

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 批改人:

年级 + 班级	18 计科 8 班	专业 (方向)	超算
学号	18340208	姓名	张洪宾
Email	2285075600@qq.com	完成时间	2020 年 10 月 1 号

1 实验目的

- 熟悉 MPI 编程
- 充分理解对 MPI 的点ToPoint通信和集合通信的特点，并进行比较
- 熟悉 Linux 的操作，学习如何封装函数为库

2 实验过程、核心代码和实验结果

2.1 实验环境

操作系统: macOS Catalina

编译器: gcc 和 g++

2.2 通过 MPI 实现通用矩阵乘法

在上次我们已经实现了串行版本的通用矩阵乘法。为了和本次实验的性能做比较，我重新跑了一下实验 1 的代码，为了方便，我将 m 、 n 、 k 取为相同的值。串行运行的所需的时间如下：

问题规模 m、n、k	512	1024	2048
运行时间	0.885625s	9.005277s	135.715959s

表 1: 不同进程数量下不同规模的问题的计算时间

可以看到, 当问题规模很大的时候, 所需的时间开销会非常巨大。于是我们想要对这个程序做并行化。

在这里我们先使用点到点的通信方式进行编程。我们很容易分析得到, 矩阵 A 乘以矩阵 B 可以分解为, 矩阵 A 的每一行视为一个矩阵, 与矩阵 B 相乘。事实上, 还可以再将矩阵 B 做分解, 但是我发现那样的通信开销过大, 性能变得很差, 于是只对一个矩阵分解即可。

在这里我们可以使用 Foster 法则, 在划分问题后我们需要考虑通信的问题。在这里我们使用的是点到点的通信方式, 即 0 号进程生成数据后, 将其他进程所需的数据逐个发送给其他进程。然后其他进程计算完毕后, 再逐个发送给 0 号进程做汇总。

在这里主要使用的函数是 MPI_Send 和 MPI_Recv。首先 0 号进程初始化矩阵, 然后 0 号进程将数据发送给各个进程, 各个部分计算完成后, 再将结果发送回来。

关键的函数的代码如下:

```

1 void Send_Matrix(int p){
2     for(int i = 1; i < p; i++){
3         MPI_Send(matrixA[i * m / p], m * n / p, MPI_INT, i, i, MPI_COMM_WORLD);
4         MPI_Send(matrixB[0], k * n, MPI_INT, i, i, MPI_COMM_WORLD);
5     }
6 }
7 void Recv_Matrix(int my_id, int p){
8     MPI_Recv(matrixA[my_id * m / p], m * n / p, MPI_INT, 0, my_id, MPI_COMM_WORLD, &
9         status_p);
10    MPI_Recv(matrixB[0], k * n, MPI_INT, 0, my_id, MPI_COMM_WORLD, &status_p);
11 }
12 void calculate(int p, int my_id){
13     for(int i = my_id * m / p; i < (my_id + 1) * m / p; i++){
14         for(int j = 0; j < k; j++){
15             for(int x = 0; x < n; x++){
16                 res[i][j] += matrixA[i][x] * matrixB[x][j];
17             }
18         }
19     }
20 }

```

```

21
22 void Recv_Res(int p){
23     for(int i = 1; i < p; i++){
24         MPI_Recv(res[i * m / p], m * k / p, MPI_INT, i, i, MPI_COMM_WORLD, &status_p)
25         ;
26     }
27 }
28 void Send_Res(int my_id, int p){
29     MPI_Send(res[my_id * m / p], m * k / p, MPI_INT, 0, my_id, MPI_COMM_WORLD);
30 }

```

在 main 函数中，我定义了一段代码，用于测量运行的时间：每个进程首先记下开始运行的时间，运行结束后再记下运行结束的时间，每个进程都可以计算得到一个局部的运行时间，然后可以用 MPI_Reduce 的方式，取出局部运行时间的最大值，就是整个 MPI 程序计算矩阵乘法所需的时间。这部分代码如下：

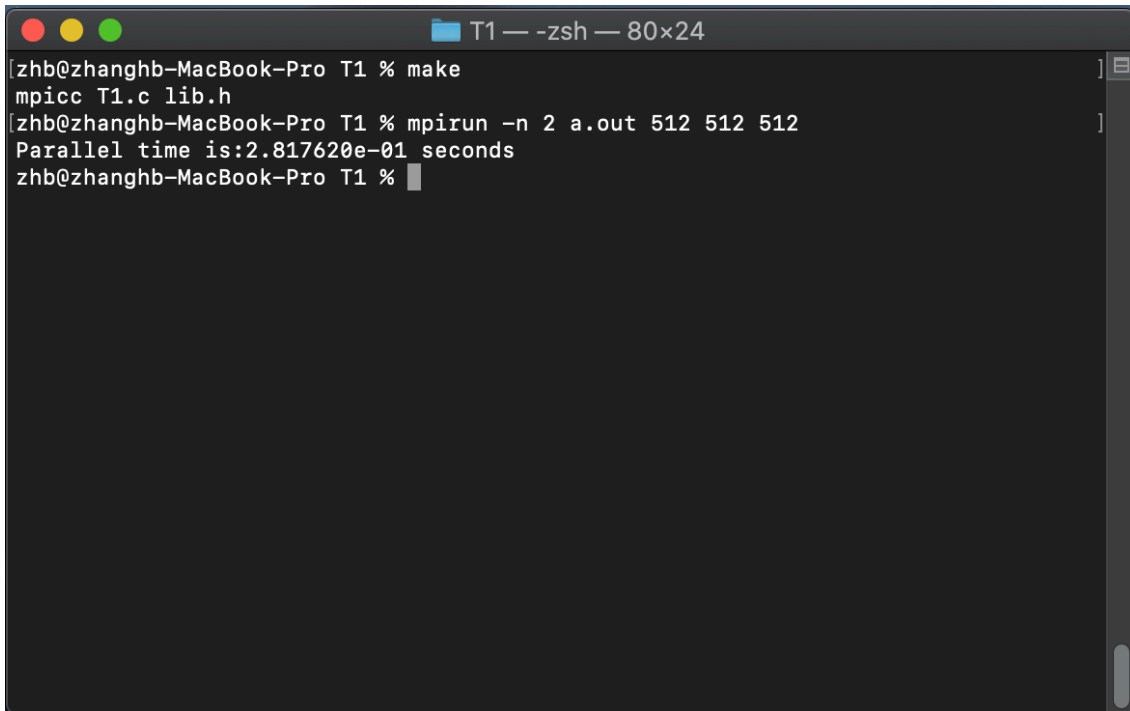
```

1 double my_start, my_end, my_elapsed, elapsed;
2 my_start = MPI_Wtime();
3 // 计算过程
4 ...
5 my_end = MPI_Wtime();
6 my_elapsed = my_end - my_start;
7 MPI_Reduce(&my_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

```

在这里我将代码存储在 lib.h 和 T1.c 两个文件中，然后用 Makefile 编译后生成可执行文件 a.out。然后我采用 main 函数传参数的方法决定 m、n、k 的值。

运行的方式如下：



```
zhb@zhanghb-MacBook-Pro T1 % make
mpicc T1.c lib.h
zhb@zhanghb-MacBook-Pro T1 % mpirun -n 2 a.out 512 512 512
Parallel time is:2.817620e-01 seconds
zhb@zhanghb-MacBook-Pro T1 %
```

图 1: 运行的方式

我先写了一个串行的程序，然后比较了串行执行的结果和并行执行的结果，发现是一致的，因而并行程序可以正确执行，没有问题。这一部分比较简单，在此不表。

在这里可以用-n 指定由多少个进程运行 MPI 程序。由于机器的限制，我只能运行至多 4 个进程。所以我用 2 个进程和 4 个进程分别跑了不同规模的问题，为了方便我取 m 、 n 、 k 为相同的值，结果如下：

线程数量 p	问题规模 n		
	512	1024	2048
2	0.2817620s	4.282550s	46.94138s
4	0.2032560s	2.203474s	26.03343s

表 2: 不同进程数量下不同规模的问题的计算时间

由此可以看到，我们在并行运行的方式下，所需要的开销大大减小了。

2.3 基于 MPI 的通用矩阵乘法优化

在这里我们使用了集合通信。基本的算法仍然不必，但是我们在做通信的时候不再采用点到点的通信方式，而是使用了集合通信。

在这里我使用了 MPI_Bcast, MPI_Scatter 和 MPI_Gather 等函数。基本的算法与第一问相同，但是具体的函数做了替换，关键部分如下：

```
1 void Send_Matrix(int p){
2     MPI_Scatter(matrixA[0], m * n / p, MPI_INT, temp[0], m * n / p, MPI_INT, 0,
3         MPI_COMM_WORLD);
4     MPI_Bcast(matrixB[0], k * n, MPI_INT, 0, MPI_COMM_WORLD);
5 }
6 void calculate(int p){
7     for(int i = 0; i < m / p; i++){
8         for(int j = 0; j < k; j++){
9             for(int x = 0; x < n; x++){
10                temp_res[i][j] += temp[i][x] * matrixB[x][j];
11            }
12        }
13    }
14 }
15
16 void Recv_Res(int p){
17     MPI_Gather(temp_res[0], m * k / p, MPI_INT, res[0], m * k / p, MPI_INT, 0,
18         MPI_COMM_WORLD);
19 }
```

然后我们可以用问题一中的方法进行编译运行，结果如下：

问题规模 n \ 线程数量 p	512	1024	2048
2	0.2442480	4.428831s	35.61942s
4	0.1451760s	2.849212s	22.43200s

表 3: 不同进程数量下不同规模的问题的计算时间

在这里可以看到，相比点到点通信，集合通信的效率更高。这也不难理解，因为我们在这个问题下，如果一次消息传递采用点到点通信，需要做很多次，而采用集合通信则只需要做一

次就行了。

2.4 改造 Lab1 成矩阵乘法库函数

这个地方的难点是要理解动态链接方式。动态链接，在可执行文件装载时或运行时，由操作系统的装载程序加载库。大多数操作系统将解析外部引用（比如库）作为加载过程的一部分。我们平时调用库一般也是用动态链接方式实现的。

Linux/macOS 下动态库文件的文件名形如 libxxx.so，其中 so 是 Shared Object 的缩写，即可以共享的目标文件。在链接动态库生成可执行文件时，并不会把动态库的代码复制到执行文件中，而是在执行文件中记录对动态库的引用。程序执行时，再去加载动态库文件。如果动态库已经加载，则不必重复加载，从而能节省内存空间。

通过查阅资料，我大概了解了 Linux 下用 gcc 生成和使用动态库的步骤：

- 编写源代码
- 将一个或多个文件共同编译链接，gcc 中要使用 -fPIC -shared 参数来生成共享库，来生成 libxxx.so。
- 通过 -L<path> -lxxx 的 gcc 选项链接生成的 libxxx.so。
- 把 libxxx.so 放入链接库的标准路径，或指定 LD_LIBRARY_PATH，才能运行链接了 libxxx.so 的程序。

于是在 Lab 1 的基础上，我做了适当地修改，类似于 C 语言的函数库，将函数拆成 matrix.h 用来存放函数原型和 matrix.c 用来存放函数体，以适应动态链接的方式。

然后使用如下命令：

```
gcc -fPIC -shared -o libmatrix.so matrix.c
```

用于生成共享库。

然后我写了一个 test.c 来调用这个库，用 #include "matrix.h" 来引用这个库，然后后面的代码与 Lab 1 大同小异，不再赘述。

然后用这条命令：

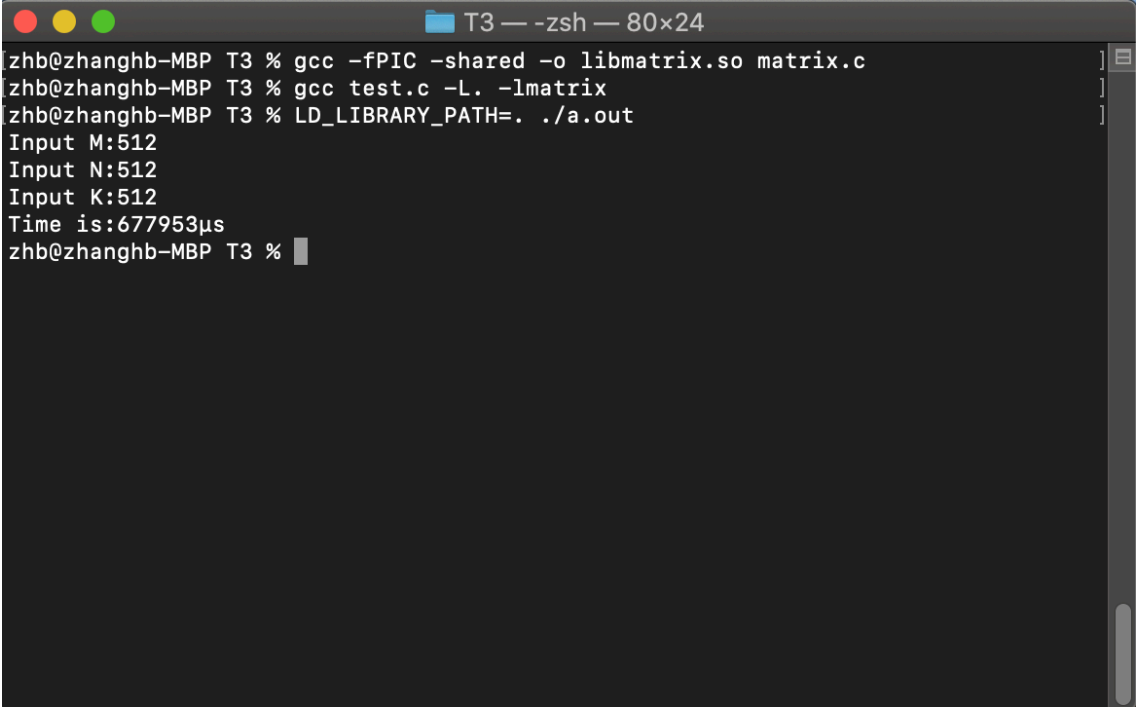
```
gcc test.c -L. -lmatrix
```

这样可以生成可执行文件 a.out。

然后输入

```
LD_LIBRARY_PATH=. ./a.out
```

可以执行程序。

A terminal window titled 'T3 - zsh - 80x24' showing the following commands and output:

```
zhb@zhanghb-MBP T3 % gcc -fPIC -shared -o libmatrix.so matrix.c
zhb@zhanghb-MBP T3 % gcc test.c -L. -lmatrix
zhb@zhanghb-MBP T3 % LD_LIBRARY_PATH=. ./a.out
Input M:512
Input N:512
Input K:512
Time is:677953µs
zhb@zhanghb-MBP T3 %
```

图 2: 运行结果

在这里如果不输入 `LD_LIBRARY_PATH=.` 会输出找不到库文件的报错信息。

也可以把共享库的路径放入环境变量中，就不用指定库的位置也可以运行程序了。

为了方便起见，我将上述过程写入 Makefile，输入 `make build` 可以编译得到共享库，输入 `make test` 可以运行，输入 `make clean` 可以清除生成的文件。

3 实验感想

这次实验难度不是特别难，但是还是收获颇丰。

- 通过对 MPI 的编程，对点到点通信和集合通信的方式有了更加深入的理解。
- 之前没有接触过编译原理，通过这次作业了解了静态链接和动态链接的差别，也熟悉了 macOS/Linux 一些相关操作，学会了自己写共享库，对计算机体系结构的理解加深了。

总而言之这次作业收获很多，我觉得非常有意义。