中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　**批改人：**

实验	基于 MPI 的并行矩阵乘法	专业（方向）	计算机科学与技术
学号	21307082	姓名	赖耿桂
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.3.27

1. 实验目的（200 字以内）

①使用 MPI 点对点通信实现并行矩阵乘法，熟悉 MPI 的使用及工作原理；

②通过设置不同的进程数量（1-16）和矩阵规模（128-2048），比较不同情况下的运行时间，并分析程序的并行性能。

2. 实验过程和核心代码（600 字以内，图文并茂）

首先，我在头文件 Generator.h 中定义了一个函数 random_matrix_generator，用来随机生成了大小 rows*cols 的矩阵(不过是使用一维数组来存储)，代码如下：

```
vector<double> random_matrix_generator(int rows, int cols){
    vector<double> mat(rows*cols);

    // 初始化随机数生成器
    random_device rd;
    default_random_engine eng(rd());
    uniform_real_distribution<double> distr(-50.0,50.0);

    // 赋值
    for(int i = 0; i < rows*cols; ++i){
        mat[i] = distr(eng);
    }

    return mat;
}
```

然后在文件 lab1_main.cpp 中，定义了一个打印输出矩阵的函数 printMatrix。通过传入矩阵的行数 rows 和列数 cols，使得一维数组 mat 可以按照 rows*cols 的规格打印输出矩阵，代码如下：

```
void printMatrix(vector<double> mat, int rows, int cols){
    for(int i = 0; i < rows; ++i){
        for(int j = 0; j < cols; ++j){
            printf("%lf ", mat[i * cols + j]);
        }
        printf("\n");
    }
    return;
}
```

在 main 函数中，声明定义了如下变量：

```
// 在运行时直接在最后输入参数：M，N，K，process_num(线程数量)
int M = atoi(argv[1]), N = atoi(argv[2]), K = atoi(argv[3]), process_num = atoi(argv[4]), myrank, rows_per_process;
// myrank：进程号；rows_per_process：发送给除 0 号进程外的其他进程的矩阵 A 的行数

vector<double> A(M * N), B(N * K), C(M * K);     // 矩阵 A、B、C，C=AB
vector<double> temp_C(M * K);        // 用来存每个非 0 号进程的计算结果
double start_time, end_time;     // 记录并行矩阵乘法的开始时刻和结束时刻
```

在终端执行该程序时，直接在命令末尾输入 M、N、K 以及 process_num，故 main 函数要有 argc 和 argv 两个参数：

```
int main(int argc, char* argv[])
```

接下来是整个实验最关键的过程——使用 MPI 点对点通信实现并行矩阵乘法：首先初始化 MPI，并获取通信子的进程数量和进程在通信子的编号，并求出每个进程划分出来的矩阵 A 的行数：

```
MPI_Init(&argc, &argv);      // MPI 初始化
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);      // 获取通信子的进程数
MPI_Comm_size(MPI_COMM_WORLD, &process_num);     // 获取进程在通信子中的编号
rows_per_process = M / process_num;      // 每个进程计算的矩阵 A 分块的行数
```

然后对于 0 号进程即 myrank 等于 0 时，需要先随机初始化矩阵 A 和 B，并输出它们：

```
// 调用随机矩阵生成函数 random_matrix_generator，初始化矩阵 A 和 B
A = random_matrix_generator(M , N);
B = random_matrix_generator(N , K);

// 输出矩阵 A、B
printMatrix(A, M, N);
printMatrix(B, N, K);
```

由于其他进程需要使用矩阵 A 的分块和矩阵 B 来进行矩阵乘法运算，因此，需要将矩阵 A 的分块和矩阵 B 发送给其他进程，注意使用 tag 参数来进行区分(此时开始计时)：

```
start_time = MPI_Wtime();   // 开始计时
// 将矩阵B发送至其他进程
for(int i = 1; i < process_num; ++i){
    MPI_Send(B.data(), N *K, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
}
// 将矩阵A分块发送至其他进程
for(int i = 1; i < process_num; ++i){
    MPI_Send(A.data() + rows_per_process * (i-1) * N, rows_per_process * N, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
}
```

此外，0 号矩阵还需要接收其他矩阵的运算结果，并将其存到矩阵 C 对应的位置：

```
// 接收来自其他进程的运算结果
for(int i = 1; i < process_num; ++i){
    MPI_Recv(temp_C.data(), rows_per_process * K, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for(int j = 0; j < rows_per_process; ++j){
        for(int k = 0; k < K; ++k){
            C[((i-1)*rows_per_process + j) * K + k] = temp_C[j * K + k]; // 将其他进程的运算结果(temp_C)存到矩阵C中
        }
    }
}
```

需要注意的是，由于 0 号进程不参与矩阵 A 的分块的相关计算，因此实际上矩阵 A 还有一部分没有参与运算，因此需要进行剩余部分的计算(此时完成了矩阵乘法，结束计时)，算出运算时间并打印输出矩阵 C 和运算时间 delta_time：

```
/* 把剩余部分算完,这是因为共有process_num个进程,除了0号进程,剩余(process_num-1)个进程
    计算了(process_num-1)*(M/process_num)行，而A共有M行，因此还需要单独计算剩余的[M-(process_num-1)*(M/process_num)]行
    下为通用矩阵乘法 */
for(int i = (process_num - 1) * rows_per_process; i < M; ++i){
    for(int k = 0; k < K; ++k){
    double tmp = 0;
    // 使用一个变量tmp是为了防止多线程环境下可能同时对 C[i * K + k]进行访问和修改，导致其值被覆盖，导致运算错误
        for(int j = 0; j < N; ++j){
            tmp += A[i * N + j] * B[j * K + k];
```

```
            }
            C[i * K + k] = tmp;
        }
    }
    end_time = MPI_Wtime();      // 结束计时
    double delta_time = end_time - start_time;  // 求出矩阵乘法所用时间
    printMatrix(C, M, K);        // 输出矩阵 C
    printf("\n 在%d 个进程并行的情况下计算大小为%d*%d 的矩阵 A 与大小为%d*%d 的矩阵 B 的乘积所用时间为：%lf s.\n", process_num, M, N, N, K, delta_time);
```

接下来是其他进程即 myrank 不为 0 时，首先要接收 0 号进程发送过来的矩阵 B 及矩阵 A 的分块：

```
    // 接收 0 号进程发过来的矩阵 B
    MPI_Recv(B.data(), N * K, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    vector<double> temp_A(rows_per_process * N);      //用来存储矩阵 A 的分块
    // 接收 0 号进程发过来的矩阵 A 的分块
    MPI_Recv(temp_A.data(), rows_per_process * N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

然后使用矩阵 A 的分块和矩阵 B 进行通用矩阵乘法运算，并将运算结果发给 0 号进程：

```
    // 用 0 号进程发送过来的矩阵 A 的分块和矩阵 B 进行通用矩阵乘法运算
    for(int i = 0; i < rows_per_process; ++i){
        for(int k = 0; k < K; ++k){
            double tmp = 0;
            // 使用一个变量 tmp 是为了防止多线程环境下可能同时对 temp_C[i * K + k]进行访问和修改，导致其值被覆盖，导致运算错误
            for(int j = 0; j < N; ++j){
                tmp += temp_A[i * N + j] * B[j * K + k];
            }
            temp_C[i * K + k] = tmp;
        }
    }
    MPI_Send(temp_C.data(), rows_per_process * K, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);    // 将运算结果发给 0 号进程
```

MPI 结束：

```
    MPI_Finalize();      // MPI 结束
```

3. 实验结果（500 字以内，图文并茂）

在终端进行编译：

```
mpicxx -g -Wall -o lab1_main lab1_main.cpp
```

然后运行(这里以两个 256*256 的方阵、4 个进程为例)：

```
2.  mpiexec -n 4 ./lab1_main 256 256 256 4
```

(注：最后的四个参数分别为 M、N、K、process_num)

注：

- 下表中矩阵规模为 X 指的是两个大小为 X*X 的方阵进行矩阵乘法
- 由于此实验结果较多，考虑到如果放终端的输出截图会很多，因此只将不同规模的矩阵在不同进程数下的矩阵乘法所用的时间填入下表。
- 下为运行示例，仅展示程序的正确性：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab1$ mpicxx -g -Wall -o lab1_main lab1_main.cpp
laigg@laigg-VirtualBox:~/codes/parallel/lab1$ mpirun -np 1 ./lab1_main 4 4 4 1
Matrix A:
43.950427 14.242636 -48.159798 7.335317
-24.064385 38.368093 -46.420910 -25.049454
46.803631 -45.329136 -7.177331 -26.938043
-28.405200 1.797058 11.611387 0.980137

Matrix B:
-28.591411 47.616534 -25.717888 -2.724265
-35.848783 9.848825 36.368612 19.423643
-48.904260 40.982632 28.756984 -48.206515
-10.799685 35.290606 8.334425 37.042823

Matrix C:
508.814267 518.192740 -1936.122156 2750.248109
1853.291749 -3554.443422 470.581276 2120.693690
928.736948 537.382173 -3283.159780 -1659.829488
169.290968 -824.403441 1137.955574 -411.148767


在1个进程并行的情况下计算大小为4*4的矩阵A与大小为4*4的矩阵B的乘积所用时间为：0.000001 s.
laigg@laigg-VirtualBox:~/codes/parallel/lab1$
```

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.025451s	0.177677s	0.952542s	14.133127s	202.594073s
2	0.013014s	0.105095s	0.885947s	14.236494s	198.659364s
4	0.026541s	0.068138s	0.569275s	7.230700s	110.573228s
8	0.216977s	0.242752s	0.790271s	6.005270s	88.905721s
16	0.380163s	0.701426s	1.511931s	6.965474s	83.117775s

1. 对于规模较大的矩阵如 2048*2048，使用多进程并行对矩阵乘法运算的优

化效果较好，而对于规模较小的矩阵如 128*128、256*256 等(512*512 和 1024*1024 的矩阵也出现了进程数增加但是运行时间不减反增的情况)，使用多进程并行对矩阵乘法的优化效果比较差，甚至随着进程数的增加，矩阵乘法所用的时间反而增加。这可能是因为对于规模较小的矩阵，使用多进程并行对其进行优化时，并行部分对性能的提高远不及多进程带来的通信开销导致的性能下降。

2. 对于 256*256、512*512 的矩阵，在进程数为 4 的时候矩阵乘法所用的时间最少，这是因为虚拟机有 4 个 CPU，每个 CPU 有一个核，在进程数为 4 时对 CPU 的利用最充分。当进程数多于 4 时，会产生竞争现象，上下文切换更加频繁，且通信开销随着进程数增加而增大，这是导致它们进程数提高但是运行时间不减反增的原因。

4. 实验感想（200 字以内）

在本次实验中，我使用 MPI 点对点通信实现并行矩阵乘法，由于此前没有使用过 MPI 编程，因此在理解点对点通信流程及实现、调用 MPI 的各种接口时遇到了一定的困难。但在完成实验之后，我对点对点通信有了更深入的理解，初步掌握了 MPI 编程。

此外，在分析不同规模的矩阵乘法在不同进程下的性能时，我也认识到矩阵规模、设备的硬件条件、进程间的通信开销等对性能也会产生较大影响，而并不是进程越多性能就一定越高。

讨论题：

-在内存受限情况下，如何进行大规模矩阵乘法计算？

①可以对矩阵进行分块，就像本次实验一样，然后依次对每批分块使用多进程并行处理。

②可以考虑进行内存优化，例如，如果在一个循环中多次使用相同的数组元素，可以将这些元素保留在缓存中，而不是在每次迭代中重新加载。

③如果是稀疏矩阵，可以对其进行压缩，减少 0 参与的运算。

-如何提高大规模稀疏矩阵乘法性能？

①可以对该矩阵进行压缩，避免对 0 进行不必要的运算。

②在条件允许的情况下可以进行向量化优化。

③可以对矩阵进行分块，分块后可以选择进行压缩或向量化等优化处理后，再使用多进程处理。