中山大学计算机学院本科生实验报告

（2021学年秋季学期）

课程名称：高性能计算程序设计基础　　批改人：

实验	**LAB2**	专业	计算机科学与技术（超算）
学号	19335162	姓名	潘思晗
Email	pansh25@mail2.sysu.edu.cn	完成日期	2021.10.06

一、实验目的

> *1.*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*。
>
> *2.*实现基于 *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、实验过程和核心代码

1.MPI实现通用矩阵乘法

（1）算法描述

通用矩阵乘法通常定义为: $C = AB$，$C_{m,n} = \sum_{n=1}^{N} A_{m,n}B_{n,k}$

串行版本`GEMM()`函数实现：传入矩阵A、B并为矩阵C申请空间，通过循环嵌套实现矩阵的相乘，最后返回计算结果的指针，如下所示

```
int** GEMM(int** matA,int** matB)
{
    int** ans = (int**)malloc(sizeof(int*)*m);
    for(int i=0;i<m;i++){
        ans[i]=(int*)malloc(sizeof(int)*k);
    }
    for(int i=0;i<m;i++){
        for(int j=0;j<k;j++){
            ans[i][j] = 0;
            for(int t=0;t<n;t++){
                ans[i][j] += matA[i][t]*matB[t][j];
            }
        }
    }
    return ans;
}
```

（2）核心代码

为实现MPI并行版本的矩阵乘法，需要按照线程数将矩阵A以行划分为线程数量的块，并发给每个线程，并将矩阵B全部发给每个线程，每个线程计算完一块之后将这一部分结果发给主线程，主线程计算最后一块再将所有结果汇总。

首先是MPI程序的初始化：

```
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

计算分块后每一块的行数并且给每个进程中都分配内存

```
int lines = m / numprocs;        //lines表示分块后的行数
block_A = new double[lines*n];   //矩阵A的分块
block_B = new double[n*k];       //矩阵B
block_C = new double[lines*k];   //计算得到的结果矩阵C
```

主进程

在主进程（0号进程）中，首先为矩阵A和B分配内存空间，并进行初始化

```
matA = new double[m*n];
matB = new double[n*k];
matC = new double[m*k];
generate_matrix(matA,m,n);
generate_matrix(matB,n,k);
```

然后标记时间，使用`MPI_Send()`将矩阵B发送给其他所有进程，而将矩阵A分块发送给其他进程

```
begin = MPI_Wtime();
for(int i=1;i<numprocs;i++){
    MPI_Send(matB, n*k, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
}//矩阵B发送给每个子进程
for(int i=1;i<numprocs;i++){
    MPI_Send(matA + (i-1)*lines*n, n*lines, MPI_DOUBLE, i, 1,
MPI_COMM_WORLD);
}//矩阵A分块发送
```

主进程还要完成剩余的矩阵计算

```
int last = lines * (numprocs-1);
if(last < m)
{
    int remaining = m - last;
    GEMM(matA+last*n,matB,matC+last*k,remaining,n,k);
}
```

最后，调用`MPI_Recv()`按顺序接收其他进程发送来的答案，并组合到矩阵C

```
for(int i=1;i<numprocs;i++){
    MPI_Recv(block_C, lines*k, MPI_DOUBLE, i, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    for(int t=0;t<lines;t++){
        for(int j=0;j<k;j++){
            matC[((i-1)*lines+t)*k+j] = block_C[t*k+j];
        }
    }
}
```

标记时间并计算得到运行总时长，最后打印出三个矩阵A、B、C以及完成矩阵乘法所用时间。

其它进程

接受矩阵B和矩阵A的分块，完成计算并将结果传送到主进程（0号进程）

```
MPI_Recv(block_B, n*k, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
MPI_Recv(block_A, n*lines, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
GEMM(block_A, block_B, block_C, lines, n, k);
MPI_Send(block_C, lines*k, MPI_DOUBLE,0,2, MPI_COMM_WORLD);
```

结束并行

```
MPI_Finalize();
```

2.基于MPI的通用矩阵乘法优化

（1）算法描述

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

点对点通信已经在上面部分实现，下面主要完成集合通信。

在集合通信中，使用`MPI_Scatter()`将矩阵 A 的分块发送给其他进程，并使用`MPI_Bcast()`将矩阵 B 广播给其他进程，然后各进程执行`GEMM()`通用矩阵乘法，并使用`MPI_Barrier()`函数确保同一个通信子中的所有进程都完成调用之前没有进程能够提前返回，最后使用`MPI_Gather()`收集每个进程的结果。

（2）核心代码

首先是MPI程序的初始化：

```
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

计算分块后每一块的行数并且给要传递的信息分配相应的内存

```
int lines = m / numprocs;        //lines表示分块后的行数
matB = new double[n*k];          //每个进程都发送的矩阵B
block_A = new double[lines*n];   //矩阵A的分块
block_C = new double[lines*k];   //计算得到的结果矩阵C
```

主进程

在主进程（0号进程）中，首先对矩阵A和B进行初始化，并为矩阵C分配相应的内存空间

```
matA = new double[m*n];
matC = new double[m*k];
generate_matrix(matA,m,n);
generate_matrix(matB,n,k);
```

然后标记时间，使用`MPI_Bcast()`将矩阵B广播给其他所有进程，而将矩阵A分块发送给其他进程，完成计算后将结果汇总

```
begin = MPI_Wtime();
MPI_Bcast(matB, n*k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(matA, lines*n, MPI_DOUBLE, block_A, lines*n,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
GEMM(block_A, matB, block_C, lines, n, k);
MPI_Gather(block_C, lines*k, MPI_DOUBLE, matC, lines*k, MPI_DOUBLE,
0, MPI_COMM_WORLD );
```

主进程还要完成剩余的矩阵计算，然后停止计时

```
int last = lines * (numprocs-1);
if(last < m)
{
    int remaining = m - last;
    GEMM(matA+last*n,matB,matC+last*k,remaining,n,k);
}
end = MPI_Wtime();
```

其它进程

接受矩阵B和矩阵A的分块，计算完毕后将结果汇总

```
MPI_Bcast(matB, n*k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(matA, lines*n, MPI_DOUBLE, block_A, lines*n,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
GEMM(block_A, matB, block_C, lines, n, k);
MPI_Gather(block_C, lines*k, MPI_DOUBLE, matC, lines*k, MPI_DOUBLE,
0, MPI_COMM_WORLD );
```

结束并行

```
MPI_Finalize();
```

3.改造Lab1成矩阵乘法库函数

（1）改造文件

函数头文件

函数头文件中，不包含`matrix_multiply()`函数的实现，仅包含其定义和库文件

```
#ifndef MATRIX_MULTIPLY_H
#define MATRIX_MULTIPLY_H
double** matrix_multiply(double** matA, double** matB, int m, int
n, int k);
#endif
```

该文件命名为`matrix_multiply.h`。

函数实现文件

函数的实现文件与头文件同名，命名为`matrix_multiply.cpp`，算法的实现如下所示

```
#include<stdlib.h>
#include "matrix_multiply.h"
double** matrix_multiply(double** matA, double** matB, int m, int
n, int k)
{
    double** ans = (double**)malloc(sizeof(double*)*m);
    for(int i=0;i<m;i++){
        ans[i]=(double*)malloc(sizeof(double)*k);
    }
    for(int i=0;i<m;i++){
        for(int j=0;j<k;j++){
            ans[i][j] = 0;
            for(int t=0;t<n;t++){
                ans[i][j] += matA[i][t]*matB[t][j];
            }
        }
    }
    return ans;
}
```

调用函数的文件此处将其命名为`main.cpp`，在此文件中调用函数`matrix_multiply()`，并输出相应结果。

注意实现文件中都要包含函数头文件`matrix_multiply.h`。

（2）生成动态链接库

首先编译`.cpp`文件，生成`.o`文件，编译命令如下：

```
g++ -c -fPIC -o matrix_multiply.o matrix_multiply.cpp
```

然后使用`-shared`参数生成`.so`动态链接库：

```
g++ -shared -o lib_MatrixMultiply.so matrix_multiply.o
```

载入动态链接库，`-L`参数指明库文件所在路径，由于我将其放在同一文件夹下，因此使用`-L.`参数表示当前路径，命令如下：

```
g++ main.cpp -L. -l_MatrixMultiply -o  main
```

最后将`.so`文件移动至 `/usr/lib` ，从而能在不同位置的程序中调用该动态库：

```
sudo mv lib_MatrixMultiply.so /usr/lib
```

三、实验结果

打印CPU相关信息如下

```
emilylyly@emilylyly-VirtualBox:~$ lscpu
架构:                x86_64
CPU 运行模式:        32-bit, 64-bit
字节序:              Little Endian
CPU:                 1
在线 CPU 列表:       0
每个核的线程数:      1
每个座的核数:        1
座:                  1
NUMA 节点:           1
厂商 ID:             GenuineIntel
CPU 系列:            6
型号:                142
型号名称:            Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
步进:                11
CPU MHz:             1992.002
BogoMIPS:            3984.00
超管理器厂商:        KVM
虚拟化类型:          完全
L1d 缓存:            32K
L1i 缓存:            32K
L2 缓存:             256K
L3 缓存:             8192K
NUMA 节点0 CPU:      0
标记:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
freq pni pclmulqdq monitor ssse3 cx16 pcid sse4_1 sse4_2 x2apic movbe
_l1d arch_capabilities
```

1.MPI实现通用矩阵乘法

检验正确性

编译和运行程序的命令如下：

```
mpicxx mpi_GEMM.cpp -o mpi_GEMM
mpirun -np 1 ./mpi_GEMM 2 4 3
```

首先用较小规模的矩阵验证一下算法的正确性。

打印所有矩阵，检验结果判断算法的实现是否正确：

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpicxx mpi_GEMM.cpp -o mpi_GEMM
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 1 ./mpi_GEMM 2 4 3
matrixA:
6.529134 4.897057 5.082188 3.366784
5.107876 4.410267 6.104697 4.097840

matrixB:
6.529134 4.897057 5.082188
3.366784 5.107876 4.410267
6.104697 4.097840 0.681325
6.325366 6.962754 9.517754

matrixC:
111.438275 101.255174 90.286448
111.386084 101.088974 88.571145
```

通过验证可知结果正确。

测试时间

令 MPI 并行进程从 1 增加到 8，矩阵规模从 512 增加到 2048

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpicxx mpi_GEMM.cpp -o mpi_GEMM
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 1 ./mpi_GEMM 512 512 512
THE TIME OF MPI_GEMM: 0.494517 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 2 ./mpi_GEMM 512 512 512
THE TIME OF MPI_GEMM: 0.520283 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 3 ./mpi_GEMM 512 512 512
THE TIME OF MPI_GEMM: 0.631959 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 4 ./mpi_GEMM 512 512 512
THE TIME OF MPI_GEMM: 0.777410 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 5 ./mpi_GEMM 512 512 512
THE TIME OF MPI_GEMM: 0.695060 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 6 ./mpi_GEMM 512 512 512
THE TIME OF MPI_GEMM: 0.730483 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 7 ./mpi_GEMM 512 512 512
THE TIME OF MPI_GEMM: 0.693505 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 8 ./mpi_GEMM 512 512 512
THE TIME OF MPI_GEMM: 0.975922 s
```

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 1 ./mpi_GEMM 1024 1024 1024
THE TIME OF MPI_GEMM: 8.655223 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 2 ./mpi_GEMM 1024 1024 1024
THE TIME OF MPI_GEMM: 8.117741 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 3 ./mpi_GEMM 1024 1024 1024
THE TIME OF MPI_GEMM: 9.313959 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 4 ./mpi_GEMM 1024 1024 1024
THE TIME OF MPI_GEMM: 7.733208 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 5 ./mpi_GEMM 1024 1024 1024
THE TIME OF MPI_GEMM: 9.337421 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 6 ./mpi_GEMM 1024 1024 1024
THE TIME OF MPI_GEMM: 9.661268 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 7 ./mpi_GEMM 1024 1024 1024
THE TIME OF MPI_GEMM: 8.829388 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 8 ./mpi_GEMM 1024 1024 1024
THE TIME OF MPI_GEMM: 8.484756 s
```

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 1 ./mpi_GEMM 2048 2048 2048
THE TIME OF MPI_GEMM: 219.689730 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 8 ./mpi_GEMM 2048 2048 2048
THE TIME OF MPI_GEMM: 207.641136 s
```

统计各计算时间如下表：

并行进程数\矩阵规模	**512 512 512**	**1024 1024 1024**	**2048 2048 2048**
1	0.494517	8.655233	219.689730
2	0.520283	8.117741	/
3	0.631959	9.313959	/
4	0.777410	7.733208	/
5	0.695060	9.337421	/
6	0.730483	9.661268	/
7	0.693505	8.829388	/
8	0.975922	8.484756	207.641136

2.基于MPI的通用乘法优化

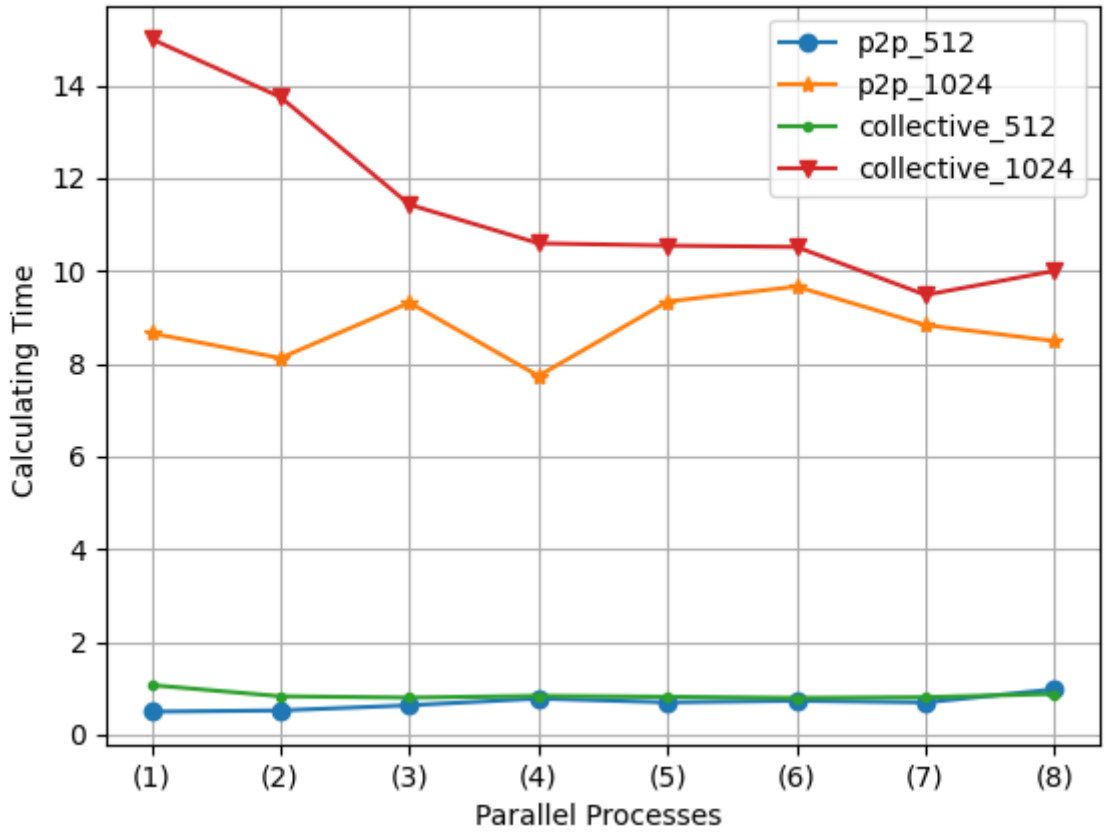
令 MPI 并行进程从 1 增加到 8，矩阵规模从 512 增加到 2048.

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 1 ./mpi_opt 512 512 512
THE TIME OF MPI_GEMM OPTIMIZATION: 1.067229 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 2 ./mpi_opt 512 512 512
THE TIME OF MPI_GEMM OPTIMIZATION: 0.825817 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 3 ./mpi_opt 512 512 512
THE TIME OF MPI_GEMM OPTIMIZATION: 0.797692 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 4 ./mpi_opt 512 512 512
THE TIME OF MPI_GEMM OPTIMIZATION: 0.834081 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 5 ./mpi_opt 512 512 512
THE TIME OF MPI_GEMM OPTIMIZATION: 0.815568 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 6 ./mpi_opt 512 512 512
THE TIME OF MPI_GEMM OPTIMIZATION: 0.789892 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 7 ./mpi_opt 512 512 512
THE TIME OF MPI_GEMM OPTIMIZATION: 0.806790 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 8 ./mpi_opt 512 512 512
THE TIME OF MPI_GEMM OPTIMIZATION: 0.876151 s
```

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 1 ./mpi_opt 1024 1024 1024
THE TIME OF MPI_GEMM OPTIMIZATION: 14.999666 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 2 ./mpi_opt 1024 1024 1024
THE TIME OF MPI_GEMM OPTIMIZATION: 13.746394 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 3 ./mpi_opt 1024 1024 1024
THE TIME OF MPI_GEMM OPTIMIZATION: 11.427158 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 4 ./mpi_opt 1024 1024 1024
THE TIME OF MPI_GEMM OPTIMIZATION: 10.593350 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 5 ./mpi_opt 1024 1024 1024
THE TIME OF MPI_GEMM OPTIMIZATION: 10.544094 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 6 ./mpi_opt 1024 1024 1024
THE TIME OF MPI_GEMM OPTIMIZATION: 10.515326 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 7 ./mpi_opt 1024 1024 1024
THE TIME OF MPI_GEMM OPTIMIZATION: 9.480355 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ mpirun -np 8 ./mpi_opt 1024 1024 1024
THE TIME OF MPI_GEMM OPTIMIZATION: 9.998999 s
```

点对点通信和集合通信性能对比

将上述数据利用python分析并对比制作图像，结果如下所示：



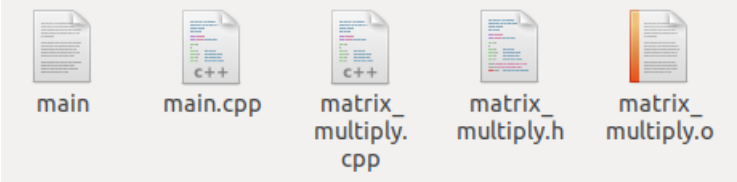
由上图可以看出，随着并发进程数量的上升，计算时间有一定的缩短。同时，对比点对点通信和集合通信，可以看到集合通信的耗时更长，因此认为在实验条件下，点对点通信的性能更优。

3.改造Lab1成矩阵乘法库函数

首先执行下列命令

```
g++ -c -fPIC -o matrix_multiply.o matrix_multiply.cpp
g++ -shared -o lib_MatrixMultiply.so matrix_multiply.o
g++ main.cpp -L. -l_MatrixMultiply -o  main
sudo mv lib_MatrixMultiply.so /usr/lib
```

执行之后可以看到下列文件（以及移动到`/usr/lib`的`.so`文件）



使用`ldd`命令判断动态链接库是否链接成功

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ ldd ./main
        linux-vdso.so.1 (0x00007ffca3df0000)
        lib_MatrixMultiply.so => /usr/lib/lib_MatrixMultiply.so (0x00007f535e411000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f535e020000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f535e816000)
```

如图所示，动态连接成功。

下面测试程序`main`能否正常调用`matrix_multiply`函数：

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab2$ ./main
ENTER 3 INTERGERS (512~2048) :512 512 512
THE TIME OF GEMM: 1.251485 s
```

程序能够成功运行，因此封装和调用动态库成功。

四、实验感想

这次实验主要使用MPI对矩阵乘法做了改变和优化，我了解了点对点通信和集合通信的区别与联系。在实验过程中也遇到了许多值得思考的问题，比如使用集合通信时由于new和delete的位置不对，增加并发进程数目时遇到了问题，通过网上查阅资料修改了申请和释放内存的位置，才解决这一问题。

除此之外，封装和生成动态链接库对我来说是新的知识，通过网上搜索和查阅资料有了一些了解，在之后的学习中还应该进一步探索。

总体而言，通过这次实验，我对于MPI程序的编写有了新的理解。

五、参考资料

1.https://zhuanlan.zhihu.com/p/78657463

2.https://en.wikipedia.org/wiki/Strassen_algorithm

3.https://blog.csdn.net/zb1593496558/article/details/79993181

4.https://www.cnblogs.com/bigben0123/p/3325942.html

5.https://www.cnblogs.com/Spiro-K/p/6378576.html