并行程序设计第 05 次作业实验报告

李鹏 2113850

摘　要:　本次实验使用 MPI 实现了梯形积分、多个数组排序和高斯消元三个并行计算实验。在不同规模下的梯形积分中，观察了计算效率和精度的变化。多个数组排序实验中，发现 ARR_NUM 对计算耗时影响最大，其次是 seg，ARR_LEN 变化对耗时几乎没有显著影响。高斯消元实验中，随着线程数的增加，总体耗时减小，但进一步增加可能导致通信和任务分配开销抵消性能提升。通过这些实验，加深了对 MPI 并行计算的理解。

关键词:　多线程编程、MPI、梯形积分、多个数组排序、高斯消元

1　实验环境

三台云服务器的配置

使用 FianlShell 连接服务器

编程语言：C、MPI

```
roc@roc-virtual-machine:~/hello$ make
mpicc -o mpi_hello mpi_hello.c
```

```
roc@roc-virtual-machine:~/hello$ mpiexec -n 6 /home/roc/hello/mpi_hello
 Hello world from processor roc, rank 0 out of 6 processors
 Hello world from processor roc1, rank 1 out of 6 processors
 Hello world from processor roc2, rank 2 out of 6 processors
 Hello world from processor roc2, rank 3 out of 6 processors
 Hello world from processor roc, rank 4 out of 6 processors
 Hello world from processor roc1, rank 5 out of 6 processors
```

2　梯形积分

2.1　　实验目的：

通过 MPI 编程实现梯形积分法，熟悉 MPI 编程方法，调整 TOTAL_NUM 来改变积分的规模，观察不同规模对计算效率及精度的影响。

2.2　　实验方法

MPI 初始化及参数设置

引入了 MPI 相关的头文件#include <mpi.h>，使用 MPI_Init 对 MPI 进行初始化。通过 MPI_Comm_rank 获取当前进程的排名 rank，以及通过 MPI_Comm_size 获取总进程数 thread_num。

梯形积分计算

在梯形积分法中，每个进程负责计算一部分的积分。具体而言，每个进程通过对分配给它的一系列积分区间进行遍历，计算这些区间内函数值的和，最终得到局部的积分结果。

接收并合并结果

主 MPI 进程需要接收其他进程的计算结果，并进行合并。使用 MPI 的 MPI_Reduce 函数来实现。在这里，我们使用 MPI_Reduce 对所有进程的局部积分结果进行求和操作，将结果保存在主进程中。

2.3　　代码分析

cal 函数用于计算局部积分，接受两个参数 begin 和 size 分别表示计算的起始位置和个数。

```
double cal(int begin, int size)
{
    double temp_size = 0.0;
    for (int i = begin; i < fmin(begin + size, TOTAL_NUM); ++i)
    {
        // 使用梯形法则计算部分面积
        temp_size += (f(BEGIN_NUM + gap * i) + f(BEGIN_NUM + gap * (i + 1))) * gap / 2;
    }
    return temp_size;
}
```

f 函数：定义被积函数。

```
double f(double x)
{
    return x * x * x + 2 * x * x + x + 1;
}
```

main 函数：主函数中进行 MPI 初始化，获取 MPI 进程的排名和总数。然后根据排名分别计算局部积分，将结果发送给主进程，主进程累加所有结果并输出总面积。

```
if (rank != 0)
{
    double buf;
    // 计算每个线程的部分面积
    buf = cal((rank - 1) * each_thread_size, each_thread_size);
    // 发送部分面积到主线程
    MPI_Send(&buf, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}

if (rank == 0)
{
    for (int i = 1; i < thread_num; ++i)
    {
        double temp_size;
        // 接收各个线程的部分面积，并累加
        MPI_Recv(&temp_size, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        totalSize += temp_size;
    }
    // 输出总面积
    printf("Total size: %lf\n", totalSize);
}
```

2.4 实验结果

```
roc@roc-virtual-machine:~$ mpiexec -n 6 -f /home/roc/test1/config /home/roc/test1/mpi_test1
Hello world from processor roc, rank 0 out of 6 processors
Hello world from processor roc, rank 1 out of 6 processors
Hello world from processor roc, rank 2 out of 6 processors
Hello world from processor roc1, rank 3 out of 6 processors
Hello world from processor roc2, rank 4 out of 6 processors
Hello world from processor roc1, rank 5 out of 6 processors
```

当 TOTAL_NUM = 1000 时，程序耗时约为 353 毫秒。

当 TOTAL_NUM = 5000 时，程序耗时约为 578 毫秒。

当 TOTAL_NUM = 10000 时，程序耗时约为 1031 毫秒。

另外，随着 TOTAL_NUM 的增大，使用更多的数据点进行梯形积分，计算的结果会更加精确而，但变化幅度也不是很大，100 个梯形计算的结果也可以接受。

通过本次实验，得到了梯形积分法的近似结果。不同 MPI 进程计算不同区间的积分，通过 MPI 通信将各进程的结果汇总。通过调整 TOTAL_NUM 来改变积分的规模，观察不同规模对计算效率及精度的影响。

3 多个数组排序

3.1 实验目的

通过 MPI 编程实现对多个数组进行排序的任务，探讨在任务不均衡情况下的性能表现。通过调整 ARR_NUM、ARR_LEN 和 seg 来改变数组的规模和排序的粒度，观察不同规模对计算效率的影响。

3.2 实验方法

初始化和任务分配：

使用 init 函数初始化包含随机整数的二维数组 arr，其中 ARR_NUM 为数组个数，ARR_LEN 为数组长度。定义了 doTask 函数，该函数负责对给定范围内的数组进行排序。任务的粗颗粒度分配由全局变量 seg 控制。

MPI 通信及任务执行：

使用 MPI 进行初始化，获取当前进程的排名和总进程数。

主进程（rank=0）在循环中等待任意一个线程的状态，一旦收到消息，就向该线程发送当前任务的起始位置，并将任务起始位置加上分配的大小 seg。这一过程重复，直到所有任务完成。其他进程在循环中等待主进程发送任务状态，一旦收到任务状态，就执行 doTask 函数对相应范围的数组进行排序，然后向主进程发送完成状态。这一过程重复，直到所有任务完成。

最后，调用 MPI_Finalize 结束 MPI 进程。

3.3 代码分析

init 函数：用于初始化包含随机整数的二维数组 arr。

```
void init() {
    srand(time(NULL));
    for (int i = 0; i < ARR_NUM; i++) {
        for (int j = 0; j < ARR_LEN; j++)
            arr[i][j] = rand() % 100;
    }
}
```

doTask 函数接收一个起始位置 begin，然后对指定范围的二维数组进行排序，对每个一维数组进行升序排序。

```
void doTask(int begin) {
    for (int i = begin; i < (begin + SEG < ARR_NUM ? begin + SEG : ARR_NUM); ++i) {
        qsort(arr[i], ARR_LEN, sizeof(int), cmp);
    }
}
```

主函数中，通过 MPI 初始化，获取当前进程的排名和总数。
在主进程（rank=0）中，通过循环等待任意一个线程的状态。一旦收到状态，就向该线程发送当前任务的起始位置，并将任务起始位置加上分配的大小 seg。其他进程在循环中等待主进程发送任务状态。一旦收到任务状态，就执行 doTask 函数对相应范围的二维数组进行排序，并向主进程发送完成状态。这一过程重复，直到所有任务完成。最后，调用 MPI_Finalize 结束 MPI 进程。

```
init();
int current_task = 0; // 当前的任务
int rank, thread_num;
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &thread_num);
MPI_Status status;
int ready;
int done = 0;

if (rank == 0) {
    while (current_task < ARR_NUM) {
        MPI_Recv(&ready, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Send(&current_task, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
        current_task += SEG;
    }
    printf("success\n");
    done = 1;
} else {
    while (!done) {
        int begin;
        MPI_Send(&ready, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Recv(&begin, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        doTask(begin);
    }
}

MPI_Finalize();
return 0;
```

3.4 实验结果

通过 MPI 编程实现了对多个数组进行排序的任务。通过调整 ARR_NUM、ARR_LEN 和 seg 来改变数组的规模和排序的粒度，观察不同规模对计算效率的影响。

```
roc@roc-virtual-machine:~$ mpiexec -n 6 -f /home/roc/test2/config /home/roc/test2/mpi_test2
Hello world from processor roc, rank 0 out of 6 processors
Hello world from processor roc1, rank 1 out of 6 processors
Hello world from processor roc1, rank 2 out of 6 processors
Hello world from processor roc2, rank 3 out of 6 processors
Hello world from processor roc1, rank 4 out of 6 processors
Hello world from processor roc, rank 5 out of 6 processors
```

经过简单测试，发现改变 ARR_NUM 对计算耗时的影响最大，其次是 seg，而改变 ARR_LEN 对计算耗时几乎没有影响。可能是因为增加 ARR_NUM 直接影响需要排序的数组总数，这导致更多的排序任务分布在 MPI 进程之间，通信开销增大；改变 seg 意味着每个任务负责对更大的数组部分进行排序，依赖排序算法；改变 ARR_LEN 影响每个单独数组的大小，对整体计算耗时的影响较小。

4 高斯消元

4.1 实验目的

通过 MPI 实现高斯消去法解线性方程组，分析不同的优化策略对性能的影响。

4.2 实验方法

初始化和任务分配:

n（矩阵规模），seg（任务粒度），线程数。使用 change() 函数初始化包含随机浮点数的矩阵 a。主进程（rank=0）负责行的划分和任务分发，确定每个进程负责的行数和范围。

MPI 通信及任务执行:

通过 MPI_Init() 初始化 MPI，获取当前进程的排名和总进程数。主进程在循环中向其他进程发送任务信息，包括每个进程负责的起始行 task。其他进程接收主进程发送的任务信息，执行 OMP_elimination 函数对相应范围的矩阵进行消元操作。

MPI 通信机制:

使用 MPI 通信机制在主进程和其他进程之间传递任务信息和执行结果。主进程等待所有其他进程执行完当前任务后，继续下一轮的任务分发。

计时：

使用 gettimeofday() 记录开始时间。主进程在所有任务完成后，再次使用 gettimeofday() 记录结束时间，计算整个 MPI 程序的运行时间。

4.3 代码分析

OMP_elimination 函数：高斯消元操作。参数 i 表示当前行数，参数 j 表示要消元的行数。计算相差倍数 temp，然后遍历当前行中的元素，将 i 后面的数值减去相对应的值乘以倍数。将第 i 个元素置为 0。

```
void OMP_elimination(int i, int j) {
    // 求出相差倍数
    float temp = a[j][i] / a[i][i];
    // 遍历这一行的所有值，将 i 后面的数值依次减去相对应的值乘以倍数
    for (int k = i + 1; k <= n; ++k) {
        a[j][k] -= a[i][k] * temp;
    }
    // 第 i 个为 0
    a[j][i] = 0.00;
}
```

change 函数：用于初始化矩阵，随机生成矩阵的元素值。使用 rand() 生成随机整数，并转换为浮点数，存储到矩阵 a 中。

```
void change() {
    srand((unsigned) time(NULL));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= n; j++) {
            a[i][j] = (float) (rand() % 10000) / 100.00;
        }
    }
}
```

主函数 main：使用 change 函数初始化矩阵 a。使用 MPI 进行初始化，获取当前进程的排名和总数。记录开始时间，并进行时间的统计。主进程（rank=0）在循环中向其他进程发送任务信息，包括每个进程负责的起始行 task。其他进程接收主进程发送的任务信息，执行 OMP_elimination 函数对相应范围的矩阵进行消元操作。主进程等待所有其他进程执行完当前任务后，继续下一轮的任务分发。输出成功消息，包括矩阵规模、总运行时间等信息。

```
if (rank == 0) {
    printf("size : %d\n", n);
    for (line = 0; line < n - 1; ++line) {
        next_task = line + 1;
        seg = (n - next_task) / (thread_num - 1) + 1;
        for (int i = 1; i < thread_num; i++) {
            int task = (i - 1) * seg + next_task;
            MPI_Send(&task, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
        // 等待所有线程
        for (int j = 1; j < thread_num; ++j) {
            MPI_Recv(&ready, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
        }
    }
    printf("success\n");
    done = 1;
    gettimeofday(&stopTime, NULL);
    double trans_mul_time = (stopTime.tv_sec - startTime.tv_sec) * 1000 + (stopTime.tv_usec - startTime.tv_usec) * 0.001;
    printf("time: %lf ms\n", trans_mul_time);
} else {
    while (!done) {
        int task;
        MPI_Recv(&task, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        int min = task + seg < n ? task + seg : n;
        for (int i = task; i < min; ++i) {
            OMP_elimination(line, i);
        }
        MPI_Send(&ready, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
}
MPI_Finalize();
return 0;
```

4.4 实验结果

```
roc@roc-virtual-machine:~/hello$ mpiexec -n 6 -f /home/roc/test3/config /home/roc/test3/mpi_test3
Hello world from processor roc1, rank 0 out of 6 processors
Hello world from processor roc, rank 1 out of 6 processors
Hello world from processor roc1, rank 2 out of 6 processors
Hello world from processor roc, rank 3 out of 6 processors
Hello world from processor roc, rank 4 out of 6 processors
Hello world from processor roc2, rank 5 out of 6 processors
total time: 2879ms
```

```
roc@roc-virtual-machine:~/hello$ mpiexec -n 8 -f /home/roc/test3/config /home/roc/test3/mpi_test3
Hello world from processor roc2, rank 0 out of 8 processors
Hello world from processor roc2, rank 1 out of 8 processors
Hello world from processor roc1, rank 2 out of 8 processors
Hello world from processor roc, rank 3 out of 8 processors
Hello world from processor roc1, rank 4 out of 8 processors
Hello world from processor roc1, rank 5 out of 8 processors
Hello world from processor roc2, rank 6 out of 8 processors
Hello world from processor roc, rank 7 out of 8 processors
total time: 2320ms
```

```
roc@roc-virtual-machine:~/hello$ mpiexec -n 12 -f /home/roc/test3/config /home/roc/test3/mpi_test3
Hello world from processor roc1, rank 0 out of 12 processors
Hello world from processor roc, rank 1 out of 12 processors
Hello world from processor roc2, rank 2 out of 12 processors
Hello world from processor roc2, rank 3 out of 12 processors
Hello world from processor roc, rank 4 out of 12 processors
Hello world from processor roc1, rank 5 out of 12 processors
Hello world from processor roc1, rank 6 out of 12 processors
Hello world from processor roc, rank 7 out of 12 processors
Hello world from processor roc2, rank 8 out of 12 processors
Hello world from processor roc1, rank 9 out of 12 processors
Hello world from processor roc, rank 10 out of 12 processors
Hello world from processor roc, rank 11 out of 12 processors
total time: 1518ms
```

根据实验结果，MPI 在不同线程数下实验高斯消元的计算耗时分别为：

当 Thread_num=6 时，耗时 2879ms

当 Thread_num=8 时，耗时 2328ms

当 Thread_num=12 时，耗时 1518ms

随着线程数增加，总体耗时逐渐减小。在线程数为 12 时，耗时最短，达到 1518ms。当 Thread_num 从 6 增加到 8 时，总体耗时显著减小，说明多线程的并行计算有助于提高计算效率。进一步增加线程数到 12，耗时继续减小，但减速效果相较前期减缓。这是由于任务的分配和通信开销可能抵消了并行计算带来的性能提升。然而，进一步增加线程数可能会导致通信和任务分配的开销增加，性能提升减缓。