpirun", "-np", str(num_processes), executable_path]

    # 将输入转换为字符串，以便通过 stdin 传递
    input_str = str(a) + " " + str(b) + " " + str(n) + "\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)

    # 打开输出文件
    with open(output_file, "a") as file:
        # 检查命令是否成功执行
        if result.returncode == 0:
```

```
            file.write("运行成功！\n")
            file.write("输出:\n" + result.stdout + "\n")
            print("运行成功！")
            print("输出:\n" + result.stdout + "\n")
        else:
            file.write("运行失败！\n")
            file.write("错误信息:\n" + result.stderr + "\n")
            print("运行失败！")
            print("错误信息:\n" + result.stderr + "\n")

# 示例：使用5个进程运行位于"./exp1"的exp1程序，输入为a=0, b=10, n=1000，输出保存到"output.txt"
run_exp1_with_input(5, "./exp1", 0, 10, 1000, "output.txt")

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
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 i in range(times):
            str_print = "进程数: \t" + str(num) + "\t 数据规模: \t" + str(size)
            print(str_print)
            with open("output.txt", "a") as file:
                file.write(str_print + "\n")
                    t += float(run_exp1_with_input(num, "./exp3", size, "output.txt"))
        t /= times
        print("average time: ", t)
```

这里是用的矩形求积分写的，后面两个修改下输入数据就可以同样适用。

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。

- 矩形求积分

运行过程截图：

```
进程数：         32      数据规模：       6400
运行成功！
输出：
input: With n=6400 trapezoids, our estimate of the area is 331691.911316 from 0.
000000 to 100.000000
time = 6.747246e-05 second


进程数：         32      数据规模：       6400
运行成功！
输出：
input: With n=6400 trapezoids, our estimate of the area is 331691.911316 from 0.
000000 to 100.000000
time = 8.177757e-05 second


进程数：         32      数据规模：       6400
运行成功！
输出：
```

输出的结果文件截图：

output-3.txt — 已锁定

```
进程数:  1       数据规模:       4800    平均花费时间:    4.6491624e-05
进程数:  2       数据规模:       4800    平均花费时间:    4.3940544e-05
进程数:  4       数据规模:       4800    平均花费时间:    3.3068658e-05
进程数:  16      数据规模:       4800    平均花费时间:    4.4846535e-05
进程数:  24      数据规模:       4800    平均花费时间:    6.2036515e-05
进程数:  32      数据规模:       4800    平均花费时间:    8.4352488e-05
进程数:  1       数据规模:       2000    平均花费时间:    2.2029877e-05
进程数:  2       数据规模:       2000    平均花费时间:    2.8729437999999994e-05
进程数:  4       数据规模:       2000    平均花费时间:    2.6130675999999995e-05
进程数:  16      数据规模:       2000    平均花费时间:    3.8957597e-05
进程数:  24      数据规模:       2000    平均花费时间:    7.0953367e-05
进程数:  32      数据规模:       2000    平均花费时间:    0.00014095304900000002
进程数:  1       数据规模:       4000    平均花费时间:    3.120899299999999e-05
进程数:  2       数据规模:       4000    平均花费时间:    3.2806398e-05
进程数:  4       数据规模:       4000    平均花费时间:    3.2424928000000005e-05
进程数:  16      数据规模:       4000    平均花费时间:    4.2843818e-05
进程数:  24      数据规模:       4000    平均花费时间:    0.00053513051500000001
进程数:  32      数据规模:       4000    平均花费时间:    8.234978e-05
进程数:  1       数据规模:       8000    平均花费时间:    6.389617899999999e-05
进程数:  2       数据规模:       8000    平均花费时间:    5.915165399999999e-05
进程数:  4       数据规模:       8000    平均花费时间:    4.796982e-05
进程数:  16      数据规模:       8000    平均花费时间:    4.5537949e-05
进程数:  24      数据规模:       8000    平均花费时间:    6.3681598e-05
进程数:  32      数据规模:       8000    平均花费时间:    0.000391674072
进程数:  1       数据规模:       16000   平均花费时间:    0.000136899951
进程数:  2       数据规模:       16000   平均花费时间:    9.2077248e-05
进程数:  4       数据规模:       16000   平均花费时间:    6.0749054000000005e-05
进程数:  16      数据规模:       16000   平均花费时间:    7.140636499999999e-05
进程数:  24      数据规模:       16000   平均花费时间:    8.7404249e-05
进程数:  32      数据规模:       16000   平均花费时间:    9.310246000000001e-05
进程数:  1       数据规模:       3200    平均花费时间:    3.5238266e-05
进程数:  2       数据规模:       3200    平均花费时间:    3.1661987e-05
进程数:  4       数据规模:       3200    平均花费时间:    3.378391299999999e-05
进程数:  16      数据规模:       3200    平均花费时间:    3.9839744e-05
进程数:  24      数据规模:       3200    平均花费时间:    9.436606999999999e-05
进程数:  32      数据规模:       3200    平均花费时间:    8.563995700000001e-05
进程数:  1       数据规模:       6400    平均花费时间:    5.106926e-05
进程数:  2       数据规模:       6400    平均花费时间:    4.229545600000001e-05
进程数:  4       数据规模:       6400    平均花费时间:    3.3974646999999993e-05
进程数:  16      数据规模:       6400    平均花费时间:    4.6181680000000004e-05
进程数:  24      数据规模:       6400    平均花费时间:    0.00010933875599999999
进程数:  32      数据规模:       6400    平均花费时间:    8.218288100000001e-05
进程数:  1       数据规模:       12800   平均花费时间:    0.00011010170699999999
进程数:  2       数据规模:       12800   平均花费时间:    8.075237300000002e-05
进程数:  4       数据规模:       12800   平均花费时间:    4.9686431000000004e-05
进程数:  16      数据规模:       12800   平均花费时间:    4.427433100000001e-05
进程数:  24      数据规模:       12800   平均花费时间:    0.00016925338899999998
进程数:  32      数据规模:       12800   平均花费时间:    8.7976451e-05
```

- 矩阵向量乘法

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

```
进程数：          1        数据规模：        4800
运行成功！
输出：
Enter N:
time=4.766810e-01


进程数：          1        数据规模：        4800
运行成功！
输出：
Enter N:
time=4.721608e-01


进程数：          1        数据规模：        4800
运行成功！
输出：
Enter N:
time=4.740689e-01


进程数：          1        数据规模：        4800
```

output-6.txt — 已锁定

```
进程数:  1    数据规模:  4800    平均花费时间:  3.4701122418132995
进程数:  2    数据规模:  4800    平均花费时间:  0.6675416246599999
进程数:  4    数据规模:  4800    平均花费时间:  0.360917371998
进程数:  16   数据规模:  4800    平均花费时间:  0.210086510866
进程数:  24   数据规模:  4800    平均花费时间:  0.058781907519
进程数:  32   数据规模:  4800    平均花费时间:  0.13113555614
进程数:  1    数据规模:  2000    平均花费时间:  2.4572018404556
进程数:  2    数据规模:  2000    平均花费时间:  0.684567543357
进程数:  4    数据规模:  2000    平均花费时间:  0.317668719069
进程数:  16   数据规模:  2000    平均花费时间:  0.161178150612
进程数:  24   数据规模:  2000    平均花费时间:  0.201452465795
进程数:  32   数据规模:  2000    平均花费时间:  0.106168910951
进程数:  1    数据规模:  4000    平均花费时间:  2.1098682815900007
进程数:  2    数据规模:  4000    平均花费时间:  1.1300072917750001
进程数:  4    数据规模:  4000    平均花费时间:  0.346156463543
进程数:  16   数据规模:  4000    平均花费时间:  0.485156657595
进程数:  24   数据规模:  4000    平均花费时间:  0.17605629283399998
进程数:  32   数据规模:  4000    平均花费时间:  0.18050668007099996
进程数:  1    数据规模:  8000    平均花费时间:  3.5287879910789
进程数:  2    数据规模:  8000    平均花费时间:  1.26080055946
进程数:  4    数据规模:  8000    平均花费时间:  0.439819391292
进程数:  16   数据规模:  8000    平均花费时间:  0.348139829189
进程数:  24   数据规模:  8000    平均花费时间:  0.044300717311
进程数:  32   数据规模:  8000    平均花费时间:  0.131940559659
进程数:  1    数据规模:  16000   平均花费时间:  1.2865658514220002
进程数:  2    数据规模:  16000   平均花费时间:  0.8729864840829998
进程数:  4    数据规模:  16000   平均花费时间:  0.8088656535800001
进程数:  16   数据规模:  16000   平均花费时间:  0.264348656147
进程数:  24   数据规模:  16000   平均花费时间:  0.156592503619
进程数:  32   数据规模:  16000   平均花费时间:  0.16327647714099996
进程数:  1    数据规模:  3200    平均花费时间:  1.5343763698413002
进程数:  2    数据规模:  3200    平均花费时间:  1.15920294985
进程数:  4    数据规模:  3200    平均花费时间:  0.05487959632200001
进程数:  16   数据规模:  3200    平均花费时间:  0.22164431202600002
进程数:  24   数据规模:  3200    平均花费时间:  0.26046181745700003
进程数:  32   数据规模:  3200    平均花费时间:  0.068370042951
进程数:  1    数据规模:  6400    平均花费时间:  1.3952683844739
进程数:  2    数据规模:  6400    平均花费时间:  1.335070819618
进程数:  4    数据规模:  6400    平均花费时间:  0.6094202065210002
进程数:  16   数据规模:  6400    平均花费时间:  0.11448619290800002
进程数:  24   数据规模:  6400    平均花费时间:  0.216848815755
进程数:  32   数据规模:  6400    平均花费时间:  0.024519654308
进程数:  1    数据规模:  12800   平均花费时间:  1.1396892763998
进程数:  2    数据规模:  12800   平均花费时间:  0.8542889177090001
进程数:  4    数据规模:  12800   平均花费时间:  0.693913913385
进程数:  16   数据规模:  12800   平均花费时间:  0.187876308063
进程数:  24   数据规模:  12800   平均花费时间:  0.129176405701
进程数:  32   数据规模:  12800   平均花费时间:  0.059905314221000004
```

- 奇偶数排列

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

```
进程数：          24       数据规模：        3200
运行成功！
输出：
Enter N:
time=1.417875e-03 second


average time:  0.00156817434
进程数：          32       数据规模：        3200
运行成功！
输出：
Enter N:
time=1.421452e-03 second


进程数：          32       数据规模：        3200
运行成功！
输出：
Enter N:
time=2.031326e-03 second


进程数：          32       数据规模：        3200
```

output-7.txt

```
进程数:  1    数据规模:    4800    平均花费时间:    0.0014629126
进程数:  2    数据规模:    4800    平均花费时间:    0.0014559268200000002
进程数:  4    数据规模:    4800    平均花费时间:    0.00144686679
进程数:  16   数据规模:    4800    平均花费时间:    0.0019241810999999997
进程数:  24   数据规模:    4800    平均花费时间:    0.00136067859
进程数:  32   数据规模:    4800    平均花费时间:    0.0035244943000000001
进程数:  1    数据规模:    2000    平均花费时间:    0.00049290657
进程数:  2    数据规模:    2000    平均花费时间:    0.00081696513
进程数:  4    数据规模:    2000    平均花费时间:    0.00094280231
进程数:  16   数据规模:    2000    平均花费时间:    0.0017347098
进程数:  24   数据规模:    2000    平均花费时间:    0.0010746001899999999
进程数:  32   数据规模:    2000    平均花费时间:    0.0018754960900000002
进程数:  1    数据规模:    4000    平均花费时间:    0.00109534267
进程数:  2    数据规模:    4000    平均花费时间:    0.00097844605
进程数:  4    数据规模:    4000    平均花费时间:    0.0014976263100000002
进程数:  16   数据规模:    4000    平均花费时间:    0.0018266914999999998
进程数:  24   数据规模:    4000    平均花费时间:    0.0014141321100000002
进程数:  32   数据规模:    4000    平均花费时间:    0.0024937630000000004
进程数:  1    数据规模:    8000    平均花费时间:    0.0021927358
进程数:  2    数据规模:    8000    平均花费时间:    0.0019005536
进程数:  4    数据规模:    8000    平均花费时间:    0.0018111707000000002
进程数:  16   数据规模:    8000    平均花费时间:    0.0021265984
进程数:  24   数据规模:    8000    平均花费时间:    0.0014441252
进程数:  32   数据规模:    8000    平均花费时间:    0.0030427934
进程数:  1    数据规模:    16000   平均花费时间:    0.0035502433
进程数:  2    数据规模:    16000   平均花费时间:    0.0028333188000000003
进程数:  4    数据规模:    16000   平均花费时间:    0.0025443314999999997
进程数:  16   数据规模:    16000   平均花费时间:    0.0024220465000000004
进程数:  24   数据规模:    16000   平均花费时间:    0.0019288537999999997
进程数:  32   数据规模:    16000   平均花费时间:    0.0047257189
进程数:  1    数据规模:    3200    平均花费时间:    0.0009315967499999997
进程数:  2    数据规模:    3200    平均花费时间:    0.0010608672900000002
进程数:  4    数据规模:    3200    平均花费时间:    0.00111937518
进程数:  16   数据规模:    3200    平均花费时间:    0.0019523621
进程数:  24   数据规模:    3200    平均花费时间:    0.0013876916
进程数:  32   数据规模:    3200    平均花费时间:    0.0032654284
进程数:  1    数据规模:    6400    平均花费时间:    0.0016624449999999998
进程数:  2    数据规模:    6400    平均花费时间:    0.0015382052399999996
进程数:  4    数据规模:    6400    平均花费时间:    0.0013594149999999999
进程数:  16   数据规模:    6400    平均花费时间:    0.0017834425
进程数:  24   数据规模:    6400    平均花费时间:    0.00128819934
进程数:  32   数据规模:    6400    平均花费时间:    0.0021108152
进程数:  1    数据规模:    12800   平均花费时间:    0.0028160810000000002
进程数:  2    数据规模:    12800   平均花费时间:    0.002466464
进程数:  4    数据规模:    12800   平均花费时间:    0.0025394679000000004
进程数:  16   数据规模:    12800   平均花费时间:    0.0019774437000000004
进程数:  24   数据规模:    12800   平均花费时间:    0.00161161407
进程数:  32   数据规模:    12800   平均花费时间:    0.002726984
```

（2）执行时间分析（表格）

- 矩形求积分

① 执行时间表格分析

	0，100，4800	0，100，6400	0,100，12800	0，100，16000
1（串行）	0.0000469	0.0000510	0.0001101	0.0001369
2（并行）	0.0000439	0.0000422	0.0000920	0.0000920
4（并行）	0.0000330	0.0000339	0.0000807	0.0000607
16（并行）	0.0000448	0.0000461	0.0000442	0.0000714
24（并行）	0.0000620	0.0001093	0.0001692	0.0000874
32（并行）	0.0000843	0.0000821	0.0000879	0.0000931

可见在增加进程数时，并行程序并不一定使得运行时间降低，一般是近似符合一个一元二次函数，先降低后增加。可能的原因有在并行执行中，涉及到进程间通信的开销以及任务划分和合并的开销，这些开销超过了并行计算的好处。

在数据量增大时不一定使得时间增大，因为也有可能让计算效率更高了，从而时间变短，但是一般情况下数据量大，运行时间就长。

② 执行加速比分析

	0，100，4800	0，100，6400	0,100，12800	0，100，16000
1（串行）	1.0	1.0	1.0	1.0
2（并行）	1.0683371	1.2085308	1.1967391	2.2553542
4（并行）	1.4212121	1.5044248	1.3643123	1.9173669
16（并行）	1.046875	1.1062907	2.4909502	1.5663616
24（并行）	0.7564516	0.4666057	0.6507092	1.4704619
32（并行）	0.5563464	0.6211937	1.2525597	1.4880435

分析这组执行加速比的数据，我们可以观察到以下几个规律和可能的原因:

1. 加速比与进程数的关系 ：

- 在一些情况下，随着进程数的增加，加速比并没有线性增长，甚至出现了下降。这可能是因为并行计算中的开销（如进程间通信、数据同步等）随着进程数的增加而增加。当这些开销超过了并行带来的性能提升时，加速比就会下降。

- 特别地，对于较大的数据量（例如 16000），加速比在 2 和 4 进程时显著提高，这表

明并行计算在处理大量数据时能够有效利用多核资源。但是，当进程数增加到一定程度后，加速比的提升幅度减小，甚至出现下降，这可能是由于并行计算的开销和硬件资源的限制。

2. 数据量对加速比的影响：

- 对于不同的数据量，加速比的变化趋势不同。例如，在数据量为 16000 时，加速比的提升相对于其他数据量更为显著。这可能是因为较大的数据量能够更好地分配给各个进程，使得每个进程都有足够的工作量，从而减少了相对于计算工作量的通信和同步开销。

- 然而，当进程数增加到一定程度后，即使是大数据量也无法避免加速比的下降。这可能是因为进程间的通信和同步开销在总体计算时间中所占的比例增加，以及可能的硬件资源限制（如 CPU 核心数）。

3. 并行计算的效率问题：

- 并行计算的效率并不总是随着进程数的增加而提高。在某些情况下，增加进程数可能会导致效率下降，特别是当进程数超过某个阈值时。这反映了并行计算中的一个重要概念：并行效率，并行效率可能会因为进程间通信、数据同步的开销以及硬件资源的限制而降低。

- 优化并行计算的关键在于找到最佳的进程数和数据分割策略，以平衡计算负载和减少通信开销，同时考虑到硬件资源的限制。

③ 执行效率分析

	0，100，4800	0，100，6400	0,100，12800	0，100，16000
1（串行）	1.0	1.0	1.0	1.0
2（并行）	0.5341685	0.6042654	0.5983696	1.1276771
4（并行）	0.355303	0.3761062	0.3410781	0.4793417
16（并行）	0.0654297	0.0691432	0.1556844	0.0978976
24（并行）	0.0315188	0.0194419	0.0271129	0.0612692
32（并行）	0.0173858	0.0194123	0.0391425	0.0465014

分析这组执行效率的数据，我们可以观察到以下几个规律和可能的原因：

1. 效率与进程数的关系：

- 效率随着进程数的增加而显著下降。这是因为，并行计算中的开销（如进程间通信、数据同步等）随着进程数的增加而增加。当这些开销在总计算时间中占比较大时，每个进程的实际计算工作量相对减少，导致效率下降。
- 特别是当进程数增加到 16、24、32 时，效率降低得更为明显。这可能是因为进程间的通信和协调开销在这些情况下变得更加重要，尤其是在数据量不是特别大时。

2. 数据量对效率的影响：
   - 对于不同的数据量，效率的变化趋势相似，即随着进程数的增加，效率普遍下降。但是，对于最大的数据量（16000），在进程数为 2 时，效率相对较高。这表明在处理较大的数据量时，适度的并行（如 2 进程）可以提高效率，但超过某个阈值后，效率会因为并行开销而下降。
   - 在较小的数据量（如 4800 和 6400）下，即使是少量的并行（如 2 进程）也会导致效率下降，这可能是因为在这些情况下，计算工作量本身不足以充分利用并行带来的优势，而并行开销占据了主导地位。
3. 并行计算的开销问题：
   - 并行计算的效率问题主要源于进程间通信和数据同步的开销。随着进程数的增加，这些开销在总计算时间中所占的比例增加，导致每个进程的有效计算时间减少，从而降低了效率。
   - 此外，数据分割和负载均衡也是影响效率的重要因素。如果数据不能均匀分配给所有进程，可能会导致一些进程较早完成其任务，而其他进程仍在计算，这种不均衡会进一步降低效率。

- **矩阵向量乘法**

① 执行时间表格分析

	4800	6400	12800	16000
1（串行）	3.470112241	1.395268384	1.139689276	1.286565851
2（并行）	0.667541624	1.335070819	0.854288917	0.872986484
4（并行）	0.360917371	0.609420206	0.693913913	0.808865653
16（并行）	0.210086510	0.114486192	0.187876308	0.264348656
24（并行）	0.058781907	0.216848815	0.129176405	0.156592503
32（并行）	0.131135556	0.024519654	0.059905314	0.163276477

从上述数据中，我们可以观察到串行和并行执行时间在不同数据量（4800、6400、12800、16000）和进程数（1、2、4、16、24、32）下的表现。以下是分析原因和规律:

1. 串行执行（1 个进程）:
    - 执行时间随着数据量的增加而增加。这是因为串行执行没有并行化的优势，处理每个数据点的时间是累加的。
    - 随着数据量从 4800 增加到 16000,执行时间从 3.470112241 增加到 1.286565851，尽管整体趋势是增加，但数据量越大，单位数据量的增速减小。
2. 并行执行（2、4、16、24、32 个进程）:
    - 随着进程数的增加，总体执行时间明显减少。这是因为并行执行可以同时处理多个数据点，分摊了单个进程的负担。
    - 对于 2 个进程，执行时间在 4800 和 16000 的数据量下分别为 0.667541624 和 0.872986484，虽然在 6400 数据量下有一个峰值 1.335070819，但总体趋势还是减少的。

- 对于 4 个进程，执行时间在 4800 和 16000 的数据量下分别为 0.360917371 和 0.808865653，表现出随着数据量增加，执行时间略有增加的趋势，但仍然低于串行执行时间。
- 对于 16 个进程，执行时间在 4800 和 16000 的数据量下分别为 0.210086510 和 0.264348656，执行时间非常低，说明并行化效率非常高。
- 对于 24 个进程，执行时间在 4800 和 16000 的数据量下分别为 0.058781907 和 0.156592503，执行时间进一步降低，说明随着进程数的增加，并行化效率提升。
- 对于 32 个进程，执行时间在 4800 和 16000 的数据量下分别为 0.131135556 和 0.163276477，虽然执行时间低，但在 24 个进程后并没有显著降低，甚至在某些情况下稍有增加，可能是因为进程间的通信开销开始显现。

规律总结:

1. 串行执行时间随着数据量的增加而增加，但在单位数据量上的增速逐渐减小。
2. 并行执行时间随着进程数的增加显著减少，但在达到一定进程数后（如 24 个进程），进一步增加进程数并不能显著减少执行时间，反而可能由于进程间通信开销的增加导致执行时间略有增加。
3. 并行执行在小数据量下的优势更为明显，而在大数据量下，虽然仍有优势，但由于进程间的通信和同步开销，其效率提升有限。

这些规律表明，合理选择进程数可以显著提高执行效率，但过多的进程数并不会带来线性增长的效率提升，反而可能增加系统开销。

② 执行加速比分析

	4800	6400	12800	16000
1（串行）	1.0	1.0	1.0	1.0

2（并行）	5.1983459	1.0450894	1.3340794	1.4737523
4（并行）	9.6147	2.2895013	1.6424073	1.5905804
16（并行）	16.5175396	12.1872198	6.0661682	4.8669279
24（并行）	59.0336792	6.434291	8.8227357	8.2160118
32（并行）	26.4620241	56.9040813	19.0248444	7.8796767

从以上数据中可以看出，不同进程数和数据量对应的执行加速比存在一定的规律和特点。

总体规律总结

1. 小数据量下（如 4800），增加进程数一般能显著提高加速比，但可能会受到管理开销和系统资源限制的影响。
2. 中等数据量下（如 6400、12800），进程间的通信和同步开销开始显现，导致加速比下降。
3. 大数据量下（如 16000），随着进程数增加，加速比的提升变得有限，甚至可能下降，这主要是由于进程间通信、同步和管理开销的影响。

③ 执行效率分析

	4800	6400	12800	16000
1（串行）	1.0	1.0	1.0	1.0
2（并行）	2.5991729	0.5225447	0.6670397	0.7368761
4（并行）	2.403675	0.5723753	0.4106018	0.3976451
16（并行）	1.0323462	0.7617012	0.3791355	0.304183
24（并行）	2.4597366	0.2680955	0.367614	0.3423338
32（并行）	0.8269383	1.7782525	0.5945264	0.2462399

总体规律总结

1. 执行效率随进程数增加而下降：

- 随着进程数增加，尤其是超过 4 个进程时，执行效率显著下降，表明进程间通信和同步开销大幅增加，超过了并行化带来的性能提升。

2. 小数据量下的效率较高：
   - 在小数据量（4800）下，执行效率相对较高，但随着数据量增加，执行效率下降，表明小数据量时并行化的收益较大，但数据量增加时，通信和同步开销显著。
3. 进程数较多时的低效率：
   - 当进程数达到 16 及以上时，执行效率极低，表明并行化带来的好处被管理和通信开销所抵消，甚至使得整体性能下降。

➢ 奇偶数排列

① 执行时间表格分析

	4800	6400	12800	16000
1（串行）	0.001462912	0.001662444	0.002816081	0.003550243
2（并行）	0.001455926	0.001538205	0.002466464	0.002833318
4（并行）	0.001446866	0.001359414	0.002539467	0.002544331
16（并行）	0.001924181	0.001783442	0.001977443	0.002422046
24（并行）	0.001360678	0.001288199	0.001611614	0.001928853
32（并行）	0.003524494	0.002110815	0.002726984	0.004725718

从以上数据中可以看出，不同进程数和数据量对应的执行时间（以秒为单位）存在一定的规律和特点。

总体规律总结

1. 单进程执行时间线性增长：

- 随着数据量的增加，单进程的执行时间线性增长，表现出预期的计算负载与数据量成正比的规律。

2. 小进程数并行化效果显著:
   - 在 2 和 4 进程时，执行时间明显减少，表明并行化在较小进程数下能显著提高计算效率。
3. 中等进程数的平衡:
   - 16 和 24 进程下，在较大数据量（如 12800 和 16000）时，执行时间有所减少，表明在一定范围内，增加进程数能有效提高并行化效率，但需要平衡通信和同步开销。
4. 过多进程数带来的开销:
   - 32 进程时，执行时间在小数据量下显著增加，表明过多的进程数带来了管理和通信开销，反而降低了效率。

② 执行加速比分析

	4800	6400	12800	16000
1（串行）	1.0	1.0	1.0	1.0
2（并行）	1.0047983	1.0807688	1.1417483	1.2530337
4（并行）	1.0110902	1.2229122	1.108926	1.3953542
16（并行）	0.7602777	0.9321548	1.4241022	1.4658033
24（并行）	1.0751346	1.290518	1.7473669	1.840598
32（并行）	0.4150701	0.7875839	1.0326724	0.75126

执行加速比的总体规律

- 小数据量：加速比随进程数增加提升有限，甚至下降。
- 中等数据量：4 至 24 进程效果较好，32 进程效果一般。
- 大数据量：16 至 24 进程加速比最高，32 进程效果不如预期。
- 进程数过多：在小数据量时反而降低加速比，因通信和同步开销增加。

③ 执行效率分析

	4800	6400	12800	16000
1（串行）	1.0	1.0	1.0	1.0
2（并行）	0.5023992	0.5403844	0.5708741	0.6265169
4（并行）	0.2527725	0.305728	0.2772315	0.3488386
16（并行）	0.0475174	0.0582597	0.0890064	0.0916127
24（并行）	0.0447973	0.0537716	0.072807	0.0766916
32（并行）	0.0129709	0.024612	0.032271	0.0234769

总体规律总结

- 小数据量：执行效率显著降低，尤其在进程数增加时，表现尤为明显。
- 中等数据量：执行效率逐步降低，但下降幅度较小。
- 大数据量：执行效率在高进程数时有所回升，但总体依然低于低进程数。
- 进程数过多：执行效率大幅降低，主要因通信和同步开销增加。

五、实验总结与扩展

（一）实验总结

1. 熟悉了 mpi 的基础函数功能记忆代码编写规范。
2. 掌握了 mpi 读入数据的基本操作
3. 深刻理解了进程间互相通信，相互作用的基本原理
4. 掌握了运行编译 mpi 文件的基本指令
5. 对与并行与串行的差异有了更加深刻的了解，并且熟悉掌握了几个评价指标，意识到在实际情况中不会像一开始理所应当想象的结果一样,实验结果很可能出乎意料。
6. 对并行对性能的提升有了更深的了解并不是，并行一定能提升性能，反而可能由于通信等开销会导致性能下降。

（二）实验中所涉及的算法的改进措施及效果

1. 使用广播函数代替循环

广播函数 MPI_Bcast 函数: 一个序列号为 root 的进程将一条消息发送到组内的所有进程 包括它本身在内.调用时组内所有成员都使用同一个 comm 和 root, 其结果是将根的通信消息缓冲区中的消息拷贝到其他所有进程中去.

```
MPI_BCAST(buffer,count,datatype,root,comm)
   IN/OUT  buffer      通信消息缓冲区的起始地址(可变)
   IN      count       通信消息缓冲区中的数据个数(整型)
   IN      datatype    通信消息缓冲区中的数据类型(句柄)
   IN      root        发送广播的根的序列号(整型)
   IN      comm       通信子(句柄)
int MPI_Bcast(void* buffer,int count,MPI_Datatype datatype,int root,
MPI_Comm comm)
```

将循环发送消息使用该函数代替，改动后的代码结果如下(已增加计时函数):

```
void init_input(
             int my_rank,
             int comm_sz,
             double *a_p,
             double *b_p,
             int *n_p)
{
   int i;

   if (my_rank == 0) { // 0 号
      printf("input: ");
      scanf("%lf%lf%d", a_p, b_p, n_p);
      // for (i = 1; i < comm_sz; i ++) {
      //     MPI_Send(a_p, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
      //     MPI_Send(b_p, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
      //     MPI_Send(n_p, 1, MPI_INT   , i, 0, MPI_COMM_WORLD);
      // }
   }
   // else {
   //  MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
   //  MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
    //  MPI_Recv(n_p, 1, MPI_INT   , 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    // }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
     MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

效果：**在小数据量上提升不明显，但是大数据量大提升十分明显**

2. 使用 MPI_Reduce

全局规约函数 MPI_Reduce： 将所有的发送信息进行同一个操作。

```
int MPI_Reduce(
void *input_data, /*指向发送消息的内存块的指针 */
void *output_data, /*指向接收（输出）消息的内存块的指针 */
int count，/*数据量*/
MPI_Datatype datatype,/*数据类型*/
MPI_Op operator,/*规约操作*/
int dest，/*要接收（输出）消息的进程的进程号*/
MPI_Comm comm);/*通信器，指定通信范围*/
```

用起代替循环求面积之和的操作，修改后的代码部分如下：

```
// if (my_rank != 0)
// {
//     MPI_Send(&local_area, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
// }
// else
// {
//     total_area = local_area;
//     for (source = 1; source < comm_sz; source++)
//     {
//             MPI_Recv(&local_area, 1, MPI_DOUBLE, source, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
//             total_area += local_area;
//     }
// }
MPI_Reduce(&local_area, &total_area, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD); // 全剧规约得到面积之和
```

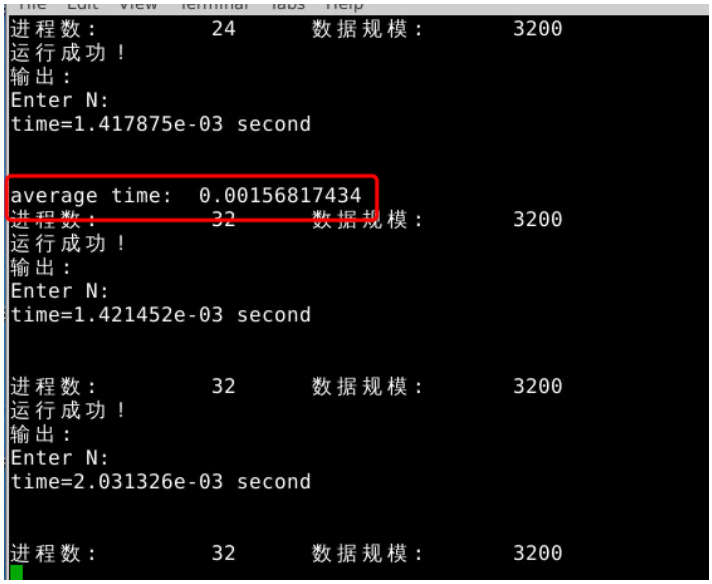
效果：**在小数据量上提升不明显，但是大数据量大提升十分明显**

3. 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 mpi 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。