中山大学计算机院本科生实验报告

（2024 学年春季学期）

课程名称：并行程序设计　　批改人：

实验	2-基于 MPI 的并行矩阵乘法（进阶）	专业（方向）	计算机科学与技术
学号	21307174	姓名	刘俊杰
Email	liujj255@mail2.sysu.edu.cn	完成日期	2024/4/2

1. 实验目的

改进上次实验中的 MPI 并行矩阵乘法(MPI-v1)，并讨论不同通信方式对性能的影响。

输入：m, n, k三个整数，每个整数的取值范围均为[128, 2048]

问题描述：随机生成$m \times n$的矩阵A及$n \times k$的矩阵B，并对这两个矩阵进行矩阵乘法运算，得到矩阵C.

输出：A, B, C三个矩阵，及矩阵计算所消耗的时间t。

要求：

1.采用 MPI 集合通信实现并行矩阵乘法中的进程间通信；

2. 使用 mpi_type_create_struct 聚合 MPI 进程内变量后通信；

3. 尝试不同数据/任务划分方式（选做）。

2.实验过程和核心代码

2.1 实验过程

①首先 0 号进程接收矩阵 A 和矩阵 B 的维度，并初始化矩阵 A、B、C。

②将矩阵 A 按行分给不同的进程计算，计算出平均每个进程负责的矩阵 A 的行数 rows_per_proc。

③利用 MPI_Scatter 将每个进程负责的矩阵 A 的行分给每个进程,将矩阵 A、B 的维度和矩阵 B 用 MPI_Bcast 广播给每个进程。

④每个进程计算出结果后，利用 MPI_Gather 将结果归到 0 号进程中。

⑤0 号进程输出结果和计算时间。

2.2 核心代码

①首先 0 号进程接收矩阵 A 和矩阵 B 的维度，并初始化矩阵 A、B、C。

```
// 只有0号进程接收输入
if (my_rank == 0) {
    if (argc != 5) { // 运行参数出错
        printf("Usage: %s <A_rows> <A_cols> <B_rows> <B_cols>\n", argv[0]);
        MPI_Abort(MPI_COMM_WORLD, 1);
        exit(1);
    }

    A_rows = atoi(argv[1]);
    A_cols = atoi(argv[2]);
    B_rows = atoi(argv[3]);
    B_cols = atoi(argv[4]);

    if (A_cols != B_rows) { // A B 维度无法进行矩阵乘法
        printf("Error: Number of columns in matrix A must be equal to number of rows in matrix B.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
        exit(1);
    }
}
```

```
// 动态分配矩阵内存
int *A, *B, *C, *D;
A = (int *)malloc(A_rows * A_cols * sizeof(int));
B = (int *)malloc(B_rows * B_cols * sizeof(int));
C = (int *)calloc(A_rows * B_cols, sizeof(int));
D = (int *)malloc(A_rows * B_cols * sizeof(int));
//total_D = (int *)malloc(A_rows * B_cols * sizeof(int));

// 0号进程初始化矩阵 A 和 B
if (my_rank == 0) {
    srand(time(NULL));
    for (int i = 0; i < A_rows; i++) {
        for (int j = 0; j < A_cols; j++) {
            A[i * A_cols + j] = rand() % 10 + 1; // 生成介于 1 和 10 之间的随机数
        }
    }
    for (int i = 0; i < B_rows; i++) {
        for (int j = 0; j < B_cols; j++) {
            B[i * B_cols + j] = rand() % 10 + 1; // 生成介于 1 和 10 之间的随机数
        }
    }
}
for(int i = 0 ;i < A_rows;i++ ){
    for(int j = 0;j< B_cols ;j++){
        C[i * B_cols +j ]=0;
        D[i * B_cols +j ]=0;
        //total_D[i * B_cols +j ]=0;
    }
}
```

②将矩阵 A 按行分给不同的进程计算，计算出平均每个进程负责的矩阵 A 的行数 rows_per_proc。

```
// 广播矩阵维度
MPI_Bcast(&A_rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&A_cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&B_rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&B_cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

③利用 MPI_Scatter 将每个进程负责的矩阵 A 的行分给每个进程,将矩阵 A、B 的维度和矩阵 B 用 MPI_Bcast 广播给每个进程。

```
// 分发矩阵 A 的数据给各个进程
int *sendbuf = NULL;
if (my_rank == 0) {
    sendbuf = (int *)malloc(A_rows * A_cols * sizeof(int));
    for (int i = 0; i < A_rows; i++) {
        for (int j = 0; j < A_cols; j++) {
            sendbuf[i * A_cols + j] = A[i * A_cols + j];
        }
    }
}

// 接收缓冲区
int *recvbuf = (int *)malloc((rows_per_proc + extra_rows) * A_cols * sizeof(int));

// 分发矩阵 A 的数据
MPI_Scatter(sendbuf, rows_per_proc  * A_cols, MPI_INT, recvbuf, (rows_per_proc ) * A_cols, MPI_INT, 0, MPI_COMM_WORLD);

// 将接收到的数据复制到本地的 A 数组
for(int i = my_rank * rows_per_proc ;i < (my_rank + 1) * rows_per_proc ;i++){
    for(int j = 0;j < A_cols;j++  ){
        A[i * A_cols + j] = recvbuf[(i-my_rank*rows_per_proc) * A_cols + j];
    }
}


// 释放发送缓冲区和接收缓冲区
free(sendbuf);
free(recvbuf);

// 广播矩阵 B 的数据
MPI_Bcast(B, B_rows * B_cols, MPI_INT, 0, MPI_COMM_WORLD);
```

④每个进程计算出结果后，利用 MPI_Gather 将结果归到 0 号进程中。

```
// 计算局部矩阵乘法
matrixMultiplication(A, B, C, rows_per_proc, B_cols, A_cols, my_rank, num_procs, extra_rows, D);
```

```c
// 每个进程执行局部的矩阵乘法
void matrixMultiplication(int *A, int *B, int *C, int rows, int cols, int common, int my_rank, int num_procs, int extra_rows ,int *D) {
    if (my_rank < extra_rows) { // 判断是否前extra_rows进程是否要多负责一个进程
        for (int i = my_rank * (rows + 1); i < (my_rank + 1) * (rows + 1); i++) { // 每个进程负责矩阵 A 的对应部分
            for (int j = 0; j < cols; j++) {
                for (int k = 0; k < common; k++) {
                    C[i * cols + j] += A[i * common + k] * B[k * cols + j];
                }
                D[i * cols + j] = C[i * cols + j];
            }
        }
    } else {
        for (int i = my_rank * rows + extra_rows; i < (my_rank + 1) * rows + extra_rows; i++) {
            for (int j = 0; j < cols; j++) {
                for (int k = 0; k < common; k++) {
                    C[i * cols + j] += A[i * common + k] * B[k * cols + j];
                }
                D[i * cols + j] = C[i * cols + j];
            }
        }
    }
}
```

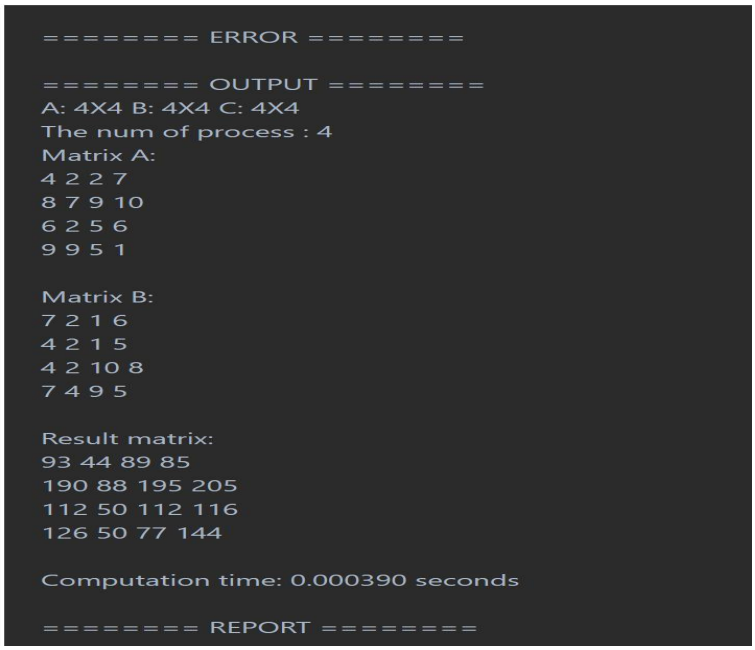
```c
// 0号进程收集局部结果
//MPI_Reduce(D, C, A_rows * B_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
// 使用 MPI_Gather 将各个进程的局部结果收集到根进程中
MPI_Gather(D + rows_per_proc * my_rank*4 , rows_per_proc * B_cols, MPI_INT, C, rows_per_proc  * B_cols, MPI_INT, 0, MPI_COMM_WORLD);
```

⑤0 号进程输出结果和计算时间。

```c
// 输出矩阵 A、B 和 C
if (my_rank == 0) {
    printf("A: %dX%d    B: %dX%d    C: %dX%d\n",A_rows,A_cols,B_rows,B_cols,A_rows,A_cols);
   printf("The num of process : %d\n",num_procs);
//      printf("Matrix A:\n");
//      printMatrix(A, A_rows, A_cols);

//      printf("Matrix B:\n");
//      printMatrix(B, B_rows, B_cols);

//      printf("Result matrix:\n");
//      printMatrix(C, A_rows, B_cols);

    end_time = MPI_Wtime();
    printf("Computation time: %f seconds\n", end_time - start_time);
}
```

3 实验结果

3.0 验证程序的计算正确性:



可以看到计算的结果是正确的

3.1 在虚拟机上进行实验

由于虚拟机资源有限，只有 4 个核，故只能跑 4 个进程

3.1.1 矩阵规模为 128

```
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 1 ./Lab2 128 128 128 128
A: 128X128    B: 128X128    C: 128X128
The num of process : 1
Computation time: 0.011057 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 2 ./Lab2 128 128 128 128
A: 128X128    B: 128X128    C: 128X128
The num of process : 2
Computation time: 0.005244 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 4 ./Lab2 128 128 128 128
A: 128X128    B: 128X128    C: 128X128
The num of process : 4
Computation time: 0.007688 seconds
```

3.1.2 矩阵规模为 256

```
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 1 ./Lab2 256 256 256 256
A: 256X256    B: 256X256    C: 256X256
The num of process : 1
Computation time: 0.067198 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 2 ./Lab2 256 256 256 256
A: 256X256    B: 256X256    C: 256X256
The num of process : 2
Computation time: 0.036430 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 4 ./Lab2 256 256 256 256
A: 256X256    B: 256X256    C: 256X256
The num of process : 4
Computation time: 0.053572 seconds
```

3.1.3 矩阵规模为 512

```
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 1 ./Lab2 512 512 512 512
A: 512X512    B: 512X512    C: 512X512
The num of process : 1
Computation time: 0.706773 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 2 ./Lab2 512 512 512 512
A: 512X512    B: 512X512    C: 512X512
The num of process : 2
Computation time: 0.388127 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 4 ./Lab2 512 512 512 512
A: 512X512    B: 512X512    C: 512X512
The num of process : 4
Computation time: 0.413725 seconds
```

3.1.4 矩阵规模为 1024

```
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 1 ./Lab2 1024 1024 1024 1024
A: 1024X1024    B: 1024X1024    C: 1024X1024
The num of process : 1
Computation time: 19.130648 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 2 ./Lab2 1024 1024 1024 1024
A: 1024X1024    B: 1024X1024    C: 1024X1024
The num of process : 2
Computation time: 6.986366 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 4 ./Lab2 1024 1024 1024 1024
A: 1024X1024    B: 1024X1024    C: 1024X1024
The num of process : 4
Computation time: 7.582389 seconds
```

3.1.5 矩阵规模为 2048

```
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 1 ./Lab2 2048 2048 2048 2048
A: 2048X2048    B: 2048X2048    C: 2048X2048
The num of process : 1
Computation time: 252.259035 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 2 ./Lab2 2048 2048 2048 2048
A: 2048X2048    B: 2048X2048    C: 2048X2048
The num of process : 2
Computation time: 138.678854 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpiexec -n 4 ./Lab2 2048 2048 2048 2048
A: 2048X2048    B: 2048X2048    C: 2048X2048
The num of process : 4
Computation time: 105.216116 seconds
```

3.1.6 实验结果

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.011057s	0.067198s	0.706773s	19.0130648s	252.25035s
2	0.005244s	0.036430s	0.388127s	6.86366s	138.678854s
4	0.007688s	0.053572s	0.412725s	7.582389s	105.216116s

可以看到进程数为 2 时的程序运行速度较进程数为 1 的快，但进程数为 4 的却比进程数为 2 的慢,可能是在虚拟机上 4 个进程已经耗尽了虚拟机的计算资源，相应其增加的进程之间通信消耗更大了，导致程序运行时间反而变长了。

3.2 在超算习堂上进行实验

超算习堂自由编程可使用多个进程资源进行实验,但似乎运行时间过久就不行

3.2.1 矩阵规模为 128

```
======== ERROR ========

======== OUTPUT ========
A: 128X128 B: 128X128 C: 128X128
The num of process : 1
Computation time: 0.015409 seconds

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
A: 128X128 B: 128X128 C: 128X128
The num of process : 2
Computation time: 0.011354 seconds

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
A: 128X128 B: 128X128 C: 128X128
The num of process : 4
Computation time: 0.007417 seconds

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
A: 128X128 B: 128X128 C: 128X128
The num of process : 8
Computation time: 0.003790 seconds

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
A: 128X128 B: 128X128 C: 128X128
The num of process : 16
Computation time: 0.002185 seconds

======== REPORT ========
```

3.2.2 矩阵规模为 256

```
======== ERROR ========

======== OUTPUT ========
A: 256X256 B: 256X256 C: 256X256
The num of process : 1
Computation time: 0.156335 seconds

======== REPORT ========
```

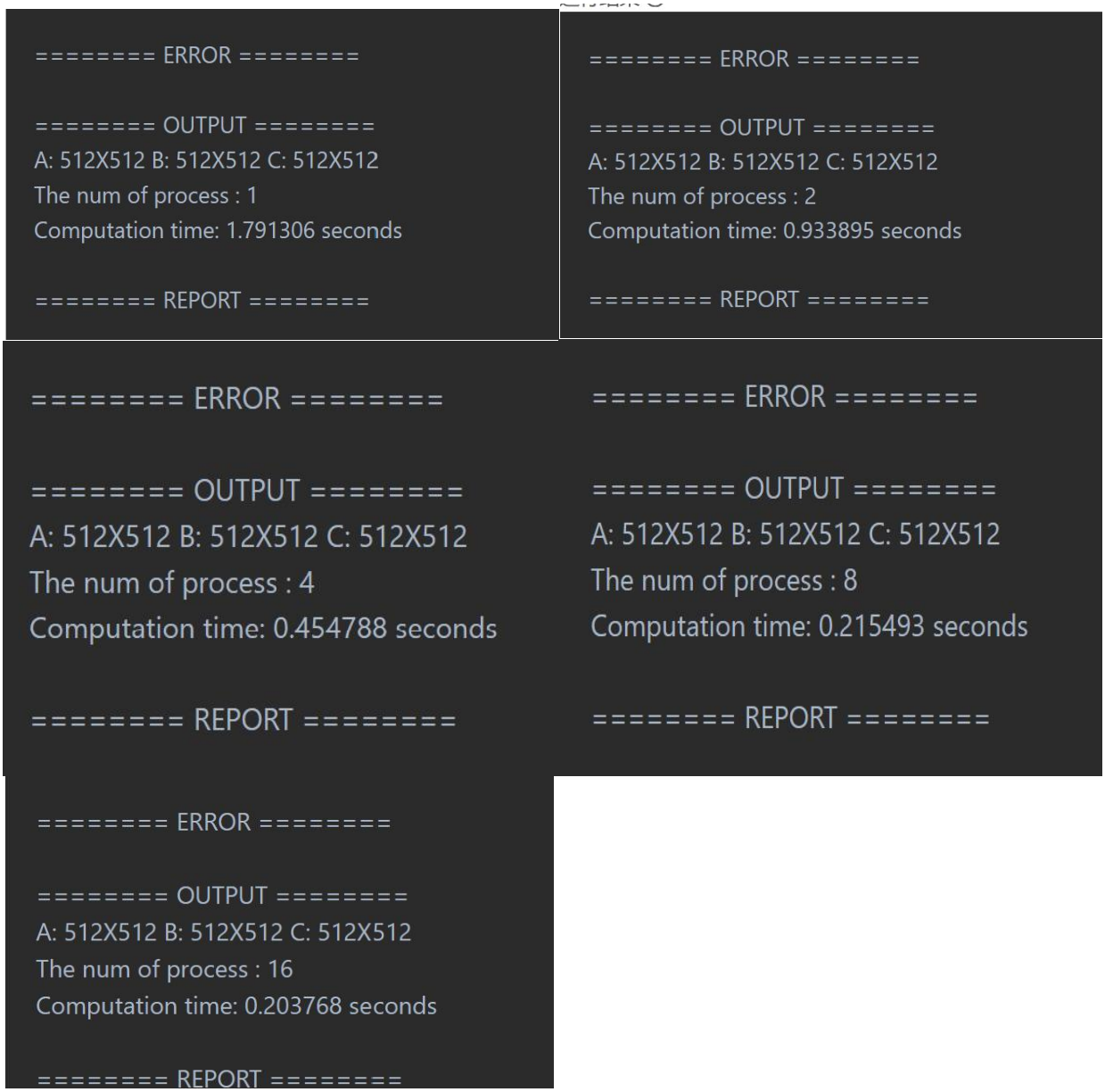
```
======== ERROR ========

======== OUTPUT ========
A: 256X256 B: 256X256 C: 256X256
The num of process : 2
Computation time: 0.085688 seconds

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
A: 256X256 B: 256X256 C: 256X256
The num of process : 4
Computation time: 0.042711 seconds

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
A: 256X256 B: 256X256 C: 256X256
The num of process : 8
Computation time: 0.043936 seconds

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
A: 256X256 B: 256X256 C: 256X256
The num of process : 16
Computation time: 0.020165 seconds

======== REPORT ========
```

3.2.3 矩阵规模为 512



3.2.4 结果对比

进程数	矩阵规模		
	128	256	512
1	0.015409s	0.156335s	1.791306s
2	0.011354s	0.085688s	0.933895s

4	0.007417s	0.042711s	0.454788s
8	0.003790s	0.043936s	0.215493s
16	0.002185s	0.020165s	0.203768s

可以看到进程数越多，程序的运行时间得到了加速。

4.实验感想

4.1 问题即解决方法

①Problem: MPI 集合通讯不能用于二维数组

Solution: 将二维数组扩展为一维数组进行传播通讯

②Problem: 对于矩阵 A 的行数除进程数除不尽即有的进程接受需要负责的到 A 的矩阵行数会比平均每个负责的要多一行,而 MPI_Scatter 只能给每个进程分发相同数目的数量，所以 MPI_Scatter 不能处理这种情况。

Solution: 可以对矩阵 A 进行填充，MPI_Scatter 将行分给对应进程后，进程从中提取对应的部分。(实验中未实现)

4.2 实验感悟

在并行计算中，通信开销往往是性能的一个瓶颈。因此，选择合适的通信方式和优化通信模式对于提高性能至关重要。在这个实验中，我使用了集合通信函

数 MPI_Gather 来收集结果，而没有直接采用点对点通信方式。这样做可以减少通信的次数，提高效率。

并且在并行计算中，如何划分数据对于负载均衡和性能的影响很大。合理的数据划分可以使得各个进程的计算量尽量均衡，避免出现单个进程负载过重或者大量空闲的情况。在矩阵乘法中，可以尝试不同的数据划分方式，比如按行划分或者按列划分，以找到最优的划分方式。

我还需要加强对 MPI 的学习，对 MPI 工具的应用更加熟练，明白对不同的问题和数据类型应该采用什么方法解决。