中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab8-并行多源最短路径搜索

姓 名 马岱

学 号 22336180

学 院 计算机学院

专 业 计算机科学与技术

2025 年 5 月 14 日

1 实验目的

- 评估并行化最短路径算法的性能。
- 探索不同最短路径算法的并行适应性。

2 实验内容

实现并行的多源最短路径搜索，具体要求如下：

- 使用 OpenMP、Pthreads 或 MPI 中的任意一种并行编程模型来实现。

  你选择的并行框架是：OpenMP（静态、动态调度）

- 选用任意一种最短路径算法作为基础进行并行化，如 Bellman-Ford、Dijkstra（针对每个源点执行）、Floyd-Warshall 或 Johnson 算法。

  你选用的算法是：Floyd-Warshall 算法

2.1 算法简述

Floyd-Warshall 算法实现，采用三重循环结构，逐层枚举中间顶点 k，再对每一对顶点 (i, j) 判断是否存在通过顶点 k 的更短路径，并实时更新距离矩阵 $dist$ 中的值。判断条件 `dist[i][k] != INF && dist[k][j] != INF` 确保只有在两条路径均存在时，才进行路径长度更新，避免浮点无穷大相加导致的无效运算。

```
    for (int k = 0; k < n; ++k)
    {
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < n; ++j)
            {
                if (dist[i][k] != INF && dist[k][j] != INF)
                {
                    dist[i][j] = std::min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }
```

分别使用 `#pragma omp parallel for schedule(dynamic)` 指令将外层 i 循环的迭代任务划分为若干子任务，由多个线程动态调度执行和 `schedule(static)` 指令将 i

循环的所有迭代任务在程序启动时按块均匀分配给各线程，每个线程完成固定数量的循环迭代。

```
    switch (parallel_strategy)
    {
    case 0: // 默认策略 - 并行化 i 循环，动态调度
        for (int k = 0; k < n; ++k)
        {
#pragma omp parallel for schedule(dynamic)
            for (int i = 0; i < n; ++i)
            {
                for (int j = 0; j < n; ++j)
                {
                    if (dist[i][k] != INF && dist[k][j] != INF)
                    {
                        dist[i][j] = std::min(dist[i][j], dist[i][k] +
                        ↪ dist[k][j]);
                    }
                }
            }
        }
        break;
    case 1: // 并行化 i 循环，静态调度
        for (int k = 0; k < n; ++k)
        {
#pragma omp parallel for schedule(static)
            for (int i = 0; i < n; ++i)
            {
                for (int j = 0; j < n; ++j)
                {
                    if (dist[i][k] != INF && dist[k][j] != INF)
                    {
                        dist[i][j] = std::min(dist[i][j], dist[i][k] +
                        ↪ dist[k][j]);
                    }
                }
            }
        }
        break;
    }
```

根据 parallel_strategy 参数，选择使用不同的 OpenMP 并行调度策略。策略 0 对应动态调度，策略 1 对应静态调度。并通过对 Floyd-Warshall 算法的 i 循环进行并行化实现程序并行化。

从输入的邻接表文件中读取图的信息，并构建用于后续算法计算的邻接矩阵。初始化邻接矩阵 dist，矩阵大小为 $n \times n$，其中 n 为顶点数，初始值全部设为无穷大（∞），表示节点间暂不可达。将所有边的权重填入邻接矩阵对应位置，即对每条边 (u, v)，设 $\texttt{dist}[i2d[u]][i2d[v]] = w$。对角线元素（自身到自身）赋值为 0。

```
// 读取邻接表文件
Graph readGraph(const std::string &filename)
{
    //...
    // 创建顶点 ID 到索引的映射
    std::vector<int> vertex_ids(vertices.begin(), vertices.end());
    std::sort(vertex_ids.begin(), vertex_ids.end());
    graph.num_vertices = vertex_ids.size();
    graph.d2i.resize(graph.num_vertices);

    for (int i = 0; i < graph.num_vertices; ++i)
    {
        graph.i2d[vertex_ids[i]] = i;
        graph.d2i[i] = vertex_ids[i];
    }
    graph.dist.resize(graph.num_vertices,
    ↪ std::vector<float>(graph.num_vertices, INF));
    //...
    for (const auto &[vertex_id, neighbors] : edges)
    {
        int u = graph.i2d[vertex_id];
        for (const auto &[neighbor_id, weight] : neighbors)
        {
            int v = graph.i2d[neighbor_id];
            graph.dist[u][v] = weight;
        }
    }
    return graph;
}
```

构建两个映射关系：顶点 ID 到索引的映射：$i2d$[vertex_id] = index 和索引到顶点 ID 的映射：$d2i$[index] = vertex_id。该映射用于将不连续的顶点编号转换为连续的邻接矩阵索引。

3 实验结果与分析

3.1 实验配置简述

- **所用算法：** Floyd-Warshall 算法
- **并行框架：** OpenMP
- **并行方式：**外层循环 k 顺序执行，确保松弛操作的依赖性；中间循环 i 使用 OpenMP 并行化，采用 `schedule(dynamic)` 动态调度方式，将邻接矩阵不同行的松弛计算任务动态分配至不同线程执行。同时使用调度策略 `static` 对比测试不同调度方式下的性能差异。
- **测试数据特征：**
  - 数据集 1：节点数量 [930], 平均度数 [29.0774] (或边数量 [13521])
  - 数据集 2：节点数量 [525], 平均度数 [55.9657] (或边数量 [14691])

数据集 1 (节点数量: [930, 平均度数: [29.0774]) 的性能数据：

表 1: 数据集 1 ([930] 节点, 平均度数 [29.0774]) 上 [Floyd-Warshall] 的并行性能 (串行时间 T_s: [填写 T1 时间] 秒)

调度方式	线程数量 (p)	运行时间 T_p (秒)	加速比 $S_p = T_s/T_p$	并行效率 $E_p = S_p/p$
Dynamic	1	0.658649	1.0000	1.0000
Dynamic	2	0.370146	1.7794	0.8897
Dynamic	4	0.305309	2.1573	0.5393
Dynamic	8	0.624189	1.0552	0.1319
Dynamic	16	0.948470	0.6944	0.0434
Static	1	0.677864	1.0000	1.0000
Static	2	0.401997	1.6862	0.8431
Static	4	0.199373	3.4000	0.8500
Static	8	0.827815	0.8189	0.1024
Static	16	0.908699	0.7460	0.0466

针对数据集 2（节点数量：[525]，平均度数：[55.9657]）的性能数据：

表 2: 数据集 2 ([525] 节点, 平均度数 [55.9657]) 上 [Floyd-Warshall] 的并行性能 (串行时间 T_s: [填写 T1 时间] 秒)

调度方式	线程数量 (p)	运行时间 T_p（秒）	加速比 $S_p = T_s/T_p$	并行效率 $E_p = S_p/p$
Dynamic	1	0.190259	1.0000	1.0000
Dynamic	2	0.115078	1.6533	0.8267
Dynamic	4	0.127913	1.4874	0.3719
Dynamic	8	0.271509	0.7007	0.0876
Dynamic	16	0.616113	0.3088	0.0193
Static	1	0.171820	1.0000	1.0000
Static	2	0.109751	1.5655	0.7828
Static	4	0.113469	1.5142	0.3786
Static	8	0.607974	0.2826	0.0353
Static	16	0.627892	0.2736	0.0171

3.2 并行性能分析

根据你的实验数据，结合你选择的算法、并行框架、并行方式以及测试数据的特征（节点数量、平均度数等），分析程序的并行性能。可辅以图表（如加速比曲线、效率曲线）进行更直观的分析。

- 运行时间随线程数增加的变化趋势。

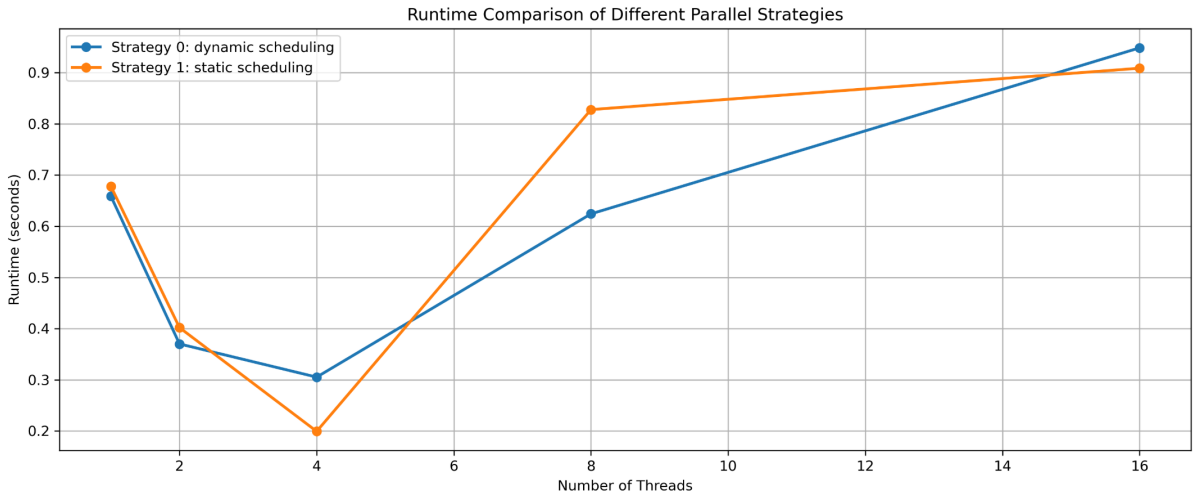

图 1: 930 节点（flower）运行时间-线程数

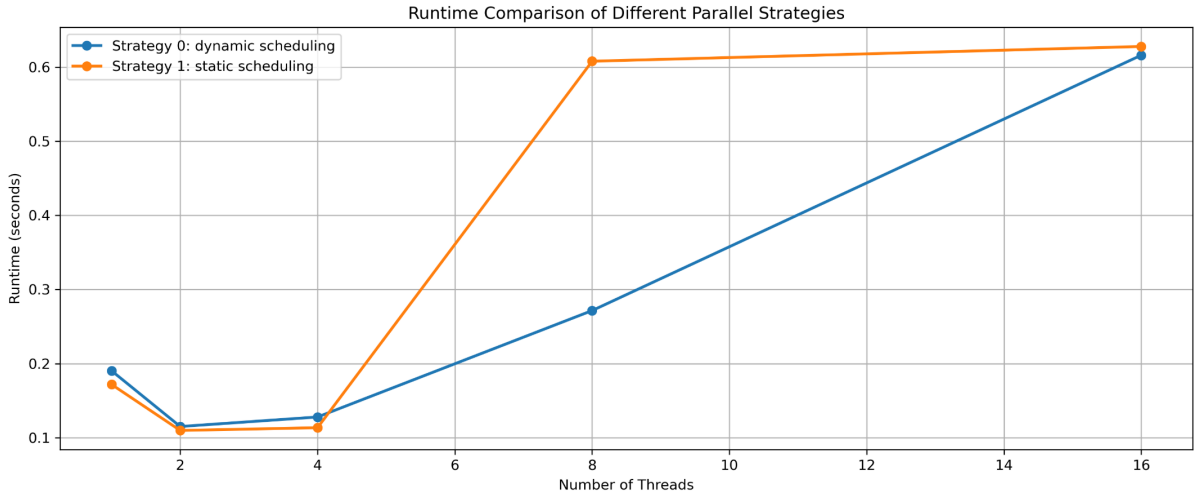

图 2: 525 节点（mouse）运行时间-线程数

回答：从实验结果来看，运行时间随着线程数的增加呈现出“先下降后上升”的典型 U 型变化趋势：在单线程情况下，两组数据集的运行时间分别较高，随着线程数从 1 增加到 4，算法能够有效将矩阵行的松弛操作分摊到多个核上，计算负载得到明显平衡，因而运行时间显著下降，尤其在 4 线程时分别达到最小值；但当线程数继续增加到 8 和 16 时，运行时间反而出现回升，且上升幅度随着线程数增长而加剧，这主要源自多线程并发访问共享内存和缓存带宽的竞争加剧，以及 OpenMP 调度与同步开销的非线性累积所致。在低线程数时，计算并行度带来的加速效益远超同步与调度成本，而一旦线程数超过了硬件的最佳并行度，额外的线程切换、线程同步和缓存不一致性导致运行效率下降。

- 加速比的变化情况。

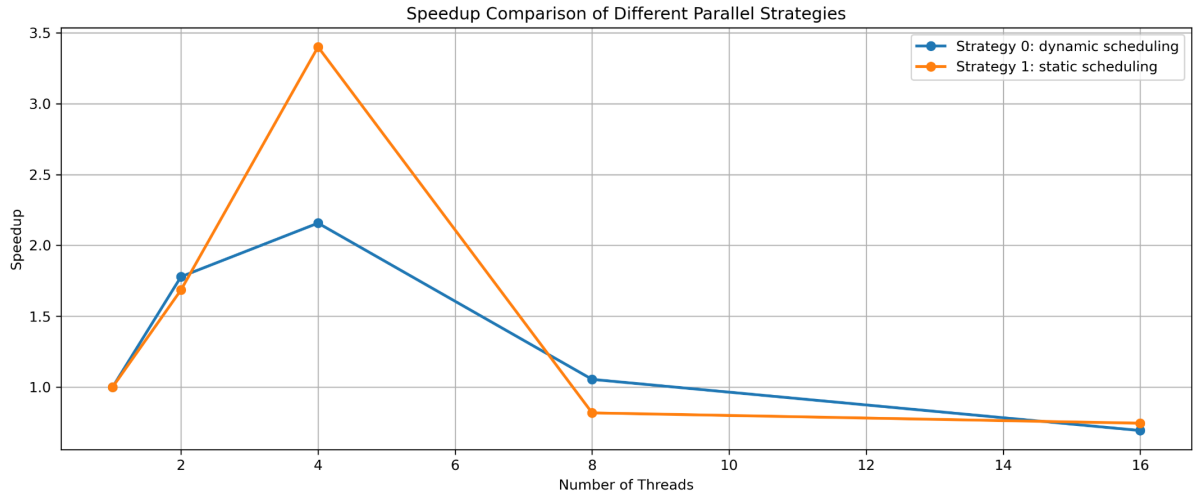

图 3: 930 节点（flower）加速比的变化情况

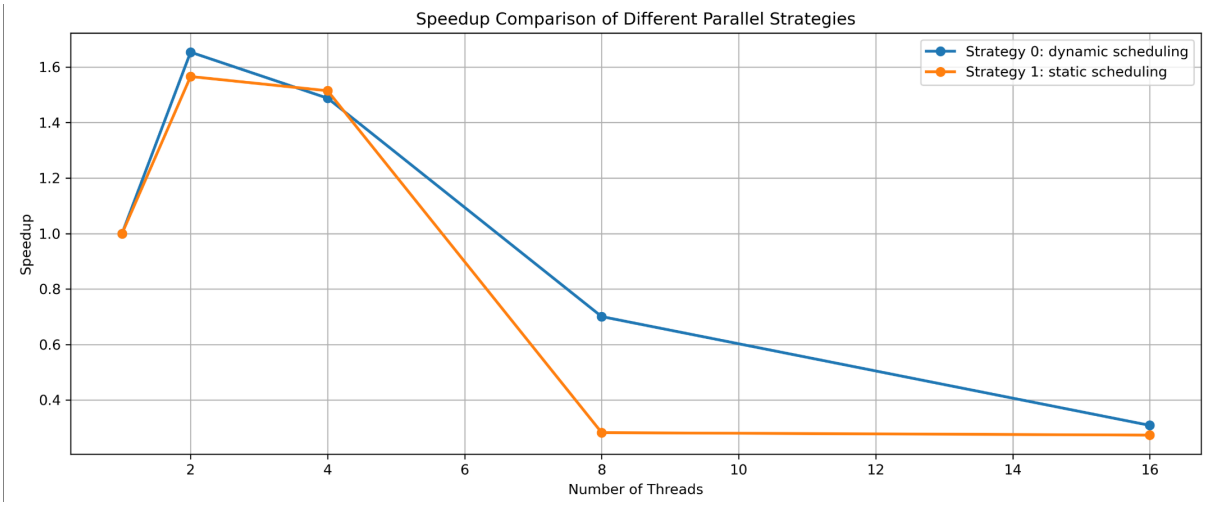

图 4: 525 节点（mouse）加速比的变化情况

回答：结合两组实验数据来看，加速比随线程数的变化也呈现出明显的“先上升后下降”趋势。以节点数 930 的数据集为例，动态调度下，随着线程数从 1 增加到 4，加速比由 1.00 提升至约 2.16；再增加到 8、16 线程时，加速比分别回落至约 1.06 和 0.69。静态调度则表现出超线性加速，在 4 线程时峰值达到 3.40，但同样在 8 和 16 线程时，加速比分别骤降至 0.82 和 0.75，表明额外线程并未带来更多计算收益，反而造成了更高的调度和同步开销。对于节点数 525 的较小数据集，动态调度的最优加速发生在 2 线程（ 1.78），在 4 线程后就开始回落，最高值低于 930 节点时的峰值；静态调度的峰值同样出现在 2 线程（ 1.69），而在 4 线程及以上均不足线性。总体而言，两种策略都能在低线程数下取得不错的并行加速，动态调度更适合负载微不平衡的场景，静态调度则在中等规模且访问局部性好的情况下可能产生超线性效果；线程数超过硬件的最佳并行度后，加速比都会迅速下降。

- 并行效率的变化情况，分析影响并行效率的主要因素。

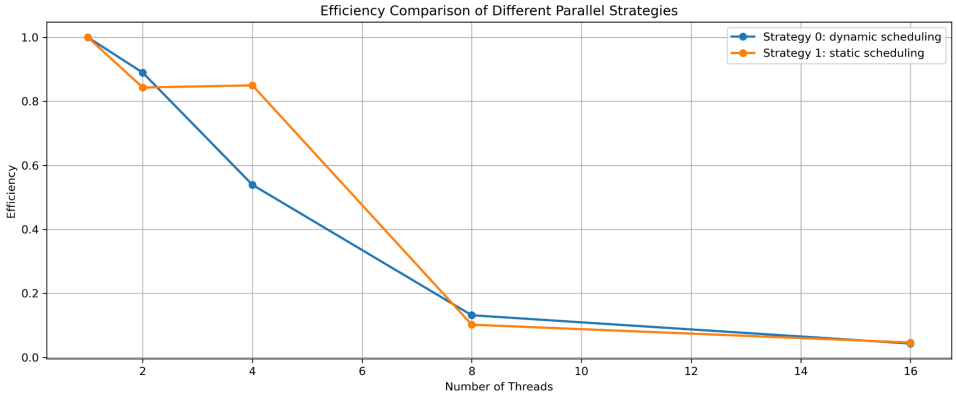

图 5: 930 节点（flower）并行效率的变化情况

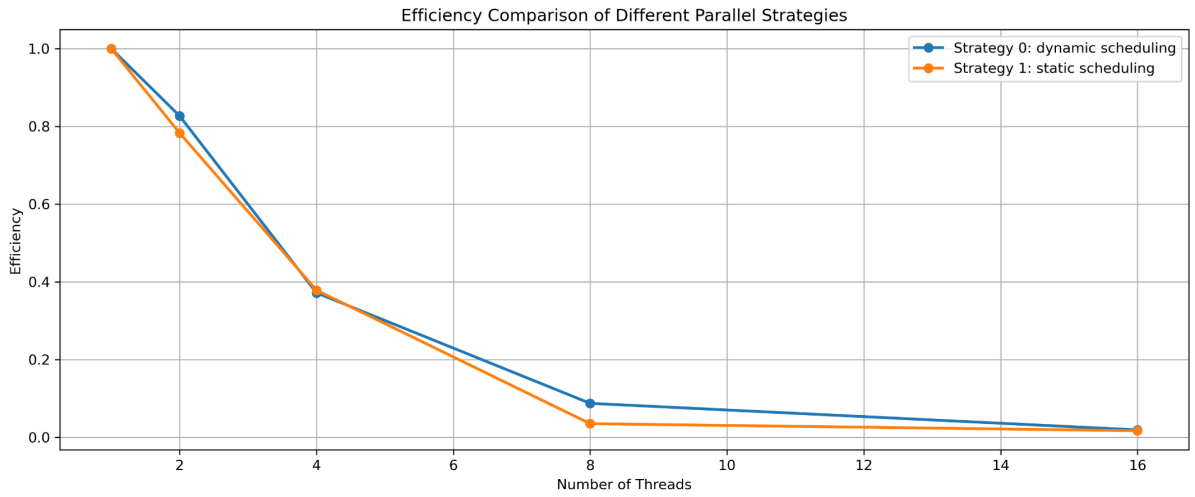

图 6: 525 节点（mouse）并行效率的变化情况

回答：从实验数据来看，并行效率随线程数增加总体呈现出显著的下降趋势：在单线程时效率为 1.00，随着线程数增至 2、4 时，效率保持在 0.5 ~ 0.9 区间（取决于数据集规模与调度策略），但当线程数进一步增加到 8、16 后，效率迅速跌至 0.1 以下，几乎丧失了并行增益。

影响并行效率的主要因素包括以下几点：

- 首先，负载均衡——在较低线程数下，任务足够大且划分合理时，各线程能充分分摊计算，但当线程过多或图的行计算量差异较大时，动态调度固然可以部分缓解不均衡，但静态调度则可能导致部分线程闲置，进而拉低整体效率；
- 其次，调度与同步开销——OpenMP 在每个迭代 k 上都会进行线程同步和任务分配，线程数越多，这部分开销以非线性方式累积，当调度成本接近或超过单线程执行时间时，并行效率便会骤降；再次，内存带宽与缓存一致性——Floyd–Warshall 算法属于内存访问密集型，矩阵元素频繁读写，当多个线程并发访问同一缓存行或内存通道时，会引发缓存抖动与带宽竞争，导致读写延迟激增；
- 最后，算法特性——$O(n^3)$ 算法本身缺乏粗粒度并行独立性，依赖于中间结果的更新，外层 k 循环无法并行，限制了可并行的总工作量，从而决定了并行效率在超出最佳线程数后必然下降。

综上所述，要提升并行效率，除了选择合适的线程数和调度策略，还需考虑改用块划分以增强缓存局部性、减少同步频率，或采用更高级的并行模型（如 MPI+OpenMP 混合、GPU 加速）来克服这些限制。

- 不同数据特征（如节点数量、平均度数）对并行性能的影响。

回答：不同数据特征对并行性能的影响主要体现在计算量与内存访问模式的差异上。

首先，节点数量越大，Floyd–Warshall 算法的三重循环计算量以 $O(n^3)$ 级数增长，导致每次松弛操作所需的浮点运算和内存读写显著增加；因此，在大节点数（如 930 节点）的数据集上，使用少量线程（2–4 线程）能够更充分地利用多核并行带来的计算加速，但由于更大规模矩阵带来的内存带宽占用和缓存擦写更为严重，当线程数继续增至 8 或 16 时，性能瓶颈会更早出现，并行效率急剧下降。

其次，平均度数反映了图的稠密度；度数越高，邻接矩阵中真正参与松弛计算的元素越多，这在中等规模（如 525 节点、平均度数 55.97）情况下能够更好地掩盖调度与同步开销，提升缓存命中率，因此静态调度在 2–4 线程时能取得接近或超线性的加速效果；而对于较稀疏图（如平均度数 29.08），行间计算量差异较大，动态调度通过动态分配剩余任务可以在低线程数下略微改善负载均衡，但其调度开销也会随线程增多而增长。

最后，边数量虽与平均度数相关，但更重要的是访问模式：稠密小图的局部性更好，缓存利用率较高；稀疏大图则频繁跨缓存行访问，导致缓存失效与带宽竞争增多。

综上所述，要在不同数据特征下实现最优并行性能，需要针对节点规模和稠密度选择合适的线程数及调度策略——中小规模稠密图偏好静态调度，大规模稀疏图可在少量线程下采用动态调度，并通过块划分、混合并行模型或异构加速等方法进一步提升性能。