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	MPI矩阵乘法进阶	专业(方向)	计算机科学与技术
学号	--	姓名	--
Email	--	完成日期	2025.4.8

实验目的

使用MPI集合通信实现并行矩阵乘法
使用mpi_type_create_struct聚合进程内变量后通信

实验过程和核心代码

首先定义需要使用的变量

```
double* mat1 = nullptr;
double* mat2 = nullptr;
double* ans = nullptr;
Messages* message = nullptr;

int m = 0;  // matrix size m*m
int avg_row = 0;

double begin, end; // timer
```

这里的`Message`为聚合的变量类, 包含每个进程处理的mat行数`avg_row`以及`mat2`矩阵变量, 由于`mat2`的大小是由用户输入决定的, 所以这里`mat2`在使用mpi_type_create_struct时需要提前知道矩阵大小`m`的值, 所以无法将`m`也包含在此聚合变量中. `m`仍需单独使用一个广播发送. 如下

```
class Messages {
public:
    int avg_row;
    double mat2[];

    static MPI_Datatype message_type;

    static void buildMPIType(int m) {
        int block_lengths[2] = {1, m * m};
        MPI_Datatype types[2] = {MPI_INT, MPI_DOUBLE};
        MPI_Aint displacements[2];

        Messages tmp;

        MPI_Aint base_address;
        MPI_Get_address(&tmp, &base_address);
        MPI_Get_address(&tmp.avg_row, &displacements[0]);
        MPI_Get_address(tmp.mat2, &displacements[1]);

        for(int i = 0; i < 2; ++i) {
            displacements[i] -= base_address;
        }

        MPI_Type_create_struct(2, block_lengths, displacements, types, &message_type);
        MPI_Type_commit(&message_type);
    }
};
// 静态变量类外声明
MPI_Datatype Messages::message_type;
```

同样, 先在0号进程中获取输入m, 并初始化矩阵

```
if(rank == 0) {
    cout << "input matrix size m: ";
    cin >> m;
    mat1 = new double[m*m];
    mat2 = new double[m*m];
    ans = new double [m*m](); // 初始化为0

    getRandomMat(mat1, m, m);
    getRandomMat(mat2, m, m);

    cout << "mat1:" << endl;
    showMatCorner(mat1, m, show_size);
    cout << "mat2:" << endl;
    showMatCorner(mat2, m, show_size);
    cout << "ans:" << endl;
    showMatCorner(ans, m, show_size);

    avg_row = m / size;

    begin = MPI_Wtime();
}
```

接下来先使用`MPI_Bcast()`来广播`m`的值, 根据`m`的值为`message`分配对应大小的空间. 接着0号进程需要为`message`进行赋值, 再进行广播聚合消息. 其他进程收到广播后提取其中的值.

```
// 广播m的值
MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
// 所有进程获得m值后进行构建
Messages::buildMPIType(m);
// 分配空间
message = (Messages*)malloc(sizeof(int) * 1 + sizeo(double) * m * m);
// 主进程为message赋值
if(rank == 0) {
    message->avg_row = avg_row;
    for(int i = 0; i < m; ++i) {
        for(int j = 0; j < m; ++j) {
            message->mat2[i * m + j] = mat2[i * m + j];
        }
    }
}
// 广播聚合消息
MPI_Bcast(message, 1, Messages::message_type, 0,MPI_COMM_WORLD);
// 非主进程从聚合消息中提取变量
if(rank != 0) {
    avg_row = message->avg_row;
    // 非主进程要为mat2分配空间
    mat2 = new double[m * m];
    for(int i = 0; i < m; ++i) {
        for(int j = 0; j < m; ++j) {
            mat2[i * m + j] = message->mat2[i * m + j];
        }
    }
}
```

接下来是进行计算, 当矩阵大小无法被进程数整除时, 主进程输出一条信息后直接退出

```
if(m % size != 0) {
    if(rank == 0) {
        cout << "size of matrix m mod process num != 0!" << endl;
    }
}
```

当进程数为1时, 直接进行串行矩阵乘法

```
if(size == 1) {
    matMul(ans, mat1, mat2, m, m);
    end = MPI_Wtime();
    cout << "ans:" << endl;
    showMatCorner(ans, m, show_size);
    printf("running time: %.5fs\n", end - begin);
}
```

若m能够整除size, 则开始计算. 由于mat1需要分avg_row行给各个进程, 所以使用`MPI_Scatter()`来分配.

接下来各个进程进行相应部分的计算获得结果, 使用`MPI_Gather()`将每个进程的`local_ans`集中到0号进程的`ans`中

最后0号进程输出运行时间, 和计算结果.

所有进程都要释放分配的空间.

```
double* local_mat1 = new double[avg_row * m];
double* local_ans = new double[avg_row * m]();
// 接收 avg_row 行 mat1
cout << "process " << rank << " scattering" <<endl;
MPI_Scatter(mat1, avg_row * m, MPI_DOUBLE,local_mat1, avg_row * m, MPI_DOUBLE, 0,MPI_COMM_WORLD);
// 计算
cout << "process " << rank << " calculating" <<endl;
matMul(local_ans, local_mat1, mat2, avg_row, m);
// 收集结果
cout << "process " << rank << " gathering" <<endl;
MPI_Gather(local_ans, avg_row * m, MPI_DOUBLE,ans, avg_row * m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if(rank == 0) {
    end = MPI_Wtime();
    cout << "ans:" << endl;
    showMatCorner(ans, m, show_size);
    printf("running time: %.5fs\n", end - begin);
    if(mat1) delete[] mat1;
    if(ans) delete[] ans;
}
if(message) delete message;
if(local_mat1) delete[] local_mat1;
if(mat2) delete[] mat2;
if(local_ans) delete[] local_ans;
```

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	0.00742	0.07731	0.61762	7.43551	134.44590
2	0.00435	0.03832	0.34216	4.47427	73.51936
4	0.01513	0.02496	0.20759	2.82278	43.82075
8	0.01172	0.03873	0.26671	2.88783	45.33672
16	0.01826	0.03929	0.27887	2.82645	46.76699

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256及以上规模的矩阵上都取得了最快的运行速度.

对比上一次的实验结果, 这一次实验结果运行速度普遍较快, 可以知道集合通信的性能比点对点通信更加高效.

实验感想

完成这次实验后, 我发现MPI的集合通信更加易于编写, 代码更加简洁. 逻辑更加清晰. 对于需要发送多个变量的情况, 可以将他们放在一个类中使用mpi_type_create_struct, 通过一次通信来发送多个消息. 在集合通信与点对点通信都能使用的场景下, 可以多考虑使用集合通信, 不但更高效, 而且编写更简单.