中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

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	--	姓名	--
Email	--	完成日期	2025.5.21

实验目的

- 使用任意并行框架实现多源最短路径搜索

实验过程和核心代码

由于需要计算所有顶点对的最短路径(All Pairs Shortest Path), 我们使用Floyd算法来实现. 要并行化Floyd算法, 首先我们要了解Floyd算法的原理.

Floyd的思想是没有直接边相连的两点i, j的距离先认为是无穷大. 不断插入节点k, 如果i, k的距离加上k, j的距离小于i, j的距离, 则更新i, j的距离. 直到所有点插入完成.

核心代码非常简单:

```
for(int k = 0; k < n; ++k) {
    for(int i = 0; i < n; ++i) {
        // ik没有连接, 无需更新ij
        if(dist[i][k] >= INF) {
            continue;
        }
        for(int j = 0; j < n; ++j) {
            dist[i][j] = min({INF, dist[i][j], dist[i][k] + dist[k][j]});
        }
    }
}
```

然后需要考虑如何并行化Floyd算法.

首先最容易想到的是并行化最内层循环, 在i和k都确定的情况下, `dist[i][j]` 的值受到 `dist[i][k]` 和 `dist[k][j]` 的影响. 而 `dist[i][k]` 的值只有在j = k时才可能发生更新, 此时 `dist[i][k] + dist[k][j] = dist[i][j] + dist[j][j]` , 而 `dist[j][j]` 为0, 所以 `dist[i][k]` 值是保持不变的. 同理 `dist[k][j]` 也只有在k = i时才能发生更新. 从算法层面讲, 我们在固定插入点为k, 起始点为i的情况下, 更新i和某个终点的距离并不会影响i与其他终点的距离. 所以此并行方法是可行的.

```
// parallel mode 2
for(int k = 0; k < n; ++k) {
    for(int i = 0; i < n; ++i) {
        if(dist[i][k] >= INF) {
            continue;
        }
        #pragma omp parallel for schedule(static)
        for(int j = 0; j < n; ++j) {
            dist[i][j] = min({INF, dist[i][j], dist[i][k] + dist[k][j]});
        }
    }
}
```

其次, 我们也会想到可不可以并行化最外层循环, 即不同线程同时处理不同的插入点k. 但这样可能会影响算法的正确性. 例如我们有4个顶点的图, 以如下方式连接:

```
1 -(1)- 2 -(1)- 3 -(1)- 4
()中的值表示距离
```

假设我们使用两个线程并行, 那么线程A执行插入1,2顶点, 线程B执行插入3,4顶点.
如果A线程在执行插入顶点2, 且i=1, j=4时, B线程还没有更新2到4的距离, 那么此时A线程更新1-4的距离仍为 `INF` . 并且B线程执行插入顶点3, i=1, j=4时, A线程没有更新1-3的距离, 那么B线程更新1-4的距离仍为 `INF` . 而实际上1-4的最短距离为3. 所以我们不能并行化最外层的循环.

最后是并行化中间的循环, 从算法层面来说, 先固定插入点, 每个线程执行某个起点和所有终点的距离更新. 每个线程的i是独立的, 只有在k等于其他线程的i时, 可能会受其他线程的影响, 而此时其他线程i=k, 所以 `dist[i][k] + dist[k][j] = dist[i][i] + dist[i][j] = 0 + dist[i][j]` , 即插入的点k就是起点i, 不会更新i到其他点的距离. 所以此并行方法可行

```
// parallel mode1
for(int k = 0; k < n; ++k) {
    #pragma omp parallel for schedule(static)
    for(int i = 0; i < n; ++i) {
        if(dist[i][k] >= INF) {
            continue;
        }
        for(int j = 0; j < n; ++j) {
            dist[i][j] = min({INF, dist[i][j], dist[i][k] + dist[k][j]});
        }
    }
}
```

由于测试文件仅用于输出结果, 不影响核心算法运行性能, 所以只添加了一个简单的 `test.txt` 作为测试文件

实验结果

虚拟机处理器个数: 4
时间单位: s

updated_flower.csv

节点数量: 930, 平均度数: 14.5

thread num \ parallel mode	1	2
1	3.2273	3.0416
2	2.7448	2.0662
4	2.4566	1.2637
8	2.0544	6.4352
16	1.6107	10.210

updated_mouse.csv

节点数量: 525, 平均度数: 28.0

thread num \ parallel mode	1	2
1	2.4747	2.5203
2	1.4789	1.6933
4	0.9116	1.2821
8	0.9328	8.0678
16	0.9201	13.4094

首先观察到在mode1(并行中间i循环)下线程数1-4时性能有明显提升, 在mouse数据下, 线程数4-16只有略微的性能下降, 而且在flower数据下线程数4-16还有不小的性能提升. 猜测可能是由于**flower数据的平均度数较小**, 中间的i循环有许多满足 `dist[i][k] >= INF` 条件从而在多个线程中快速跳过.

在mode2(并行最内层j循环)下两个数据集都是在线程数为4时达到了最好的性能, 在线程数大于4时性能明显下降, 甚至比一个线程的执行性能都要慢几倍. 可能时由于内层整个循环的执行次数为 度数 $\times$ 度数. 在线程数超过虚拟机核数时需要频繁的创建和销毁线程, 带来的开销远大于并行带来的性能提升.

在mode1下, 由于mouse数据的节点数更少, 所以mouse数据下运行的速度更快. 在mode2下, 在线程数为1-4时也是mouse的性能更快. 由于Floyd算法三个循环都是执行节点数的次数, 所以节点数对性能影响较大, 节点数越少性能越好.

实验感想

在本次实验中, 我先学习实现了Floyd算法, 然后对此算法进行分析, 分别判断每层循环并行的可行性. 我学习到需要对算法进行理解, 才能进一步分析并行化是否会影响算法正确性, 以及如何在不影响算法正确性的前提下去实现并行算法以提高性能. 此外, 我也了解到不同的并行方式对算法性能也要非常大的影响, 要在进行充分分析和实验后来得到该算法较好的并行方式.