Problem 3: List Ranking with Optimal Work and Depth

1 Problem Statement

Given a linked list represented as an array where each element points to its successor, determine the rank (distance from the head) of each node. We seek a parallel algorithm with:

- **Work:** $O(n)$
- **Depth:** $O(\log n)$

2 Solution Overview

The naive sequential algorithm has $O(n)$ work but $O(n)$ depth, as we must traverse the list sequentially. To achieve $O(\log n)$ depth, we need parallelism. The challenge is maintaining $O(n)$ total work while achieving logarithmic depth.

3 Algorithm: Independent Set with Pointer Jumping

3.1 Key Idea

The algorithm uses a divide-and-conquer approach:

1. Select an independent set of nodes (no two consecutive)
2. Remove these nodes, creating shortcuts in the list
3. Recursively solve on the shortened list (size $\approx n/2$)
4. Compute ranks of removed nodes from their neighbors' ranks

3.2 Data Structure

For each node i:

- `next`$[i]$: pointer to successor (`NIL` for tail)
- `dist`$[i]$: distance to `next`$[i]$ (initially 1, or 0 for tail)
- `rank`$[i]$: distance from head (output)

Algorithm 1 List Ranking with $O(n)$ Work, $O(\log n)$ Depth

```
1: function ListRank(next[], dist[])
2:     if list has only one node then
3:         rank[head] ← 0
4:         return
5:     end if
6:     // Phase 1: Select independent set
7:     S ← ∅                                              ▷ Independent set
8:     for each node i in parallel do
9:         if i has odd index and next[i] ≠ NIL then
10:            S ← S ∪ {i}                                ▷ Deterministic selection
11:        end if
12:    end for
13:    // Phase 2: Shortcut around independent set
14:    for each node i ∈ S in parallel do
15:        j ← next[i]                                    ▷ Successor of i
16:        k ← next[j]                                    ▷ Successor's successor
17:        for each predecessor p of i in parallel do
18:            next[p] ← j                                ▷ Skip node i
19:            dist[p] ← dist[p] + dist[i]                ▷ Update distance
20:        end for
21:    end for
22:    // Phase 3: Recurse on reduced list
23:    L' ← list with nodes in S removed
24:    ListRank(L')
25:    // Phase 4: Restore ranks of removed nodes
26:    for each node i ∈ S in parallel do
27:        p ← predecessor of i in original list
28:        rank[i] ← rank[p] + dist[p]
29:    end for
30: end function
```

3.3 Algorithm

3.4 Optimized Deterministic Version

For the independent set selection, we use a simple deterministic rule:

- Select every other node (odd positions) that has a successor
- This guarantees an independent set of size $\Theta(n)$

More sophisticated approaches (e.g., based on node degrees or colors) can improve practical performance.

4 Complexity Analysis

4.1 Work Complexity

Let $W(n)$ denote the total work for a list of size n.

- **Phase 1:** $O(n)$ work to select independent set
- **Phase 2:** $O(n)$ work to shortcut (constant work per node)
- **Phase 3:** $W(n/2)$ for recursion on reduced list
- **Phase 4:** $O(n)$ work to restore ranks

Recurrence: $W(n) = W(n/2) + O(n)$
Solution: $W(n) = O(n) + O(n/2) + O(n/4) + \cdots = O(n)$

4.2 Depth Complexity

Let $D(n)$ denote the depth for a list of size n.

- **Phase 1:** $O(1)$ depth (parallel selection)
- **Phase 2:** $O(1)$ depth (parallel shortcuts)
- **Phase 3:** $D(n/2)$ for recursion
- **Phase 4:** $O(1)$ depth (parallel rank computation)

Recurrence: $D(n) = D(n/2) + O(1)$
Solution: $D(n) = O(\log n)$

5 Alternative: Pointer Jumping (Suboptimal Work)

For comparison, the simpler pointer jumping algorithm:
This achieves:

- **Work:** $O(n \log n)$ (not optimal)
- **Depth:** $O(\log n)$

Algorithm 2 Basic Pointer Jumping

```
1: for k = 1 to ⌈log_2 n⌉ do
2:     for each node i in parallel do
3:         if next[i] ≠ NIL then
4:             dist[i] ← dist[i] + dist[next[i]]
5:             next[i] ← next[next[i]]
6:         end if
7:     end for
8: end for
```

6 Conclusion

The independent set approach achieves optimal complexity bounds:

- $O(n)$ work (matching sequential lower bound)
- $O(\log n)$ depth (optimal for comparison-based algorithms)

This demonstrates that list ranking can be efficiently parallelized without increasing the total work beyond the sequential complexity.