中山大学计算机院本科生实验报告

（2023 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	21311359	姓名	何凯迪
Email	hekd@mail2.sysu.edu.cn	完成日期	2023/9/11

1. 实验目的

（1）通过 MPI 实现通用矩阵乘法

（2）基于 MPI 的通用矩阵乘法优化

（3）改造 Lab1 成矩阵乘法库函数

（4）构造 MPI 版本矩阵乘法加速比和并行效率表

2. 实验过程和核心代码

矩阵乘法：

```
void matrixMultiply(double A[], double B[], double C[], int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C[i*size + j] = 0.0;
            for (int k = 0; k < size; k++) {
                C[i*size + j] += A[i*size + k] * B[k*size + j];
            }
        }
    }
}
```

（1）通过 MPI 实现通用矩阵乘法（点对点通信）

初始化 MPI 环境

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

生成随机矩阵

```
void initializeMatrix(double matrix[], int size, unsigned int seed) {
    mt19937 rng(seed); // 使用Mersenne Twister 19937伪随机数生成器
    uniform_real_distribution<double> dist(0.0, 1.0);

    for (int i = 0; i < size; i++) {
        matrix[i] = dist(rng); // 生成随机数填充矩阵
    }
}
```

将矩阵 B 全部发送，将矩阵 A 分块发送。

```
// 广播矩阵B给所有进程
MPI_Bcast(B, N * N, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);

if (size > 1) {
        if (rank == MASTER) {
                // 将A的各行分发给其他进程
                for (int dest = 1; dest < size; dest++) {
                        int startIndex = (dest-1) * localSize;

                        MPI_Send(&A[startIndex], localSize * N, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);

                }
```

其它进程接收主进程发送的矩阵 A 部分

```
 } else {
     // 接收分配给本地进程的A的子矩阵
     MPI_Recv(A, localSize * N, MPI_DOUBLE, MASTER, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
     // 矩阵乘法
     matrixMultiply(A, B, C, localSize);
 }
```

将计算结果发回主进程，并清空内存

```
else {
  // 发送本地计算的结果给主进程
  MPI_Send(C, localSize * N, MPI_DOUBLE, MASTER, 0, MPI_COMM_WORLD);

  // 清理内存
  delete[] A;
  delete[] B;
  delete[] C;
```

主进程接收计算结果并输出时间

```
if (rank == MASTER) {
    // 收集各进程的结果
    for (int source = 1; source < size; source++) {
        int startIndex = source * localSize;
        MPI_Recv(&C[startIndex * N], localSize * N, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    endTime = MPI_Wtime();
    cout << "矩阵乘法执行时间: " << (endTime - startTime)*1000 << " ms" << endl;
```

打印矩阵（由于过于庞大，故只打印小部分行列，其它由省略号替代）

```
void print_matrix(double mat[], int N, int outputSize){
        for (int i = 0; i < outputSize; i++) {
            for (int j = 0; j < outputSize; j++) {
                cout << mat[i * N + j] << " ";
            }
            cout << " ... ";
            for (int j = N - outputSize; j < N; j++) {
                cout << mat[i * N + j] << " ";
            }
            cout << endl;
        }
        cout << " ... " << endl;
        for (int i = N - outputSize; i < N; i++) {
            for (int j = 0; j < outputSize; j++) {
                cout << mat[i * N + j] << " ";
            }
            cout << " ... ";
            for (int j = N - outputSize; j < N; j++) {
                cout << mat[i * N + j] << " ";
            }
            cout << endl;
        }
}
```

（2）基于 MPI 的通用矩阵乘法优化

初始化 MPI 环境

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

生成随机矩阵

```
void initializeMatrix(double matrix[], int size, unsigned int seed) {
    mt19937 rng(seed); // 使用Mersenne Twister 19937伪随机数生成器
    uniform_real_distribution<double> dist(0.0, 1.0);

    for (int i = 0; i < size; i++) {
        matrix[i] = dist(rng); // 生成随机数填充矩阵
    }
}
```

开始计时。将矩阵 A 分块发送，将矩阵 B 完整发送，在各个进程中做矩阵乘法，再由主进程聚集接收。结束计时。

```
double startTime = MPI_Wtime(); // 记录开始时间

// 分发数据块到各进程
MPI_Scatter(A, blockSize * matrixSize, MPI_DOUBLE, localA, blockSize * matrixSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(B, matrixSize * matrixSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// 进行局部矩阵乘法计算
matrixMultiply(localA, B, localC, matrixSize, blockSize);

// 收集结果到根进程
MPI_Gather(localC, blockSize * matrixSize, MPI_DOUBLE, C, blockSize * matrixSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

double endTime = MPI_Wtime(); // 记录结束时间
```

在主进程打印矩阵，输出运行时间，释放内存，结束 mpi。

```
// 打印结果
if (rank == 0) {

    if(pr == 1){
    std::cout << "Matrix A:" << std::endl;
    printMatrix(A, matrixSize, printRows, printCols);

    std::cout << "Matrix B:" << std::endl;
    printMatrix(B, matrixSize, printRows, printCols);

    std::cout << "Matrix C:" << std::endl;
    printMatrix(C, matrixSize, printRows, printCols);
    }

    std::cout << "Execution time: " << (endTime - startTime)*1000 << " ms" << std::endl;

}

delete[] A;
delete[] B;
delete[] C;
delete[] localA;
delete[] localC;

MPI_Finalize();
```

尝试用 mpi_type_create_struct 聚合 MPI 进程内变量后通信。

```
struct MatrixBlock {
    int rows;
    int cols;
    double data[1]; // 实际上是一个长度为1的动态数组，用于存储矩阵数据
};
```

```
// 创建自定义数据类型
MPI_Datatype matrixBlockType;
int blockLengths[2] = {1, blockSize * matrixSize}; // 两个字段，第一个是 rows，第二个是 data
MPI_Aint displacements[2] = {offsetof(MatrixBlock, rows), offsetof(MatrixBlock, data)};
MPI_Datatype types[2] = {MPI_INT, MPI_DOUBLE};
MPI_Type_create_struct(2, blockLengths, displacements, types, &matrixBlockType);
MPI_Type_commit(&matrixBlockType);
```

（3）改造 Lab1 成矩阵乘法库函数

创建三个文件如下：

Matrix_multiply.h

定义了一个名为 Matrix 的结构体，用来表示矩阵。声明各函数。

```
#ifndef MATRIX_MULTIPLY_H
#define MATRIX_MULTIPLY_H

typedef struct {
    int rows;
    int cols;
    float **data;
} Matrix;

Matrix createRandomMatrix(int rows, int cols);
Matrix matrixMultiply(const Matrix *A, const Matrix *B);
Matrix createMatrix(int rows, int cols);
void printMatrix(const Matrix* mat, int maxRows, int maxCols);
void freeMatrix(Matrix *mat);

#endif
```

Matrix_multiply.c

生成随机矩阵。

```
Matrix createRandomMatrix(int rows, int cols) {
        Matrix mat;
        mat.rows = rows;
        mat.cols = cols;
        mat.data = (float**)malloc(rows * sizeof(float*));

        for (int i = 0; i < rows; i++) {
                mat.data[i] = (float*)malloc(cols * sizeof(float));
                for (int j = 0; j < cols; j++) {
                        mat.data[i][j] = (float)rand() / RAND_MAX; // 生成0到1之间的随机浮点数
                }
        }

        return mat;
}
```

矩阵乘法。

```
Matrix matrixMultiply(const Matrix* A, const Matrix* B) {
        if (A->cols != B->rows) {
                fprintf(stderr, "Matrix dimensions are incompatible for multiplication\n");
                exit(1);
        }

        Matrix C;
        C.rows = A->rows;
        C.cols = B->cols;
        C.data = (float**)malloc(C.rows * sizeof(float*));

        for (int i = 0; i < C.rows; i++) {
                C.data[i] = (float*)calloc(C.cols, sizeof(float));
                for (int k = 0; k < A->cols; k++) {
                        for (int j = 0; j < C.cols; j++) {
                                C.data[i][j] += A->data[i][k] * B->data[k][j];
                        }
                }
        }

        return C;
}
```

打印矩阵。

```
void printMatrix(const Matrix* mat, int maxRows, int maxCols) {
        int numRows = mat->rows;
        int numCols = mat->cols;

        for (int i = 0; i < numRows && i < maxRows; i++) {
                for (int j = 0; j < numCols && j < maxCols; j++) {
                        printf("%f ", mat->data[i][j]);
                }

                if (numCols > maxCols) {
                        printf("...");
                }

                printf("\n");
        }

        if (numRows > maxRows) {
                printf("...\n");
        }
}
```

释放内存。

```
void freeMatrix(Matrix *mat) {
    for (int i = 0; i < mat->rows; i++) {
        free(mat->data[i]);
    }
    free(mat->data);
}
```

example.c

接收命令行输入的矩阵阶数。

```
int rows_A = atoi(argv[1]);
int cols_A = atoi(argv[2]);
int cols_B = atoi(argv[3]);
```

创建随机矩阵 A、B 并打印

```
// 创建矩阵 A 和 B
Matrix A = createRandomMatrix(rows_A, cols_A);
Matrix B = createRandomMatrix(cols_A, cols_B);

// 打印矩阵 A 和 B
printf("Matrix A:\n");
printMatrix(&A, 2, 2);

printf("Matrix B:\n");
printMatrix(&B, 2, 2);
```

开始计时，计算矩阵 C，结束计时，打印矩阵 C，释放内存，打印运行时间。

```
// 记录开始时间
clock_t start_time = clock();

// 执行矩阵乘法
Matrix C = matrixMultiply(&A, &B);

// 记录结束时间
clock_t end_time = clock();

// 打印矩阵 C
printf("Matrix C (Result of A * B):\n");
printMatrix(&C, 2, 2);

// 释放矩阵内存
freeMatrix(&A);
freeMatrix(&B);
freeMatrix(&C);

// 计算并打印运行时间（以秒为单位）
double execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
printf("Execution time: %f seconds\n", execution_time);
```

编译共享库文件 libmatrix_multiply.so:

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ gcc -shared -o libmatrix_multiply.so matrix_
multiply.c -fPIC
```

得到.so 文件:

编译示例程序 matrix_example，并链接共享库:

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ gcc -o example example.c -L. -lmatrix_multip
ly -Wl,-rpath,.
```

运行示例程序:

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ ./example 128 128 128
```

（4）构造 MPI 版本矩阵乘法加速比和并行效率表

点对点通信耗时表：

ms	128	256	512	1024	2048
1	10.9134	89.5312	820.043	7819.95	56870.9
2	10.9885	89.1781	790.544	7577.68	58277.3
4	0.891447	3.84641	27.6093	217.141	1813.53
8	0.462294	1.27578	6.56438	26.664	182.961
16	57.9724	120.557	549.413	1407.94	4468.92

集合通信耗时表：

ms	128	256	512	1024	2048
1	10.6483	88.0289	782.022	7550.34	58492.5
2	6.09255	47.0726	385.895	3880.22	29368.9
4	3.06177	22.4988	192.472	1963.3	16764.1
8	2.41971	16.7313	114.515	1049.48	9721.05
16	108.474	204.259	1024.62	1776.91	14208.2

通过额外的实验，结果表明产生了一定的通信开销。（代码如下）

```
int main(int argc, char** argv) {
    int N = 512; // N阶矩阵的大小
    int num_procs, my_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // 根进程负责生成随机矩阵
    vector<vector<double>> matrix(N, vector<double>(N));
    if (my_rank == 0) {
        srand(time(nullptr));
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                matrix[i][j] = rand() / (double)RAND_MAX;
            }
        }
    }

    // 开始计时
    double start_time = MPI_Wtime();

    // 广播矩阵给所有进程
    for (int i = 0; i < N; i++) {
        MPI_Bcast(matrix[i].data(), N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }

    // 停止计时
    double end_time = MPI_Wtime();

    // 计算总通信时间
    double total_comm_time;
    MPI_Reduce(&end_time, &total_comm_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    if (my_rank == 0) {
        cout << "Total communication time: " << total_comm_time - start_time << " seconds." << endl;
    }

    // 汇总所有部分矩阵到根进程
    vector<vector<double>> result(N, vector<double>(N));
    for (int i = 0; i < N; i++) {
        MPI_Gather(matrix[i].data(), N, MPI_DOUBLE, result[i].data(), N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
```

3. 实验结果

（1）通过 MPI 实现通用矩阵乘法（点对点通信）

打印矩阵：（避免冗余故仅以 128 阶矩阵展示一次矩阵打印结果）

```
kaddy@kaddy-VirtualBox:~/HPC$ mpic++ test.cpp -o test
kaddy@kaddy-VirtualBox:~/HPC$ mpirun -np 1 ./test
矩阵乘法执行时间: 10.9615 ms
矩阵A:
0.890155 0.130707  ... 0.765111 0.579899
0.378123 0.622393  ... 0.295178 0.997285
 ...
0.924782 0.405923  ... 0.887176 0.00911665
0.407753 0.0901721  ... 0.895029 0.137411
矩阵B:
0.509721 0.183354  ... 0.0936705 0.598044
0.444104 0.955429  ... 0.483161 0.202417
 ...
0.753194 0.305282  ... 0.668279 0.624943
0.551577 0.901771  ... 0.90597 0.859044
矩阵C:
33.5702 32.9351  ... 33.8723 34.4393
29.3279 32.9851  ... 32.037 32.2694
 ...
29.4444 30.7737  ... 29.8258 31.4652
30.3666 30.26  ... 31.3203 32.4697
```

128 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test.cpp -o test
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test
矩阵乘法执行时间: 10.9134 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test
矩阵乘法执行时间: 10.9885 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test
矩阵乘法执行时间: 0.891447 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test
矩阵乘法执行时间: 0.462294 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test
矩阵乘法执行时间: 57.9724 ms
```

256 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test.cpp -o test
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test
矩阵乘法执行时间: 89.5312 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test
矩阵乘法执行时间: 89.1781 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test
矩阵乘法执行时间: 3.84641 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test
矩阵乘法执行时间: 1.27578 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test
矩阵乘法执行时间: 120.557 ms
```

512 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test.cpp -o test
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test
矩阵乘法执行时间: 820.043 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test
矩阵乘法执行时间: 790.544 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test
矩阵乘法执行时间: 27.6093 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test
矩阵乘法执行时间: 6.56438 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test
矩阵乘法执行时间: 549.413 ms
```

1024 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test.cpp -o test
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test
矩阵乘法执行时间: 7819.95 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test
矩阵乘法执行时间: 7577.68 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test
矩阵乘法执行时间: 217.141 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test
矩阵乘法执行时间: 26.664 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test
矩阵乘法执行时间: 1407.94 ms
```

2048 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test.cpp -o test
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test
矩阵乘法执行时间: 56870.9 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test
矩阵乘法执行时间: 58277.3 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test
矩阵乘法执行时间: 1813.53 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test
矩阵乘法执行时间: 182.961 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test
矩阵乘法执行时间: 4468.92 ms
```

（2）基于 MPI 的通用矩阵乘法优化（集合通信）

打印矩阵：（避免冗余故仅以 128 阶矩阵展示一次矩阵打印结果）

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test1.cpp -o test1
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -np 1 ./test1
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: 11.0896 ms
```

128 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test1.cpp -o test1
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test1
Execution time: 10.6483 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test1
Execution time: 6.09255 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test1
Execution time: 3.06177 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test1
Execution time: 2.41971 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test1
Execution time: 108.474 ms
```

256 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test1.cpp -o test1
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test1
Execution time: 88.0289 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test1
Execution time: 47.0726 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test1
Execution time: 22.4988 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test1
Execution time: 16.7313 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test1
Execution time: 204.259 ms
```

512 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test1.cpp -o test1
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test1
Execution time: 782.022 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test1
Execution time: 385.895 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test1
Execution time: 192.472 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test1
Execution time: 114.515 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test1
Execution time: 1024.62 ms
```

1024 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test1.cpp -o test1
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test1
Execution time: 7550.34 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test1
Execution time: 3880.22 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test1
Execution time: 1963.3 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test1
Execution time: 1049.48 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test1
Execution time: 1776.91 ms
```

2048 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpic++ test1.cpp -o test1
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 1 ./test1
Execution time: 58492.5 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 2 ./test1
Execution time: 29368.9 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 4 ./test1
Execution time: 16764.1 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 8 ./test1
Execution time: 9721.05 ms
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ mpirun -n 16 ./test1
Execution time: 14208.2 ms
```

（3）改造 Lab1 成矩阵乘法库函数

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ ./example 128 128 128
Matrix A:
0.840188 0.394383 ...
0.749771 0.368664 ...
...
Matrix B:
0.088661 0.719976 ...
0.546715 0.569322 ...
...
Matrix C (Result of A * B):
34.042606 34.583607 ...
30.390951 33.131092 ...
...
Execution time: 0.007107 seconds
```

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ ./example 256 256 256
Matrix A:
0.840188 0.394383 ...
0.730729 0.328374 ...
...
Matrix B:
0.775621 0.507780 ...
0.117806 0.580322 ...
...
Matrix C (Result of A * B):
65.003365 63.487816 ...
62.481018 60.882561 ...
...
Execution time: 0.056603 seconds
```

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ ./example 512 512 512
Matrix A:
0.840188 0.394383 ...
0.551443 0.933420 ...
...
Matrix B:
0.262807 0.319440 ...
0.897824 0.659894 ...
...
Matrix C (Result of A * B):
135.775009 128.181335 ...
132.034348 126.801277 ...
...
Execution time: 0.457786 seconds
```

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ ./example 1024 1024 1024
Matrix A:
0.840188 0.394383 ...
0.180136 0.359247 ...
...
Matrix B:
0.384496 0.447241 ...
0.628011 0.321799 ...
...
Matrix C (Result of A * B):
264.159363 257.258392 ...
256.461914 251.636154 ...
...
Execution time: 3.666153 seconds
```

```
kaddy@kaddy-VirtualBox:~/HPC/exp_1$ ./example 2048 2048 2048
Matrix A:
0.840188 0.394383 ...
0.130165 0.018988 ...
...
Matrix B:
0.802008 0.288009 ...
0.826367 0.722746 ...
...
Matrix C (Result of A * B):
510.334869 520.425842 ...
508.902740 516.492981 ...
...
Execution time: 29.202531 seconds
```

（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	0.993165582	1.003959492	1.037314811	1.031971527	0.975867104
4	12.24234307	23.27656178	29.70169472	36.01323564	31.35922758
8	23.60705525	70.17761683	124.9231458	293.2774527	310.8361891
16	0.188251651	0.742646217	1.492580263	5.554178445	12.72587113

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	0.496582791	0.501979746	0.518657406	0.515985763	0.487933552
4	3.060585767	5.819140445	7.42542368	9.00330891	7.839806896
8	2.950881906	8.772202104	15.61539323	36.65968159	38.85452364
16	0.011765728	0.046415389	0.093286266	0.347136153	0.795366945

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	1.747757507	1.870066663	2.026514985	1.945853586	1.991647627
4	3.477824918	3.912604228	4.063042936	3.845739316	3.489152415
8	4.400651318	5.26133056	6.828991835	7.194362923	6.017096919
16	0.098164537	0.430967057	0.763231247	4.249140362	4.116812826

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	0.873878754	0.935033331	1.013257492	0.972926793	0.995823814
4	0.86945623	0.978151057	1.015760734	0.961434829	0.872288104
8	0.550081415	0.65766632	0.853623979	0.899295365	0.752137115
16	0.006135284	0.026935441	0.047701953	0.265571273	0.257300802

讨论点对点通信矩阵乘法和集合通信矩阵乘法分别在强扩展和弱扩展情况下的扩展性：

点对点通信矩阵乘法：

强扩展性：在强扩展情况下，点对点通信矩阵乘法的性能通常会表现得较好。这是因为每个处理单元（例如，CPU 或 GPU）都负责执行矩阵乘法中的一部分工作，并且通信需求较小。随着处理单元的数量增加，总的计算能力也会增加，因此矩阵乘法的执行速度会线性提高。

弱扩展性：在弱扩展情况下，点对点通信矩阵乘法可能会遇到性能瓶颈。这是因为虽然处理单元的数量增加，但每个处理单元的工作量也随之增加。如果通信开销相对较大，那么随着处理单元数量的增加，通信时间可能会占据总执行时间的较大部分，从而限制了弱扩展性。

集合通信矩阵乘法：

强扩展性：集合通信矩阵乘法通常在强扩展情况下表现良好。这是因为在集合通信模型中，处理单元可以独立地执行计算任务，然后在需要时将结果进行集合（聚合）。每个处理单元之间的通信要求较低，因为它们可以并行工作，因此强扩展性较好。

弱扩展性：在弱扩展情况下，集合通信矩阵乘法也可以获得较好的性能，因为每个处理单元的工作负载相对较小，通信开销相对较低。处理单元数量的增加会导致计算速度线性提高，因此在这种情况下扩展性也较好。

4. 实验感想

//可以写写过程中遇到的问题，你是怎么解决的。以及可以写你对此次实验的一些理解……
本次实验耗时很长，debug 占了很大一部分。最开始对 MPI 不熟练，同时装了 openmpi 和 mpich，结果编译运行后出现了各种错误，后来才发现问题，然后 remove 了 mpich，但是 openmpi 环境仍然使我在明确写明 if(rank==0)的情况下，还是在 N 进程跑出 N 个结果，最后选择了 mpich 才使输出结果正常。最初尝试点对点通信中，我令根进程只负责生成矩阵、分发数据以及收集数据和结果输出，结果导致并行结果极差，所以我改为根进程也参与运算，结果稍好。但由结果不难看出，当规模较小时，在我电脑上并行的效果并不太好，即使增大规模后也只稍优了一点点。在集合通信中这个问题被更加放大了。大概是我的电脑太老旧了硬件，设施没跟上，故使得并行效果不佳。
总的来说，本次实验收获颇丰。

距离上次写完感想提交已经过了两三天，我终于知道为什么并行效率如此之差了。虚拟机默认设置一直只分到了一个核，所以根本并行不起来！重新设置分给它 8 核后一切都正常了...