中山大学计算机院本科生实验报告

（2025学年春季学期）

课程名称：并行程序设计
批改人：

实验	2-基于MPI的并行矩阵乘法（进阶）	专业（方向）	计算机科学与技术
学号	22336087	姓名	胡瑞康
Email	hurk3@mail2.sysu.edu.cn	完成日期	2025.3.26

代码介绍

行块划分版本

在行块划分版本中，我们采用了MPI的`Scatterv`和`Gatherv`函数来分发和收集数据。每个进程负责矩阵A的一部分行，并计算该部分行与矩阵B的乘积。以下是行块划分版本代码的关键部分：

数据结构定义

```
typedef struct {
    int m;
    int n;
    int k;
} MatrixDims;
```

这里定义了一个结构体`MatrixDims`，用于存储矩阵的维度信息：矩阵A的行数`m`，列数`n`，矩阵B的列数`k`。

创建自定义MPI数据类型

```
MPI_Datatype mpi_matrix_dims;
int blocklengths[3] = {1, 1, 1};
MPI_Datatype types[3] = {MPI_INT, MPI_INT, MPI_INT};
MPI_Aint displacements[3];
```

为方便在进程间传递`MatrixDims`结构体，使用`MPI_Type_create_struct`来创建一个自定义的数据类型。结构体包含三个整数类型的字段：`m`、`n`和`k`，这些字段的大小相同，因此每个字段的块长为1。`displacements`数组则记录每个字段的偏移量。

计算结构体成员的偏移量

```
MPI_Aint base_address;
MPI_Get_address(&dims, &base_address);
MPI_Get_address(&dims.m, &displacements[0]);
MPI_Get_address(&dims.n, &displacements[1]);
MPI_Get_address(&dims.k, &displacements[2]);
displacements[0] = MPI_Aint_diff(displacements[0], base_address);
displacements[1] = MPI_Aint_diff(displacements[1], base_address);
displacements[2] = MPI_Aint_diff(displacements[2], base_address);
```

通过 `MPI_Get_address` 获取结构体及其成员变量在内存中的地址，并计算出各个字段相对于结构体起始位置的偏移量，这样可以正确地创建自定义数据类型。

自定义MPI数据类型的创建与提交

```
MPI_Type_create_struct(3, blocklengths, displacements, types, &mpi_matrix_dims);
MPI_Type_commit(&mpi_matrix_dims);
```

通过 `MPI_Type_create_struct` 创建自定义数据类型 `mpi_matrix_dims`，然后通过 `MPI_Type_commit` 提交该数据类型，使其可供MPI使用。

矩阵B的广播与矩阵A的分发

```
if (rank == 0) {
    B = (double*)malloc(dims.n * dims.k * sizeof(double));
    srand(time(NULL));
    for (int i = 0; i < dims.n * dims.k; i++) {
        B[i] = (double)(rand() % 10);
    }
}
MPI_Bcast(B, dims.n * dims.k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

在进程0中，随机生成矩阵B，并通过 `MPI_Bcast` 广播给所有进程。`MPI_Bcast` 是一种集合通信操作，用于将数据从一个进程（根进程）广播到所有其他进程。

接下来，通过 `MPI_Scatterv` 将矩阵A按行块划分并分发给各个进程。

计算矩阵乘法

```
MPI_Barrier(MPI_COMM_WORLD); // 同步所有进程
start = MPI_Wtime();

// 矩阵乘法计算 - 使用缓存优化
for (int i = 0; i < local_rows; i++) {
    for (int l = 0; l < dims.n; l++) {
        double temp = local_A[i * dims.n + l];
        for (int j = 0; j < dims.k; j++) {
            local_C[i * dims.k + j] += temp * B[l * dims.k + j];
        }
    }
}
finish = MPI_Wtime();
```

所有进程使用 `MPI_Barrier` 确保同步，保证每个进程在开始计算前已经准备好。然后开始进行矩阵乘法计算。为了优化缓存的命中率，采用了调整循环顺序的技巧，使得对同一列的访问更加集中，提高了计算效率。

收集计算时间

```
double local_time = finish - start;
double max_time;
MPI_Reduce(&local_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

使用 `MPI_Reduce` 收集各个进程的计算时间，并将所有进程中的最大时间传递给进程0。这样可以确保程序能够报告整个矩阵乘法的最大计算时间。

收集结果矩阵

```
if (rank == 0) {
    C = (double*)malloc(dims.m * dims.k * sizeof(double));
}
MPI_Gatherv(local_C, local_rows * dims.k, MPI_DOUBLE,
            C, send_counts, displs, MPI_DOUBLE,
            0, MPI_COMM_WORLD);
```

在所有进程完成矩阵乘法计算后，进程0通过 `MPI_Gatherv` 收集所有进程的计算结果，最终拼接成矩阵C。`MPI_Gatherv` 也是一种集合通信操作，与 `MPI_Scatterv` 类似，可以将不同进程的数据收集到一个数组中，且可以指定每个进程接收或发送的数据量。

输出与释放资源

```
if (rank == 0) {
    printf("\n矩阵乘法计算耗时：%f 秒\n", max_time);
    // 打印矩阵A、B和C
}
free(B);
free(local_A);
free(local_C);
if (rank == 0) {
    free(A);
    free(C);
}
MPI_Type_free(&mpi_matrix_dims);
MPI_Finalize();
```

进程0输出矩阵乘法的计算时间，并根据矩阵规模决定是否打印矩阵内容。最后，释放所有分配的内存并调用 `MPI_Type_free` 释放自定义数据类型，最后使用 `MPI_Finalize` 结束MPI程序。

块划分版本

进程数要求为完全平方数

在本实现中，首先检查进程数是否为完全平方数。因为矩阵被分配到一个二维网格（如 4x4, 3x3 等）中，所以进程数必须是完全平方数。否则，程序会输出错误信息并结束。

```
int sqrt_p = (int)sqrt(size);
if (sqrt_p * sqrt_p != size) {
    if (rank == 0) printf("错误：进程数必须为完全平方数（如1、4、9、16）！\n");
    MPI_Finalize();
    return 1;
}
```

二维网格上的进程划分

进程在二维网格中按行列划分（类似矩阵的分块），通过 `rank` 来计算每个进程在网格中的行列位置。

```
int row = rank / sqrt_p;  // 行索引
int col = rank % sqrt_p;  // 列索引
```

矩阵分块

矩阵被分成 `sqrt_p x sqrt_p` 个小块，每个进程负责计算一个矩阵块。每个块的大小通过矩阵的维度和进程数来计算：

```
int block_m = m / sqrt_p;  // 每个块的行数
int block_n = n / sqrt_p;  // A的块列数，B的块行数
int block_k = k / sqrt_p;  // 每个块的列数
```

每个进程计算自己对应的矩阵块的乘法，并将结果存储到 `local_C` 中。

矩阵A和矩阵B的分发

矩阵A和矩阵B按块分发给每个进程。进程0将每个块的数据发送给对应的进程：

```
for (int i = 0; i < sqrt_p; i++) {
    for (int j = 0; j < sqrt_p; j++) {
        int proc = i * sqrt_p + j;
        // 填充local_A和local_B
        for (int ii = 0; ii < block_m; ii++) {
            for (int jj = 0; jj < block_n; jj++) {
                local_A[ii * block_n + jj] = A[(i * block_m + ii) * n + (j * block_n +
jj)];
            }
        }
        for (int ii = 0; ii < block_n; ii++) {
            for (int jj = 0; jj < block_k; jj++) {
                local_B[ii * block_k + jj] = B[(i * block_n + ii) * k + (j * block_k +
jj)];
            }
        }
        // 如果不是进程0，发送数据
        if (proc != 0) {
            MPI_Send(local_A, block_m * block_n, MPI_DOUBLE, proc, 0, MPI_COMM_WORLD);
            MPI_Send(local_B, block_n * block_k, MPI_DOUBLE, proc, 1, MPI_COMM_WORLD);
        }
    }
}
```

对于其他进程，进程0会将其计算的矩阵块通过 `MPI_Send` 发送给它们：

```
MPI_Recv(local_A, block_m * block_n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(local_B, block_n * block_k, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

本地矩阵乘法计算

每个进程计算其对应块的矩阵乘法，并将结果存储到 `local_C` 中。具体的计算过程与之前的版本类似，主要计算自己所负责的行列块。

```
for (int i = 0; i < block_m; i++) {
    for (int j = 0; j < block_k; j++) {
        local_C[i * block_k + j] = 0.0;
        for (int l = 0; l < block_n; l++) {
            local_C[i * block_k + j] += local_A[i * block_n + l] * local_B[l * block_k +
j];
        }
    }
}
```

结果的收集与重组

进程0将自己的计算结果先存储到矩阵C中，然后使用 `MPI_Recv` 从其他进程接收它们的计算结果，并按进程的二维网格坐标重新填充矩阵C：

```
if (rank != 0) {
    MPI_Send(local_C, block_m * block_k, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
} else {
    // 进程0先填入自己的块
    for (int i = 0; i < block_m; i++) {
        for (int j = 0; j < block_k; j++) {
            C[i * k + j] = local_C[i * block_k + j];
        }
    }
    // 接收其他进程的块
    for (int proc = 1; proc < size; proc++) {
        double *temp_C = (double*)malloc(block_m * block_k * sizeof(double));
        MPI_Recv(temp_C, block_m * block_k, MPI_DOUBLE, proc, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int r = proc / sqrt_p;
        int c = proc % sqrt_p;
        for (int i = 0; i < block_m; i++) {
            for (int j = 0; j < block_k; j++) {
                C[(r * block_m + i) * k + (c * block_k + j)] = temp_C[i * block_k + j];
            }
        }
        free(temp_C);
    }
}
```

进程0输出结果

在所有进程完成计算后，进程0输出计算时间，并根据矩阵的大小决定是否打印矩阵内容。

运行测试

使用修改版的`evaluate.py`脚本进行自动化测试

```
import subprocess
import re
import os
import signal

# 要测试的程序及可执行文件名
implementations = {
    "行块划分": {
        "source": "row_distributed.cpp",
        "binary": "row_exec",
        "process_counts": [1, 2, 4, 8, 16],
        "matrix_sizes": [128, 256, 512, 1024, 2048]
    },
    "块划分": {
        "source": "block_distributed.cpp",
        "binary": "block_exec",
        "process_counts": [1, 4, 16],
        "matrix_sizes": [128, 256, 512, 1024, 2048]
    }
}

pattern = re.compile(r"矩阵乘法计算耗时：([\d\.]+) 秒")

def get_error_details(output, error):
    """从输出中提取错误详情"""
    error_details = []
    if error:
        error_details.append("错误输出：")
        error_details.append(error)
    if output:
        # 查找MPI错误信息
        mpi_errors = re.findall(r"\[.*?\] .*?(?=\n|$)", output)
        if mpi_errors:
            error_details.append("MPI错误信息：")
            error_details.extend(mpi_errors)
    return "\n".join(error_details)

# 执行每个版本的测试
for label, config in implementations.items():
    print(f"\n正在编译 {label}...")
    compile_command = f"mpic++ {config['source']} -o {config['binary']}"
    try:
        subprocess.run(compile_command, shell=True, check=True)
    except subprocess.CalledProcessError as e:
        print(f"编译失败：{e}")
```

```
        continue

    results = {}

    for p in config['process_counts']:
        for size in config['matrix_sizes']:
            input_str = f"{size} {size} {size}\n"
            command = f"OMPI_ALLOW_RUN_AS_ROOT=1 OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1 mpirun --
oversubscribe -np {p} ./{config['binary']}"
            print(f"\n运行 {label}：进程数 = {p}, 矩阵规模 = {size} x {size}")
            try:
                result = subprocess.run(
                    command,
                    input=input_str,
                    text=True,
                    shell=True,
                    capture_output=True,
                    check=True
                )
                output = result.stdout
                match = pattern.search(output)
                if match:
                    time_consumed = match.group(1)
                else:
                    time_consumed = "N/A"
                results[(p, size)] = time_consumed
                print(f"测试结果：耗时 = {time_consumed} 秒")
            except subprocess.CalledProcessError as e:
                error_msg = f"进程退出码 {e.returncode}"
                if e.returncode < 0:
                    error_msg += f" (信号 {abs(e.returncode)})"
                error_details = get_error_details(e.stdout, e.stderr)
                print(f"运行出错：{error_msg}")
                if error_details:
                    print(error_details)
                results[(p, size)] = f"Error: {error_msg}"
            except Exception as e:
                error_msg = f"未知错误：{str(e)}"
                print(error_msg)
                results[(p, size)] = error_msg

    # 打印结果表格
    print(f"\n{label} - 测试结果表格（单位：秒）\n")
    header = "|进程数|" + "|".join(str(s) for s in config['matrix_sizes']) + "|"
    separator = "|" + " :-: |" * (len(config['matrix_sizes']) + 1)
    print(header)
    print(separator)
    for p in config['process_counts']:
        row = f"|{p}|"
        for size in config['matrix_sizes']:
            row += f"{results[(p, size)]}|"
        print(row)
```

结果展示

表格总结

Lab1 行块划分 点对点通信结果表格：

进程数	128	256	512	1024	2048
1	0.010381	0.088377	0.700413	5.529363	44.570371
2	0.005298	0.044211	0.348689	2.839320	22.808313
4	0.002873	0.021984	0.183779	1.525855	12.001745
8	0.001803	0.013555	0.103347	0.841628	6.742776
16	0.001659	0.010262	0.078286	0.609859	5.012189

Lab2 行块划分 集合通信结果表格：

进程数	128	256	512	1024	2048
1	0.010380	0.084439	0.688953	5.513323	44.325613
2	0.005167	0.043282	0.349028	2.824103	22.729453
4	0.002814	0.022436	0.185242	1.509896	11.857379
8	0.001862	0.014087	0.109864	0.840141	6.607636
16	0.001249	0.014432	0.065695	0.580559	4.983197

Lab2 块划分 集合通信结果表格：（只包含能被整除的进程数）

进程数	128	256	512	1024	2048
1	0.011211	0.094224	0.794860	12.106422	102.604563
4	0.001438	0.012656	0.102716	0.950956	8.610196
16	0.000277	0.002030	0.019576	0.188162	1.502471

行块划分情况下通信模式的比较（点对点通信 vs 集合通信）

从表格中可以看到，行块划分在点对点通信和集合通信情况下的表现略有不同：

- **点对点通信**（Lab1）：
  - 对于较小的矩阵（128和256），行块划分的效率差异不大，但是随着矩阵规模增大，集合通信的性能逐渐占优，主要是因为点对点通信中的每次数据交换都是单独的，随着进程数增加，开销也增加。

    - 随着进程数的增加，点对点通信的时间减少幅度比集合通信要小，这表明点对点通信在较大矩阵和更多进程的情况下较为“昂贵”。
- **集合通信**（Lab2）：
    - 集合通信通过`MPI_Bcast`、`MPI_Scatterv`和`MPI_Gatherv`等集合操作实现了更加高效的数据传输，尤其在矩阵规模增大时，性能提高明显。
    - 例如，当进程数为16时，矩阵大小为2048x2048时，集合通信耗时仅为4.983秒，而点对点通信则为5.012秒，集合通信的表现更加优秀。

Lab2的行块划分与块划分的比较

- **行块划分 vs 块划分**:
    - 行块划分方式使用了更直接的数据分发方式，主要是按行分配矩阵的子矩阵，适合简单的矩阵乘法，但随着矩阵规模的增大，可能会导致某些进程的负载不均衡，尤其是在多进程情况下，某些进程会等待其他进程完成计算，导致效率降低。
    - 块划分则把矩阵划分成更小的块，每个进程计算自己负责的矩阵块。通过二维网格（比如4x4、9x9等）分配进程，有助于均衡每个进程的工作量。尤其是矩阵规模较大时，块划分能够显著减少每个进程需要处理的数据量，提升整体性能。
- **性能对比**:
    - 对比两个版本的结果可以发现，**块划分**方式在矩阵规模大时（如2048x2048）表现得更加高效，尤其是在高进程数（如16个进程）下。对于2048x2048的矩阵，块划分比行块划分要快很多，尤其在集合通信模式下。
    - 例如，当进程数为16时，块划分（集合通信）耗时为1.502秒，而行块划分（集合通信）耗时为4.983秒，差距明显。