Lab8 - 并行多源最短路径搜索

实验要求

使用OpenMP/Pthreads/MPI中的一种实现无向图上的多源最短路径搜索，并通过实验分析在不同进程数量、数据下该实现的性能。

输入：

1. 邻接表文件，其中每行包含两个整型（分别为两个邻接顶点的ID）及一个浮点型数据（顶点间的距离）。上图（a）中为一个邻接表的例子。注意在本次实验中忽略边的方向，都视为无向图处理；邻接表中没有的边，其距离视为无穷大。
2. 测试文件，共n行，每行包含两个整型（分别为两个邻接顶点的ID)。

问题描述：计算所有顶点对之间的最短路径距离。

输出：多源最短路径计算所消耗的时间t；及n个浮点数，每个浮点数为测试数据对应行的顶点对之间的最短距离。

要求：使用OpenMP/Pthreads/MPI中的一种实现并行多源最短路径搜索，设置不同线程数量（1-16）通过实验分析程序的并行性能。讨论不同数据（节点数量，平均度数等）及并行方式对性能可能存在的影响。

实验过程

1. 实现思路

考虑在Floyd算法上进行并行化，串行伪代码可写为如下，可以看出，当遍历到第 t 列时，需要查询的位置$D[t][w]$以及更新的位置$D[v][w]$都不在第 t 列中，则可选择在第 t 列中划分线程更新$D[v][w]$，相当于固定 t，划分 v，使用$D[t][w]$这一行和$D[v][t]$这一元素来更新其他行$D[v][w]$，各个线程不会互相干扰。

```
for t in range(n):
    for v in range(n):
        if v != t:
            for w in range(n):
                if w != t:
                    D[v][w] = min(D[v][w],D[v][t] + D[t][w])
```

2. 代码实现

展示主要部分代码，读取数据部分

```
FILE* file = fopen("updated_flower.csv", "r");
if (file == NULL) {
    perror("Unable to open file");
    return EXIT_FAILURE;
}
// 动态调整最大节点数
int maxNode = 0;
int source, target;
double distance;
char line[256];
```

```
// 跳过第一行
fgets(line, sizeof(line), file);
// 确定最大编号
while (fgets(line, sizeof(line), file)) {
    sscanf(line, "%d,%d,%lf", &source, &target, &distance);
    if (source > maxNode) { maxNode = source;}
    if (target > maxNode) { maxNode = target;  }
}
printf("maxNode:  %d\n", maxNode);
// 计算矩阵大小
int size = maxNode + 1;
double init_value = 2 * size;// 观察csv文件，最长距离不会超过2*size
// 分配矩阵内存
double** matrix = malloc(size * sizeof(double*));
for (int i = 0; i < size; i++) {
    matrix[i] = malloc(size * sizeof(double));
}
// 初始化矩阵
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        if (i == j) {
            matrix[i][j] = 0.0;  // 对角线元素为0
        }
        else {
            matrix[i][j] = init_value; // 其他元素初始化为指定值
        }
    }
}
// 回到文件开头重新读取数据
rewind(file);
// 跳过第一行标题
fgets(line, sizeof(line), file);
// 读取填充矩阵
while (fgets(line, sizeof(line), file)) {
    sscanf(line, "%d,%d,%lf", &source, &target, &distance);
    printf("%d,%d,%lf\n", source, target, distance);
    matrix[source][target] = distance;
    matrix[target][source] = distance;
}
fclose(file);
```

Floyd并行化部分，使用openmp动态调度

```
void floyd(double **matrix,int  n,int num_threads) {
#   pragma omp parallel num_threads(num_threads)
    for (int t = 0; t < n; t++) {
#   pragma omp for schedule(dynamic)
        for (int v = 0; v < n; v++) {
            if (v != t) {
                for (int w = 0; w < n; w++) {
                    if (w != t) {
                        matrix[v][w] = min(matrix[v][w], matrix[v][t] +
matrix[t][w]);
                    }
                }
```

```
            }
        }
    }

}
```

3. 运行结果

编译运行指令：

```
gcc -g -Wall -fopenmp -o opf opfloyd.c
./opf 4
```

结果分析

使用数据集updated_flower.csv，对每个进程数计算采50次，得到平均运行时间如下：

进程数	运行时间/s
1	4.087
2	2.842
4	2.300
8	2.383
16	2.363

使用数据集updated_mouse.csv，结果如下：

进程数	运行时间/s
1	0.709
2	0.503
4	0.366
8	0.893
16	0.467

可以看到线程数增加使得平均运行时间降低，但线程数在8之后运行时间反而增加，分析原因是虚拟机中每个处理器只有4个内核，在多线程到8和16的时候也只有4个内核在执行任务，而线程切换带来的开销也使时间增多。