中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

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	--	姓名	--
Email	--	完成日期	2025.3.27

实验目的

使用MPI点对点通信实现并行矩阵乘法
设置线程数量(1-16)及矩阵规模(128-2048)
根据运行时间, 分析程序并行性能

实验过程和核心代码

采取为每个进程分配计算矩阵指定行数乘法来实现并行. 假设进程数为p, 矩阵大小为$n \times n$, 计算方法如下:

$$row = ceil(\frac{n}{p})$$

在无法整除的情况下, 最后一个进程计算的行数要额外确定.
在通信时要确定进程计算的起始行, 对每个进程i, 起始行即为$i \times n$. 我在实现时使用了三次发送.

- 第一次发送i进程需要处理的行数以及每行元素的个数
- 第二次发送相应行的mat1矩阵元素
- 第三次发送整个mat2矩阵元素

代码如下:

```
if(size > 1) {
    for(int i = 1; i < size; ++i) {
        begin_row = row_per_p * i;
        end_row = min(row_per_p * (i + 1) - 1, m - 1);
        // msg[0] 处理行数, msg[1] 一行元素个数
        int msg[2] = {end_row - begin_row + 1, m};
        // 需要处理几行
        printf("process: %d, calculate %d rows(%d to %d)\n", i, msg[0], begin_row, end_row);
        MPI_Send(&msg[0], 2, MPI_INT, i, 1, MPI_COMM_WORLD);
        // 处理对应行mat1
        MPI_Send(&mat1[begin_row*m], msg[0] * m, MPI_DOUBLE, i, 2, MPI_COMM_WORLD);
        // 整个mat2
        MPI_Send(mat2, m * m, MPI_DOUBLE, i, 3, MPI_COMM_WORLD);
    }
}
```

这里预留了$0 \sim row_per_p - 1$行, 给主进程进行计算.

```
begin_row = row_per_p * 0;
end_row = min(row_per_p * (0 + 1) - 1, m - 1);
printf("process: %d, calculate %d rows(%d to %d)\n", 0, end_row - begin_row + 1, begin_row, end_row);
matMul(ans, mat1, mat2, end_row - begin_row + 1, m);
```

子进程接收对应的数据并进行计算:

```
// 子进程
if(rank != 0) {
    int msg[2];
    // 接收行数及每行元素个数
    MPI_Recv(&msg, 2, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
    double* mat1 = new double[msg[0] * msg[1]];
    double* mat2 = new double[msg[1] * msg[1]];
    double* ans = new double [msg[0] * msg[1]](); // 初始化为0
    // 接收 msg[0] 行mat1
    MPI_Recv(&mat1[0], msg[0] * msg[1], MPI_DOUBLE, 0, 2, MPI_COMM_WORLD, &status);
    // 接收 整个 mat2
    MPI_Recv(&mat2[0], msg[1] * msg[1], MPI_DOUBLE, 0, 3, MPI_COMM_WORLD, &status);
    // 计算
    matMul(ans, mat1, mat2, msg[0], msg[1]);
    // 发送结果
    MPI_Send(&ans[0], msg[0] * msg[1], MPI_DOUBLE, 0, 4, MPI_COMM_WORLD);

    delete mat1;
    delete mat2;
    delete ans;
}
```

等待主进程将自己的部分计算完成后就开始接收子进程的结果:

```
// 主进程计算部分
begin_row = row_per_p * 0;
end_row = min(row_per_p * (0 + 1) - 1, m - 1);
printf("process: %d, calculate %d rows(%d to %d)\n", 0, end_row - begin_row + 1, begin_row, end_row);
matMul(ans, mat1, mat2, end_row - begin_row + 1, m);
// 接收结果
if(size > 1) {
    for(int i = 1; i < size; ++i){
        begin_row = row_per_p * i;
        end_row = min(row_per_p * (i + 1), m - 1);
        MPI_Recv(&ans[begin_row*m], (end_row - begin_row + 1)*m, MPI_DOUBLE, i, 4, MPI_COMM_WORLD, &status);
    }
}
```

其他函数实现如下:

```
// 获取随机数矩阵
void getRandomMat(double* mat, int m, int n) {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> distrib(1, 100);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            mat[i*n+j] = distrib(gen);
        }
    }
}
// 矩阵乘法 rows行
void matMul(double* ans, double* mat1, double* mat2, int rows, int n) {
    for(int i = 0; i < rows; ++i) {
        for(int j = 0; j < n; ++j) {
            for(int k = 0; k < n; ++k)
                ans[i*n + j] += mat1[i*n + k] * mat2[k*n + j];
        }
    }
}
// 打印前n行, 前n列结果
void showMatCorner(double* mat, int m, int n) {
    for(int i = 0; i < n; ++i) {
        for(int j = 0; j < n; ++j) {
            printf("%12.3f ", mat[i*m + j]);
        }
        cout << endl;
    }
}
```

实验结果

虚拟机核数: 4
时间单位: s
相乘矩阵mat1, mat2的大小都为: 矩阵规模 × 矩阵规模

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.00764	0.06666	0.65120	7.79679	180.78437
2	0.00384	0.04135	0.43442	7.50840	95.39312
4	0.00320	0.04005	0.29141	4.58821	66.43409
8	0.01941	0.04718	0.33140	4.65259	66.59794
16	0.04450	0.08319	0.42977	6.55712	66.57667

结果分析

在进程数小于虚拟机核数4时, 在2048大规模的矩阵下达到了较好的加速比. 在小规模矩阵上由于进程创建, 进程间通信等开销导致加速比较小.

当进程数设置为4, 即虚拟机拥有核数时, 在所有规模的矩阵上都取得了最快的运行速度.

当进程数超过4时, 可以看到小规模矩阵运行时间明显变大. 这是由于多个进程程共用同一个核, 需要操作系统进行调度, 产生额外开销, 在小规模矩阵时影响较明显, 大规模矩阵下影响较小.

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. 可以使用分块矩阵乘法, 将大规模矩阵分割成小矩阵, 然后对其进行乘法运算. 这样可以在内存受限时逐部分计算, 从而得到最后的结果. 但这样需要分割的额外开销.

> 2. 若能完整存储行列, 则可以每次取一行和每列进行计算. 不必将整个矩阵加载到内存内.

- 如何提高大规模稀疏矩阵乘法性能

> 1. 使用多进程/多线程计算方式加速计算

> 2. 可以使用向量化操作, 利用指令级并行(多发射, 流水线)来加速计算

> 3. 使用高效的存储格式(CSR, CSC, COO)等减少内存占用并加速计算.

实验感想

在本次实验中使用了MPI来实现矩阵乘法的并行化. 提高了对MPI的熟悉程度, 对不同线程下矩阵乘法的性能有了一个直观的感受.