

中山大学计算机院本科生实验报告

(2024 学年秋季学期)

课程名称：高性能计算程序设计

批改人：

实验	通用矩阵乘法	专业（方向）	信息与计算科学
学号	22336049	姓名	陳日康
Email	chenih5@mail2.sysu.edu.cn	完成日期	2024 年 10 月 7 日

1. 实验目的

- 1. 通过 MPI 实现通用矩阵乘法
- 2. 基于 MPI 通用矩阵乘法优化
- 3. 将“实验 0”改造成矩阵乘法库函数
- 4. 构造 MPI 矩阵乘法加速比和并行效率表

2. 实验过程和核心代码

(1). 通过 MPI 实现通用矩阵乘法（点对点通信）

(i). 矩阵乘法函数 multiplyMatrices

```
void multiplyMatrices(const double* matA, const double* matB, double* matC, int dimension) {
    for (int i = 0; i < dimension; ++i) {
        for (int j = 0; j < dimension; ++j) {
            double sum = 0.0;
            for (int k = 0; k < dimension; ++k) {
                sum += matA[i * dimension + k] * matB[k * dimension + j];
            }
            matC[i * dimension + j] = sum;
        }
    }
}
```

执行矩阵乘法，将两个矩阵 matA 和 matB 相乘，结果存储在 matC 中。for (int k = 0; k < dimension; ++k): 执行矩阵乘法的核心逻辑，累加 matA[i][k]与 matB[k][j]的乘积，最终计算得到 matC[i][j]。

(ii). 矩阵广播函数 MPI_Bcast

```
MPI_Bcast(matB, MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, MASTER_PROCESS, MPI_COMM_WORLD);
```

MPI_Bcast 是一种单对多的通信模式，主进程将数据广播给所有其他进程。将矩阵 matB 从主进程广播给所有其他进程，使每个进程都能获得完整的 matB 矩阵。MATRIX_SIZE * MATRIX_SIZE: 广播的数据总量，即整个矩阵的大小。MASTER_PROCESS: 广播的发起者是主进程（编号为 0 的进程）。MPI_COMM_WORLD: 使用全局通信器，表示所有进程参与通信。

(iii). 矩阵发送函数 MPI_Send 与接收函数 MPI_Recv

```
if (numProcesses > 1) {
    if (processRank == MASTER_PROCESS) {
        for (int dest = 1; dest < numProcesses; ++dest) {
            int startIdx = (dest - 1) * localMatSize;
            MPI_Send(&matA[startIdx * MATRIX_SIZE], localMatSize * MATRIX_SIZE, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(matA, localMatSize * MATRIX_SIZE, MPI_DOUBLE, MASTER_PROCESS, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        multiplyMatrices(matA, matB, matC, localMatSize);
        MPI_Send(matC, localMatSize * MATRIX_SIZE, MPI_DOUBLE, MASTER_PROCESS, 0, MPI_COMM_WORLD);
    }
} else {
    multiplyMatrices(matA, matB, matC, MATRIX_SIZE);
}
```

MPI_Send 是点对点通信，主进程将部分数据发送给指定的目标进程，将矩阵 matA 的一部分发送给其他进程，进行并行计算。&matA[startIdx * MATRIX_SIZE]：发送矩阵 matA 中从 startIdx 开始的子矩阵（对应一个进程的工作部分）。dest：接收数据的目标进程号。0：消息标签，用于标识消息的类型。MPI_Recv 是另一种点对点通信，用于从指定进程接收数据。从主进程接收分配给当前进程的 matA 子矩阵，用于并行计算。MASTER_PROCESS：发送数据的源进程号，即主进程。

(iv). 主进程汇总所有结果

```
if (processRank == MASTER_PROCESS) {
    for (int source = 1; source < numProcesses; ++source) {
        int startIdx = (source - 1) * localMatSize;
        MPI_Recv(&matC[startIdx * MATRIX_SIZE], localMatSize * MATRIX_SIZE, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

这里 MPI_Recv 用于接收每个子进程计算出的部分结果（即矩阵乘法的局部结果）。主进程从各个子进程接收计算结果，将子矩阵 matC 拼接成完整的结果矩阵。

(2). 基于 MPI 通用矩阵乘法优化

(i). 矩阵块乘法 multiplyBlock

```
void multiplyBlock(double* matrixA, double* matrixB, double* matrixC, int dim, int blockRows) {
    for (int row = 0; row < blockRows; ++row) {
        for (int col = 0; col < dim; ++col) {
            matrixC[row * dim + col] = 0.0;
            for (int inner = 0; inner < dim; ++inner) {
                matrixC[row * dim + col] += matrixA[row * dim + inner] * matrixB[inner * dim + col];
            }
        }
    }
}
```

计算局部矩阵块的乘积，即每个进程处理 matrixA 的部分行与整个 matrixB 的乘积，结果存储在 matrixC 中。matrixA[row * dim + inner]与 matrixB[inner * dim + col]实现标准的矩阵乘法，其中 matrixA 按行划分，matrixB 被所有进程共享。blockRows 是每个进程处理的 matrixA 子矩阵的行数，dim 是矩阵的全局维度。

(iii). 进程初始化与通信 MPI_Init 和 MPI_Comm_rank/size

```
MPI_Init(&argc, &argv);

int processRank, worldSize;
MPI_Comm_rank(MPI_COMM_WORLD, &processRank);
MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
```

MPI_Init: 初始化 MPI 环境, 开启并行处理。MPI_Comm_rank: 获取当前进程的编号 processRank, 每个进程都有唯一的编号。MPI_Comm_size: 获取总进程数 worldSize, 即当前程序运行时使用的进程数量。

(iii). 矩阵分块通信 MPI_Scatter 和 MPI_Bcast

```
// Scatter matrix A to all processes
MPI_Scatter(fullMatrixA, rowsPerProc * matrixDim, MPI_DOUBLE, localBlockA, rowsPerProc * matrixDim, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Broadcast matrix B to all processes
MPI_Bcast(fullMatrixB, matrixDim * matrixDim, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

MPI_Scatter 将矩阵 A 的不同行块分发给不同的进程, 每个进程接收 rowsPerProc 行。MPI_Bcast 将矩阵 B 广播给所有进程, 所有进程都共享完整的 matrixB, 以便在矩阵乘法中使用。

(iv). 结果收集 MPI_Gather

```
// Gather results from all processes
MPI_Gather(localBlockC, rowsPerProc * matrixDim, MPI_DOUBLE, resultMatrix, rowsPerProc * matrixDim, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

将各个进程计算的部分结果 (矩阵块 localBlockC) 收集到主进程中, 形成完整的结果矩阵 resultMatrix。

(3). 将 “实验 0” 改造成矩阵乘法库函数

将 Lab1 的单进程矩阵乘法改进为一个库函数 matrix_multiply, 在 Linux 系统中将此函数编译为 .so 文件, 由 MPI 调用。分为三个程序: 定义矩阵乘法函数的头文件 matrix_multiply.h、实现矩阵乘法的文件 matrix_multiply.c 和一个测试文件 mpi_program.c, 该测试文件用于测试标准的库函数 matrix_multiply 是否成功生成并由 MPI 直接调用。

创建头文件 matrix_multiply.h, 声明 matrix_multiply 函数

```
#ifndef matrix_multiply_h
#define matrix_multiply_h
#include<stdio.h>
#include<stdlib.h>

void matrix_multiply(double**A, double**B, double**C, int M, int N, int K);

#endif
```

编写矩阵乘法的实现文件 matrix_multiply.c:

```
#include "matrix_multiply.h"
#include<stdio.h>
#include<stdlib.h>
void matrix_multiply(double**A,double**B,double**C,int M,int N,int K){
    for(int m=0;m<M;m++){
        for(int k=0;k<K;k++){
            for(int n=0;n<N;n++){
                C[m][k]+=A[m][n]*B[n][k];
            }
        }
    }
}
```

mpi_program.c:

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

初始化 MPI 环境，获取当前进程的编号和总进程数。

```
// Broadcast dimensions to all processes
MPI_Bcast(&rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

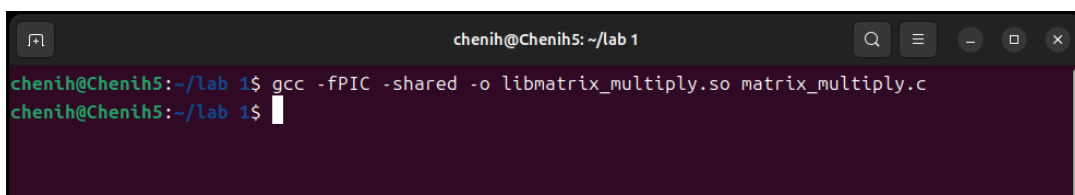
将矩阵的行数和列数从主进程 (rank 0)广播给所有其他进程。MPI_Bcast 将 rows 和 cols 的值发送给所有其他进程，确保每个进程都知道矩阵的尺寸。1 表示广播一个元素（即行数和列数）。

```
// Perform matrix multiplication A * B = C
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        C[i * cols + j] = 0;
        for (int k = 0; k < cols; k++) {
            C[i * cols + j] += A[i * cols + k] * B[k * cols + j];
        }
    }
}
```

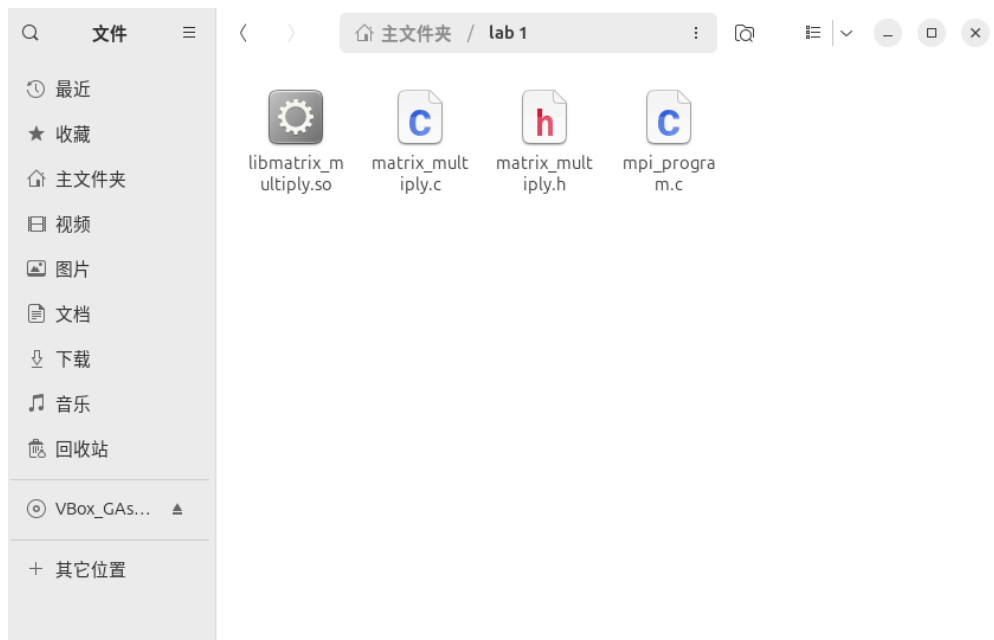
在主进程(rank 0)上进行矩阵乘法运算，采用三重循环实现矩阵乘法计算 $A*B=C$ 。内层循环计算矩阵乘法的点积，将 A 的第 i 行与 B 的第 j 列相乘并累加结果。

使用 GCC 编译 matrix_multiply.c 文件为共享库。执行以下命令：

```
gcc -fPIC -shared -o libmatrix_multiply.so matrix_multiply.c
```



```
chenih@Chenih5: ~/lab 1
chenih@Chenih5:~/lab 1$ gcc -fPIC -shared -o libmatrix_multiply.so matrix_multiply.c
chenih@Chenih5:~/lab 1$
```



编译 MPI 程序，并确保链接共享库：

```
mpicc -o mpi_program mpi_program.c -L. -lmatrix_multiply
```

使用 MPI 运行程序：

```
mpirun -np 1 ./mpi_program
```

```
chenih@Chenih5: ~/lab 1
chenih@Chenih5:~/lab 1$ gcc -fPIC -shared -o libmatrix_multiply.so matrix_multiply.c
chenih@Chenih5:~/lab 1$ mpicc -o mpi_program mpi_program.c -L. -lmatrix_multiply
chenih@Chenih5:~/lab 1$ mpirun -np 1 ./mpi_program
Enter number of rows and columns: 
```

(4). 构造 MPI 版本矩阵乘法加速比和并行效率表

并行效率通常用符号 E 表示，它是处理器数量 P 和加速比 S 的比值，公式为： $E = \frac{S}{P}$ ，详细表格见实验结果部分。

3. 实验结果

(1). 通过 MPI 实现通用矩阵乘法（点对点通信）

打印运行结果：

```
PS D:\Code\C++> g++ lab1_1.cpp -o test -I D:\MPI\Include -L D:\MPI\Lib\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Matrix A:
0.890155 0.130707 ...
0.378123 0.622393 ...
...
Matrix B:
0.509721 0.183354 ...
0.444104 0.955429 ...
...
Matrix C:
33.5702 32.9351 ...
29.3279 32.9851 ...
...
Execution time: 4.8931 ms
```

为简化结果，只打印首 2*2 矩阵，其他由省略号替代。由 showOutput 控制是否打印矩阵结果。

128 阶：

```
PS D:\Code\C++> g++ lab1_1.cpp -o test -I D:\MPI\Include -L D:\MPI\Lib\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 4.6387 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 5.2552 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 1.2797 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 1.5353 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 3.3009 ms
```

256 阶：

```
PS D:\Code\C++> g++ lab1_1.cpp -o test -I D:\MPI\Include -L D:\MPI\Lib\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 36.8656 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 38.2394 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 3.1067 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 3.1941 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 4.8734 ms
```

512 阶：

```
PS D:\Code\C++> g++ lab1_1.cpp -o test -I D:\MPI\Include -L D:\MPI\Lib\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 492.906 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 432.102 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 14.299 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 6.0991 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 9.4854 ms
```

1024 阶：

```
PS D:\Code\C++> g++ lab1_1.cpp -o test -I D:\MPI\Include -L D:\MPI\Lib\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 3414.06 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 3461.73 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 101.02 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 32.9793 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 48.6308 ms
```

2048 阶：

```
PS D:\Code\C++> g++ lab1_1.cpp -o test -I D:\MPI\Include -L D:\MPI\Lib\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 48390.6 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 47514.8 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 913.527 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 167.458 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 163.27 ms
```

点对点通信耗时表：

n\i	128	256	512	1024	2048
1	4.6387	36.8656	492.906	3414.06	48390.6
2	5.2552	38.2394	432.102	3461.73	47514.8
4	1.2797	3.1067	14.299	101.02	913.527
8	1.5353	3.1941	6.0991	32.9793	167.458
16	3.3009	4.8734	9.4854	48.6308	163.27

(2). 基于 MPI 通用矩阵乘法优化

打印运行结果：

```
PS D:\Code\C++> g++ lab1_2.cpp -o test -I D:\MPI\Include -L D:\MPI\Lib\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Matrix A:
0.796543 0.183435 ...
0.164656 0.534089 ...
...
Matrix B:
0.294434 0.0528546 ...
0.904388 0.019393 ...
...
Matrix C:
29.4989 32.1585 ...
34.4891 36.7104 ...
...
Execution time: 9.1493 ms
```

同样由于简化结果，只打印首 2*2 矩阵结果，并由 printResults 控制是否打印矩阵结果。

128 阶:

```
PS D:\Code\C++> g++ lab1_2.cpp -o test -I D:\\MPI\\Include -L D:\\MPI\\Lib\\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 7.4748 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 4.3267 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 3.2012 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 2.1944 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 2.483 ms
```

256 阶:

```
PS D:\Code\C++> g++ lab1_2.cpp -o test -I D:\\MPI\\Include -L D:\\MPI\\Lib\\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 61.7449 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 30.607 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 16.6804 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 10.3417 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 11.2102 ms
```

512 阶:

```
PS D:\Code\C++> g++ lab1_2.cpp -o test -I D:\\MPI\\Include -L D:\\MPI\\Lib\\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 619.085 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 317.945 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 168.523 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 115.525 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 123.604 ms
```

1024 阶:

```
PS D:\Code\C++> g++ lab1_2.cpp -o test -I D:\\MPI\\Include -L D:\\MPI\\Lib\\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 5871.86 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 3436.32 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 2059.59 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 1561.49 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 1590.31 ms
```

2048 阶:

```
PS D:\Code\C++> g++ lab1_2.cpp -o test -I D:\\MPI\\Include -L D:\\MPI\\Lib\\x64 -lmsmpi
PS D:\Code\C++> mpiexec -n 1 test
Execution time: 169534 ms
PS D:\Code\C++> mpiexec -n 2 test
Execution time: 37037.8 ms
PS D:\Code\C++> mpiexec -n 4 test
Execution time: 46175.4 ms
PS D:\Code\C++> mpiexec -n 8 test
Execution time: 26988.1 ms
PS D:\Code\C++> mpiexec -n 16 test
Execution time: 20848.1 ms
```


集合通信耗时表：

n\i	128	256	512	1024	2048
1	7.4748	61.7449	619.085	5871.86	169534
2	4.3267	30.607	317.945	3436.32	37037.8
4	3.2012	16.6804	168.523	2059.59	46175.4
8	2.1944	10.3417	115.525	1561.49	26988.1
16	2.483	11.2102	123.604	1590.31	20848.1

(3). 将“实验 0”改造成矩阵乘法库函数

```
chenih@Chenih5: ~/lab1
chenih@Chenih5:~/lab1$ gcc -fPIC -shared -o libmatrix_multiply.so matrix_multiply.c
chenih@Chenih5:~/lab1$ mpicc -o mpi_program mpi_program.c -L. -lmatrix_multiply
chenih@Chenih5:~/lab1$ mpirun -np 4 ./mpi_program
Enter number of rows and columns: 100 100 100

Matrix A (first 2x2 with ellipsis):
1.00  7.00 ...
8.00  0.00 ...
...

Matrix B (first 2x2 with ellipsis):
0.00  2.00 ...
2.00  8.00 ...
...

Matrix C (first 2x2 with ellipsis):
1977.00 2433.00 ...
1945.00 2480.00 ...
...

Execution time: 0.003406 seconds
chenih@Chenih5:~/lab1$
```

运行程序，可以成功执行。

(4). 构造 MPI 版本矩阵乘法加速比和并行效率表

点对点通信加速比：

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	1	1	1	1	1
2	0.883	0.964	1.141	0.986	1.018

4	3.625	11.866	34.471	33.796	52.971
8	3.021	11.542	80.816	103.521	288.972
16	1.405	7.565	51.965	70.204	296.384

点对点通信并行效率：

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	1	1	1	1	1
2	0.442	0.482	0.571	0.493	0.509
4	0.906	2.967	8.618	8.449	13.243
8	0.378	1.443	10.102	12.94	36.122
16	0.088	0.473	3.248	4.388	18.524

集合通信加速比：

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	1	1	1	1	1
2	1.728	2.017	1.947	1.709	4.577
4	2.335	3.702	3.674	2.851	3.672
8	3.406	5.97	5.359	3.76	6.282
16	3.01	5.508	5.009	3.692	8.132

集合通信并行效率：

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	1	1	1	1	1
2	0.864	1.009	0.974	0.855	2.289
4	0.584	0.926	0.919	0.713	0.918

8	0.426	0.746	0.66	0.47	0.785
16	0.188	0.344	0.313	0.231	0.508

分类讨论两种矩阵乘法分别在强扩展和弱扩展情况下的扩展性：

强扩展是指在保持问题规模不变的前提下，增加进程数量，观察程序性能的提升情况。弱扩展是指在保持每个进程处理相同工作量的前提下，随着进程数量的增加，相应地增大问题规模。

点对点通信在强扩展下的表现较差，主要由于通信开销随着进程数的增加而显著上升；而在弱扩展下，点对点通信的扩展性较好，每个进程处理的工作量保持不变，通信频率的增长是相对线性的，通信开销相对可控。集合通信在强扩展情况下的扩展性优于点对点通信，集合通信利用 **MPI** 中的诸如 **MPI_Bcast**、**MPI_Reduce** 等操作，允许多个进程同时参与通信，能够更有效地减少通信开销；在弱扩展情况下，集合通信的扩展性依然较好，集合通信可以通过一次高效的通信操作完成数据的分发和归约，保持较高的效率，但需要考虑硬件和通信带宽等因素对超大规模问题的影响。

4. 实验感想

在这次实验中，我通过 **MPI** 实现了并行矩阵乘法，并进一步探索了点对点通信和集合通信在不同扩展性情况下的表现。实验中，通过将大规模矩阵分块并分发给多个进程处理，实现了矩阵乘法的并行计算，大大提升了运算效率。在第三个任务中，我将实验中的矩阵乘法封装为库函数，提高了代码的复用性和可维护性，让我获益良多。通过实验数据，我分析了 **MPI** 矩阵乘法的加速比和并行效率，探讨了两种矩阵乘法的扩展性，我进一步认识到通信开销对并行计算性能的影响。通过这次实验，我更深入理解了并行计算中的通信开销对性能的影响，也认识到合理选择进程数和通信方式对于优化并行计算至关重要。