中山大学计算机院本科生实验报告

2022 学年秋季学期

课程名称：并行程序设计 批改人：

实验	并行程序设计（1）：通过MPI实现通用矩阵乘法	专业（方向）	计算机科学与技术（人工智能与大数据方向）
学号	20337025	姓名	崔璨明
Emai	897369624@qq.com	完成日期	2023/3/18

1.实验目的

1. 通过MPI实现通用矩阵乘法：通过MPI点对点通信的方式实现通用矩阵乘法（Lab1），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。尝试用`mpi_type_create_struct`聚合MPI进程内变量后通信。
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。
4. 构造MPI版本矩阵乘法加速比和并行效率表：分别构造MPI版本的标准矩阵乘法和优化后矩阵乘法的加速比和并行效率表格。并分类讨论两种矩阵乘法分别在强扩展和弱扩展情况下的扩展性。

2.实验过程和核心代码

2-1 通过MPI点对点通信的方式实现通用矩阵乘法

通过MPI点对点通信的方式来实现并行的通用矩阵乘法的思路大致为：在主进程中将矩阵划分成不同大小的部分，再将每一部分分发给子线程进行计算，子进程全部计算结束后，主进程统计计算结果。可以手动对矩阵进行划分，然后通过MPI中的`MPI_Recv()`函数和`MPI_Send()`函数来实现点对点的通信。核心代码解析如下：

```
MPI_Init(NULL,NULL);//初始化MPI
MPI_Comm_rank(MPI_COMM_WORLD,&pid);//获取当前进程的进程id
MPI_Comm_size(MPI_COMM_WORLD,&process_num);//进程数目
line = M/process_num;//将数据划分为对应的行数
```

在主进程中，对矩阵进行初始化（为随机数），然后通过`MPI_send()`将矩阵A分块发送，将矩阵B整个发送给子进程。再接受子进程的计算结果，将计算结果传递给矩阵C，计算剩下的数据：

```
//send matrix N to sub processes
for (int i=1;i<process_num;i++){
```

```
    MPI_Send(b,N*K,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
}
// send each row of A to sub processes
for (int i=1;i<process_num;i++){
    MPI_Send(a+(i-1)*line*N,N*line,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
}
// receive result
for (int i=1;i<process_num;i++){
    MPI_Recv(result,line*K,MPI_DOUBLE,i,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    for(int l=0;l<line;l++){
        for(int k=0;k<K;k++){
            c[((i-1)*line+l)*K+k]=result[l*K+k];
        }
    }
}
//计算剩下的
for (int i=(process_num-1)*line;i<M;i++){
    for (int j=0;j<K;j++){
        double tmp=0;
         for (int k=0;k<N;k++)
            tmp += a[i*N+k]*b[k*K+j];
         c[i*K+j] = tmp;
    }
}
```

在子进程中，调用`MPI_Recv()`接受从主进程传来的矩阵，进行计算，计算结果存在矩阵`result`中：

```
MPI_Recv(b,N*K,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);//接收矩阵B
MPI_Recv(temp,N*line,MPI_DOUBLE,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);//接收a的分
块
//计算结果
for(int i=0;i<line;i++){
    for(int j=0;j<N;j++){
        double tmp=0;
        for(int k=0;k<N;k++)
            tmp += temp[i*N+k]*b[k*K+j];
        result[i*K+j] = tmp;
    }
}
//传回主进程
MPI_Send(result, line*K, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

2-2 基于MPI的通用矩阵乘法优化

上述的实现是采用了点对点通信的方式，接下来实现MPI集合通信的通用矩阵乘法。思路和上述大致相同，只不过主进程不需要和每个子进程进行一对一的信息传输，分发A时，可以采用`MPI_Scatter()`将A平均地分配给各个进程；采用`MPI_Bcast()`将B直接广播给所有进程；收集计算结果时可以使用`MPI_Gather()`。主进程关键代码如下：

```
//send division of a to sub process
MPI_Scatter(a, line*N, MPI_DOUBLE, local_matrix, line*N, MPI_DOUBLE, 0,
MPI_COMM_WORLD );
//broadcast b to every process
MPI_Bcast(b, N*K, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
//calculate local results
for(int i= 0; i< M;i++){
    for(int j=0;j<N;j++){
        double tmp = 0;
        for(int k=0;k<N;k++)
            tmp += a[i*N+k] * b[k*K+ j];
        result[i*K+ j ] = tmp;
    }
}
//确保子进程都执行完成
MPI_Barrier();
//Collect data into a process
MPI_Gather( result, line*K, MPI_DOUBLE, c, line*K, MPI_DOUBLE, 0, MPI_COMM_WORLD
);
//计算剩余部分的结果
for(int i = (process_num-1)*line;i<M;i++){
    for(int j=0;j<N;j++){
        double tmp = 0;
        for(int k=0;k<N;k++)
            tmp += a[i*N+k]*b[k*K+j];
        c[i*K+j] = tmp;
    }
}
```

2-3 改造Lab1成矩阵乘法库函数

我将Lab1的矩阵乘法程序gemm.c进行了改进，将其分为了三个程序：定义矩阵乘法函数的头文件`matrix_multiply.h`、实现矩阵乘法的文件`matrix_multiply.c`和一个测试文件`test.c`，该测试文件用于测试标准的库函数`matrix_multiply`是否成功生成并能直接调用：

`matrix_multiply.c`文件：

lab2 > C matrix_multiply.c > matrix_multiply(double **, double **, double **, int, int, int)

```
#include "matrix_multiply.h"
#include<stdio.h>
#include<stdlib.h>
void matrix_multiply(double**A,double**B,double**C,int M,int N,int K){
    for(int m=0;m<M;m++){
        for(int k=0;k<K;k++){
            for(int n=0;n<N;n++){
                C[m][k]+=A[m][n]*B[n][k];
            }
        }
    }
}
```

`matrix_multiply.h`文件：

lab2 > C matrix_multiply.h > ...

```
#ifndef matrix_multiply_h
#define matrix_multiply_h
#include<stdio.h>
#include<stdlib.h>
void matrix_multiply(double**A,double**B,double**C,int M,int N,int K);
#endif
```

生成步骤如下：

1、使用`-fPIC`编译为位置独立的代码，并通过`-shared`生成动态链接库：

`gcc matrix_multiply.c -fPIC -shared -o libmatrix_multiply.so`

2、告诉编译器要链接的库，编译测试程序`test.c`：

`gcc test.c -L. -lmatrix_multiply -o test`

3、运行程序，可以成功执行，说明Lab1的矩阵乘法成功地改造成为了一个标准的库函数`matrix_multiply()`。

4、此外，通过ldd命令可以查看是否正确链接：

```
cui@cui-VirtualBox:~/parall/lab2$ ldd ./test
        linux-vdso.so.1 (0x00007ffe5e352000)
        libmatrix_multiply.so => /usr/lib/libmatrix_multiply.so (0x00007fcbf46d6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fcbf42e5000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fcbf4adb000)
cui@cui-VirtualBox:~/parall/lab2$
```

2-4 构造MPI版本矩阵乘法加速比和并行效率表

该部分见实验结果部分。

3.实验结果

执行命令：

```
//编译
mpicxx mpi_gemm_1.cpp -o mpi1
//执行，-np后面代表并行的线程数，后面三个参数分别表示M、N、K
mpiexec -np 2 ./mpi1 128 128 128
```

MPI点对点通信实验结果（括号内为加速比，运行时间（ms））：

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	(1, 12.17ms)	(1, 118.13ms)	(1, 969.1ms)	(1, 45710.1ms)	(1, 522257ms)
2	(1, 12.08ms)	(0.9, 132.19ms)	(1.1, 904.22ms)	(1.2, 37361.2ms)	(1.2, 440346ms)
4	(0.94, 12.93ms)	(1.13, 104.23ms)	(1.2, 778.90ms)	(1.3, 35855.6ms)	(1.3, 423514ms)
8	(0.96, 12.56ms)	(0.82, 142.87ms)	(0.8 , 1204.43ms)	(0.76, 59584.4ms)	(0.8, 653782ms)
16	(1, 12.16ms)	(0.25, 472.05ms)	(0.75, 1287.15ms)	(0.62, 73261.8ms)	(0.78, 640075ms)

MPI集合通信实验结果（括号内为加速比，运行时间（ms））：

Comm_size (num of processes)	Order of Matrix (Speedups, milliseconds)				
	128	256	512	1024	2048
1	(1, 11.31ms)	(1, 67.71ms)	(1, 745.72ms)	(1, 25867.9ms)	(1,265940ms)
2	(1.37, 8.20ms)	(1.1, 62.83ms)	(1.26, 589.87ms)	(1.2, 22373.7ms)	(1.1, 245343ms)
4	(0.54, 21.10ms)	(1.2, 56.25ms)	(1.62, 458.77ms)	(1.98, 13038.1ms)	(1.7, 157321ms)
8	(0.33, 34.44ms)	(0.77, 86.74ms)	(1.78, 417.07ms)	(1.91, 13536.4ms)	(1.45, 183487ms)
16	(0.19, 58.28ms)	(0.49,135.74ms)	(1.64, 452.54ms)	(1.59, 16181.3ms)	(1.7, 156355ms)

通过表格进行对比可知，MPI实现的并行的矩阵乘法都比串行的矩阵乘法有明显的性能提升。由实验数据显示，当矩阵的规模较大时，集合通信的MPI矩阵乘法要比点对点通信的MPI矩阵乘法速度更快，表现更好。

从实验结果中可以看到，**矩阵乘法的性能是随着并行的进程数的增加而提升的，且当矩阵较大时提升更加明显**，但由于cpu核心数目的限制（我只给进行实验的虚拟机分配了四个cpu核，如下图所示），因此电脑核心数不够导致扩展到8核心或16核心时，便不会带来性能提升。

```
cui@cui-VirtualBox:~$ cat /proc/cpuinfo | grep "physical id" | sort | uniq | wc
-l
1
cui@cui-VirtualBox:~$ cat /proc/cpuinfo | grep "physical id"
physical id     : 0
physical id     : 0
physical id     : 0
physical id     : 0
cui@cui-VirtualBox:~$ cat /proc/cpuinfo| grep "cpu cores"| uniq
cpu cores       : 4
cui@cui-VirtualBox:~$
```

我们可以看到，对于MPI点对点通信实现的矩阵乘法，它的强扩展性能通常比较好，因为每个进程只需要与其他进程进行点对点通信，通信量相对较小。随着进程数量的增加，点对点通信的数量和数据量也会增加，但由于每个进程与其他进程通信的数据量相对较小，所以总体的通信开销不会增长太快。因此，MPI点对点通信实现的矩阵乘法在强扩展情况下的表现通常比较好。

而对于MPI集合通信实现的矩阵乘法在弱扩展情况下的表现可能会更好一些。虽然每个进程需要与所有其他进程进行通信，但通信模式相对固定，通信量较大但是可控。

改造Lab1成矩阵乘法库函数：

可见程序正确地动态链接到了该库并正确运行。

```
cui@cui-VirtualBox:~/parall/lab2$ gcc matrix_multiply.c -fPIC -shared -o libmatrix_multiply.so
cui@cui-VirtualBox:~/parall/lab2$ gcc test.c -L. -lmatrix_multiply -o test
cui@cui-VirtualBox:~/parall/lab2$ ./test
input three integer(512 ~2048):
2 3 4
s
e

matrix_1:
49.30 86.20 3.20
26.70 72.90 22.60

matrix_2:
55.20 73.90 29.80 50.10
78.70 74.00 37.10 41.00
12.20 68.20 47.70 9.50

result:
9544.34 10240.31 4819.80 6034.53
7486.79 8909.05 4578.27 4541.27

using time: 0.021000  ms
```

4.实验感想

由于我对MPI编程不是很熟悉，因此本次实验对我来说是一个不小的挑战，但在完成实验的过程中，我学到许多新的知识，如点对点通信和集合通信的流程以及实现、MPI的各种接口的调用、如何将自己编写的程序生成动态链接库等等。

除此之外，本次实验过程中我还遇到了一个c语言常见的问题，即如何将二维数组作为实参传入函数中。若二维数组是`melloc`动态申请的，则传入的参数要写为`double**`，若为静态的`double a[][]`，则传入的参数应该为`(double*a)[]`，即指针数组。一开始没有注意到这个小问题，导致编译出来的动态链接库调用时会产生段错误的问题。