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	1-MPI 矩阵乘法	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	2024/3/28

1. 实验目的

使用 MPI 点对点通信实现并行矩阵乘法，以探究并行计算在提高矩阵乘法效率方面的作用。

研究不同进程数量和矩阵规模下程序的运行时间，分析并行性能随进程数量和矩阵规模的变化。

讨论:

在内存受限情况下，如何进行大规模矩阵乘法计算？

如何提高大规模稀疏矩阵乘法性能？

2. 实验过程和核心代码

2.0 实验思路

输入矩:阵 A\ B
结果矩阵: C
将矩阵 A 的不同行和整个矩阵 B 发送给

2.1 MPI 初始化和矩阵维度初始化

使用 0 号进程从运行参数中获得矩阵维度

```
int my_rank, num_procs;
int A_rows, A_cols, B_rows, B_cols;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

double start_time, end_time;

// 只有0号线程接收输入
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);
    }
}
```

2.2 利用 MPI_Send 和 MPI_Recv 将矩阵维度信息广播到进程

```
// 利用点对点和循环实现 广播矩阵维度
if (my_rank == 0) {
    int dimensions[4] = {A_rows, A_cols, B_rows, B_cols};
    for (int dest = 1; dest < num_procs; dest++) {
        MPI_Send(dimensions, 4, MPI_INT, dest, 0, MPI_COMM_WORLD);
    }
} else {
    int dimensions[4];
    MPI_Recv(dimensions, 4, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    A_rows = dimensions[0];
    A_cols = dimensions[1];
    B_rows = dimensions[2];
    B_cols = dimensions[3];
}
```

2.3 计算每个进程平均负责的 A 的行数并初始化矩阵 A、B

```
// 计算每个线程负责的行数
int rows_per_proc = A_rows / num_procs;
int extra_rows = A_rows % num_procs;

// 动态分配矩阵内存
int **A, **B, **C;
A = (int **)malloc(A_rows * sizeof(int *));
B = (int **)malloc(B_cols * sizeof(int *));
C = (int **)malloc(A_rows * sizeof(int *));
for (int i = 0; i < A_rows; i++) {
    A[i] = (int *)malloc(A_cols * sizeof(int));
    C[i] = (int *)calloc(B_cols, sizeof(int));
}
for (int i = 0; i < B_cols; i++) {
    B[i] = (int *)malloc(B_cols * sizeof(int));
}

// 0号进程初始化矩阵 A 和 B
if (my_rank == 0) {
    srand(time(NULL));
    //printf("Matrix A:\n");
    for (int i = 0; i < A_rows; i++) {
        for (int j = 0; j < A_cols; j++) {
            A[i][j] = rand() % 10 + 1; // 生成介于 1 和 10 之间的随机数
        }
    }

    //printf("Matrix B:\n");
    for (int i = 0; i < B_rows; i++) {
        for (int j = 0; j < B_cols; j++) {
            B[i][j] = rand() % 10 + 1; // 生成介于 1 和 10 之间的随机数
        }
    }
}
```

2.4 将每个进程负责矩阵 A 的对应行发送出去

需要注意行数除进程数除不尽的情况,前 extra_rows 的进程要多负责一行

```
// 发送矩阵 A 的部分数据给其他线程
if (my_rank == 0) {
    for (int dest = 1; dest < num_procs; dest++) {
        if(dest < extra_rows){// 判断是否前extra_rows线程是否要多负责一个进程
            for (int i = dest * (rows_per_proc + 1) ; i < (dest + 1) * (rows_per_proc + 1); i++) {
                MPI_Send(A[i], A_cols, MPI_INT, dest, 0, MPI_COMM_WORLD);
            }
        }
        else{
            for (int i = dest * rows_per_proc + extra_rows; i < (dest + 1) * rows_per_proc + extra_rows; i++) {
                MPI_Send(A[i], A_cols, MPI_INT, dest, 0, MPI_COMM_WORLD);
            }
        }
    }

} else {

        if(my_rank < extra_rows){// 判断是否前extra_rows线程是否要多负责一个进程
            for (int i = my_rank * (rows_per_proc + 1) ; i < (my_rank + 1) * (rows_per_proc + 1); i++) {
                MPI_Recv(A[i], A_cols, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
        }
        else{
            for (int i = my_rank * rows_per_proc + extra_rows; i < (my_rank + 1) * rows_per_proc + extra_rows; i++) {
                MPI_Recv(A[i], A_cols, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
        }


}
```

2.5 将整个矩阵 B 发给其他进程

```
// 发送矩阵 B 的数据给其他线程
if (my_rank == 0) {
    for (int dest = 1; dest < num_procs; dest++) {
        for (int i = 0; i < B_rows; i++) {
            MPI_Send(B[i], B_cols, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    }
} else {
    for (int i = 0; i < B_rows; i++) {
        MPI_Recv(B[i], B_cols, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

2.6 每个进程执行矩阵乘法

```
// 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);
```

```
// 每个线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) {
    if(my_rank < extra_rows){ // 判断是否前extra_rows线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++) { // 每个线程负责矩阵 A 的对应部分
            for (int j = 0; j < cols; j++) {
                for (int k = 0; k < common; k++) {
                    C[i][j] += A[i][k] * B[k][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][j] += A[i][k] * B[k][j];
                }
            }
        }
    }
}
```

2.7 0 号进程输出结果

```
// 0号进程输出结果和运算时间
if (my_rank == 0) {
    end_time = MPI_Wtime();

    // printf("Matrix A:\n");
    // for (int i = 0; i < A_rows; i++) {
    //     for (int j = 0; j < A_cols; j++) {
    //         printf("%d ", A[i][j]);
    //     }
    //     printf("\n");
    // }
    // printf("\n");

    // printf("Matrix B:\n");
    // for (int i = 0; i < B_rows; i++) {
    //     for (int j = 0; j < B_cols; j++) {
    //         printf("%d ", B[i][j]);
    //     }
    //     printf("\n");
    // }
    // printf("\n");

    // printf("Result matrix:\n");
    // for (int i = 0; i < A_rows; i++) {
    //     for (int j = 0; j < B_cols; j++) {
    //         printf("%d ", C[i][j]);
    //     }
    //     printf("\n");
    // }
    // printf("\n");

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

2.8 MPI_Finalize()并释放内存

```
// 释放内存
for (int i = 0; i < A_rows; i++) {
    free(A[i]);
    free(C[i]);
}
for (int i = 0; i < B_cols; i++) {
    free(B[i]);
}
free(A);
free(B);
free(C);

MPI_Finalize();
return 0;
```

3. 实验结果

3.1 实验结果正确性验证(四个进程)

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

======== OUTPUT ========
Matrix A:
3 3 8 5 6
2 5 10 10 6
7 9 1 9 1
4 5 2 4 9
9 10 1 7 9

Matrix B:
7 2 1 2 1
5 8 3 4 5
10 8 4 10 10
3 8 3 10 2
2 1 9 5 5

Result matrix:
143 140 113 178 138
181 210 141 254 177
133 167 74 155 85
103 105 120 133 102
162 171 145 183 128
Computation time: 0.000029 seconds

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

可以看到计算结果正确

3.2 运行时间结果

由于虚拟机资源有限，最多可以允许 4 个进程

3.2.1 1 个进程

```
ljj@ljj-virtual-machine:~/parrallel_programming$ ^C
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 1 ./Lab1 128 128 128 128
Computation time: 0.015423 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 1 ./Lab1 256 256 256 256
Computation time: 0.127436 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 1 ./Lab1 512 512 512 512
Computation time: 1.092201 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 1 ./Lab1 1024 1024 1024 1024
Computation time: 10.044151 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 1 ./Lab1 2048 2048 2048 2048
Computation time: 89.042888 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$
```

3.2.2 2 个进程

```
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 2 ./Lab1 128 128 128 128
Computation time: 0.007757 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 2 ./Lab1 256 256 256 256
Computation time: 0.084623 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 2 ./Lab1 512 512 512 512
Computation time: 0.714060 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 2 ./Lab1 1024 1024 1024 1024
Computation time: 5.965219 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 2 ./Lab1 2048 2048 2048 2048
Computation time: 55.334119 seconds
```

3.2.3 4 个进程

```
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 4 ./Lab1 128 128 128 128
Computation time: 0.006912 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 4 ./Lab1 256 256 256 256
Computation time: 0.043915 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 4 ./Lab1 512 512 512 512
Computation time: 0.408103 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 4 ./Lab1 1024 1024 1024 1024
Computation time: 3.375850 seconds
ljj@ljj-virtual-machine:~/parrallel_programming$ mpirun -np 4 ./Lab1 2048 2048 2048 2048
Computation time: 32.002999 seconds
```

3.3 综合对比

进程数	128	256	512	1024	2048
1	0.015423s	0.127436s	1.092201s	10.044151s	89.042888s
2	0.007757s	0.084623s	0.714060s	5.965219s	55.334119s
4	0.006912s	0.043915s	0.408103s	3.375850s	32.002999s

4. 实验感想

4.1 感想

通过实验，我深入理解了 MPI 点对点通信的概念和实现，并学会了如何利用 MPI 实现并行矩阵乘法。

在观察不同进程数量和矩阵规模下程序的运行时间后，我发现并行计算能够有效提高计算效率，特别是在处理大规模矩阵时。通过分析并行性能，我了解到在选择进程数量时需要考虑到计算节点的数量、通信开销等因素，并且需要权衡计算能力和通信开销之间的关系。

4.2 讨论

4.2.1 在内存受限情况下，方法进行大规模矩阵乘法计算的方法：

①使用分布式存储：将矩阵分块存储在不同的计算节点上，通过并行计算和通信来实现大规模矩阵乘法，减少单个节点的内存压力。
②压缩存储：对矩阵进行压缩存储，例如利用稀疏矩阵的特点进行存储，减少内存占用，从而提高大规模矩阵乘法计算的效率。

4.2.2 提高大规模稀疏矩阵乘法性能的方法包括：

①矩阵分块：将稀疏矩阵划分为若干块，只对非零元素进行计算，减少计算量，从而提高计算效率。
②使用专门的稀疏矩阵存储格式：选择合适的稀疏矩阵存储格式，例如压缩稀疏矩阵存储格式，可以减少存储空间，并提高计算效率。