中山大学计算机院本科生实验报告

（2024 学年春季学期）

课程名称：并行程序设计　　批改人：

实验	8 -并行多源最短路径搜索	专业（方向）	计算机科学与技术
学号	21307174	姓名	刘俊杰
Email	Liujj255@mail2.sysu.edu.cn	完成日期	2024/5/15

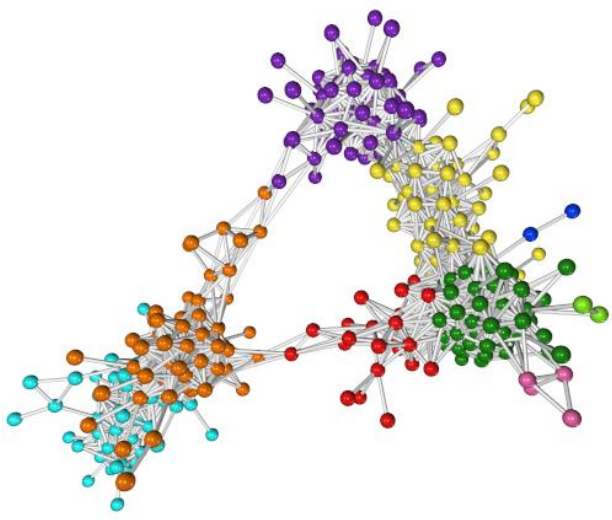
1. 实验目的

1. 使用任意并行框架实现多源最短路径搜索

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。

source	target	distance
0	1	1.0940993
2	3	1.107531
4	5	1.0544219
6	7	1.0324641
8	9	1.0407621
10	11	1.0787251
12	13	1.0722107
14	15	1.0137336
16	17	1.0486645
18	19	1.0966587
20	21	1.1073433
22	23	1.0598305
24	25	1.0783051
26	27	1.0860217

（a）邻接表



(b) 图可视化

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

2.1 实验思路

Floyd-Warshall 算法用于计算加权图中所有节点对之间的最短路径。该算法基于动态规划思想，通过中间节点的逐步添加来更新当前最短路径的长度。在单线程版本中，算法的时间复杂度为 O(n^3)，其中 n 是节点数量。为了加速该算法，可以利用多线程技术并行计算不同节点之间的最短路径。

2.2 实验过程

设计并实现多线程版本的 Floyd-Warshall 算法： 在本实验中，设计了使用多线程的 Floyd-Warshall 算法。通过将节点分配给不同的线程，每个线程负责计算部分节点之间的最短路径。这样可以提高算法的运行效率。

①生成随机邻接矩阵： 在实验中，生成了一个随机的邻接矩阵作为输入数据。这个邻接矩阵表示了一个加权有向图，其中每个元素代表两个节点之间的距离或权重。

②应用多线程 Floyd-Warshall 算法计算最短路径： 将生成的随机邻接矩阵作为输入，利用设计的多线程版本 Floyd-Warshall 算法计算所有节点对之间的最短路径。

③将结果输出到 CSV 文件中： 计算完成后，将得到的最短路径结果输出到 CSV 文件中，以便后续分析和可视化。

2.3 核心代码

①生成了一个随机的邻接矩阵作为输入数据。这个邻接矩阵表示了一个加权有向图，其中每个元素代表两个节点之间的距离或权重。

```c
// 生成随机邻接矩阵并输出到CSV文件
void generate_random_graph_to_csv(int n, const char *filename) {
    srand(time(NULL)); // 使用当前时间作为随机数种子

    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "Fail to open %s\n", filename);
        return;
    }

    // 写入标题行
    fprintf(file, "source,target,distance\n");

    // 生成随机邻接矩阵并将其写入文件
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j) {
                double distance = 0 + ((double)rand() / RAND_MAX) * (10 - 0);; // 生成1到10的随机距离
                fprintf(file, "%d,%d,%.2lf\n", i, j, distance);
            }
        }
    }

    fclose(file);
    printf("%s saved\n", filename);
}
```

②将邻接矩阵输出到 csv 文件中

```c
// 将邻接矩阵输出到CSV文件中
void print_graph_to_csv(double **graph, int n, const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "fail to open %s\n", filename);
        return;
    }

    // 写入标题行
    fprintf(file, "source,target,distance\n");

    // 写入邻接矩阵数据
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (graph[i][j] != INF) {
                fprintf(file, "%d,%d,%.2lf\n", i, j, graph[i][j]);
            }
            else fprintf(file, "%d,%d,%.2lf\n", i, j, 0.0);
        }
    }

    fclose(file);
    printf("saved %s\n", filename);
}

// 释放邻接矩阵的动态分配内存
```

③从 csv 文件中读取邻接矩阵

```c
// 从CSV文件中读取邻接矩阵
void read_graph_from_csv(double **graph, int n, const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        fprintf(stderr, "fail to open %s\n", filename);
        return;
    }

    // 跳过第一行（标题行）
    char line[256];
    fgets(line, sizeof(line), file);

    // 读取邻接矩阵数据
    while (fgets(line, sizeof(line), file) != NULL) {
        int source, target;
        double distance;
        sscanf(line, "%d,%d,%lf", &source, &target, &distance);
        graph[source][target] = distance;
    }

    fclose(file);
}
```

④使用 pthread 多线程实现并行 Floyd-Warshall 算法

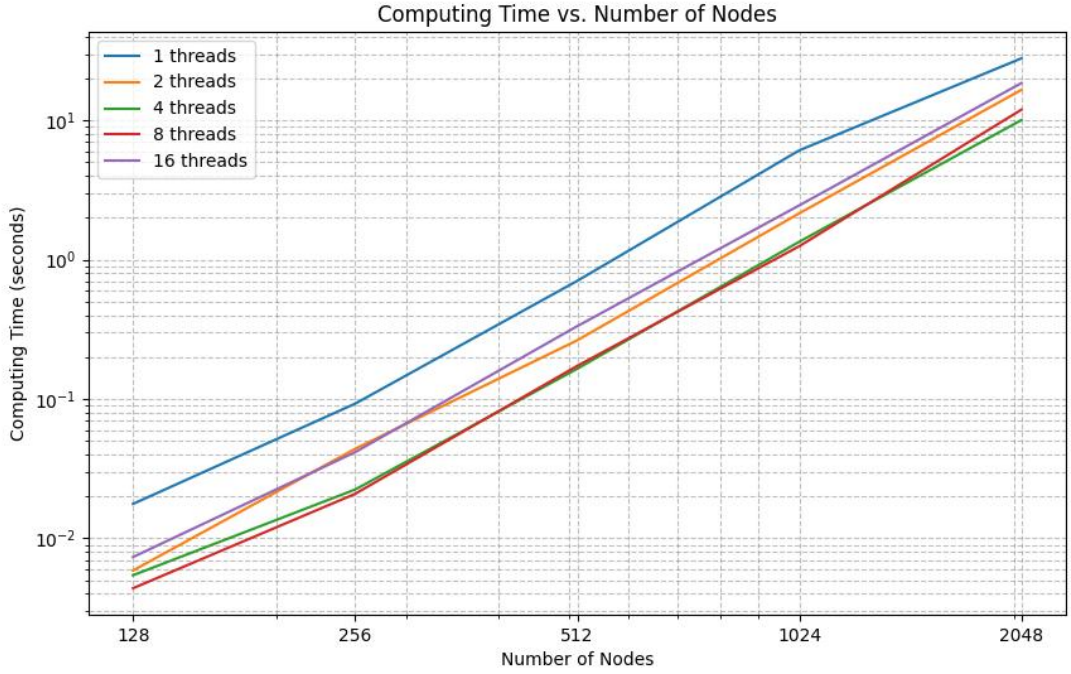
```c
// Floyd-Warshall算法实现，计算所有节点对之间的最短路径
void *floyd_warshall(void *arg) {
    int thread_id = *(int *)arg;
    int per = n / num_threads;
    int extra = n % num_threads;
    int start = thread_id < extra ? (1 + per) * thread_id : extra + per * thread_id ;
    int end = thread_id < extra ? (1 + per) * (thread_id + 1) : extra + per * (thread_id + 1);
    for (int k = 0; k < n; k++) {
        for (int i = start; i < end; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
return NULL;
}

void floyd_warshall_parallel(int num_threads) {

    pthread_t threads[num_threads];
    int thread_ids[num_threads];

    for (int i = 0; i < num_threads; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, floyd_warshall, &thread_ids[i]);
    }

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

3. 实验结果

3.1 实验结果展示

```
------------------
ljj@ljj-virtual-machine:~/parrallel_programming/Lab8$ gcc -o parallel floyd_parallel.c -pthread
ljj@ljj-virtual-machine:~/parrallel_programming/Lab8$ ./parallel
------------------
The num_threads:1
The num of nodes:128
  Computing time = 0.017671s
The num of nodes:256
  Computing time = 0.092672s
The num of nodes:512
  Computing time = 0.705672s
The num of nodes:1024
  Computing time = 6.108950s
The num of nodes:2048
  Computing time = 27.934701s
------------------
------------------
The num_threads:2
The num of nodes:128
  Computing time = 0.005877s
The num of nodes:256
  Computing time = 0.043937s
The num of nodes:512
  Computing time = 0.264180s
The num of nodes:1024
  Computing time = 2.157002s
The num of nodes:2048
  Computing time = 16.602200s
------------------
------------------
The num_threads:4
The num of nodes:128
  Computing time = 0.005432s
The num of nodes:256
  Computing time = 0.022412s
The num of nodes:512
  Computing time = 0.165022s
The num of nodes:1024
  Computing time = 1.348110s
The num of nodes:2048
  Computing time = 10.071274s
------------------
------------------
The num_threads:8
The num of nodes:128
  Computing time = 0.004384s
The num of nodes:256
  Computing time = 0.020871s
The num of nodes:512
  Computing time = 0.173033s
The num of nodes:1024
  Computing time = 1.251947s
The num of nodes:2048
  Computing time = 11.966243s
------------------
------------------
The num_threads:16
The num of nodes:128
  Computing time = 0.007342s
The num of nodes:256
  Computing time = 0.041568s
The num of nodes:512
  Computing time = 0.333765s
The num of nodes:1024
  Computing time = 2.459176s
The num of nodes:2048
  Computing time = 18.564995s
```

可视化对比：



3.2 实验结果分析

①随着节点数量的增加，计算时间呈指数增长： 无论线程数量如何，随着节点数量从 128 增加到 2048，计算时间都呈现出指数级增长。这表明在 Floyd-Warshall 算法中，节点数量对计算时间有显著影响，因为节点数量的增加会导致计算量的大幅增加。
②多线程并行计算显著提高了计算效率： 在所有节点数量下，随着线程数量的增加，计算时间明显减少。这表明使用多线程并行计算可以有效地利用多核处理器的优势，加快了计算速度。特别是在节点数量较大时，多线程并行计算的优势更加明显。
③线程数量对计算效率的影响有限： 尽管增加线程数量可以提高计算效率，但是当线程数量增加到一定程度后，进一步增加线程数量对计算效率的提升有限。在节点数量较小的情况下，增加线程数量可以显著降低计算时间，但是在节点数量较大时，增加线程数量对计算时间的影响逐渐减弱，甚至达到饱和状态。

4. 实验感想

通过本次实验我掌握多线程编程的基本技巧， 在实现多线程版本的 Floyd-Warshall 算法时，我学会了如何使用 pthread 库来创建和管理多个线程，以及如何合理地分配任务和同步线程之间的数据访问。这些基本技巧对于编写高效的并行程序非常重要。