中山大学数据科学与计算机院本科生实验报告

（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**　　批改人：

年级+班级	**18级计7**	**专业（方向）**	**计科**
学号	18340166	**姓名**	王若琪
Email	wangrq29@mail2.sysu.edu.cn	**完成日期**	2020.9.27

---

1. 实验目的

（1）通过 MPI 实现通用矩阵乘法的并行版本，MPI 并行进程（rank size）从1增加至8，矩阵规模从 512 增加至 2048.

（2）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），在 Linux 系统中将此函数编译为 .so 文件，由其他程序调用。

（4）通过本次实验，熟悉并掌握 MPI 并行编程模型，并理解各种通信原理。

（5）通过本次实验，学会库函数的封装过程，并应用实现。

---

2. 实验过程和核心代码

2.1 点对点通信并行矩阵乘法的实现

为了实现并行的通用矩阵乘法，需将矩阵进行划分，然后将每一块分配到不同的进程中进行运算，随后再统计计算结果。这一步中，我使用了点到点的通信方式进行矩阵分块以及计算结果的发送和接收。

首先，并行开始：

```
    MPI_Init(&argc, &argv); // 并行开始
    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，核心代码如下：

```
    int num_lines = M / numprocs; // 计算分块后每一块的行数
```

在 0 号进程中，首先对矩阵 A，B，C 都进行内存分配，并初始化 A 和 B，具体代码见 p2p.cpp。

接下来仍在0号进程中将 B 发送给每一个进程，用点对点的通信方式 MPI_Send()，向每一个进程都发送一个完整的矩阵 B，核心代码如下：

```
for (int i = 1; i < numprocs; i++) // 将 B 发送给其他进程
{
    MPI_Send(B, N * K, MPI_INT, i, 0, MPI_COMM_WORLD);
}
```

0号进程用点对点的通信方式 MPI_Send()，将 A 的分块发送给其他进程，核心代码如下：

```
for (int i = 1; i < numprocs; i++) // 将 A 的分块发送给其他进程
{
    MPI_Send(A + (i - 1) * num_lines * N, N * num_lines, MPI_INT, i, 1,
MPI_COMM_WORLD);
}
```

0 号进程用点对点的通信方式 MPI_Recv()，按顺序接收其他进程发送来的答案，并组合到矩阵 C，核心代码如下：

```
for (int i = 1; i < numprocs; i++)
{
    MPI_Recv(buffer_C, num_lines * K, MPI_INT, i, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    for (int j = 0; j < num_lines; j++)
    {
        for (int l = 0; l < K; l++)
        {
            C[((i - 1) * num_lines + j) * K + l] = buffer_C[j * K + l];
        }
    }
}
```

0 号进程算完没有除尽的矩阵部分，核心代码如下：

```
int lastRow = (num_lines * (numprocs - 1));
if (lastRow < M) // 计算完没有除尽的部分
{
    int remainRows = M - lastRow;
    gemm(A + lastRow * N, B, C + lastRow * K, remainRows, N, K);
}
```

0号进程的最后，用 MPI_Wtime() 计算时间并输出。

其他进程中，先用点对点的通信方式 MPI_Recv() 接收来自 0 号进程的矩阵 B，再用 MPI_Recv() 接收来自 0 号进程的矩阵 A 的分块，再进行 gemm 通用矩阵乘法，最后用 MPI_Send() 发送这一部分的计算结果到 0 号进程中。此部分核心代码如下：

```
else
{
    MPI_Recv(buffer_B, N * K, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);  // 接收来自 0 号进程的矩阵 B
    MPI_Recv(buffer_A, N * num_lines, MPI_INT, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE); // 接收来自 0 号进程的矩阵 A 的分块
    gemm(buffer_A, buffer_B, buffer_C, num_lines, N, K);
    MPI_Send(buffer_C, num_lines * K, MPI_INT, 0, 2, MPI_COMM_WORLD); // 发送
这一部分的计算结果到 0 号进程中
}
```

最后释放内存，输出矩阵及运行时间即可。

并行结束：

```
    MPI_Finalize();
```

2.2 集合通信方式的实现

因为在上文中已经介绍了点对点通信方式的实现，所以这里重点介绍集合通信方式的使用。并且初始化等过程都与点对点通信方式中的一样，这里不做赘述。

集合通信中，需要用到 MPI_Scatter() 将矩阵 A 的分块发送给其他进程，并用到 MPI_Bcast() 将 B 广播给其他进程，然后各进程执行自己的 gemm 过程，并用 MPI_Barrier() 确保他们都执行完，再用 MPI_Gather() 收集每个进程算出的答案，此部分核心代码如下：

```
    MPI_Scatter(A, num_lines * N, MPI_INT, buffer_A, num_lines * N, MPI_INT,
0, MPI_COMM_WORLD);  // 将 A 的分块发送给其他进程
    MPI_Bcast(B, N * K, MPI_INT, 0, MPI_COMM_WORLD); // 将 B 广播给其他进程
    gemm(buffer_A, B, buffer_C, num_lines, N, K);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Gather(buffer_C, num_lines * K, MPI_INT, C, num_lines * K, MPI_INT,
0, MPI_COMM_WORLD);  // 从其他进程中收集数组
```

最后，在 0 号进程中对矩阵中没算完的部分计算结果：

```
    int lastRow = num_lines * numprocs;
    if (rank == 0 && lastRow < M)  // 计算完没有除尽的部分
    {
        int remainRows = M - lastRow;
        gemm(A + lastRow * N, B, C + lastRow * K, remainRows, N, K);
    }
```

最后释放内存并输出矩阵及时间即可。

并行结束：

```
    MPI_Finalize();
```

2.3 封装动态库

先用以下语句进行封装：

```
gcc gemm.c -fPIC -shared -o libgemm.so
gcc main.c -L. -lgemm -o main
```

再将 .so 文件移到 /usr/lib 中去，以便于在不同位置的程序中都能够调用动态库。

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

用 ldd 命令测试是否动态链接成功，命令及输出如下：

```
wrq@wrq-Inspiron-7490:~/Desktop/temp$ ldd ./main
    linux-vdso.so.1 (0x00007fff6e7bf000)
    libgemm.so => /usr/lib/libgemm.so (0x00007f8e97004000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8e96c13000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f8e97409000)
```

---

3. 实验结果

3.1 点对点通信方法测试结果

先编译运行点对点通信矩阵乘法程序，

编译和运行命令如下（以进程数为 1，矩阵规模为 512，512，512为例）：

```
mpicxx p2p.cpp -o p2p
mpirun -np 1 ./p2p 512 512 512
```

令 MPI 并行进程从 1 增加到 8，矩阵规模从 512 增加到 2048，统计各计算时间如下表：

并行进程数 \ 矩阵规模(M,N,K)	512 512 512	1024 1024 1024	2048 2048 2048
1	0.478493 s	3.779454 s	167.387097 s
2	0.541307 s	4.435025 s	121.225100 s
3	0.415344 s	3.939758 s	94.989280 s
4	0.363690 s	3.253953 s	82.586394 s
5	0.406041 s	4.051126 s	129.155472 s
6	0.339566 s	4.233559 s	110.260892 s
7	0.380680 s	3.174078 s	99.023560 s
8	0.340434 s	3.670143 s	90.876045 s

3.2 集合通信方式测试结果

先编译运行集合通信矩阵乘法程序，

编译和运行命令如下（以进程数为 1，矩阵规模为 512，512，512为例）：

```
mpicxx collective.cpp -o collective
mpirun -np 1 ./collective 512 512 512
```

令 MPI 并行进程从 1 增加到 8，矩阵规模从 512 增加到 2048，统计各计算时间如下表：

并行进程数 \ 矩阵规模(M,N,K)	512 512 512	1024 1024 1024	2048 2048 2048
1	0.499899 s	4.982820 s	152.048721 s
2	0.342366 s	3.343490 s	102.437190 s
3	0.209000 s	2.424158 s	70.721167 s
4	0.175314 s	1.863039 s	55.744010 s
5	0.258079 s	3.292307 s	72.704992 s
6	0.226359 s	2.622452 s	62.976411 s
7	0.191614 s	2.310509 s	62.330494 s
8	0.189683 s	2.010206 s	55.762272 s

3.3 比较两种实现方式的性能

两种方式的并行矩阵乘法都较串行时有明显的性能提升。但是两种方式的优化性能有明显差异：由上文中的两个表格对比，可以看出，用集合通信实现矩阵乘法的运行时间在总体上比用点对点通信实现矩阵乘法的运行时间短，无论矩阵规模和进程数大小。所以，针对此类问题，我认为集合通信的性能明显优于点对点通信。

3.4 封装动态库结果

经测试，动态链接成功：

```
wrq@wrq-Inspiron-7490:~/Desktop/temp$ ldd ./main
    linux-vdso.so.1 (0x00007fff6e7bf000)
    libgemm.so => /usr/lib/libgemm.so (0x00007f8e97004000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8e96c13000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f8e97409000)
```

并且该测试程序能够成功执行完毕。

---

4. 实验感想

- 在编写点对点通信实现矩阵乘法的过程中，我遇到指针错误的情况，一开始，我以为是 new 和 delete 的位置出了问题，但是经过仔细检查，并没有任何差错。最后请教两位同学，发现是因为我在程序有一个位置中将 M 和 N 搞混了导致运行出错。这个错误反映出我写代码时对算法没有清

晰的梳理，之后如果遇到类似的实验，我会先将算法在心里或纸上梳理一遍，防止出现这种粗心大意的问题。

- 本次实验我还学会了封装动态链接库，这是我之前的编程学习中完全没有接触过的内容，这次掌握并熟悉了相关操作，经查阅资料动态链接库不仅能够方便编程，还能够提升程序性能，这其中的学问很深，我会继续学习和探索类似问题。