Lab 8: 并行多源最短路径搜索实验报告

蔡可豪
22336018

1. 实验目的

本实验旨在通过使用 OpenMP 实现并行的 Floyd-Warshall 算法，来计算无向图中所有顶点对之间的最短路径。实验的重点是分析在不同线程数量下算法的性能表现，并探讨数据特征（如节点数量、平均度数）以及并行化策略对性能的潜在影响。

2. 实验环境

- **操作系统**: macOS Sonoma (具体版本根据用户环境而定，此处为示例)
- **编译器**: Clang (通过 `brew install llvm` 安装的版本，支持 OpenMP)
- **并行框架**: OpenMP
- **CPU**: (根据用户环境而定，例如 Apple M1 Pro)

3. 算法描述

3.1 Floyd-Warshall 算法

Floyd-Warshall 算法是一种动态规划算法，用于找到图中所有顶点对之间的最短路径。算法的核心思想是，对于任意一对顶点 `(i, j)`，其最短路径要么是直接连接它们的边，要么是通过某个中间顶点 `k` 连接的路径 `(i, k)` 和 `(k, j)` 之和。算法通过迭代地考虑所有可能的中间顶点来更新最短路径。

设 `dist[i][j]` 为从顶点 `i` 到顶点 `j` 的最短路径长度。算法的迭代过程如下：

```
for k from 0 to num_vertices - 1:
  for i from 0 to num_vertices - 1:
    for j from 0 to num_vertices - 1:
      if dist[i][k] + dist[k][j] < dist[i][j]:
        dist[i][j] = dist[i][k] + dist[k][j]
```

3.2 OpenMP 并行化

在本实验中，我们使用 OpenMP 对 Floyd-Warshall 算法的内两层循环（`i` 和 `j` 循环）进行并行化。外层的 `k` 循环保持串行，因为 `k` 的每次迭代都依赖于前一次迭代完成的 `dist` 矩阵的更新。通过 `#pragma omp parallel for schedule(static)` 指令，可以将计算不同 `i`（或 `j`）的工作分配给多个线程，从而尝试加速计算过程。

```cpp
// Floyd-Warshall 算法
for (int k = 0; k < num_vertices; ++k) {
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < num_vertices; ++i) {
        for (int j = 0; j < num_vertices; ++j) {
            if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

4. 实验步骤

1. **编写代码**：实现 C++ 程序（`main.cpp`），包含读取邻接表、读取测试用例、并行 Floyd-Warshall 算法以及输出结果的功能。
2. **准备数据**：
   - `lab8/data/adj_list.txt`：存储图的邻接表，每行格式为 `source target weight`。
   - `lab8/data/test_pairs.txt`：存储需要查询最短路径的顶点对，每行格式为 `source target`。
3. **编译代码**：由于 macOS 默认的 Clang 可能不完全支持 OpenMP，首先通过 `brew install llvm` 安装了包含完整 OpenMP 支持的 LLVM/Clang。然后使用以下命令编译：

```bash
export PATH="/opt/homebrew/opt/llvm/bin:$PATH"
export LDFLAGS="-L/opt/homebrew/opt/llvm/lib"
export CPPFLAGS="-I/opt/homebrew/opt/llvm/include"
clang++ -std=c++17 -fopenmp lab8/src/main.cpp -o lab8/src/apsp
```

4. **运行实验**：使用不同的线程数（1, 2, 4, 8, 16）运行编译好的程序，并将输出重定向到结果文件中。

```bash
mkdir -p lab8/results
for i in 1 2 4 8 16; do
    echo "Running with $i threads...";
    ./lab8/src/apsp lab8/data/adj_list.txt lab8/data/test_pairs.txt $i > lab8/results/output_threads_$i.txt;
done
```

5. 实验结果与分析

5.1 输入数据

测试使用的图结构如下（来自 `adj_list.txt`）：

```
0 1 0.7214398
0 2 0.0547731
0 4 0.2237872
```

```
0 5 0.2278107
1 2 0.5384615
1 6 0.193787
1 7 0.010355
1 8 0.0295858
5 7 0.8267477
5 9 0.1489362
5 10 0.0243161
7 10 0.557377
7 11 0.4036885
7 12 0.0245902
7 13 0.0061475
```

根据邻接表，图中最大顶点 ID 为 13，因此程序推断顶点数量为 14 (0-13)。

测试查询的顶点对如下 (来自 `test_pairs.txt`):

```
0 13
1 10
5 8
```

5.2 运行时间

不同线程数下 Floyd-Warshall 算法的运行时间如下：

线程数	运行时间 (ms)
1	0.050
2	0.253
4	0.233
8	0.895
16	1.504

5.3 最短路径结果

对于测试的顶点对，计算出的最短路径如下（所有线程数下结果一致）：

- Shortest distance between 0 and 13: 0.609737
- Shortest distance between 1 and 10: 0.567732
- Shortest distance between 5 and 8: 0.621634

5.4 性能分析

从运行时间数据可以看出，对于本次实验使用的小规模图（14个顶点），增加 OpenMP 线程数并没有带来性能提升，反而导致了运行时间的显著增加。单线程运行时间最短（0.050 ms），而16线程时运行时间最长（1.504 ms）。

原因分析：

1. **并行开销（Overhead）**：OpenMP 在启动并行区域、创建和管理线程、以及在并行循环结束时进行线程同步都需要一定的开销。对于计算量较小的问题，这些开销可能会超过并行计算本身带来的收益。
2. **数据规模**：Floyd-Warshall 算法的时间复杂度为 O(V^3)，其中 V 是顶点数量。当 V 较小时（如本例中的 14），总的计算量相对较小。即使内层循环被并行化，每个线程分配到的实际计算任务也非常少，使得并行开销占比更大。
3. **`schedule(static)` 策略**：静态调度将迭代平均分配给线程。对于负载非常均匀的任务，这通常是高效的。但在本例中，由于计算量本身很小，调度的开销和线程管理的开销可能占据了主导地位。
4. **算法特性**：Floyd-Warshall 算法的外层 `k` 循环是串行的，这限制了整体的并行度。并行化主要发生在内两层循环。如果图非常稀疏或者 V 很小，内层循环的迭代次数可能不足以充分利用多核优势。

对性能可能存在影响的因素讨论：

- **节点数量 (V)**：随着节点数量 V 的显著增加，Floyd-Warshall 算法的总计算量会以 V^3 的速度增长。在这种情况下，并行计算的收益更有可能超过并行开销。对于大规模图，预计 OpenMP 并行化能带来显著的性能提升。
- **平均度数/图的密度**：图的密度（边的数量）主要影响邻接矩阵的初始化和某些图算法的性能，但对于 Floyd-Warshall 算法（基于邻接矩阵表示，并假设不存在的边权重为无穷大），其核心计算部分的复杂度主要由顶点数 V 决定。不过，在实际应用中，如果图非常稀疏，可能更适合使用针对稀疏图的 SSSP 算法（如多次 Dijkstra 或 Bellman-Ford）并进行并行化。
- **并行方式/调度策略**：
  - 对于 Floyd-Warshall，主要的并行点在于内两层循环。可以尝试不同的 `schedule` 子句（如 `dynamic`，`guided`）并调整 `chunk_size`，但这对于当前的小规模问题不太可能产生质的改变。
  - 更高级的并行化策略，如分块 Floyd-Warshall，可能在特定架构和大规模图上表现更好，但实现更复杂。
- **硬件环境**：CPU核心数、缓存大小、内存带宽等都会影响并行程序的性能。在核心数较少或缓存较小的系统上，并行开销的影响可能更为明显。

6. 结论

本实验成功使用 OpenMP 实现了并行的 Floyd-Warshall 算法。实验结果表明，对于本次使用的小规模测试图（14 个顶点），由于并行开销远大于并行计算带来的收益，增加线程数反而导致了性能下降。

这突出说明了并行化并非总是能带来性能提升，尤其是在处理小规模问题时。算法的特性、数据规模以及并行化引入的额外开销都是评估并行性能时需要综合考虑的关键因素。对于 Floyd-Warshall 算法，其并行化的效果更可能在顶点数量远大于本实验所用数据集时显现出来。

未来的工作可以包括：

- 在更大规模的图数据集上测试该并行实现的性能。
- 比较不同并行调度策略的影响。
- 与其他并行最短路径算法（如并行 Dijkstra）进行性能对比。

7. 附录：代码

核心 C++ 代码 (`lab8/src/main.cpp`) 如下：

```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include <limits>
#include <omp.h>
#include <chrono>
#include <iomanip>

const double INF = std::numeric_limits<double>::infinity();

// 函数：读取邻接表文件
bool read_adjacency_list(const std::string& filename, int& num_vertices,
std::vector<std::tuple<int, int, double>>& edges) {
    std::ifstream infile(filename);
    if (!infile.is_open()) {
        std::cerr << "Error: Cannot open adjacency list file: " << filename <<
std::endl;
        return false;
    }

    std::string line;
    num_vertices = 0;
    while (std::getline(infile, line)) {
        std::istringstream iss(line);
        int u, v;
        double weight;
        if (iss >> u >> v >> weight) {
            edges.emplace_back(u, v, weight);
            num_vertices = std::max({num_vertices, u + 1, v + 1});
        } else {
            std::cerr << "Warning: Skipping invalid line in adjacency list: " <<
line << std::endl;
        }
    }
    infile.close();
    return true;
}

// 函数：读取测试用例文件
bool read_test_cases(const std::string& filename, std::vector<std::pair<int, int>>&
test_pairs) {
    std::ifstream infile(filename);
    if (!infile.is_open()) {
        std::cerr << "Error: Cannot open test case file: " << filename << std::endl;
        return false;
    }

    std::string line;
    while (std::getline(infile, line)) {
        std::istringstream iss(line);
        int u, v;
```

```
        if (iss >> u >> v) {
            test_pairs.emplace_back(u, v);
        } else {
            std::cerr << "Warning: Skipping invalid line in test case file: " << line << std::endl;
        }
    }
    infile.close();
    return true;
}


int main(int argc, char* argv[]) {
    if (argc < 4) {
        std::cerr << "Usage: " << argv[0] << " <adjacency_list_file> <test_cases_file> <num_threads>" << std::endl;
        return 1;
    }

    std::string adj_list_filename = argv[1];
    std::string test_cases_filename = argv[2];
    int num_threads = std::stoi(argv[3]);

    if (num_threads <= 0 || num_threads > 16) {
        std::cerr << "Error: Number of threads must be between 1 and 16." << std::endl;
        return 1;
    }

    omp_set_num_threads(num_threads);

    int num_vertices;
    std::vector<std::tuple<int, int, double>> edges;

    if (!read_adjacency_list(adj_list_filename, num_vertices, edges)) {
        return 1;
    }

    if (num_vertices == 0) {
        std::cout << "No vertices found in the graph." << std::endl;
        return 0;
    }

    std::vector<std::vector<double>> dist(num_vertices, std::vector<double>(num_vertices, INF));

    for (int i = 0; i < num_vertices; ++i) {
        dist[i][i] = 0;
    }

    for (const auto& edge : edges) {
        int u = std::get<0>(edge);
```

```
        int v = std::get<1>(edge);
        double weight = std::get<2>(edge);
        if (u >= 0 && u < num_vertices && v >= 0 && v < num_vertices) {
            dist[u][v] = weight;
            dist[v][u] = weight; // 无向图
        }
    }

    auto start_time = std::chrono::high_resolution_clock::now();

    // Floyd-Warshall 算法
    for (int k = 0; k < num_vertices; ++k) {
        #pragma omp parallel for schedule(static)
        for (int i = 0; i < num_vertices; ++i) {
            for (int j = 0; j < num_vertices; ++j) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    auto end_time = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> elapsed_time_ms = end_time - start_time;

    std::cout << "Time taken for Floyd-Warshall: " << std::fixed << std::setprecision(3) << elapsed_time_ms.count() << " ms" << std::endl;
    std::cout << "Number of threads used: " << num_threads << std::endl;

    std::vector<std::pair<int, int>> test_pairs;
    if (!read_test_cases(test_cases_filename, test_pairs)) {
        return 1;
    }

    std::cout << "\nShortest distances for test pairs:" << std::endl;
    for (const auto& pair : test_pairs) {
        int u = pair.first;
        int v = pair.second;
        if (u >= 0 && u < num_vertices && v >= 0 && v < num_vertices) {
            std::cout << "Shortest distance between " << u << " and " << v << ": ";
            if (dist[u][v] == INF) {
                std::cout << "INF" << std::endl;
            } else {
                std::cout << std::fixed << std::setprecision(6) << dist[u][v] << std::endl;
            }
        } else {
            std::cerr << "Warning: Invalid test pair (" << u << ", " << v << ") for the given graph size." << std::endl;
        }
```

```
144     }
145
146     return 0;
147 }
```