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	基于 MPI 通用矩阵乘法	专业（方向）	信息与计算科学
学号	22336044	姓名	陈圳煌
Email	2576926165@qq.com	完成日期	2024.10.09

1. 实验目的

基于 MPI 库，分别使用点对点通信以及集合通信的方法实现通用矩阵乘法，并对比两者的运行性能。并尝试在 Windows 系统下将自己的矩阵乘法函数改造成库函数并进行调用。

2. 实验过程和核心代码

点对点通信（test2_p2p.cpp 代码文件）：

```
#include<iostream>
#include<mpi.h>
#include<cstring>
#include<random>
#include<ctime>
#include<chrono>
using namespace std;

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

void matrix_initialize(double *matrix,int m,int n){
    random_device rd;
    mt19937 generator(rd());

    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
```

```
            //uniform_real_distribution<> distribution(0.0,5.0);
            matrix[i*n+j]=0.05;
        }
    }
}

int main(int argc, char *argv[]){
    int comm_sz;
    int my_rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    int M=2048;
    int N=2048;
    int K=2048;

    double max_time;
    double min_time;
    double avg_time;
    double *A=NULL;
    double *B=new double[N*K];
    double *C=new double[M*K];
    double *local_A=new double[(M/comm_sz)*N];
    double *local_C=new double[(M/comm_sz)*K];

    if(my_rank==0){
        A=new double[M*N];
        matrix_initialize(A,M,N);
        matrix_initialize(B,N,K);
    }

    auto start=MPI_Wtime();

    if(my_rank==0){
        for(int p=1;p<comm_sz;p++){
            MPI_Send(A+p*(M/comm_sz)*N, (M/comm_sz)*N, MPI_DOUBLE, p, 0,
MPI_COMM_WORLD);
        }
        memcpy(local_A, A, (M/comm_sz)*N*sizeof(double));
    }else{
        MPI_Recv(local_A, (M/comm_sz)*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
    }


    if(my_rank==0){
        for(int p=1;p<comm_sz;p++){
            MPI_Send(B, N*K, MPI_DOUBLE, p, 0, MPI_COMM_WORLD);
        }
    }else{
        MPI_Recv(B, N*K, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    }

    matrix_multiply(M/comm_sz, N, K, local_A, B, local_C);

    if(my_rank!=0){
        MPI_Send(local_C, (M/comm_sz)*K, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
    }else{
        memcpy(C, local_C, (M/comm_sz)*K*sizeof(double));
        for(int p=1;p<comm_sz;p++){
            MPI_Recv(C+p*(M/comm_sz)*K, (M/comm_sz)*K, MPI_DOUBLE, p, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }
    auto end=MPI_Wtime();
    auto my_time=end-start;

    MPI_Reduce(&my_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);
    MPI_Reduce(&my_time, &min_time, 1, MPI_DOUBLE, MPI_MIN, 0,
MPI_COMM_WORLD);
    MPI_Reduce(&my_time, &avg_time, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
    printf("Process %d: Used time: %.4fms\n",my_rank,1000*my_time);

    if(my_rank==0){
        cout << "Matrix Multiplication Finished" << endl;
        printf("Used
time:\nMax_time: %.4fms\nMin_time: %.4fms\nAvg_time: %.4fms\n",1000*max
_time,1000*min_time,1000*avg_time/comm_sz);
        delete []A;
    }
```

```
    delete []local_A;
    delete []local_C;
    delete []B;
    delete []C;

    MPI_Finalize();
    return 0;
}
```

采用 MPI_Send 和 MPI_Recv 实现进程间的点对点通信，为了保证数组的连续性，均采用一维数组来进行矩阵计算。首先将 A 矩阵平均分成 M/Comm_sz 份，分别发送到其他几个进程（0 进程的部分矩阵直接复制）为 local_A，然后将 B 矩阵完整地发送到其他几个进程，将每个进程的 local_A 与 B 进行矩阵乘法后将每个进程的结果矩阵 local_C 按照地址回传到 0 进程组合成 C。

集合通信（test2_col.cpp 代码文件）：

```
#include<iostream>
#include<mpi.h>
#include<cstring>
#include<random>
#include<ctime>
#include<chrono>
using namespace std;

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

void matrix_initialize(double *matrix,int m,int n){
    random_device rd;
    mt19937 generator(rd());

    for(int i=0;i<m;i++){
```

```
        for(int j=0;j<n;j++){
            uniform_real_distribution<> distribution(0.0,5.0);
            matrix[i*n+j]=0.05;
        }
    }
}

int main(int argc, char *argv[]){
    int comm_sz;
    int my_rank;
    string str;

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

    int M=1024*2;
    int N=1024*2;
    int K=1024*2;

    double max_time;
    double min_time;
    double avg_time;
    double *A=NULL;
    double *B=new double[N*K];
    double *C=new double[M*K];
    double *local_A=new double[(M/comm_sz)*N];
    double *local_C=new double[(M/comm_sz)*K];

    if(my_rank==0){
        A=new double[M*N];
        matrix_initialize(A,M,N);
        matrix_initialize(B,N,K);
    }

    auto start=MPI_Wtime();
    MPI_Scatter(A, (M/comm_sz)*N, MPI_DOUBLE, local_A, (M/comm_sz)*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
    // auto start1=chrono::high_resolution_clock::now();
    MPI_Bcast(B, N*K, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    // auto start2=chrono::high_resolution_clock::now();
    matrix_multiply(M/comm_sz, N, K, local_A, B, local_C);
```

```
    // auto start3=chrono::high_resolution_clock::now();
    MPI_Gather(local_C, (M/comm_sz)*K, MPI_DOUBLE, C, (M/comm_sz)*K,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
    auto end=MPI_Wtime();
    auto my_time=end-start;

    MPI_Reduce(&my_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);
    MPI_Reduce(&my_time, &min_time, 1, MPI_DOUBLE, MPI_MIN, 0,
MPI_COMM_WORLD);
    MPI_Reduce(&my_time, &avg_time, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
    printf("Process %d: Used time: %.4fms\n",my_rank,1000*my_time);

    if(my_rank==0){
        cout << "Matrix Multiplication Finished" << endl;
        printf("Used
time:\nMax_time: %.4fms\nMin_time: %.4fms\nAvg_time: %.4fms\n",1000*max
_time,1000*min_time,1000*avg_time/comm_sz);
        delete []A;
    }

    delete []local_A;
    delete []local_C;
    delete []B;
    delete []C;

    MPI_Finalize();
    return 0;
}
```

集合通信则使用 MPI_Scatter、MPI_Bcast 和 MPI_Gather 进行通信，首先将 A 通过 MPI_Scatter 平均分给所有进程，接着将 B 广播到各个进程(即完整传输），接着将矩阵乘法的结果 local_C 通过 MPI_Gather 函数回传到主进程。

以上均使用 MPI_Wtime（）来计时，在多进程的程序中，表现比使用 C 语言的函数库要更好。

用 mpi_type_create_struct 聚合 MPI 进程内变量后通信（test_struct.cpp 代码文件）：

```
#include<iostream>
#include<mpi.h>
#include<cstring>
#include<random>
#include<ctime>
#include<chrono>
using namespace std;
//自定义结构体
struct Matrix {
    int rows;
    int cols;
    double data[8][8];
};
Matrix mat;
//建立MPI 类型
void Build_mpi_type(MPI_Datatype &mpi_matrix_type) {
    MPI_Aint array_of_displacements[3]={0};
    MPI_Datatype array_of_types[3]={MPI_INT, MPI_INT, MPI_DOUBLE};
    MPI_Aint addr1, addr2, addr3;
    //计算每个元素的地址
    MPI_Get_address(&mat.rows,&addr1);
    MPI_Get_address(&mat.cols,&addr2);
    MPI_Get_address(&mat.data,&addr3);
    //计算偏移量
    array_of_displacements[1]=addr2-addr1;
    array_of_displacements[2]=addr3-addr2;

    int array_of_blocklengths[3]={1, 1, 8*8};
    //构造自定义MPI 结构体
    MPI_Type_create_struct(3, array_of_blocklengths,
array_of_displacements, array_of_types, &mpi_matrix_type);
    MPI_Type_commit(&mpi_matrix_type);
}

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    int my_rank, comm_sz;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    Matrix A, B, C;
```

```
    //初始化
    if (my_rank==0){
        A.rows=8;
        A.cols=8;
        B.rows=8;
        B.cols=8;

        for(int i=0;i<8;i++){
            for(int j=0;j<8;j++){
                A.data[i][j]=5.28/(i+1);
                B.data[i][j]=6.33/(j+1);
            }
        }
    }

    //创建自定义 MPI 数据类型，将结构体元素整理成按连续的地址空间排列
    MPI_Datatype mpi_matrix_type;
    Build_mpi_type(mpi_matrix_type);

    auto start=MPI_Wtime();
    //广播A 和B 结构体
    MPI_Bcast(&A, 1, mpi_matrix_type, 0, MPI_COMM_WORLD);
    MPI_Bcast(&B, 1, mpi_matrix_type, 0, MPI_COMM_WORLD);
    //将矩阵分块
    int part_row=A.rows/comm_sz;
    int start_row=my_rank*part_row;
    int end_row=start_row+part_row;

    //矩阵运算
    //每个进程进行特定行的矩阵乘法即可
    for(int i=start_row;i<end_row;i++){
        for(int j=0;j<B.cols;j++){
            C.data[i][j]=0.0;
            for (int k=0;k<A.cols;k++){
                C.data[i][j]+=A.data[i][k]*B.data[k][j];
            }
        }
    }

    //结果回传
    if(my_rank!=0){
        MPI_Send(&C.data[start_row][0], (end_row-start_row)*B.cols,
MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

```
    }else{
        for(int p=1;p<comm_sz;p++){
            int recv_start_row=p*part_row;
            int recv_end_row=recv_start_row+part_row;
            MPI_Recv(&C.data[recv_start_row][0],
(recv_end_row-recv_start_row)*B.cols, MPI_DOUBLE, p, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }
    }
    auto end=MPI_Wtime();
    auto my_time=end-start;

    if(my_rank==0){
        cout << "Matrix Multiplication Finished" << endl;
        printf("Used time: %.4fms\n",1000*my_time);
        for(int i=0;i<8;i++){
            for(int j=0;j<8;j++){
                cout << C.data[i][j] << " ";
            }
            cout << endl;
        }
    }

    //清理自定义类型
    MPI_Type_free(&mpi_matrix_type);

    MPI_Finalize();
    return 0;
}
```

将矩阵包装成结构体，结构体包含矩阵的大小（包括行和列）以及一个二维数组，在进行进程间通信时，首先为自定义的结构体创建一个 MPI 数据类型，使结构体内各个元素按连续的地址空间排列。在传输时，使用 MPI_Bcast 传输整个结构体。对于矩阵运算部分，对 A 矩阵进行行分块，每个进程只需要计算分到的部分矩阵即可，由于回传数据如果按结构体回传，则需要另外定义多个结构体进行储存，因此采用点对点通信进行部分传输。

将数据聚合成结构体进行进程间通信适合复杂数据，在本次实验中，矩阵乘法不需要使用结构体，因此后续的测试中仅使用数组进行进程间通信。

测试：

```
Matrix Multiplication Finished
Used time: 2.0944ms
267.379 133.69 89.1264 66.8448 53.4758 44.5632 38.197 33.4224
133.69 66.8448 44.5632 33.4224 26.7379 22.2816 19.0985 14.6223
89.1264 44.5632 29.7088 22.2816 17.8253 14.8544 12.7323 9.7482
66.8448 33.4224 22.2816 16.7112 13.369 11.1408 9.54926 7.31115
53.4758 26.7379 17.8253 13.369 10.6952 8.91264 7.63941 5.84892
44.5632 22.2816 14.8544 11.1408 8.91264 7.4272 6.36617 4.8741
38.197 19.0985 12.7323 9.54926 7.63941 6.36617 5.45672 4.1778
29.2446 14.6223 9.7482 7.31115 5.84892 4.8741 4.1778 3.65558
```

改造矩阵乘法库函数：

首先将在 matrix_multiply.c 文件中复现实验 0 中的矩阵乘法函数，接着在 matrix_multiply.h 文件中给出定义，然后在主函数 main.cpp 中进行调用。

```
//matrix_multiply,c
#include<stdio.h>
#include<stdlib.h>
#include"matrix_multiply.h"

void matrix_multiply(double **A, double **B, double **C,int M,int N,int K){
    for(int i=0;i<M;i++){
        for(int l=0;l<N;l++){
            for(int j=0;j<K;j++){
                C[i][j]+=A[i][l]*B[l][j];
            }
        }
    }
}

//matrix_multiply.h
#ifndef MATRIX_MULTIPLY_H
#define MATRIX_MULTIPLY_H

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

#endif

//main.cpp
#include<iostream>
#include"matrix_multiply.h"
```

```
using namespace std;
int main(){
    int M,N,K;
    cin >> M >> N >> K ;
    double **A=new double*[M];
    double **B=new double*[N];
    double **C=new double*[M];
    for(int i=0;i<M;i++){
        A[i]=new double[N];
        C[i]=new double[K];
    }
    for(int i=0;i<N;i++){
        B[i]=new double[K];
    }
    for(int i=0;i<M;i++){
        for(int j=0;j<N;j++){
            A[i][j]=100.001/(i+2);
        }
    }
    for(int i=0;i<N;i++){
        for(int j=0;j<K;j++){
            B[i][j]=500.112/(j+3);
        }
    }
    for(int i=0;i<M;i++){
        for(int j=0;j<K;j++){
            C[i][j]=0.0;
        }
    }

    matrix_multiply(A,B,C,M,N,K);
    for(int i=0;i<M;i++){
        for(int j=0;j<K;j++){
            cout << C[i][j] << " ";
        }
        cout << endl;
    }
    for(int i=0;i<M;i++){
        delete []A[i];
        delete []C[i];
    }
    for(int i=0;i<N;i++){
        delete []B[i];
```

```
    }
    delete []A;
    delete []B;
    delete []C;
}
```

接着在命令行输入指令

```
g++ -c matrix_multiply.c
```

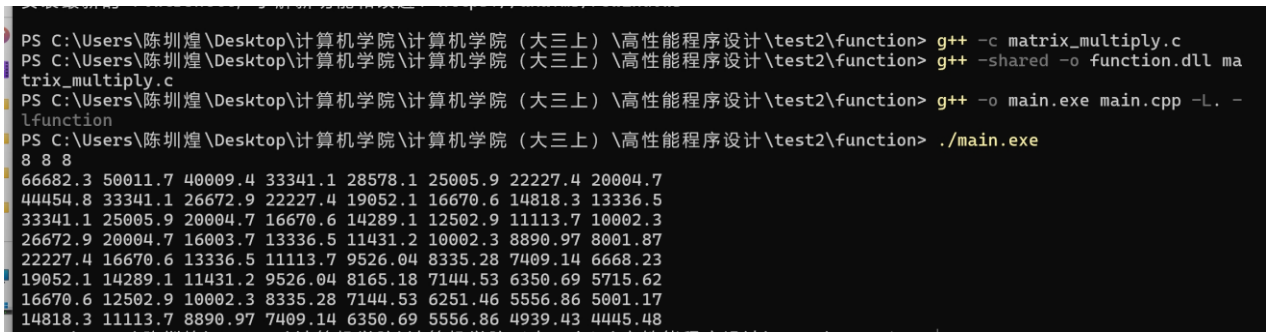
编译源文件生成 matrix_multiply.o 文件，接着运行

```
g++ -shared -o function.dll matrix_multiply.c
```

生成一个名为 function.dll 的动态链接库，接着运行

```
g++ -o main.exe main.cpp -L. -lfunction
```

编译主程序并连接动态库，最后运行 main.exe 文件。



经验证，程序成功链接动态库，并调用矩阵乘法的函数。（相关代码文件存于 function 子文件夹下）

3. 实验结果

将 A，B 矩阵的每个元素都设置为 0.05，避免随机浮点出现的误差。

以下测试时间为了避免 0 号进程因本地软件影响，均使用平均时间。

MPI 点对点通信进行矩阵浮点运算的运行时间

Comm_size (num of processes)	Order of Matrix (milliseconds)				
	128	256	512	1024	2048
1	6.9091	54.1928	491.7677	4499.8109	45370.2624
2	4.4252	27.4489	252.9164	2473.2121	25089.5369
4	3.9246	16.9178	124.2158	1288.8180	14246.0652
8	4.3322	13.8860	91.8016	800.3426	7914.0132
16	6.3995	14.0604	83.2280	706.1632	7032.4188

加速比

Comm_size (num of processes)	Order of Matrix (Speedups)				
	128	256	512	1024	2048
1	1.0	1.0	1.0	1.0	1.0
2	1.56	1.97	1.94	1.82	1.81
4	1.76	3.20	3.96	3.49	3.18
8	1.59	3.90	5.36	5.62	5.73
16	1.08	3.85	5.91	6.37	6.45

并行效率

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	1.0	1.0	1.0	1.0	1.0
2	0.78	0.99	0.94	0.91	0.91
4	0.445	0.8	0.99	0.87	0.80
8	0.20	0.49	0.67	0.70	0.72

16	0.07	0.24	0.37	0.40	0.40

MPI 集合通信进行矩阵浮点运算的运行时间

Comm_size (num of processes)	Order of Matrix (milliseconds)				
	128	256	512	1024	2048
1	6.8540	53.7850	502.9912	4492.0857	44924.5677
2	3.5785	27.3920	253.0607	2302.5493	23436.7563
4	2.2378	15.2677	126.8790	1303.9005	13790.7960
8	1.4408	9.8178	77.0336	808.2844	7504.2865
16	1.5053	9.4530	80.9273	709.5663	6754.1034

加速比

Comm_size (num of processes)	Order of Matrix (Speedups)				
	128	256	512	1024	2048
1	1.0	1.0	1.0	1.0	1.0
2	1.92	1.96	1.99	1.95	1.92
4	3.06	3.52	3.96	3.45	3.26
8	4.76	5.48	6.53	5.56	5.99
16	4.55	5.69	6.22	6.33	6.65

并行效率

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	1.0	1.0	1.0	1.0	1.0
2	0.96	0.98	0.99	0.98	0.96
4	0.77	0.88	0.99	0.86	0.82
8	0.60	0.68	0.82	0.70	0.75
16	0.28	0.36	0.39	0.40	0.42

将规模提升至 4096*4096，在 Comm_sz=8 时，点对点通信的运行时间为 87.892.8737ms，集合通信的运行时间为 79470.0881ms，差异为 7s 左右。

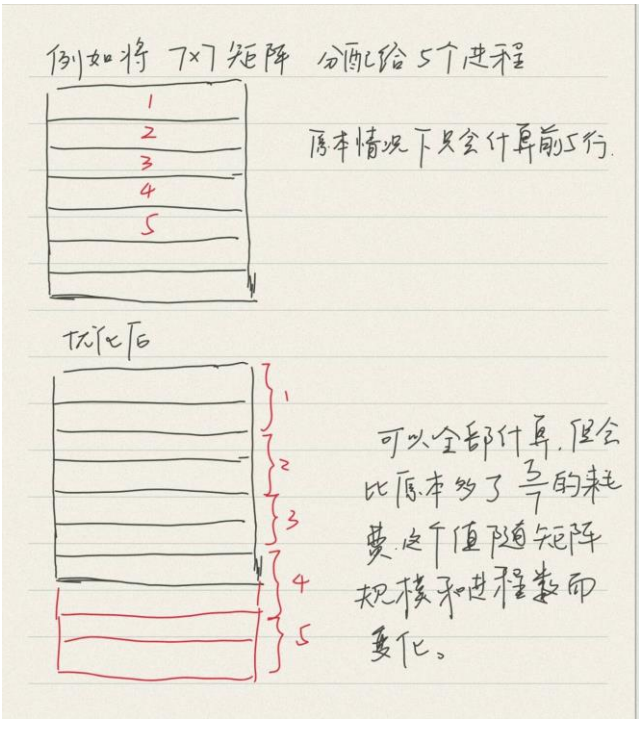
根据多次测试的结果，可以得出：

对于点对点通信实现并行矩阵乘法，随着节点数的增加，通信的需求线性增加，因此在强扩展下上受到限制，而在弱扩展性上每对节点之间的通信可以并行进行，性能基本保持稳定，可能会出现一定程度下降。

对于集合通信实现并行矩阵乘法，通过广播的方式，使得一个节点可以同时与多个节点通信，因此在强扩展下上表现较好，但在弱扩展性上，随着节点增加，可能会出现消息传递的拥堵或者延迟。

对于矩阵规模大小 M 如果不被进程数整除的解决思路：

在进行实验的过程中，发现了如果矩阵规模大小 M 不能被进程数 Comm_sz 整除的话，在进行运算时矩阵的最后一部分不满 M/Comm_sz 大小的那部分数据不参与矩阵运算，从而导致最后一部分数据出错。由于处理较为复杂且不影响本次实验，在此提供个人对此的思路，首先在进程间通信之前，先判断 M/Comm_sz 是否为 0，是的话照常运算，否的话将每个进程应该分配到矩阵的行数加一，（local_A 为 M/Comm_sz+1，local_C 为 M/Comm_sz+1），最后一个进程的最后空余行则全部补 0，将 local_C 矩阵回传时，需要将多余部分舍去。在这样的处理下，可以避免传统方法将最后剩余部分全部分配给最后一个进程而导致各个进程运行时间不均衡的问题。但是缺点是，会增加一部分损耗。



实验过程中发现的问题：

在验证 4096x4096 规模大小下的两种通信的矩阵乘法运行时间后，尝试进行 8192x8192 规模的计算，然而将规模提升至 8192*8192，在 Comm_sz=8 时，点对点通信的运行时间为 888018.0844 ms，集合通信的运行时间为 1198213.7019 ms，差异将近 5 分钟。与得到的结论截然相反。求助老师后，决定进行分阶段测试，将程序各个部分分别测试时间，验证哪一部分出现异常。

点对点通信，进程数 4，4096*4096 规模

```
Matrix Multiplication Finished
Used time :
1:32.8734ms
2:112.6118ms
3:109904.3491 ms
4:364.2879ms
```

集合通信，进程数 4，4096*4096 规模

```
Matrix Multiplication Finished
Used time :
1:17.1270ms
2:61.5181ms
3:108803.0615 ms
4:5916.7177ms
```

发现在传送 A、B 和矩阵运算并没有很大的时间差异，而是在回传 C 的过程中，使用点对点传回和 MPI_Gather 有较大的差。

关闭电脑其他应用一段时间，以防止干扰，重新进行测试：

点对点通信：989033.1893ms

```
P2P Matrix Multiplication Finished
Used time :
1:111.5962ms
2:1125.2891ms
3:983888.6359 ms
4:3907.6681ms
```

集合通信：985896.9883ms

```
COL Matrix Multiplication Finished
Used time :
1:114.8976ms
2:932.2589ms
3:980273.0460 ms
4:4576.7678ms
```

发现本次相差不大，推测先前出现差异过大可能是因为其他无关进程影响。

4. 实验感想

在这次实验中，我学习了用 MPI 库来实现矩阵乘法的并行计算，相比于之前的大大提高了效率。同时也了解了点对点通信和集合通信，并且知道了两者之间实现方式的不同，在不同规模下的运行性能差异以及在不同的扩展情况下的扩展性。最后我通过测试发现了在个人计算机上运行大规模矩阵的矩阵乘法会出现的问题，并尝试探究其中的原理，大大提高了我对这个实验的理解以及对探索问题的积极性。