# 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 | **并行多源最短路径搜索** | 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 | 2024.5.17 |

## 1. 实验目的（200 字以内）

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。

## 2. 实验过程和核心代码（600 字左右，图文并茂）

2 个文档的思路基本上完全一致，下面以 mouse 数据集为例讲解实验过程和展示核心代码，flower 数据集的实验的不同会进行附加说明：

（导入的头文件和定义的常量，flower 数据集仅 FILE_PATH 不同，为 updated_flower.csv）

```
# include <fstream>
# include <iostream>
# include <vector>
# include <sstream>
# include <cstdlib>
# include <omp.h>
# include <string.h>
# include <time.h>

# define INF 0x3f3f3f3f
# define FILE_PATH "updated_mouse.csv"
using namespace std;
```

提供的实验文档是邻接表形式即给出两个节点的 ID 和它们之间的距离，因此

我定义了一个结构体 Args 用来存储每一行的信息(由于 flower 数据集的节点 ID 范围为[1,930]，因此在读取节点 ID 的时候要让 node 减一，防止出现段错误，后面打印输出的时候加一即可)：

```
struct Args{
    int source, target;     // 边的两个端点
    double distance;        // 两个端点之间的距离
};
```

然后需要读入 csv 文件，由于读出的每一行数据是使用逗号分割的字符串，因此，我定义了如下函数用来分离处节点 ID 和距离：

```
Args processLine(string line){
    Args result;
    int comma_num = 0;   // 已经遇到的逗号数量
    int num_start_index = 0;    // 每个数字的起始索引
    for( int i = 0; i < line.size(); ++i ){
        if( line[i] == ','  && comma_num < 2){
            string numStr = line.substr(num_start_index, i - num_start_index);  // 读取数字部分的字符串
            int node = stoi(numStr);    // 转换为整数
            if(comma_num == 0){     // 遇到第一个逗号，其前面是原来表格中的 source 项
                result.source = node;
            }
            else{       // 遇到第二个逗号，2 个逗号之间是原来表格中的 target 项
                result.target = node;
            }
            num_start_index = i + 1;        // 更新下一个数字的起始索引
            comma_num++;        // 逗号数量加一
        }
        else{
            if (comma_num == 2 ){       // 已经遇到两个逗号，剩下的是 distance 项
                string numStr = line.substr(num_start_index, line.size() - num_start_index);    // 获取 distance 的数字字符串
                double dist = stod(numStr);     // 转为浮点数
                result.distance = dist;
                break;
            }
        }
    }
```

```
    return result;
}
```

然后在 main 函数中读入指定的 csv 文件并收集信息，记录最大最小节点的编号：

```
int MaxNode = 0;        // 记录数据集中的最大编号
int MinNode = INF;      // 记录数据集中的最小编号
vector<Args> Messages; // 存储表格信息
int thread_num;         // 线程数
cout << "请输入线程数：" << endl;
cin >> thread_num;

ifstream fin(FILE_PATH); // 用于读入表格
/*从给定的 csv 文件中读取信息*/
if (!fin.is_open())
{
    cerr << "Failed to open the file" << endl;
    exit(-1);
}
string line;             // 用于存储表格的每一行数据
bool isFirstLine = true; // 判断是否是标题行
while (getline(fin, line))
{
    // 读取每一行
    if (isFirstLine)
    { // 标题行不进行处理
        isFirstLine = false;
        continue;
    }
    Args Line = processLine(line);
    Messages.emplace_back(Line);
    MaxNode = max(Line.source, MaxNode);
    MaxNode = max(Line.target, MaxNode);
    MinNode = min(Line.source, MinNode);
    MinNode = min(Line.target, MinNode);
}
```

接下来做计算最短路径前的准备，需要将前面读取的信息存储到一个距离矩阵中，因此，我定义了以下的初始化函数 init：

```
void init(vector<Args>& messages, vector<vector<double>>& distances, int nodeNum){
    distances.resize(nodeNum);
    for( int i = 0; i < nodeNum; ++i){
        distances[i].assign(nodeNum, INF);    // 先将所有距离全都初始化为无穷大
        distances[i][i] = 0;        // 节点与自身的距离为0
    }

    // 通过查阅messages 中的节点之间的距离信息，修改对应的表项
    for( size_t i = 0; i < messages.size(); ++i ){
        int src = messages[i].source, tar = messages[i].target;
        double dist = messages[i].distance;
        distances[src][tar] = dist;
        distances[tar][src] = dist;
    }
    return;
}
```

在 main 函数中：

```
/*使用Dijkstra 算法前的准备工作*/
int NodeNum = MaxNode + 1;          // 从0 开始到MaxNode，共有（MaxNode+1）个节点
vector<vector<double>> Distances;   // 用来存储节点之间的距离
init(Messages, Distances, NodeNum); // 初始化距离矩阵
```

在我的实验中，我使用 Dijkstra 算法来计算最短路径(使用 openmp 进行并行化处理)：

```
vector<double> dijkstra(vector<vector<double>> distances, int start, int thread_num){
    int nodeNum = distances.size(); // 获取图中顶点的总数
    vector<bool> visited; // 标记数组，记录顶点是否已经被访问
    visited.assign(nodeNum, false); // 初始化所有顶点未被访问
    visited[start] = true; // 起点顶点被标记为已访问
    vector<double> dist = distances[start]; // 初始化起点到其他顶点的最短距离

    for(int i = 0; i < nodeNum; ++i){ // 对每个顶点运行Dijkstra 算法
        int min_dist, min_dist_index;
        min_dist = INF; // 初始化最小距离为无穷大

        // 并行计算每个顶点的最短路径
        #pragma omp parallel for num_threads(thread_num) shared(dist, visited)
        for(int j = 0; j < nodeNum; ++j){
            // 寻找当前未访问的顶点中，距离起点最近的顶点
            if(!visited[j] && dist[j] < min_dist){
```

```
                min_dist = dist[j];
                min_dist_index = j;
            }
        }

        // 标记找到的最近顶点为已访问
        visited[min_dist_index] = true;

        // 更新从起点到所有未访问顶点的最短路径
        #pragma omp parallel for num_threads(thread_num) shared(dist, visited)
        for(int k = 0; k < nodeNum; ++k){
            // 如果通过新访问的顶点可以找到更短的路径，则更新这个路径
            if(!visited[k] && dist[k] > dist[min_dist_index] + distances[min_dist_index][k]){
                dist[k] = dist[min_dist_index] + distances[min_dist_index][k];
            }
        }
    }
    // 返回从起点到所有顶点的最短路径长度数组
    return dist;
}
```

在 main 函数中，对所有点都使用 Dijkstra 算法分别计算出到所有点的距离，并存储在一个 shortestDistances 数组中，并对该部分进行计时：

```
/*使用Dijkstra 算法计算节点之间的最短距离*/
double start_time, using_time; // 计时
vector<vector<double>> shortestDistances;
start_time = omp_get_wtime();

for (int i = 0; i < NodeNum; ++i)
{
    vector<double> dist = dijkstra(Distances, i, thread_num);
    shortestDistances.emplace_back(dist);
}
using_time = omp_get_wtime() - start_time;
```

接下来，我打算随机生成 n 组数据，每组有 2 个节点，作为测试数据。然后对每组数据，都写入一个测试数据文件 test_mouse_data.csv(flower 数据集为 test_flower.csv)，然后将节点信息和两个节点之间的最短距离写入到一个测试

结果文件 test_mouse_result.csv(flower 数据集为 test_flower_result.csv)，注意 flower 数据集输出的时候节点 ID 要加一，因此我编写了函数 getRandomNum 来生成随机节点 ID：

```
int getRandomNum(int bottom, int top){
    int random_num = rand() % (top - bottom + 1) + bottom;  //生成[bottom, top]范围内的随机整数
    return random_num;
}
```

在 main 函数中：

```
string test_data = "test_mouse_data.csv";     // 随机生成的测试数据
string test_result = "test_mouse_result.csv"; // 测试数据的结果

ofstream data(test_data);
ofstream result(test_result);

if (!data.is_open() || !result.is_open())
{
    cerr << "Failed to open the file for writing." << endl;
    exit(-1);
}

for (int i = 0; i < NodeNum; ++i)
{
    int node_1 = getRandomNum(MinNode, MaxNode), node_2 = getRandomNum(MinNode, MaxNode);
                        // 随机生成 2 个节点
    data << to_string(node_1) << "," << to_string(node_2) << endl;
                        // 写入测试数据
    result << to_string(node_1) << "," << to_string(node_1) << "," << to_string(shortestDistances[node_1][node_2]) << endl; // 写入测试结果
    cout << "The shortest distance betwwen " << node_1 << " and " << node_2 << " is: " << shortestDistances[node_1][node_2] << endl;
}
cout << FILE_PATH << endl
     << thread_num << " threads using time: " << using_time << " s" << endl;
```

最后关闭文件：

```
// 关闭文件
data.close();
result.close();
fin.close();
```

## 3. 实验结果（500 字左右，图文并茂）

- 编译：

```
g++ -fopenmp filename.cpp -o filename
```

(此处的 filename 为 mouse 或 flower，我分别在 mouse.cpp 和 flower.cpp 对两个数据集进行处理)

- 运行：

```
./filename
```

- 运行截图，用于展示程序正确性：

```
The shortest distance betwwen 390 and 351 is: 8.64452
The shortest distance betwwen 349 and 502 is: 7.49502
The shortest distance betwwen 31 and 362 is: 5.31564
The shortest distance betwwen 102 and 299 is: 1.01199
The shortest distance betwwen 57 and 166 is: 1.10239
The shortest distance betwwen 427 and 442 is: 5.38669
The shortest distance betwwen 491 and 310 is: 8.57767
The shortest distance betwwen 318 and 248 is: 4.23837
The shortest distance betwwen 284 and 420 is: 2.11735
The shortest distance betwwen 351 and 230 is: 2.19954
The shortest distance betwwen 91 and 178 is: 6.4204
The shortest distance betwwen 104 and 103 is: 1.04374
The shortest distance betwwen 8 and 518 is: 4.24188
The shortest distance betwwen 103 and 147 is: 1.0041
The shortest distance betwwen 280 and 97 is: 1.1109
The shortest distance betwwen 249 and 145 is: 1.01136
The shortest distance betwwen 425 and 50 is: 3.20503
The shortest distance betwwen 122 and 433 is: 6.35971
The shortest distance betwwen 412 and 202 is: 7.48361
The shortest distance betwwen 207 and 446 is: 1.08039
The shortest distance betwwen 368 and 109 is: 6.47993
The shortest distance betwwen 364 and 334 is: 5.42004
The shortest distance betwwen 396 and 134 is: 5.4303
The shortest distance betwwen 35 and 132 is: 5.28085
The shortest distance betwwen 29 and 386 is: 2.13303
The shortest distance betwwen 362 and 97 is: 4.25345
The shortest distance betwwen 16 and 466 is: 2.13728
The shortest distance betwwen 177 and 25 is: 5.42397
The shortest distance betwwen 437 and 257 is: 6.53681
The shortest distance betwwen 149 and 192 is: 1.01115
The shortest distance betwwen 354 and 398 is: 3.22809
The shortest distance betwwen 315 and 231 is: 9.77377
The shortest distance betwwen 426 and 414 is: 2.17328
The shortest distance betwwen 140 and 313 is: 6.50395
The shortest distance betwwen 91 and 347 is: 3.21648
The shortest distance betwwen 212 and 437 is: 7.50157
The shortest distance betwwen 434 and 28 is: 5.38591
The shortest distance betwwen 246 and 305 is: 5.38455
The shortest distance betwwen 162 and 281 is: 7.62623
The shortest distance betwwen 415 and 191 is: 8.62934
The shortest distance betwwen 120 and 252 is: 6.37835
The shortest distance betwwen 265 and 136 is: 4.28039
The shortest distance betwwen 194 and 442 is: 3.25031
The shortest distance betwwen 161 and 83 is: 7.52706
The shortest distance betwwen 151 and 287 is: 1.10366
updated_mouse.csv
1 threads using time: 4.13954 s
laigg@laigg-VirtualBox:~/codes/parallel/lab8$
```

不同线程下不同数据集所用时间：

| 数据集 | 线程数 | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 |
| mouse | 4.13954 s | 2.59557 s | 2.47797 s | 31.0336 s | 49.077 s |
| flower | 28.0119 s | 19.5677 s | 17.4176 s | 132.315 s | 208.868 s |

分析：

1) 线程数的影响：

可以看到不管是哪个数据集从 1 到 4 个线程，随着线程数的增加，效率是越来越高的，然而在 8-16 个线程，效率则是大大降低，甚至比单线程还低，这可能是因为：

①我的虚拟机只有 4 个处理器，在 4 个线程的时候可以使得资源得到最有效的利用，而在 4 个线程以上时，则会产生资源竞争现象。

②两个数据集的节点数(mouse：524；flower：930)均不能被 8 或 16 整除，而我又使用的是 openmp 静态调度，这就导致在进行任务分配时会出现问题，导致效率大幅降低。

2) 数据规模的影响：

可以看到对于节点数量越多的数据集(mouse：524；flower：930)，其相同线程数下需要处理的数据更多，所用时间自然也就越多。

## 4. 实验感想（300 字左右）

在本次实验中，不同数据集之间的处理存在一些差别，给我造成了一点小阻碍，主要是 flower 数据集的节点 ID 是从 1 开始到 930，这就导致我使用处理 mouse 数据集的程序在处理 flower 数据集时出现了越界错误，这也说明了在涉及到处理不同数据集时不能想当然，要仔细分析每个数据集的具体情况。

此外在本次实验中，我使用了 openmp 来进行并行化处理，但是在未增加共享变量声明时出现了段错误，这可能是因为在默认情况下，dist 和 visited

数组是私有的，每个线程都会有一个副本，当多个线程尝试同时修改同一个共享数据时，就会发生竞态条件，导致段错误。