并行化编程书面报告

520021910266 张月宸

May 20, 2022

1 并行化思路

在 Part1 中，首先需要用 Dijkstra 算法计算出源节点 X 和 n 个必经节点所组成的点集中两两节点之间的最短距离和路径，即求在图中 (n+1) 个定点全排列上的两点的最短距离。

对于一个节点数为 V，中间定点个数为 N 的图，在没有预计算的情况下，使用 Dijkstra 算法计算一个全排列序列上相邻两点间最短距离的时间复杂度为 $O(N\times V^2)$。因此当图的规模变大时，计算的时间会变久，可以实施并行化，将总的计算量分给多个线程去共同完成来提高这部分的执行效率。

本次报告中，对此部分进行并行化和评测。

2 并行化实施

在原代码中，为求得两两定点间的最短距离，进行了循环嵌套。原代码如下：

```
vector<int> FixedSP::getFixedPointShortestPath(int source, vector<int> intermediates)
{
    ......
    /*将源节点推入定点集*/
    intermediates.push_back(source);

    N = intermediates.size();
    /*循环计算定点集全排列序列上两点间最短距离和路径*/
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (i != j) {
                min_distance[intermediates[i]][intermediates[j]] = getMinDistance(
                    intermediates[i], intermediates[j]);
            }
        }
    }

    intermediates.pop_back();
    ......
}
```

为实施并行化，定义 M 为线程数，构造 M 个线程，将 N×N（N 为定点数）循环分为 M 个 (_d)×N 循环，共同完成。

```
vector<int> getFixedPointShortestPath(int source)
{
    ......
    intermediates.push_back(source);
    N = intermediates.size();

    /*计算一个线程中负责的单位范围*/
    _d = (N % M == 0) ? N / M : (N / M + 1);

    /*构建 M 个线程*/
    vector<thread> threads;
    for (int i = 0; i < M; i++) {
        threads.emplace_back(slaveMinDistance, i); //传递线程序号
    }

    /*等待线程全部运行完*/
    for (int i = 0; i < M; i++)
        threads[i].join();

    intermediates.pop_back();
    ......
}

void slaveMinDistance(int id) { //参数是自定义的线程序号
    /*确立本线程负责的范围*/
    int start = id * _d;
    int end = start + _d;
    for (int i = start; i < end && i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (i != j) {
                min_distance[intermediates[i]][intermediates[j]] = getMinDistance(
                    intermediates[i], intermediates[j]);
            }
        }
    }
}
```

3 性能评测与结论

3.1 硬件环境

机型：LAPTOP-HBLDMH1F
处理器：Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz
RAM：16.0 GB

操作系统：64 位操作系统, 基于 x64 的处理器

3.2 评测方案设计

3.2.1 输入数据设计

为评测不同数据规模和不同线程数下的执行效率，分别选取两组实验数据：

- 25×25 邻接矩阵，4 个必经点。
- 200×200 邻接矩阵，8 个必经点。

3.2.2 评测内容

分别测量了同一数据集在线程数分别为 1、2、4、8 时的执行时间（不包括 I/O 消耗的时间，仅得出最短路径及距离的时间），作为对比执行效率的依据。

3.3 评测结果

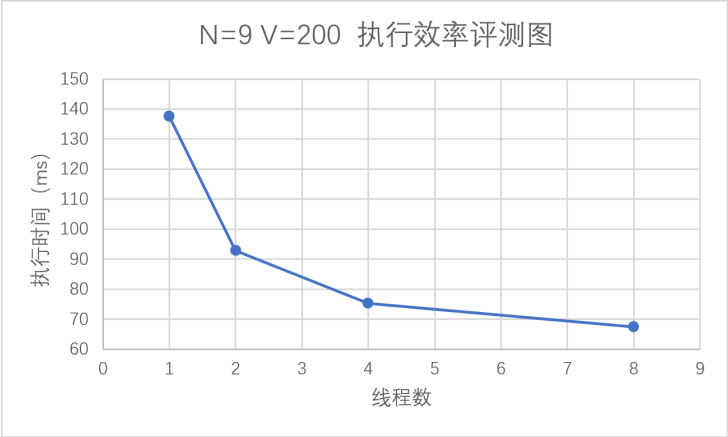

Figure 1: N=8，V=200 执行效率评测图

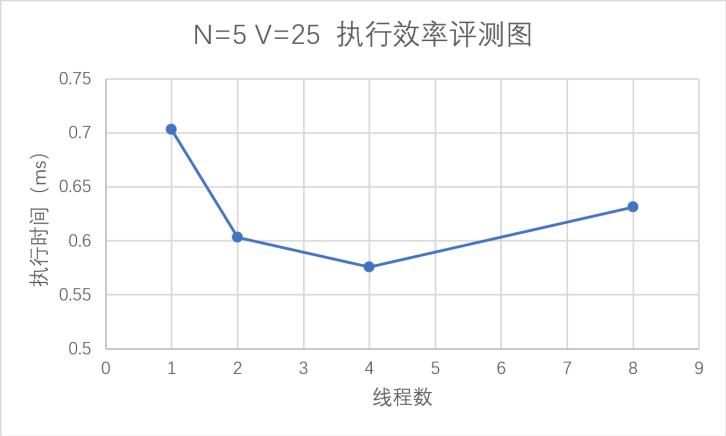

Figure 2: N=4，V = 25 执行效率评测图

3.4 结论

由图一和图二可见，N=9、v=200 时，随着线程数上升，能够获得更高的执行效率；N=5、V=25 时，在线程数为 1、2、4 时，随着线程数增多，执行效率变大，而当线程数为 8 时，执行效率反而降低了。这应该是因为随着线程数增多，线程创建和切换的开销增大，当数据量相对较小时，这一开销消耗的时间对执行效率的影响较大，因此线程数增多，执行效率反而下降了。