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	**MPI 通信编程**	专业（方向）	**信息与计算科学**
学号	**22336313**	姓名	**郑鸿鑫**
Email	**Zhenghx57@mail2.sysu.edu.cn**	完成日期	**2024/9/30**

1. 实验目的

通过实现和优化使用 MPI 进行通用矩阵乘法的程序来深入理解并行计算的基本概念和 MPI 库的应用。我会探索点对点通信和集合通信的不同策略，并通过构建加速比和并行效率表来分析程序的性能，从而学习如何提升并行程序的扩展性和效率。此外，我将练习将算法封装成库函数，并在 Linux 系统中编译和运行 MPI 程序，这将帮助我提高编程的能力。

2. 实验过程和核心代码

a. 点对点通信代码：

```
    if (rank == 0) {
        A = build_Matrix(m, n); // 根进程构建完整的矩阵 A
        C = build_Matrix(m, k); // 根进程构建完整的矩阵 C

        // 使用随机浮点值填充矩阵 A 和 B
        fill_Matrix(m, n, A, (float)(rand() % 100));
        fill_Matrix(n, k, B, (float)(rand() % 100));

        // 根进程将矩阵 B 发送给其他所有进程
        for (int i = 1; i < size; i++) {
            MPI_Send(B, n * k, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
        }
    } else {
        // 非根进程接收矩阵 B
        MPI_Recv(B, n * k, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    // 使用点对点通信分发矩阵 A
    if (rank == 0) {
        for (int i = 1; i < size; i++) {
            MPI_Send(A + i * rows_per_proc * n, rows_per_proc * n, MPI_FLOAT, i, 0,
MPI_COMM_WORLD);
        }
        // 根进程保留矩阵 A 的自己的部分
        for (int i = 0; i < rows_per_proc * n; i++) {
            sub_A[i] = A[i];
        }
    } else {
        // 非根进程接收矩阵 A 的一部分
        MPI_Recv(sub_A, rows_per_proc * n, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    }

    // 同步所有进程，然后开始计时
    MPI_Barrier(MPI_COMM_WORLD);
    double start_time = MPI_Wtime();

    // 执行矩阵乘法
    matrix_multiply(sub_A, B, sub_C, rows_per_proc, n, k);

    // 同步所有进程，然后结束计时
    MPI_Barrier(MPI_COMM_WORLD);
    double end_time = MPI_Wtime();
    double local_time_spent = end_time - start_time;

    // 计算所有进程中花费时间最长的一个
    double total_time_spent;
    MPI_Reduce(&local_time_spent, &total_time_spent, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);

    // 使用点对点通信收集结果到矩阵 C
    if (rank == 0) {
        for (int i = 0; i < rows_per_proc * k; i++) {
            C[i] = sub_C[i];
        }
        for (int i = 1; i < size; i++) {
            MPI_Recv(C + i * rows_per_proc * k, rows_per_proc * k, MPI_FLOAT, i, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    } else {
        MPI_Send(sub_C, rows_per_proc * k, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
    }
```

b. 集合通信代码:

```
    if (rank == 0) {
        A = build_Matrix(m, n); // 根进程构建完整的矩阵 A
        C = build_Matrix(m, k); // 根进程构建完整的矩阵 C

        // 使用随机浮点值填充矩阵 A 和 B
        fill_Matrix(m, n, A, (float)(rand() % 100));
        fill_Matrix(n, k, B, (float)(rand() % 100));
    }

    // 广播矩阵 B
    MPI_Bcast(B, n * k, MPI_FLOAT, 0, MPI_COMM_WORLD);

    // 分发矩阵 A
    MPI_Scatter(A, rows_per_proc * n, MPI_FLOAT, sub_A, rows_per_proc * n, MPI_FLOAT,
0, MPI_COMM_WORLD);

    // 同步所有进程，然后开始计时
    MPI_Barrier(MPI_COMM_WORLD);
    double start_time = MPI_Wtime();

    // 执行矩阵乘法
    matrix_multiply(sub_A, B, sub_C, rows_per_proc, n, k);
    // 同步所有进程，然后结束计时
    MPI_Barrier(MPI_COMM_WORLD);
    double end_time = MPI_Wtime();
    double local_time_spent = end_time - start_time;

    // 计算所有进程中花费时间最长的一个
    double total_time_spent;
    MPI_Reduce(&local_time_spent, &total_time_spent, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);

    // 收集矩阵 C
    MPI_Gather(sub_C, rows_per_proc * k, MPI_FLOAT, C, rows_per_proc * k, MPI_FLOAT,
0, MPI_COMM_WORLD);
```

说明：

在使用 mpi 实现矩阵乘法时，大部分代码都是复用 Lab1 中的矩阵乘法，如矩阵的定义，填充随机数还有矩阵的乘法，故此处仅展示关键代码，分别展示了点对点通信和集合通信的不同代码段，而不是完整的代码，完整的代码见 Code 文件夹。

c. 利用 mpi_TYPE_create_struct 聚合进程内变量后通信：

先定义一个结构体：

```
typedef struct {
    float* A;
    float* B;
    float* C;
    int m;
    int n;
    int k;
} MatrixData;
```

再接着用 MPI_TYPE_create_struct 来创建自定义结构体

```
MatrixData md;
md.m = m;
md.n = n;
md.k = k;

md.A = build_Matrix(m, n);
md.B = build_Matrix(n, k);
md.C = build_Matrix(m, k);

float* sub_A = build_Matrix(rows_per_proc, n);
float* sub_C = build_Matrix(rows_per_proc, k);

if (rank == 0) {
    fill_Matrix(m, n, md.A, (float)(rand() % 100));
    fill_Matrix(n, k, md.B, (float)(rand() % 100));
}

// 创建 MPI 数据类型
MPI_Datatype MatrixDataType;
```

```c
int blocklengths[3] = {1, 1, 1};
MPI_Aint offsets[3];
offsets[0] = offsetof(MatrixData, A);
offsets[1] = offsetof(MatrixData, B);
offsets[2] = offsetof(MatrixData, C);
MPI_Datatype types[3] = {MPI_FLOAT, MPI_FLOAT, MPI_FLOAT};

MPI_Type_create_struct(3, blocklengths, offsets, types, &MatrixDataType);
MPI_Type_commit(&MatrixDataType);
```

遇到的问题：

由于结构体中传递的是指向矩阵对应的第一个元素的首地址的指针，而在进程之间传递指针并不能真正的传递矩阵的数据，因为在不同的进程中都有自己的地址空间，传递指针在其他进程中没有任何意义。所以不能用 MPI_TYPE_create_struct 来聚合变量传递。

d. 将 Lab1 中矩阵乘法代码改造为库函数：

首先建立 matrix_multiply.h 和 matrix_multiply.c 函数分别给出函数的声明与具体实现：

头文件：

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#ifndef LAB2_MATRIX_MULTIPLY_H
#define LAB2_MATRIX_MULTIPLY_H
float* build_Matrix(int m, int n);
void fill_Matrix(int m, int n, float* A, float value);
void matrix_multiply(float* A, float* B, float* C, int m, int n, int k);
#endif //LAB2_MATRIX_MULTIPLY_H
```

具体实现：

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "matrix_multiply.h"
// 构建一个 1D 数组，用于表示大小为 m x n 的矩阵
float* build_Matrix(int m, int n) {
    return (float*)malloc(m * n * sizeof(float));
}

// 使用固定值或随机值填充矩阵
void fill_Matrix(int m, int n, float* A, float value) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            A[i * n + j] = value; // 分配固定值或生成随机值
        }
    }
}

// 对以 1D 数组表示的浮点矩阵进行矩阵乘法
```

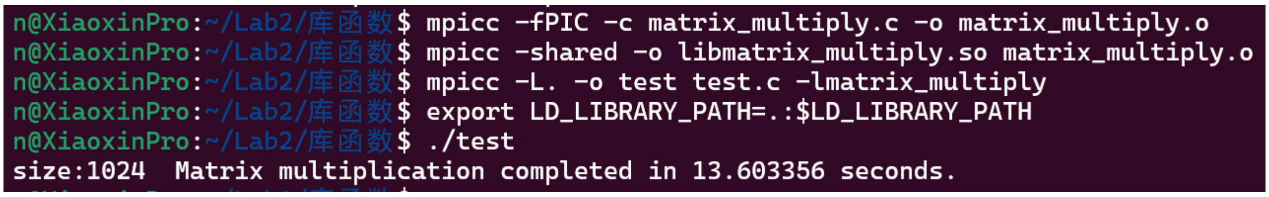
```
void matrix_multiply(float* A, float* B, float* C, int m, int n, int k) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            C[i * k + j] = 0.0f;
            for (int p = 0; p < n; p++) {
                C[i * k + j] += A[i * n + p] * B[p * k + j];
            }
        }
    }
}
```

编写 test.c 测试文件：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "matrix_multiply.h"
#include <mpi.h>
#define SIZE 1024
int main(int argc, char** argv) {
    int rank, size;
    MPI_Init(&argc, &argv); // 初始化 MPI 环境
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // 获取当前进程的排名
    MPI_Comm_size(MPI_COMM_WORLD, &size); // 获取进程总数
    int m,n,k;
    m = n = k = SIZE; // 定义矩阵的行数和列数
    int rows_per_proc = m / size; // 每个进程处理的行数
    int remaining_rows = m % size; // 剩余的行数，用于处理不能均匀分配的情况

    float* A = NULL; // 矩阵 A
    float* B = build_Matrix(n, k); // 矩阵 B
    float* C = NULL; // 结果矩阵 C
    float* sub_A = build_Matrix(rows_per_proc, n); // 子矩阵 A，存储当前进程的矩阵 A 部
分
    float* sub_C = build_Matrix(rows_per_proc, k); // 子矩阵 C，存储当前进程的矩阵 C 部
分

    if (rank == 0) {
        A = build_Matrix(m, n); // 根进程构建完整的矩阵 A
        C = build_Matrix(m, k); // 根进程构建完整的矩阵 C

        // 使用随机浮点值填充矩阵 A 和 B
        fill_Matrix(m, n, A, (float)(rand() % 100));
        fill_Matrix(n, k, B, (float)(rand() % 100));
    }

    // 广播矩阵 B
    MPI_Bcast(B, n * k, MPI_FLOAT, 0, MPI_COMM_WORLD);

    // 分发矩阵 A
    MPI_Scatter(A, rows_per_proc * n, MPI_FLOAT, sub_A, rows_per_proc * n, MPI_FLOAT,
0, MPI_COMM_WORLD);

    // 同步所有进程，然后开始计时
    MPI_Barrier(MPI_COMM_WORLD);
    double start_time = MPI_Wtime();
```

```c
    // 执行矩阵乘法
    matrix_multiply(sub_A, B, sub_C, rows_per_proc, n, k);

    // 同步所有进程，然后结束计时
    MPI_Barrier(MPI_COMM_WORLD);
    double end_time = MPI_Wtime();
    double local_time_spent = end_time - start_time;

    // 计算所有进程中花费时间最长的一个
    double total_time_spent;
    MPI_Reduce(&local_time_spent, &total_time_spent, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);

    // 收集矩阵 C
    MPI_Gather(sub_C, rows_per_proc * k, MPI_FLOAT, C, rows_per_proc * k, MPI_FLOAT,
0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("size:%d  ", m);
        printf("Matrix multiplication completed in %f seconds.\n", total_time_spent);
    }

    // 释放分配的内存
    free(sub_A);
    free(sub_C);
    free(B);
    if (rank == 0) {
        free(A);
        free(C);
    }

    MPI_Finalize(); // 终止 MPI 环境
    return 0;
}
```

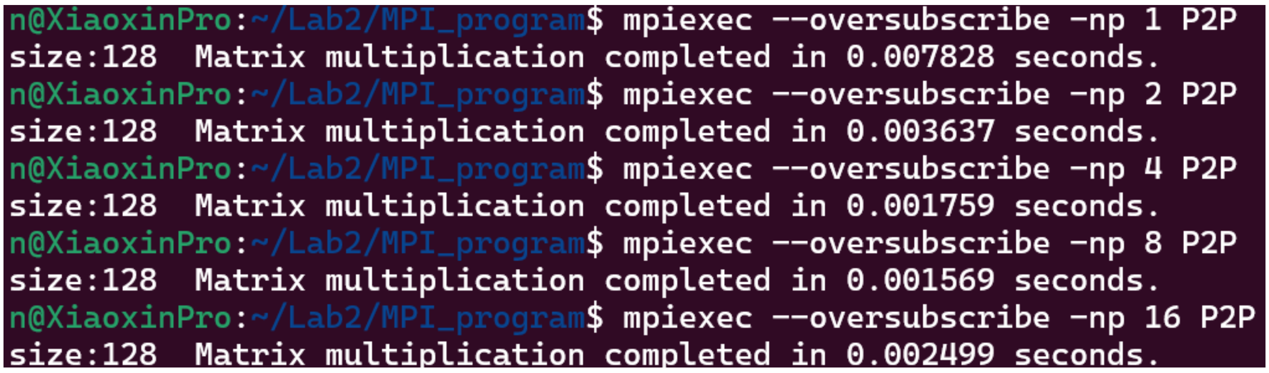
先对 matrix_multiply.c 编译输出.o 文件，然后生成 lib 开头.so 结尾的共享库文件，将库文件在编译时链接到 test.c 文件，然后尝试运行可执行文件，可以成功打印矩阵运算的时间，说明已经成功调用共享库的文件，详细的命令与结果如下图所示：



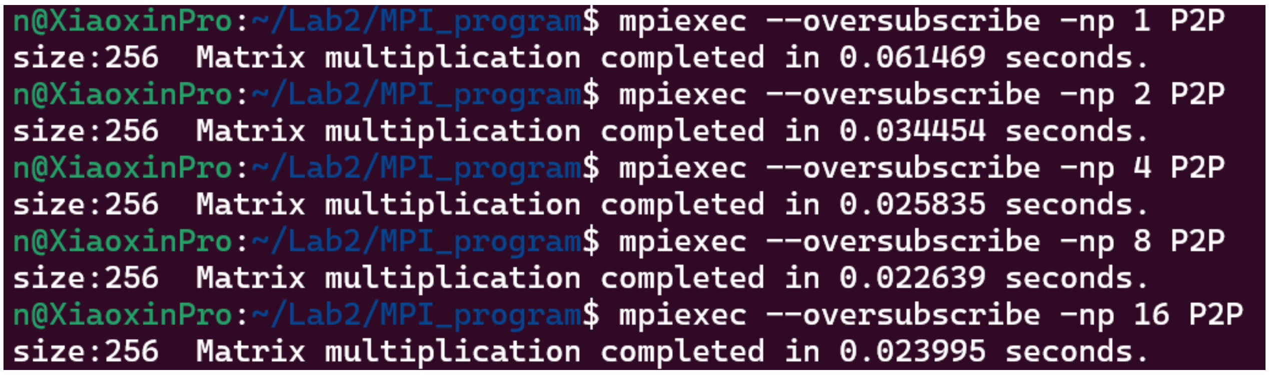
```
n@XiaoxinPro:~/Lab2/库函数$ mpicc -fPIC -c matrix_multiply.c -o matrix_multiply.o
n@XiaoxinPro:~/Lab2/库函数$ mpicc -shared -o libmatrix_multiply.so matrix_multiply.o
n@XiaoxinPro:~/Lab2/库函数$ mpicc -L. -o test test.c -lmatrix_multiply
n@XiaoxinPro:~/Lab2/库函数$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
n@XiaoxinPro:~/Lab2/库函数$ ./test
size:1024  Matrix multiplication completed in 13.603356 seconds.
```

3. 实验结果

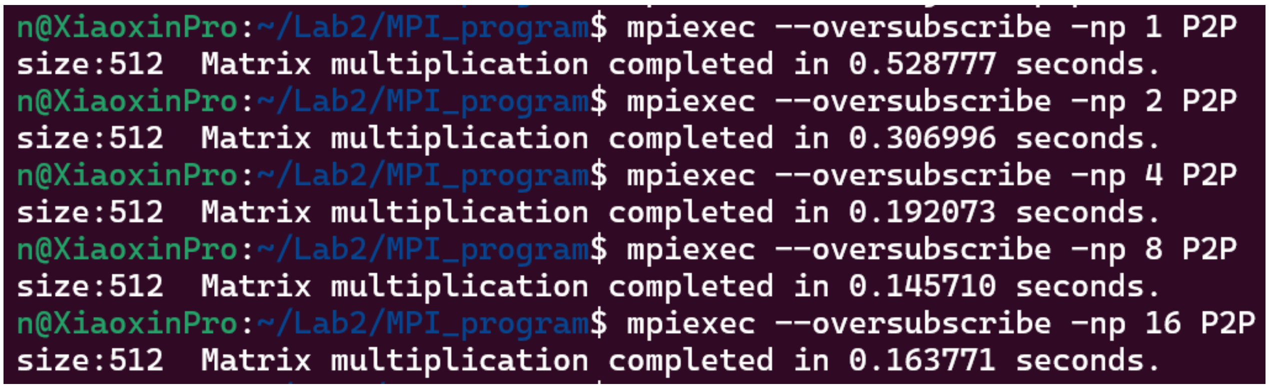
- **点对点通信方式：**



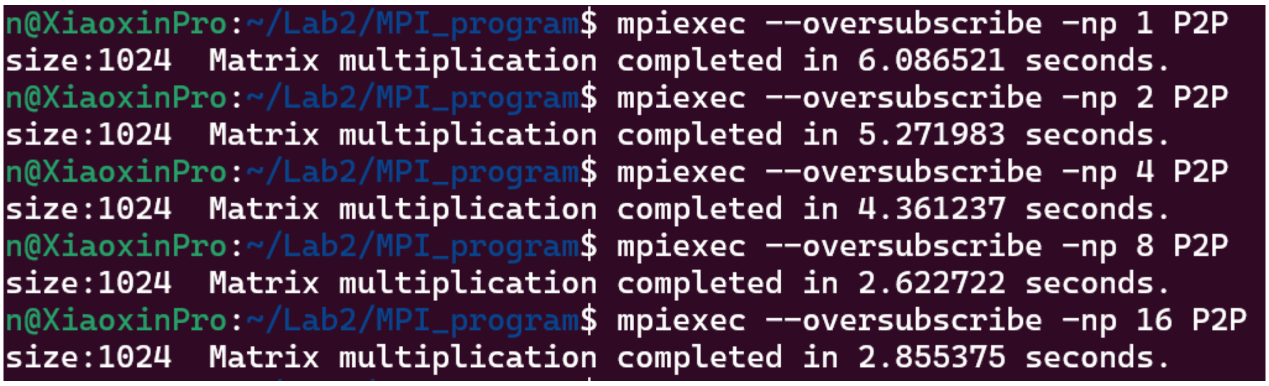
```
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 1 P2P
size:128  Matrix multiplication completed in 0.007828 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 2 P2P
size:128  Matrix multiplication completed in 0.003637 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 4 P2P
size:128  Matrix multiplication completed in 0.001759 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 8 P2P
size:128  Matrix multiplication completed in 0.001569 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 16 P2P
size:128  Matrix multiplication completed in 0.002499 seconds.
```



```
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 1 P2P
size:256  Matrix multiplication completed in 0.061469 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 2 P2P
size:256  Matrix multiplication completed in 0.034454 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 4 P2P
size:256  Matrix multiplication completed in 0.025835 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 8 P2P
size:256  Matrix multiplication completed in 0.022639 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 16 P2P
size:256  Matrix multiplication completed in 0.023995 seconds.
```

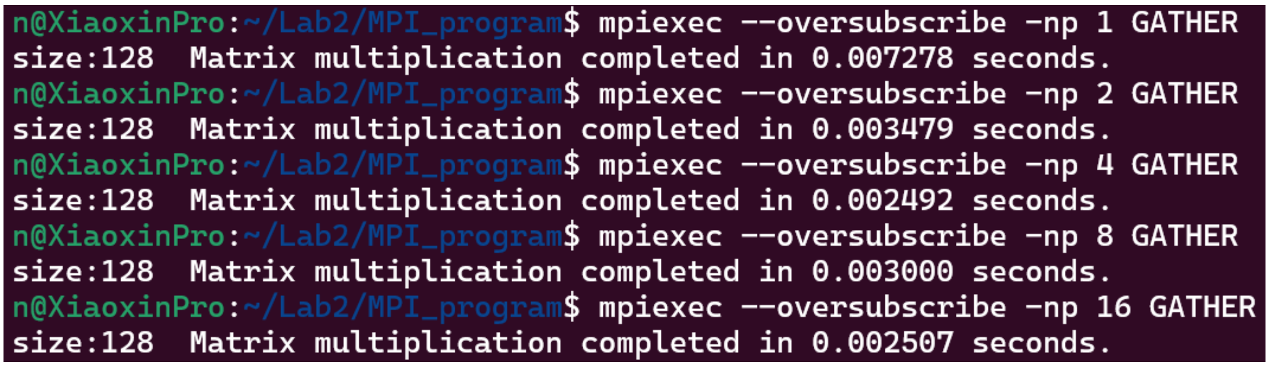


```
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 1 P2P
size:512  Matrix multiplication completed in 0.528777 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 2 P2P
size:512  Matrix multiplication completed in 0.306996 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 4 P2P
size:512  Matrix multiplication completed in 0.192073 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 8 P2P
size:512  Matrix multiplication completed in 0.145710 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 16 P2P
size:512  Matrix multiplication completed in 0.163771 seconds.
```

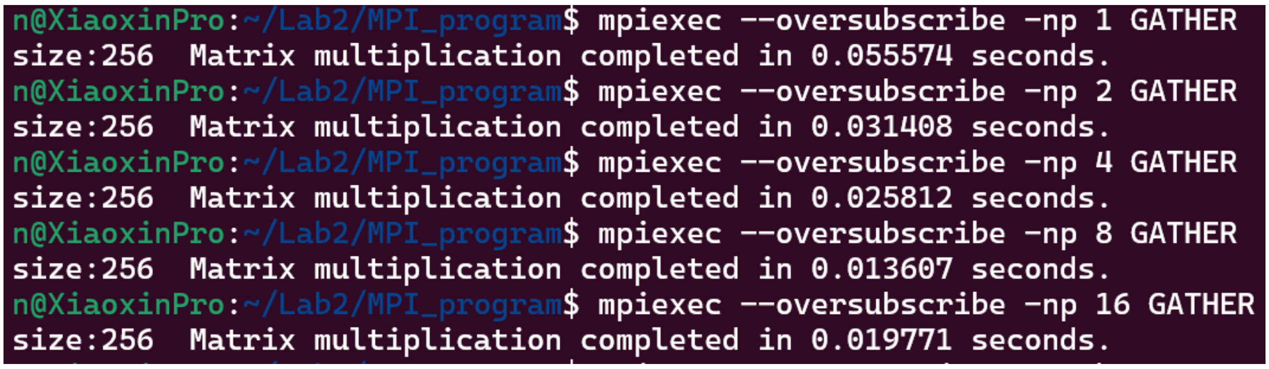


```
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 1 P2P
size:1024  Matrix multiplication completed in 6.086521 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 2 P2P
size:1024  Matrix multiplication completed in 5.271983 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 4 P2P
size:1024  Matrix multiplication completed in 4.361237 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 8 P2P
size:1024  Matrix multiplication completed in 2.622722 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 16 P2P
size:1024  Matrix multiplication completed in 2.855375 seconds.
```

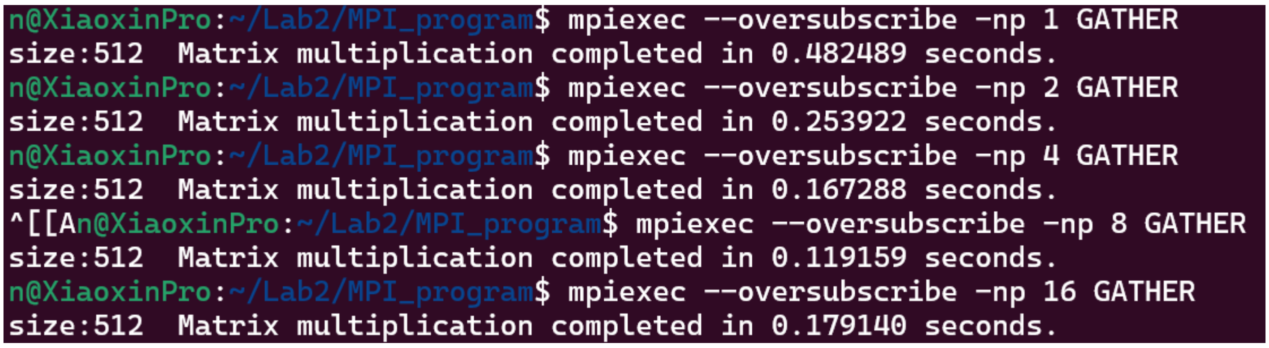
- 集合通信方式:



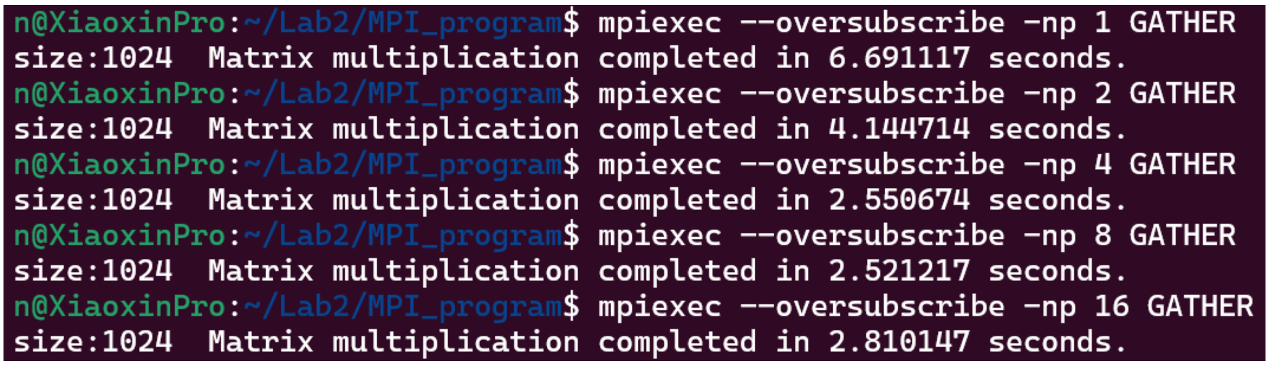
```
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 1 GATHER
size:128  Matrix multiplication completed in 0.007278 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 2 GATHER
size:128  Matrix multiplication completed in 0.003479 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 4 GATHER
size:128  Matrix multiplication completed in 0.002492 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 8 GATHER
size:128  Matrix multiplication completed in 0.003000 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 16 GATHER
size:128  Matrix multiplication completed in 0.002507 seconds.
```



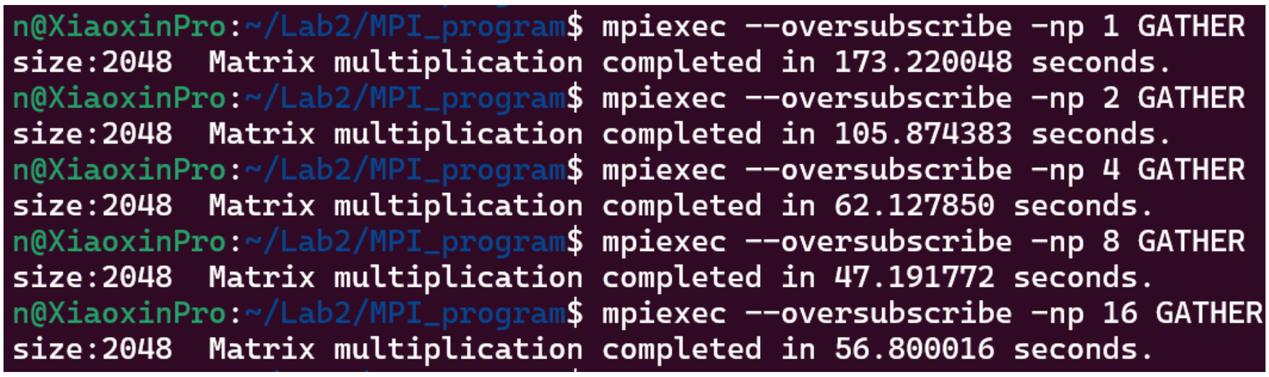
```
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 1 GATHER
size:256  Matrix multiplication completed in 0.055574 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 2 GATHER
size:256  Matrix multiplication completed in 0.031408 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 4 GATHER
size:256  Matrix multiplication completed in 0.025812 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 8 GATHER
size:256  Matrix multiplication completed in 0.013607 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 16 GATHER
size:256  Matrix multiplication completed in 0.019771 seconds.
```



```
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 1 GATHER
size:512  Matrix multiplication completed in 0.482489 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 2 GATHER
size:512  Matrix multiplication completed in 0.253922 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 4 GATHER
size:512  Matrix multiplication completed in 0.167288 seconds.
^[[An@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 8 GATHER
size:512  Matrix multiplication completed in 0.119159 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 16 GATHER
size:512  Matrix multiplication completed in 0.179140 seconds.
```



```
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 1 GATHER
size:1024  Matrix multiplication completed in 6.691117 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 2 GATHER
size:1024  Matrix multiplication completed in 4.144714 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 4 GATHER
size:1024  Matrix multiplication completed in 2.550674 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 8 GATHER
size:1024  Matrix multiplication completed in 2.521217 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 16 GATHER
size:1024  Matrix multiplication completed in 2.810147 seconds.
```



```
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 1 GATHER
size:2048  Matrix multiplication completed in 173.220048 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 2 GATHER
size:2048  Matrix multiplication completed in 105.874383 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 4 GATHER
size:2048  Matrix multiplication completed in 62.127850 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 8 GATHER
size:2048  Matrix multiplication completed in 47.191772 seconds.
n@XiaoxinPro:~/Lab2/MPI_program$ mpiexec --oversubscribe -np 16 GATHER
size:2048  Matrix multiplication completed in 56.800016 seconds.
```

说明：
此次实验为了便于比较运行的时间，使用的代码为最朴素的矩阵乘法，而没有采用编译优化或者调整循环顺序、分块矩阵等优化算法。因为那样会使各运行时间的差距缩小，难以判断加速比。

结果分析：
两种通信方式的相同之处：可以看到不管是点对点通信还是集合通信，随着创建进程数的增加，运行时间都会减少，但是从 8 进程到 16 进程时，运行时间不降反升。这是因为本地笔记本 CPU 只有 8 个处理器核心，所以在运行时必须使用 oversubcribe 参数来使单个节点可以进行更多的进程。这也是为什么 16 进程反而运行时间更长，因为会出现多个进程竞争同一核心的资源的情况，导致总体性能反而下降。
两种通信方式的不同之处：可以看到在相同的矩阵规模和相同的进程数的条件下，点对点通信的运行时间总是比集合通信的运行时间略长。因为集合通信涉及多个进程，通常可以利用底层的优化和并行，所以总体性能会更好，对于大规模的数据传输和同步运行速度更快。

表 1 点对点通信运行时间

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	0. 0078	0. 061	0. 53	6.09	236. 44
2	0. 0036	0. 034	0. 31	5.27	115. 16
4	0. 0018	0. 026	0. 19	4.36	72. 71
8	0. 0016	0. 023	0. 15	2. 62	53. 19
16	0. 0025	0. 024	0. 16	2. 86	49. 63

表 2 集合通信运行时间

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	0. 0072	0. 056	0. 48	6.69	173.22
2	0. 0035	0. 031	0. 25	4.14	105.87
4	0. 0025	0. 026	0. 17	2.55	62.13
8	0. 0030	0. 014	0. 12	2.52	47.19
16	0. 0025	0. 020	0. 18	2.81	56.80

运行时间单位为秒（s）,表中标红的时间表示本来应该相较前一版本缩短，反而增加了的异常现象，可以看到大部分都出现在由 8 线程到 16 进程时，原因上面已经解释过。

4. 实验感想

通过这次实验，我深刻体会到了 MPI 在并行计算中的强大作用。在实现矩阵乘法的过程中，我不仅学习到了如何使用 MPI 进行点对点通信和集合通信，还通过比较不同通信策略的性能，对并行计算中的扩展性和效率有了更直观的认识。此外，将算法封装成库函数并编译为共享库的过程，让我对代码的编译，链接运行等都有了更深入的了解。