

Parallel-Programming Task8

刘森元, 21307289

中山大学计算机学院

Codes on <https://github.com/Myocardial-infarction-Jerry/Parallel-Programming/tree/main/Task8>.

Project built by CMake.

```
1 | > cd Task8
2 | > cmake . && make
3 | > bin/ShortestPath datasets/updated_mouse.csv 8
4 | > ctest -V
```

1 Environment

11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz

NVIDIA GeForce RTX 3080 Ti O12G

Windows Subsystem for Linux @ Ubuntu 22.04 LTS

2 Task: 使用任意并行框架实现多源最短路径搜索

Note

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。

问题描述：计算所有顶点对之间的最短路径距离。

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）。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）。

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。

3 Theory

1. 并行计算模型

并行计算模型通常基于多线程或多进程的方法，允许同时执行多个计算任务。理论上，这种方法能够通过分布在多个核心或处理器上分布计算负载，来加速数据处理过程。特别是对于图算法，如Dijkstra算法，可以独立计算各个顶点作为起点的最短路径问题，这提供了很好的并行化潜力。

2. Amdahl定律

Amdahl定律是并行计算中的一个重要理论，它描述了加速比的理论最大值。根据这个定律，程序的加速比受限于程序中无法并行化的部分。即使只有很小一部分代码无法并行化，也会显著影响整体的性能提升。在实际应用中，要分析算法中哪些部分可以并行化，哪些是固有的串行部分。

3. 资源竞争与同步

在多线程环境中，资源竞争和线程之间的同步是必须处理的问题。当多个线程尝试同时访问和修改相同的数据时，可能会导致数据不一致或竞态条件。因此，需要合理设计数据访问策略和同步机制，以保证数据的完整性和计算的正确性。

4. 工作负载平衡

在并行处理中，确保所有处理单元（如CPU核心）工作负载均衡也是提高效率的关键。如果一部分核心负载过重而其他核心闲置，将不能充分发挥并行计算的优势。有效的负载平衡策略可以提高资源的利用率和减少总体的计算时间。

5. 理论上的时间复杂度分析

对于并行Dijkstra算法，如果能够完全并行，理论上的时间复杂度为 $O((V+E)\log Vp)$ $O(p(V+E)\log V)$ ，其中 pp 是并行度（线程数）。但实际上，由于线程创建、管理和同步的开销，以及Amdahl定律的限制，真实的性能提升可能低于理论值。

4 Code

⚠ Caution

源代码详见 [ShortestPath.cpp](#)

```
1  class DirectedGraph {
2  protected:
3      class Node;
4      class Edge;
5
6      ...
7
8      std::map<std::string, Node *> nodes;
9
10     void dijkstra(Node *start);
11
12 public:
13     ...
14
15     enum class Mode {
16         PARALLEL,
17         SEQUENTIAL
18     };
19 }
```

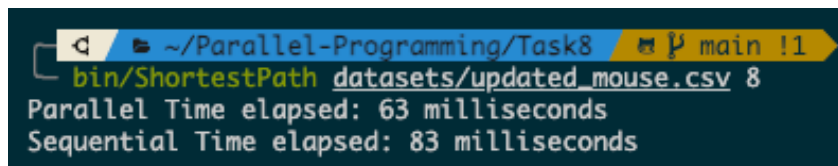
```

20     void solveShortestPath(const int &threadsLimit = 8, Mode mode = Mode::PARALLEL) {
21         if (mode == Mode::SEQUENTIAL) {
22             for (auto &node : nodes)
23                 dijkstra(node.second);
24
25             return;
26         }
27
28         std::vector<std::thread> threads;
29         int threadsCount = 0;
30         for (auto &node : nodes) {
31             threads.push_back(std::thread(&DirectedGraph::dijkstra, this, node.second));
32             threadsCount++;
33             if (threadsCount == threadsLimit) {
34                 for (auto &thread : threads)
35                     thread.join();
36                 threads.clear();
37                 threadsCount = 0;
38             }
39         }
40
41         for (auto &thread : threads)
42             thread.join();
43         threads.clear();
44     }
45 };

```

5 Result

以 8 线程 `updated_mouse.csv` 为例



```

~/Parallel-Programming/Task8 main !1
bin/ShortestPath datasets/updated_mouse.csv 8
Parallel Time elapsed: 63 milliseconds
Sequential Time elapsed: 83 milliseconds

```

对于完整数据有

Threads	Flower	Mouse
1	67 ms	120 ms
2	60 ms	101 ms
4	37 ms	87 ms
8	30 ms	63 ms
16	24 ms	52 ms

数据表明，当增加线程数时，`Flower` 和 `Mouse` 的计算时间都有所下降，但降幅随线程数的增加而减少。这种现象可以从多个并行计算的理论角度来分析：

1. Amdahl定律的影响

Amdahl定律指出，程序的加速比受限于其串行部分的比例。根据表格，尽管增加了线程数量，加速比逐渐降低，这可能意味着算法中还存在一定的串行工作，或者并行部分的效率并未完全线性增加。例如，即使将线程数从8增加到16，时间的减少并不是前者的一半，这表明随着线程数的增加，额外线程的效益递减。

2. 线程管理和同步开销

随着线程数量的增加，线程管理（如创建和销毁线程）和线程间同步（如等待数据处理完成）的开销也会增加。这些开销可能在较高的线程数下变得更加显著，从而影响到整体的性能提升。特别是在线程数超过处理器核心数的情况下，可能会导致线程频繁切换，增加上下文切换的成本。

3. 工作负载分配

如果数据或任务在多个线程之间分配不均，一些线程可能会比其他线程先完成任务，从而导致资源闲置。在这种情况下，即使增加了线程数，也无法有效利用所有的线程资源，导致性能提升不明显。

4. 硬件资源限制

硬件配置（如CPU核心数）也会对并行计算的效果产生重要影响。例如，如果CPU核心数少于线程数，那么并不能有效地并行执行所有线程，因为多个线程将共享同一核心。这可能解释了为什么从8线程到16线程的性能提升不如从1线程到4线程的提升大。

5. 数据访问瓶颈

对于一些并行算法，尤其是数据密集型任务，内存访问速度可能成为瓶颈。多个线程同时访问内存时可能会导致内存带宽饱和，或者缓存一致性机制导致的额外开销，这也可能影响并行执行的效率。

综上所述，这些数据反映了在并行化多源最短路径计算过程中存在的一系列挑战，包括程序的串行部分、线程管理和同步的开销、工作负载的不均衡分配、硬件资源的限制，以及数据访问的瓶颈等。理解这些因素对于设计高效的并行算法和优化现有算法至关重要。

对于并行化图算法如Dijkstra算法，输入数据的特征（如节点数量和平均度数）以及并行方式（如数据分割和任务调度策略）对性能的影响是显著的。这些因素共同决定了算法的效率、可扩展性和实际的运行时间。

1. 节点数量和图的密度

- ***节点数量：**节点数量的增加会导致计算负载增加，因为每个节点都需要进行路径计算。在并行处理中，更多的节点意味着可以分配更多的并行任务，但同时也增加了管理这些任务的复杂性。
- **图的密度（平均度数）**：**图的密度即每个节点的平均边数。密度较高的图意味着每个节点有更多的边，这增加了每次迭代中的计算量。对于密集图，每个节点的处理可能需要更多时间，这可能会影响并行算法的效率，特别是当内存访问和同步开销较大时。

2. 并行方式

- **数据分割：**如何将图数据分割并分配给各个处理单元是提高并行处理效率的关键。理想情况下，数据应该均匀分配，以确保所有处理器或线程的负载均衡。不平衡的负载可能导致某些线程过早空闲，而其他线程还在忙碌，从而影响整体性能。
- **任务调度：**任务调度策略决定了任务在处理器上的执行顺序。有效的调度可以减少等待时间和上下文切换，提高CPU利用率。例如，动态调度可以根据各线程的实际运行情况动态分配任务，以避免某些线程过载而其他线程空闲。