

中山大学数据科学与计算机学院本科生实验报告（2020学年秋季学期）

- 课程名称：高性能计算程序设计
- 任课教师：黄聃
- 年级/专业：18级/计算机
- 学号/姓名：18340236/朱煜
- Email：zhuy85@mail2.sysu.edu.cn
- 完成日期：2020.10.08

一、实验目的

1、通过 MPI 实现通用矩阵乘法

- 通过 MPI 实现通用矩阵乘法（Lab1）的并行版本，MPI 并行进程（rank size）从1增加至8，矩阵规模从512增加至2048。
- 通用矩阵乘法（GEMM）通常定义为：

$$C = AB$$
$$C_{m,n} = \sum_{k=1}^N A_{m,k} B_{k,n}$$

输入：M, N, K 三个整数（512 ~ 2048）

问题描述：随机生成 M * N 和 N * K 的两个矩阵A, B，对这两个矩阵做乘法得到矩阵C

输出：A, B, C 三个矩阵以及矩阵计算的时间

2、基于 MPI 的通用矩阵乘法优化

- 分别采用 MPI 点对点通信和 MPI 集合通信实现矩阵乘法中的进程之间通信，并比较两种实现方式的性能。

3、改造 Lab1 成矩阵乘法库函数

- 将 Lab1 的矩阵乘法改造为一个标准的库函数 `matrix_multiply`（函数实现文件和函数头文件），输入参数为三个完整定义矩阵（A,B,C），定义方式没有具体要求，可以是二维矩阵，也可以是 struct 等。在 Linux 系统中将此函数编译为 .so 文件，由其他程序调用

二、实验过程

1、通过 MPI 实现通用矩阵乘法

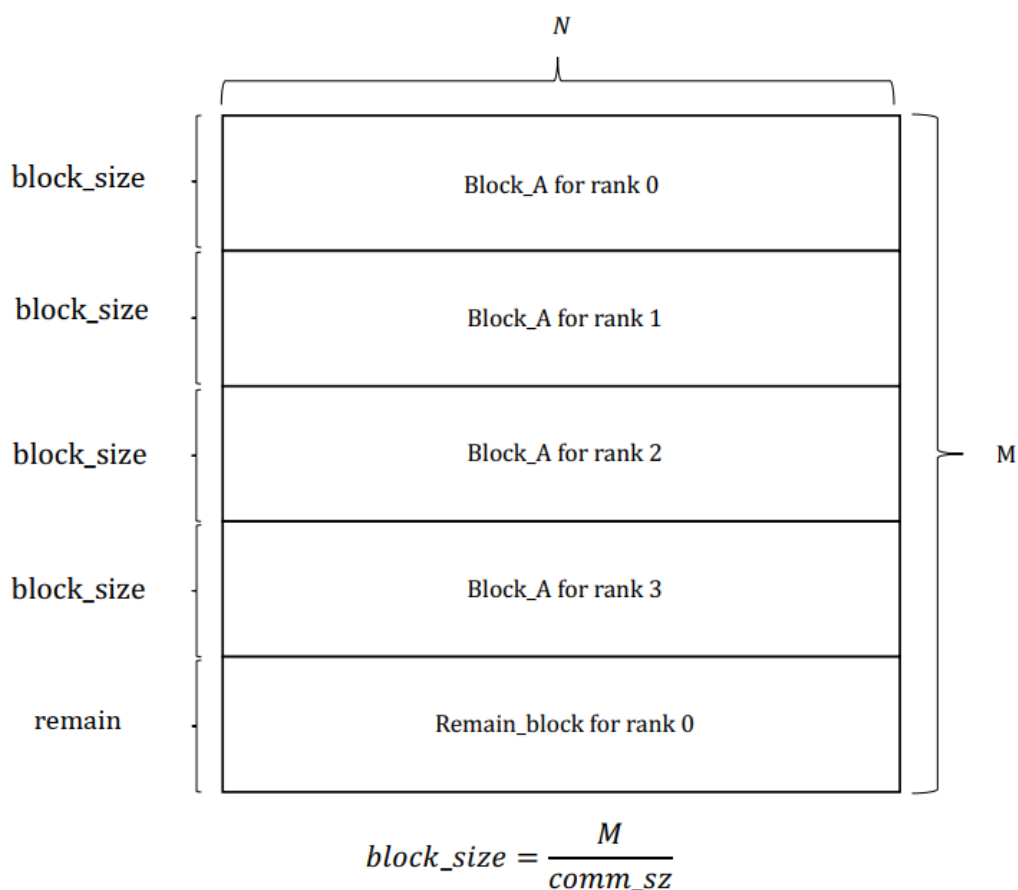
- 为了方便数据传输，使用一维数组保存矩阵，矩阵元素使用 `int` 型，随机初始化矩阵

```
void FillMatrix(int *matrix, int row, int col)
{
    for (int i = 0; i < row; ++i)
        for (int j = 0; j < col; ++j)
            matrix[i * col + j] = random(0, 9);
}
```

- 根据GEMM实现简单的矩阵乘法

```
void MatrixMul(int *A, int *B, int *C, int m, int n, int k)
{
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            int temp = 0;
            for (int z = 0; z < n; ++z)
                temp += A[i * n + z] * B[z * k + j];
            C[i * k + j] = temp;
        }
    }
}
```

- 本次MPI采用点对点通信，将A矩阵根据进程数按行划分，每部分A矩阵由0号进程使用 `MPI_Send` 函数发送给各进程，同时将B矩阵全部进行发送，各进程使用 `MPI_Recv` 函数接收A矩阵块与B矩阵进行计算，得到结果再发送回0号进程，最后由0号进程整合成C矩阵。当进程数为4时，矩阵A的划分方式如下图



- 0号进程进行矩阵划分并发送给其他进程，计算完属于自己部分矩阵后计算划分后剩余的部分矩阵，最后对所有结果进行整合，0号进程的具体代码如下

```
if (my_rank == 0)
{
    A = new int[m * n];
    C = new int[m * k];
    FillMatrix(A, m, n);
    FillMatrix(B, n, k);
```

```

start = MPI_Wtime();
// 发送A矩阵块与B矩阵
for (int i = 1; i < comm_sz; ++i)
{
    printf("rank 0 send A to %d\n", i);
    MPI_Send(A + i * block_size * n, block_size * n, MPI_INT, i, 0,
MPI_COMM_WORLD);
    printf("rank 0 send B to %d\n", i);
    MPI_Send(B, n * k, MPI_INT, i, 1, MPI_COMM_WORLD);
}
// 计算属于0号线程的那部分矩阵
MatrixMul(A, B, C, block_size, n, k);
// 无法整除 计算剩余矩阵
int remain = m - block_size * comm_sz;
if (remain > 0)
    MatrixMul(A + block_size * comm_sz * n, B, C + block_size *
comm_sz * k, remain, n, k);
// 接收来自其他核的结果
for (int i = 1; i < comm_sz; ++i)
{
    MPI_Recv(C + i * block_size * k, block_size * k, MPI_INT, i, i,
MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
    printf("rank 0 recv C from %d\n", i);
}
end = MPI_Wtime();
PrintMatrix(A, B, C, m, n, k);
printf("Computation time is %fs\n", end - start);
}

```

其余进程运行的具体代码如下

```

else
{
    // 接收A矩阵块
    MPI_Recv(block_A, block_size * n, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUSES_IGNORE);
    printf("rank %d recv A from 0\n", my_rank);
    // 接收B矩阵
    MPI_Recv(B, n * k, MPI_INT, 0, 1, MPI_COMM_WORLD,
MPI_STATUSES_IGNORE);
    printf("rank %d recv B from 0\n", my_rank);
    // 计算矩阵
    MatrixMul(block_A, B, block_C, block_size, n, k);
    printf("rank %d send C to 0\n", my_rank);
    // 发送计算结果
    MPI_Send(block_C, block_size * k, MPI_INT, 0, my_rank,
MPI_COMM_WORLD);
}

```

- 为验证正确性，使用小的矩阵维度进行矩阵的打印，当矩阵规模大时，为了方便调试，进行运算过程的 `printf` 显示当前运行进度，将矩阵打印的取消，运行结果如下，则运算正确

```
Matrix A:
1 7 0 7
5 7 1 3
6 1 5 4
5 7 5 4
Matrix B:
6 0 7 1
8 8 6 6
8 8 8 4
1 1 5 0
Matrix C:
69 63 84 43
97 67 100 51
88 52 108 32
130 100 137 67
Computation time is 0.000207s
```

调节进程数与矩阵规模进行输入，其运算时间结果（单位：s）如下，由于采用的虚拟机设置为4核，因此在大于4时基本无法进行加速，具体分析见实验结果

线程数\矩阵规模	512*512*512	1024*1024*1024	2048*2048*2048
1	0.591688	5.256663	99.408401
2	0.358821	3.317517	55.798389
4	0.172018	2.064348	33.793644
6	0.240423	2.304043	32.763830
8	0.208036	2.198216	34.742263

2、基于 MPI 的通用矩阵乘法优化

(1) MPI 点对点通信

- 实现过程见 1、通过 MPI 实现通用矩阵乘法

(2) MPI 集合通信

- 集合通信的矩阵初始化与矩阵的计算方式与MPI点对点通信的实现相同，同上
- 集合通信划分A矩阵的方式与点对点通信相同，两者区别在于通信方式，集合通信使用 MPI_Scatter 函数与 MPI_Bcast 进行矩阵块的发送，使用 MPI_Gather 函数进行矩阵的整合，具体实现代码如下

```
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
printf("rank 0 Scatter\n");
// 对矩阵A进行划分并发送
MPI_Scatter(A, block_size * n, MPI_INT, block_A, block_size * n,
MPI_INT, 0, MPI_COMM_WORLD);
printf("rank 0 Bcast\n");
// 广播矩阵B
MPI_Bcast(B, n * k, MPI_INT, 0, MPI_COMM_WORLD);
```

```

MatrixMul(block_A, B, block_C, block_size, n, k);

MPI_Barrier(MPI_COMM_WORLD);
// 整合矩阵C
MPI_Gather(block_C, block_size * k, MPI_INT, C, block_size * k, MPI_INT,
0, MPI_COMM_WORLD);
printf("rank 0 Gather");
// 计算划分剩下的矩阵
if (my_rank == 0 && block_size * comm_sz < m)
{
    int remain = m - block_size * comm_sz;
    MatrixMul(A + block_size * comm_sz * n, B, C + block_size * comm_sz
* k, remain, n, k);
}

```

- 为验证正确性，使用小的矩阵维度进行矩阵的打印，当矩阵规模大时，为了方便调试，进行运算过程的 printf 显示当前运行进度，将矩阵打印的取消，运行结果如下，可知运算正确

```

Matrix A:
1 7 0 7
5 7 1 3
6 1 5 4
5 7 5 4
Matrix B:
6 0 7 1
8 8 6 6
8 8 8 4
1 1 5 0
Matrix C:
69 63 84 43
97 67 100 51
88 52 108 32
130 100 137 67
Computation time is 0.001743s

```

调节进程数与矩阵规模进行输入，其运算时间结果（单位：s）如下，由于采用的虚拟机设置为4核，因此在大于4时基本无法进行加速，具体分析见实验结果

线程数\矩阵规模	512*512*512	1024*1024*1024	2048*2048*2048
1	0.530271	8.354856	97.127116
2	0.303783	4.090406	56.036649
4	0.228195	2.609817	36.431014
6	0.253376	2.890945	37.299923
8	0.182572	2.395501	35.665196

3、改造 Lab1 成矩阵乘法库函数

- 改写 lab1 的矩阵乘法，分别整合成 MatrixMulLib.h 与 MatrixMulLib.cpp 文件，保留 GEMM, Strassen, OptimizationMul 三种矩阵运算方式，采用二维指针作为矩阵输入，具体 matrix_multiply 代码如下

```
void matrix_multiply(int **A, int **B, int **C, int M, int N, int K)
{
    if ((M > 2048) || (N > 2048) || (K > 2048))
    {
        cout << "Error" << endl;
        return;
    }
    int length = 1;
    while (length < M || length < N || length < K)
        length *= 2;
    int **Atemp = new int *[length];
    int **Btemp = new int *[length];
    int **Ctemp = new int *[length];
    for (int i = 0; i < length; ++i)
    {
        Atemp[i] = new int[length];
        Btemp[i] = new int[length];
        Ctemp[i] = new int[length];
    }
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            Atemp[i][j] = A[i][j];
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < K; ++j)
            Btemp[i][j] = B[i][j];
    clock_t start = clock();
    Mul(Atemp, Btemp, Ctemp, M, N, K);
    clock_t end = clock();
    cout << "GEMM运算时间: " << 1000 * (end - start) / CLOCKS_PER_SEC << "ms"
    << endl;
    FillMatrix(Atemp, Btemp, Ctemp, M, N, K, length);
    start = clock();
    Strassen(Atemp, Btemp, Ctemp, length);
    end = clock();
    cout << "Strassen运算时间: " << 1000 * (end - start) / CLOCKS_PER_SEC <<
    "ms" << endl;
    if (length < 4)
        exit(0);
    FillMatrix(Atemp, Btemp, Ctemp, M, N, K, length);
    start = clock();
    OptimizationMul(Atemp, Btemp, Ctemp, length);
    end = clock();
    cout << "OptimizationMul运算时间: " << 1000 * (end - start) /
    CLOCKS_PER_SEC << "ms" << endl;
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < K; ++j)
            C[i][j] = Ctemp[i][j];
    return;
}
```

- 使用以下命令将上述文件编译成 `.so` 文件以便其他文件调用

```
g++ -ggdb -Wall -shared -fpic -o libMM.so MatrixMulLib.cpp
```

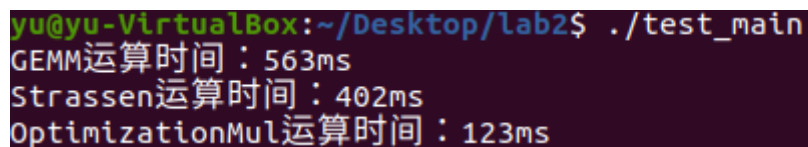
使用 `gedit` 在 `ld` 的动态链接库设置中添加生成的 `libMM.so` 文件路径，编写简单的函数调用 `matrix_multiply` 函数，具体代码如下

```
#include "MatrixMulLib.h"
int main()
{
    int length = 512;
    int **Atemp = new int *[length];
    int **Btemp = new int *[length];
    int **Ctemp = new int *[length];
    for (int i = 0; i < length; ++i)
    {
        Atemp[i] = new int[length];
        Btemp[i] = new int[length];
        Ctemp[i] = new int[length];
    }
    for (int i = 0; i < length; ++i)
        for (int j = 0; j < length; ++j)
            Atemp[i][j] = 1;
    for (int i = 0; i < length; ++i)
        for (int j = 0; j < length; ++j)
            Btemp[i][j] = 2;
    matrix_multiply(Atemp, Btemp, Ctemp, length, length, length);
}
```

使用以下命令将上述编写的 `test_mul.cpp` 进行编译

```
g++ test_mul.cpp -ldl -o test_main -L. -lMM
```

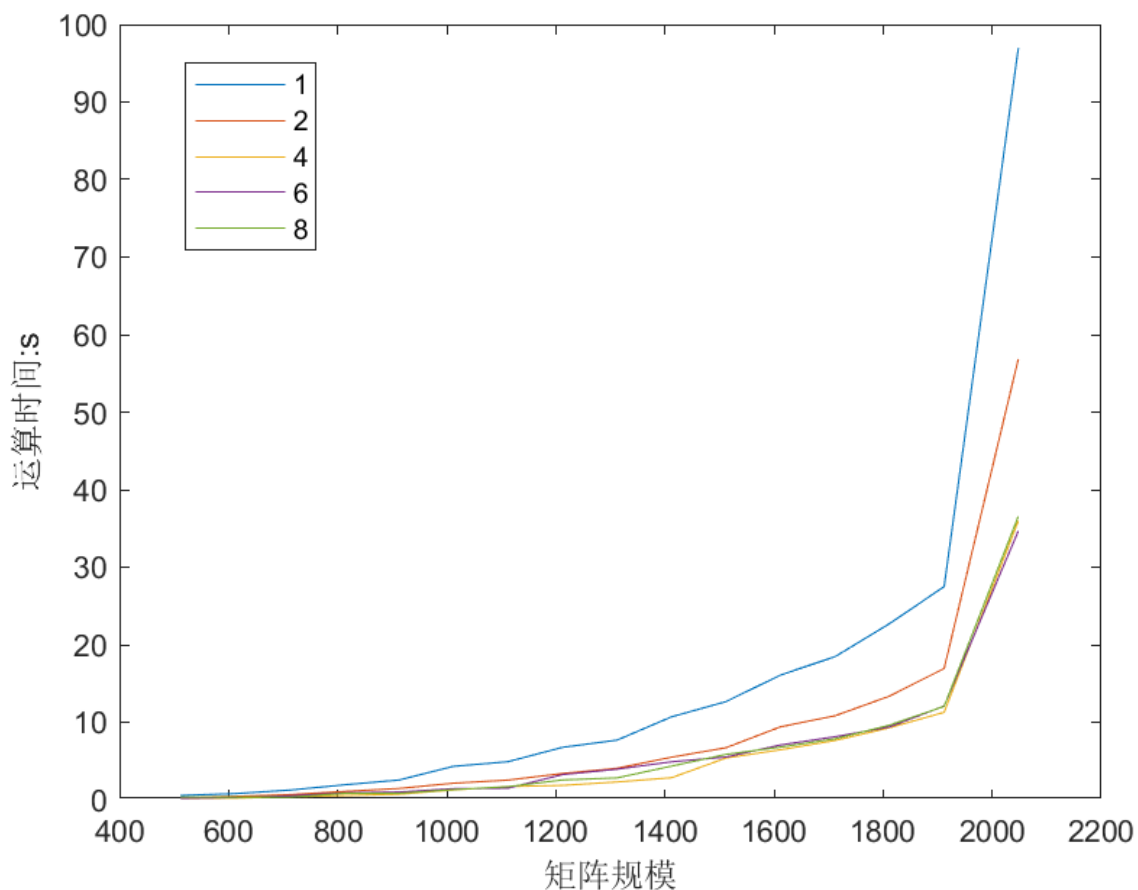
运行结果如下，可见其他程序可正常调用该函数进行矩阵运算。



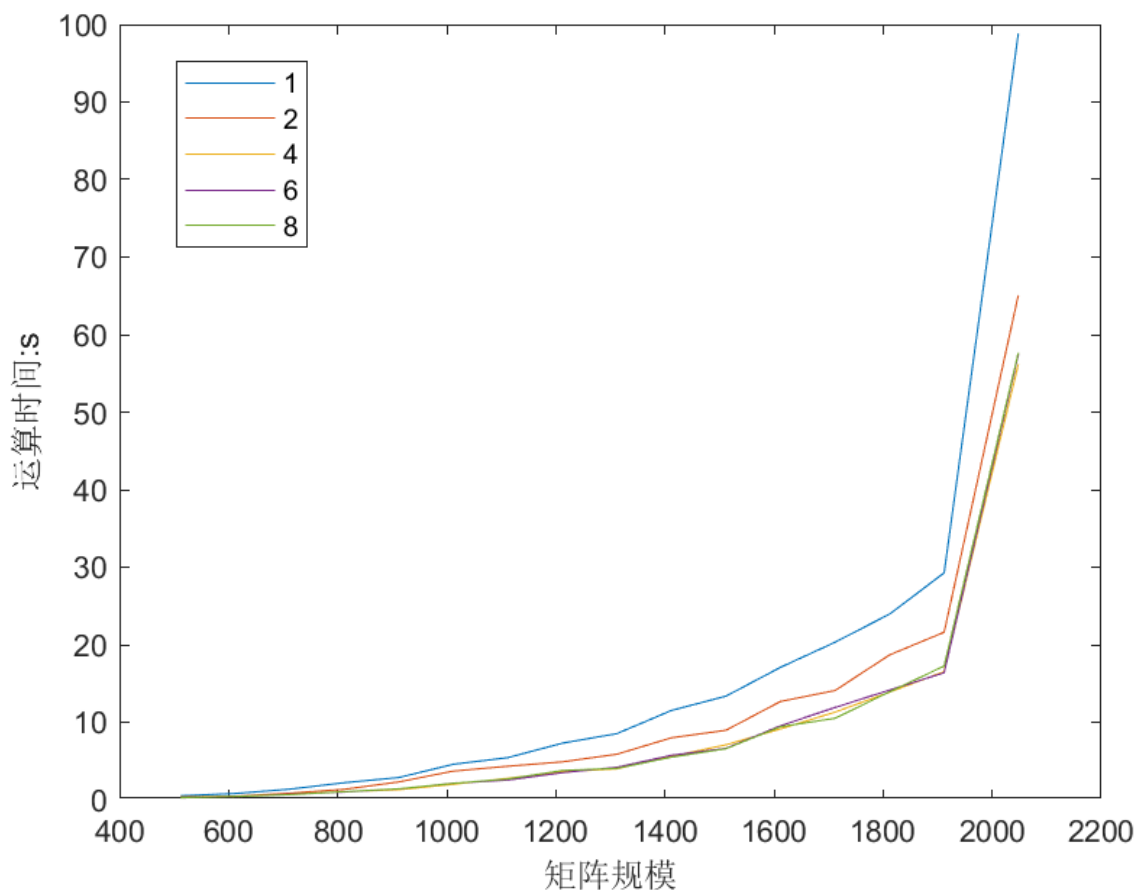
```
yu@yu-VirtualBox:~/Desktop/lab2$ ./test_main
GEMM运算时间：563ms
Strassen运算时间：402ms
OptimizationMul运算时间：123ms
```

三、实验结果

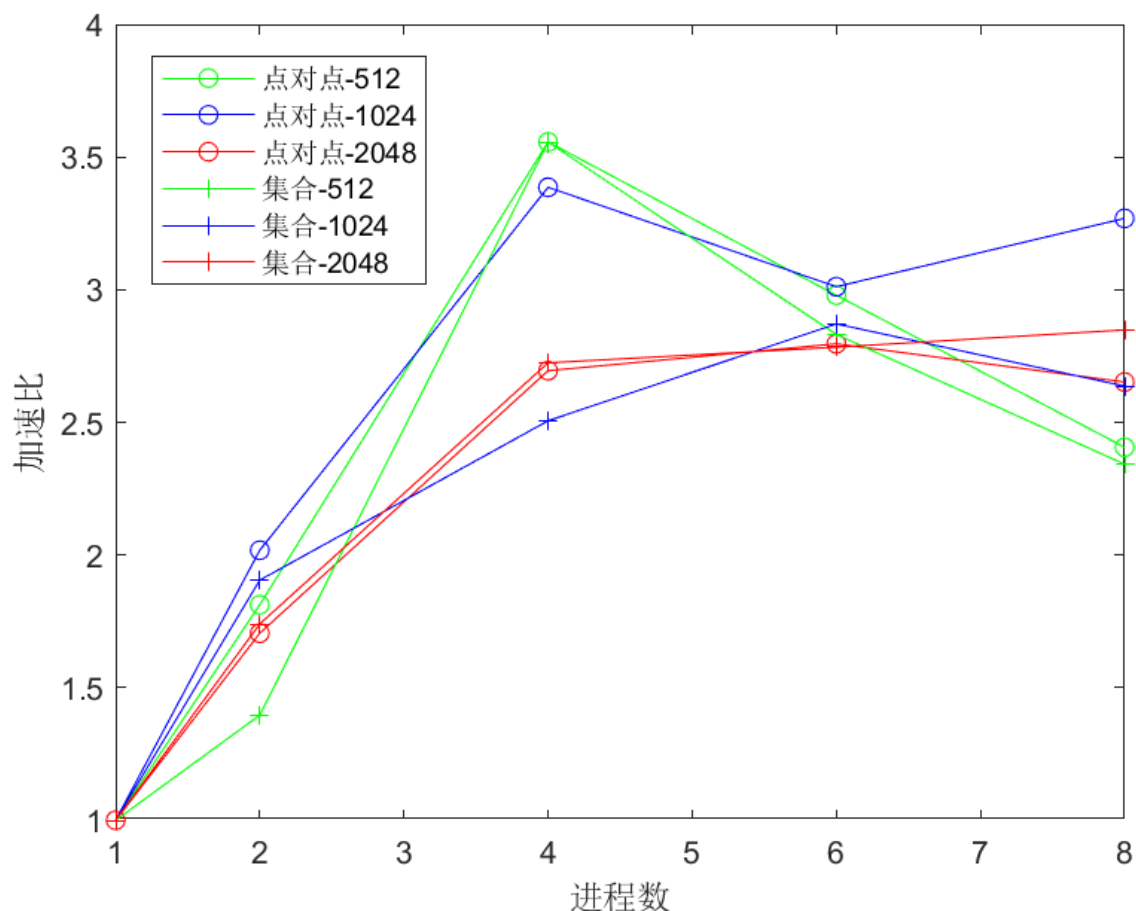
- MPI点对点通信运算时间与矩阵的规模关系如下图，运算时间随着矩阵规模大致成线性增长，而当矩阵规模达到2048左右时，运算时间增长了几倍，这是由于矩阵规模增大超过缓存界限，使得多线程在同时进行计算时，L3缓存发生了大量miss，增加了大量的矩阵读取时间，影响了运行效率。由于采用的虚拟机环境为4核4线程，因此在进程数大于4时，使用超进程运行程序时，只是创建了多个进程，将原有的时间戳重新划分给超出的进程，同时计算的上限仍为4进程，因此运算时间基本没有变化。



- MPI集合通信运算时间与矩阵的规模关系如下图，运算时间随着矩阵规模大致成线性增长，而当矩阵规模达到2048左右时，运算时间增长了几倍，这与点对点通信相同，L3缓存发生了大量miss，增加了大量的矩阵读取时间，影响了运行效率。由于计算的上限仍为4进程，因此在超过8进程时运算时间基本没有变化。



- 将点对点通信与集合通信的加速比进行比较，由于本次实验采用的集合通信方式为 `MPI_Scatter` 与 `MPI_Gather`，其具体过程与点对点通信过程大致相似，并没有采取树型结构或其他有效地通信效率，导致计算效果两者基本上没有区别，由于计算的上限仍为4进程，在进程数大于4时出现的加速比下降现象原因可能是进程之间的频繁切换花费了时间。



四、实验感想

- 由于本次实验采用的是多进程，因此出现了很多在串行程序中不会出现的问题，而其中一个较为重要的问题则是 `printf` 函数。在串行程序中，调试是一件较为简单的事情，而在多线程时便成了问题。在学习了少部分时间 `attach` 方法调试多进程后，我仍然选择使用 `printf` 函数来打印进程运行的状态。而 `printf` 是先输出到缓冲区，再从缓冲区输出到终端（文件，或者显示器）。所以最终的输出和系统有关系，并不能直接反应运行顺序。在忘记使用 `flush` 命令刷新缓冲区后导致我程序的错误进行运行，在调试中出现了许多难以预想到的问题。因此在多进程时要切记记住协调好各进程运行的相对顺序，可用 `mpi_barrier` 来同步各个进程，以得到正确的运行结果。
- 根据实验结果来看，多进程在运行过程中会共享相同的缓存，而由 lab1 可知，内存的优化与否是影响程序运行重要方面，多线程在运行过程中因为时序的问题，导致内存优化更加困难，但为了不出现像实验中矩阵规模在2000左右出现的性能锐减，保证多线程的高效工作，多线程需要更加完善的内存优化策略。