中山大学计算机学院本科生实验报告

课程名称：并行程序设计与算法

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	21307035	姓名	邓栩瀛
Email	dengxy66@mail2.sysu.edu.cn	完成日期	2024.5.16

1、实验目的

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。

2、实验过程和核心代码

本实验使用Pthreads实现

执行命令

```
g++ -std=c++11 -o main.out main.cpp -lpthread
./main.out
```

1.数据结构

邻接表的结构 Edge、邻接表的存储 graph、测试样例 tests、顶点数 num_vertices 和最短路径距离数组 shortest_distances

```
struct Edge {
    int target;
    double distance;
    Edge(int t, double d) : target(t), distance(d) {}
};

vector<vector<Edge>> graph;
vector<vector<double>> shortest_distances;
vector<pair<int, int>> tests;
int num_vertices;
```

2.文件读取

loadGraph 函数：从邻接表文件中读取数据，构建无向图的邻接表

```
void loadGraph(const string& filename) {
    ifstream file(filename);
    string line;

    getline(file, line);//跳过表头

    while (getline(file, line)) {
        stringstream ss(line);
        int source, target;
        double distance;
        char comma;
        ss >> source >> comma >> target >> comma >> distance;

        num_vertices = max(num_vertices, max(source, target) + 1);
        if (graph.size() <= max(source, target)) {
            graph.resize(max(source, target) + 1);
        }

        graph[source].emplace_back(target, distance);
        graph[target].emplace_back(source, distance);
    }
}
```

loadTests 函数：从测试文件中读取指定数量的测试样例

```
void loadTests(const string& filename, int n) {
    ifstream file(filename);
    string line;
```

```
    int count = 0;

    while (count < n && getline(file, line)) {
        stringstream ss(line);
        int source, target;
        char comma;
        ss >> source >> comma >> target;
        tests.emplace_back(source, target);
        count++;
    }
}
```

3.Dijkstra算法实现

在 dijkstra 函数中，通过多线程并行计算来加速算法的执行，每个线程负责计算部分顶点的最短路径

```
void* dijkstra(void* arg) {
    int tid = *(int*)arg;
    int start = tid * (num_vertices / MAX_THREADS);
    int end = (tid == MAX_THREADS - 1) ? num_vertices : (tid + 1) * (num_vertices /
MAX_THREADS);
    for (int i = start; i < end; ++i) {
        vector<double>& dist = shortest_distances[i];
        dist[i] = 0.0;
        vector<bool> visited(num_vertices, false);
        for (int j = 0; j < num_vertices - 1; ++j) {
            int u = -1;
            double min_dist = INF;
            for (int k = start; k < end; ++k) {
                if (!visited[k] && dist[k] < min_dist) {
                    min_dist = dist[k];
                    u = k;
                }
            }
            if (u == -1)
                break;
            visited[u] = true;
            for (const Edge& edge : graph[u]) {
                int v = edge.target;
                double alt = dist[u] + edge.distance;
                if (alt < dist[v]) {
                    dist[v] = alt;
                }
            }
        }
    }
    pthread_exit(NULL);
}
```

4.主函数

加载邻接表和测试数据，初始化最短路径数组

```
loadGraph("updated_mouse.csv");
loadTests("updated_flower.csv", num_tests);
shortest_distances.resize(num_vertices, vector<double>(num_vertices, INF));
```

创建多个线程，每个线程执行 Dijkstra 算法的一部分

```
for (int i = 0; i < MAX_THREADS; ++i) {
    thread_ids[i] = i;
    pthread_create(&threads[i], NULL, dijkstra, (void*)&thread_ids[i]);
}
```

等待所有线程完成计算

```
for (int i = 0; i < MAX_THREADS; ++i) {
    pthread_join(threads[i], NULL);
}
```

3、实验结果

线程数	测试样例数			
	128	512	2048	4096
1	1.81088	1.81262	1.81348	1.81763
2	0.210625	0.209174	0.209151	0.209599
4	0.038106	0.0362129	0.0377109	0.0380461
8	0.00580716	0.00600815	0.0058279	0.00630593
16	0.00155497	0.00164604	0.00156593	0.0018301

实验结果分析：

1. 随着线程数量的增加，总体运行时间显著减少，从线程数从1增加到16的过程中，可以看到总体运行时间大幅度减少，说明多线程并行计算确实能够有效提高程序的性能，在这种情况下，随着线程数量的增加，任务可以更好地分配和并行执行，从而加快了程序的执行速度。
2. 对于小规模测试样例，线程数量的增加带来的性能提升更加明显，例如，对于128个测试样例，从1个线程到16个线程，总体运行时间下降了一个数量级。
3. 对于大规模测试样例，线程数量的增加对性能提升的效果逐渐减弱，例如，对于4096个测试样例，随着线程数量增加，总体运行时间的降低幅度减小，可能是因为任务规模增大的同时，线程间的竞争和通信成本增加，从而限制了性能的进一步提升。

4、实验感想

1. 通过并行计算，将任务分解成多个子任务，并让多个线程同时执行这些子任务，可以显著提高程序的执行速度，这在处理大规模数据或者需要复杂计算的任务中非常重要。
2. 虽然增加线程数量在通常情况下能够提高性能，但是过多的线程也有可能会导致性能下降，因为线程创建、销毁和上下文切换的开销会增加，在选择线程数量时，需要兼顾任务的规模、计算机硬件以及系统的特性等因素。
3. 在这个实验中，调优也是非常关键的，为了进一步的优化，需要充分了解任务的特点、数据结构、并行算法以及线程之间的通信和同步方式，从而设计出高效的并行计算方案。
4. 通过性能分析，可以了解程序的瓶颈所在，从而针对性地进行优化，在这个实验中，通过比较不同线程数量下的运行时间，来发现程序的性能瓶颈和优化空间，从而调整并行策略来提高性能。