Parallel and (Nearly) Work-Efficient Dynamic Programming

Xiangyun Ding, Yan Gu, Yihan Sun

[University of California, Riverside]

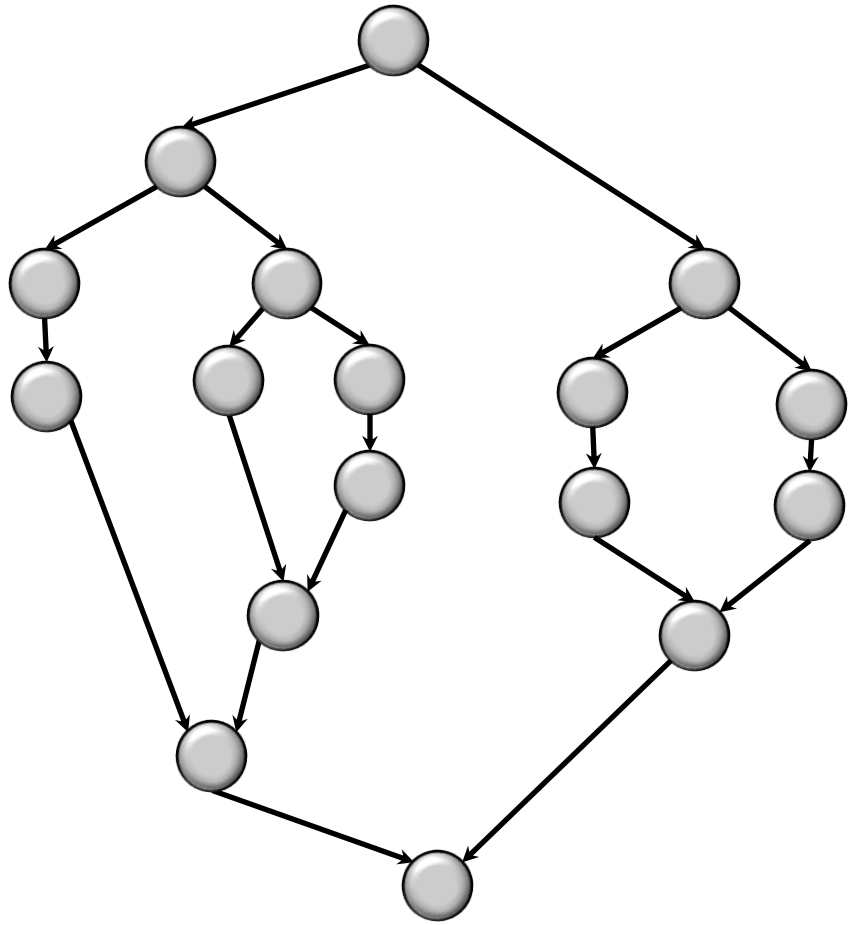
Jun 19, 2024

Full version paper: https://arxiv.org/abs/2404.16314

Code: https://github.com/ucrparlay/Parallel-Work-Efficient-Dynamic-Programming

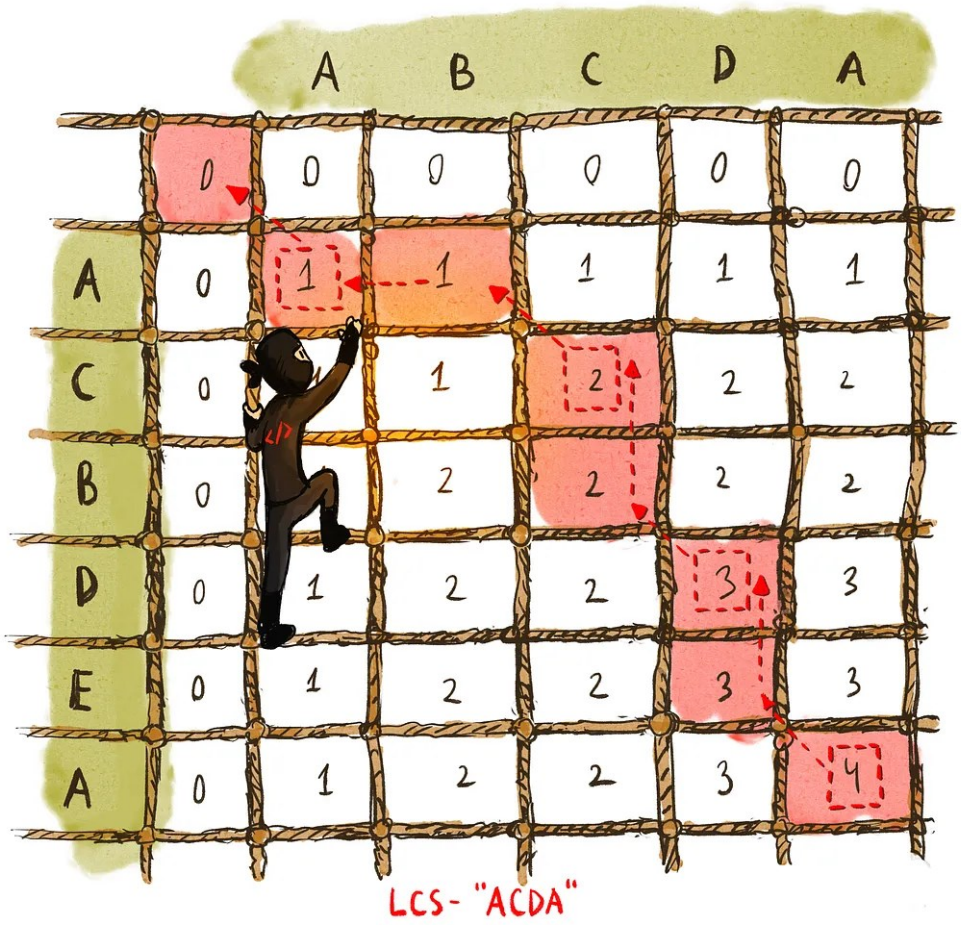
Computational Model

- Shared-memory multi-core setting
- Work-span model
  - Work: the total number of operations
  - Span (Depth): the length of the longest dependency chain
- An algorithm is
  - Work-Efficient: if it has $O(W)$ work
  - Nearly Work-Efficient: if it has $\tilde{O}(W)$ work
- where W is the work of the best-known sequential algorithm



Dynamic Programming (DP)

The Knapsack Problem



Longest Common Subsequence (LCS)

Example: Longest Common Subsequence (LCS)

- The longest common subsequence of $A[1 \dots n]$ and $B[1 \dots m]$
- Example: A B C B D A B
  B D C A B A
- $D[i, j]$: the LCS of $A[1 \dots i]$ and $B[1 \dots j]$

$$D[i,j] = \begin{cases} D[i-1, j-1] + 1 & \text{if } A[i] = B[j] \\ \max\{D[i-1, j], D[i, j-1]\} & \text{otherwise.} \end{cases}$$

DP **State**

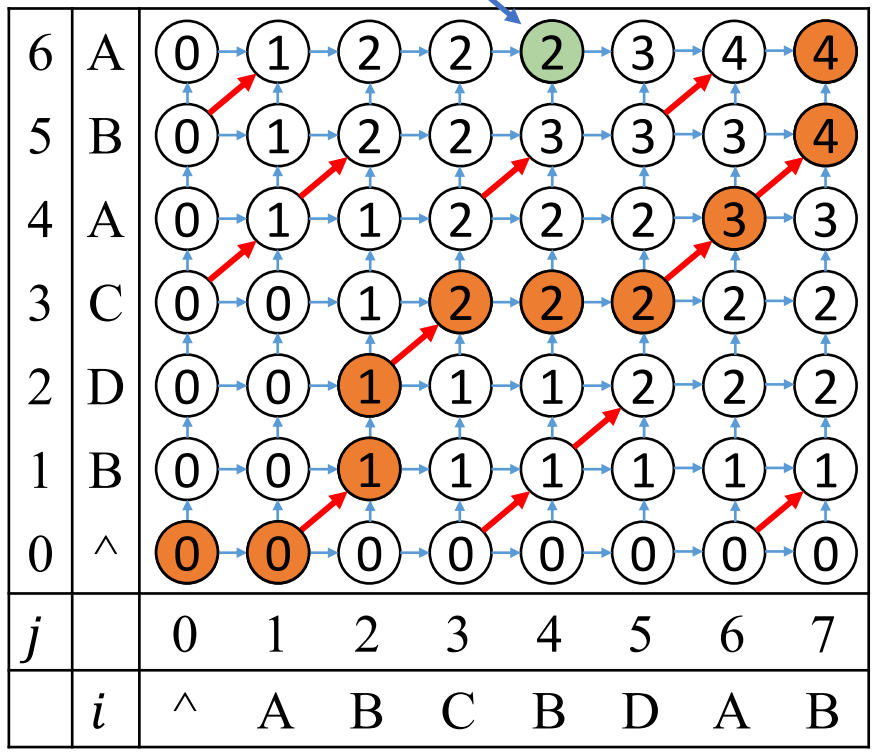
DP **Recurrence**

Example: Longest Common Subsequence (LCS)

$$D[i,j] = \begin{cases} D[i-1,j-1]+1 & \text{if } A[i]=B[j] \\ \max\{D[i-1,j], D[i,j-1]\} & \text{otherwise.} \end{cases}$$

DP **State**

DP **Recurrence**



Transition/Relaxation:
Use one DP state to update another DP state

DP **DAG**:
The DAG formed by DP states and DP transitions

Existing attempts on parallelizing DP

- Many existing works (Longest Increasing Subsequence (LIS): [Krusche et al. 2009], [Shen et al. 2022], [Cao et al. 2023], [Gu et al. 2023], Longest Common Subsequence (LCS): [Lu et al. 1994], [Babu et al. 1997], [Xu et al. 2005], [Tchendji et al. 2020], Least Weight Subsequence (LWS): [Apostolico et al. 1990], [Larmore et al. 1995], Optimal Alphabetic Tree (OAT): [Rytter 1988], [Larmore et al. 1993], [Larmore et al. 1996])
- However, most of these algorithms use asymptotically more work than the best sequential algorithms
- Example: parallel LCS
  - Existing works use $O(mn)$ work [Lu et al. 1994], [Babu et al. 1997], [Xu et al. 2005], [Tchendji et al. 2020]
  - But there exist faster sequential algorithms [Hirschberg 1977], [Hunt et al. 1977]
- Only some recent works tried to match the "fastest" sequential DP algorithms, but only on specific problems
  - [Shen et al. SPAA 2022], [Cao et al. SPAA 2023], [Gu et al. SPAA 2023]

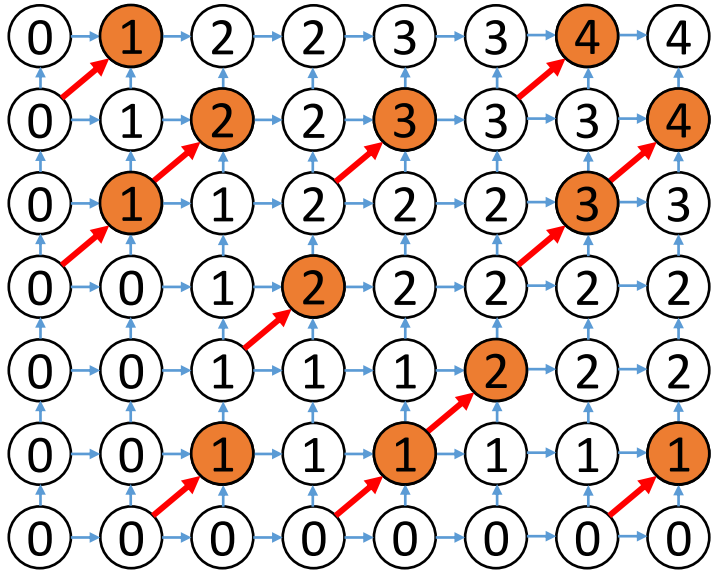
Not **Work-Efficient**!

Why work-efficient parallel DP is so hard?

- Surprisingly, one major reason we realized is that, sequential DP algorithms are extremely well-optimized!
  - Fancy techniques to skip edges/vertices in the DP DAG
  - Usually save a polynomial factor of time by avoiding these computations

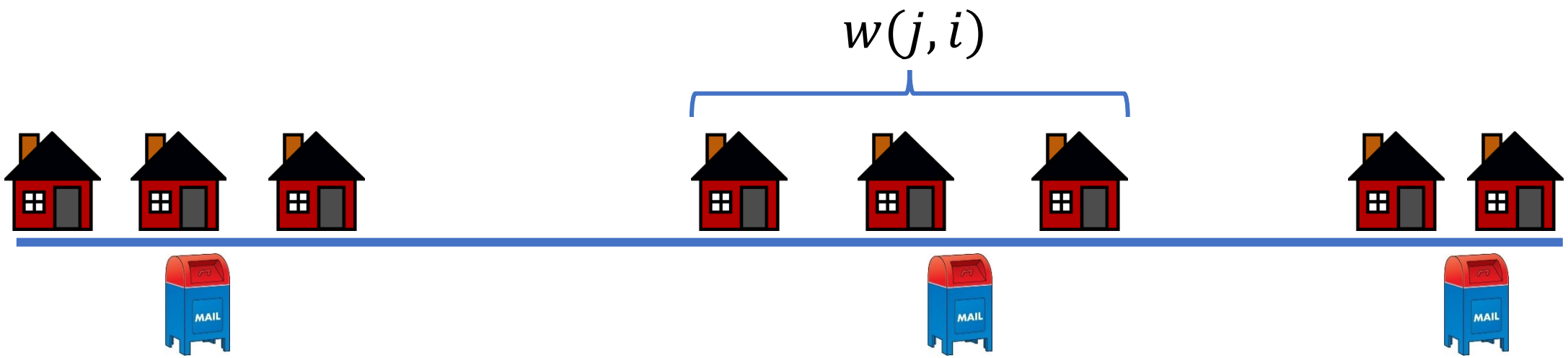
Example: Longest Common Subsequence (LCS)

- The existing parallel solutions have $O(mn)$ work
- Sequential optimization: **Sparsification** [Hirschberg 1977], [Hunt et al. 1977]
  - LCS can be computed in $\tilde{O}(L)$ work, where L is the number of "red arrows" (pairs (i, j) such that $A[i] = B[j]$)
  - Skip edges and vertices in the DP DAG



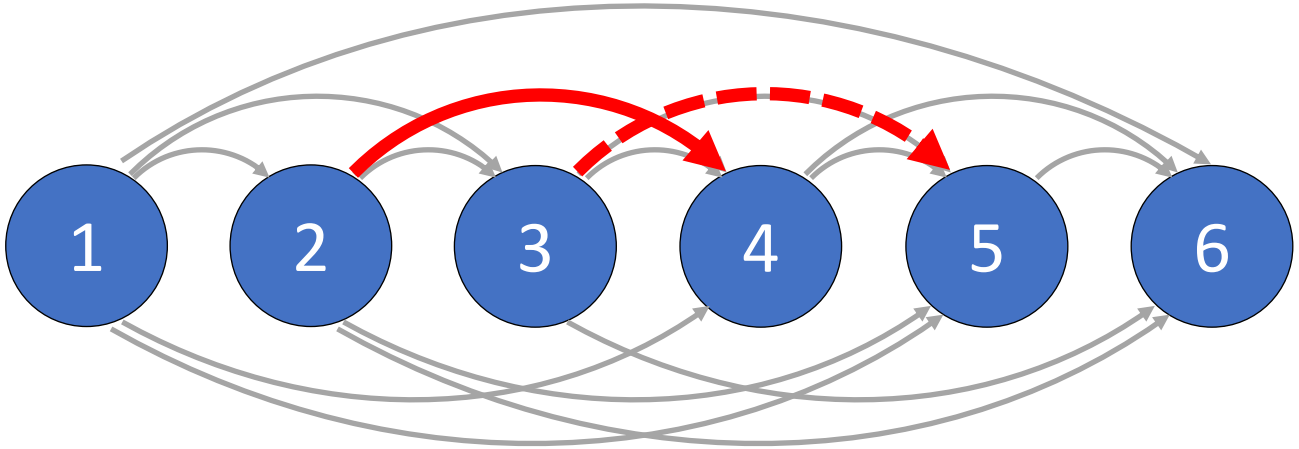
Another Example: LWS with Decision Monotonicity

- The least weight subsequence (LWS) problem (1D-clustering)
- DP recurrence: $D[i] = \min_{j<i} D[j] + w(j,i)$
- Example: the mailbox problem
  - There are n villages in a row, and we want to build some mailboxes
  - $w(j,i)$ is the cost to build one mailbox for villages from $j+1$ to i



Another Example: LWS with Decision Monotonicity

- DP recurrence: $D[i] = \min_{j<i} D[j] + w(j, i)$
- $O(n)$ states and $O(n^2)$ transitions in the DP DAG
- Decision Monotonicity: the **best decisions** are **non-decreasing**
  - Best decision: $D[i]$ is computed from $\text{best}[i]$
  - $\text{best}[4] \leq \text{best}[5]$
- Sequential algorithms can achieve $\tilde{O}(n)$ work
  - Skip useless edges in the DP DAG



Our Goal

Sequential DP Algorithms:

✓ **Highly optimized**

✗ **Iterative (no parallelism)**

Parallel DP Algorithms:

✓ **Good parallelism**

✗ **Not work-efficient**

- Bridge this gap
- **Parallel** and (nearly) **Work-Efficient** DP algorithms!

Contributions in this paper

- A general framework to parallelize almost all DP algorithms
  - The Cordon Algorithm
- Techniques to achieve work-efficiency
  - Prefix-doubling
  - Parallel data structures
- New parallel algorithms for classic DP problems
  - Sparse LCS
  - Convex/Concave (generalized) LWS
  - Optimal Alphabetic Tree (OAT)
  - GAP Edit-Distance
  - LWS on trees, etc.

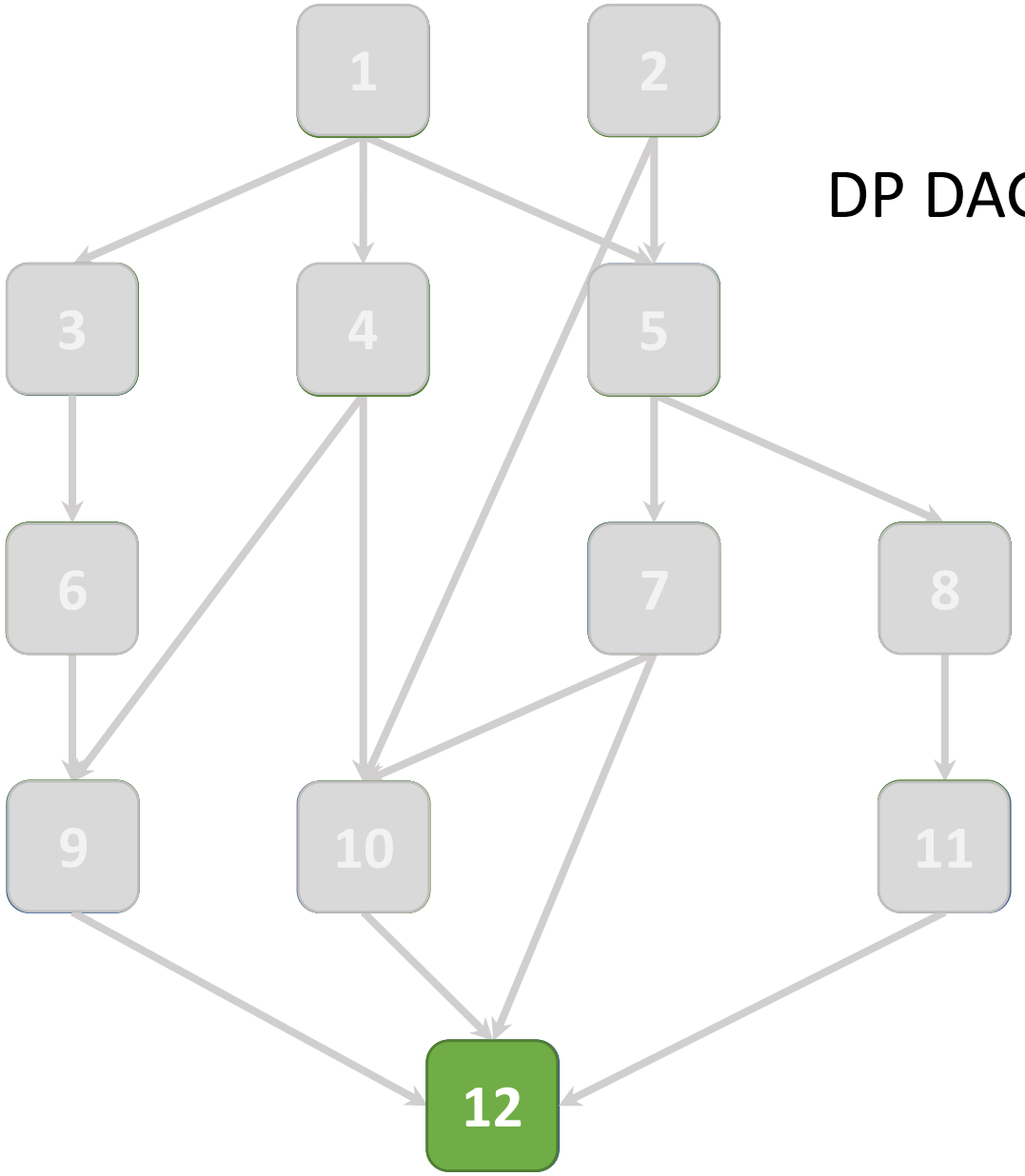
Background

- The phase-parallel framework [Shen, 2022]
  - Break the computation into several rounds
  - In each round, work on the frontier in parallel
- In this idea, parallel LCS will require $O(n)$ rounds to finish

1 : Finalized DP States

1 1 : Tentative DP States

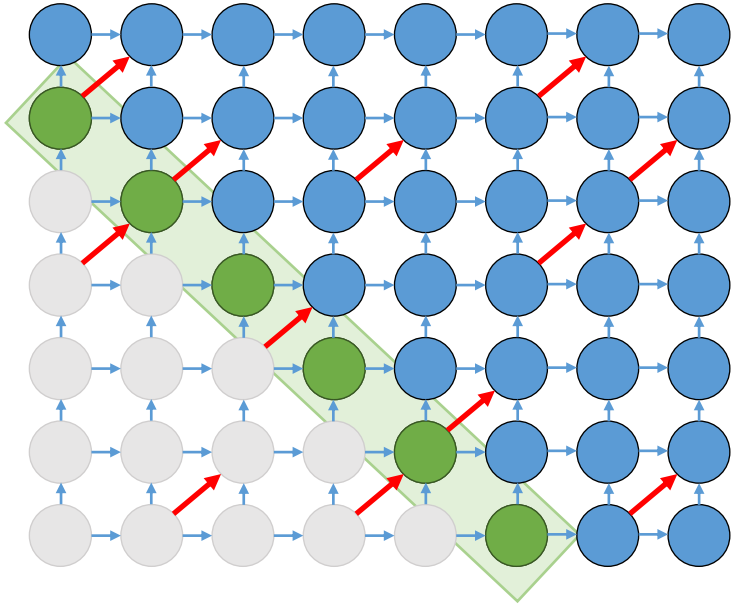
1 : Ready DP states



DP DAG

Background

- The phase-parallel framework [Shen, 2022]
  - Break the computation into several rounds
  - In each round, work on the frontier in parallel
- In this idea, parallel LCS will require $O(n)$ rounds to finish



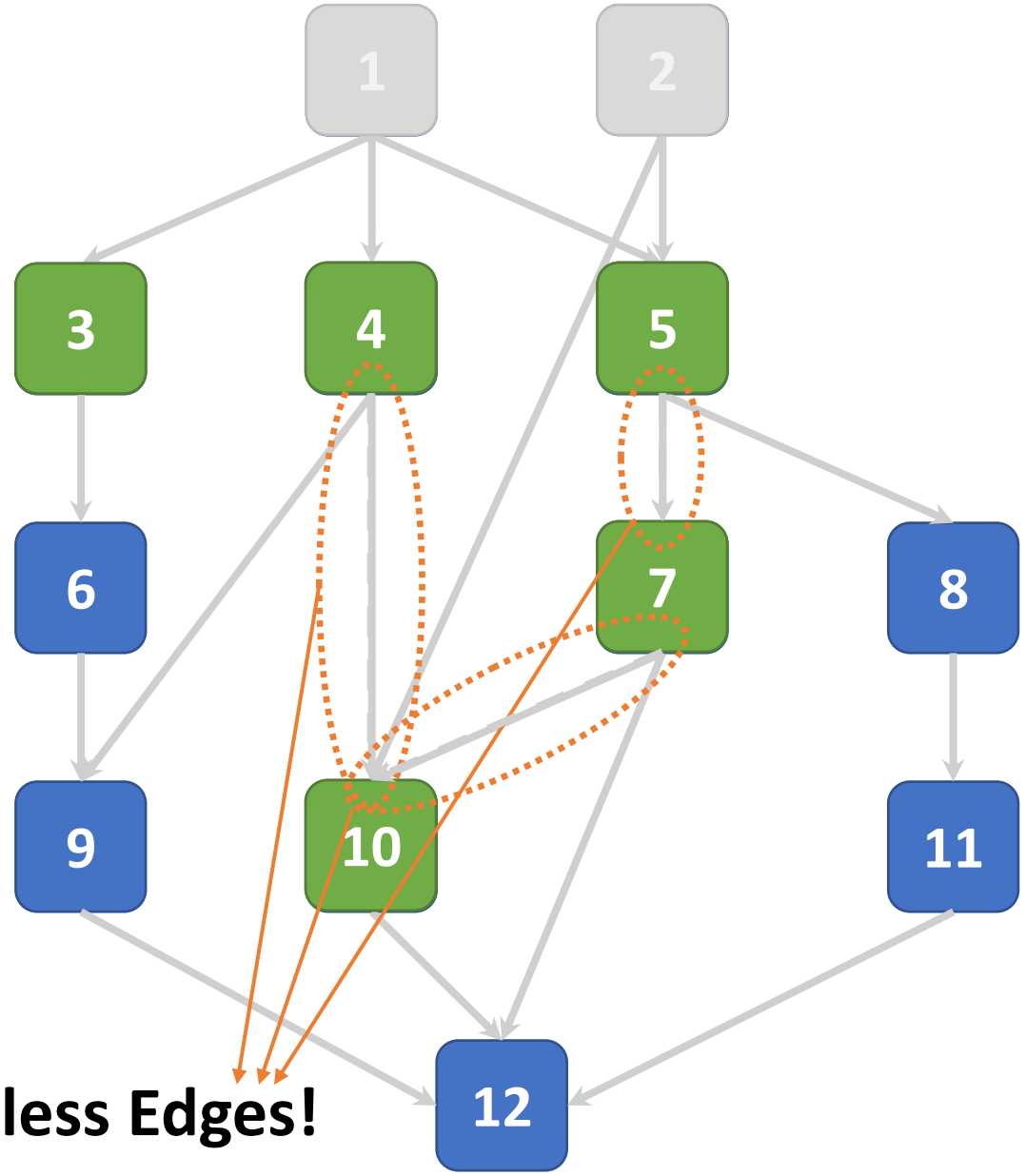
1 : Finalized DP States

1 1 : Tentative DP States

1 : Ready DP states

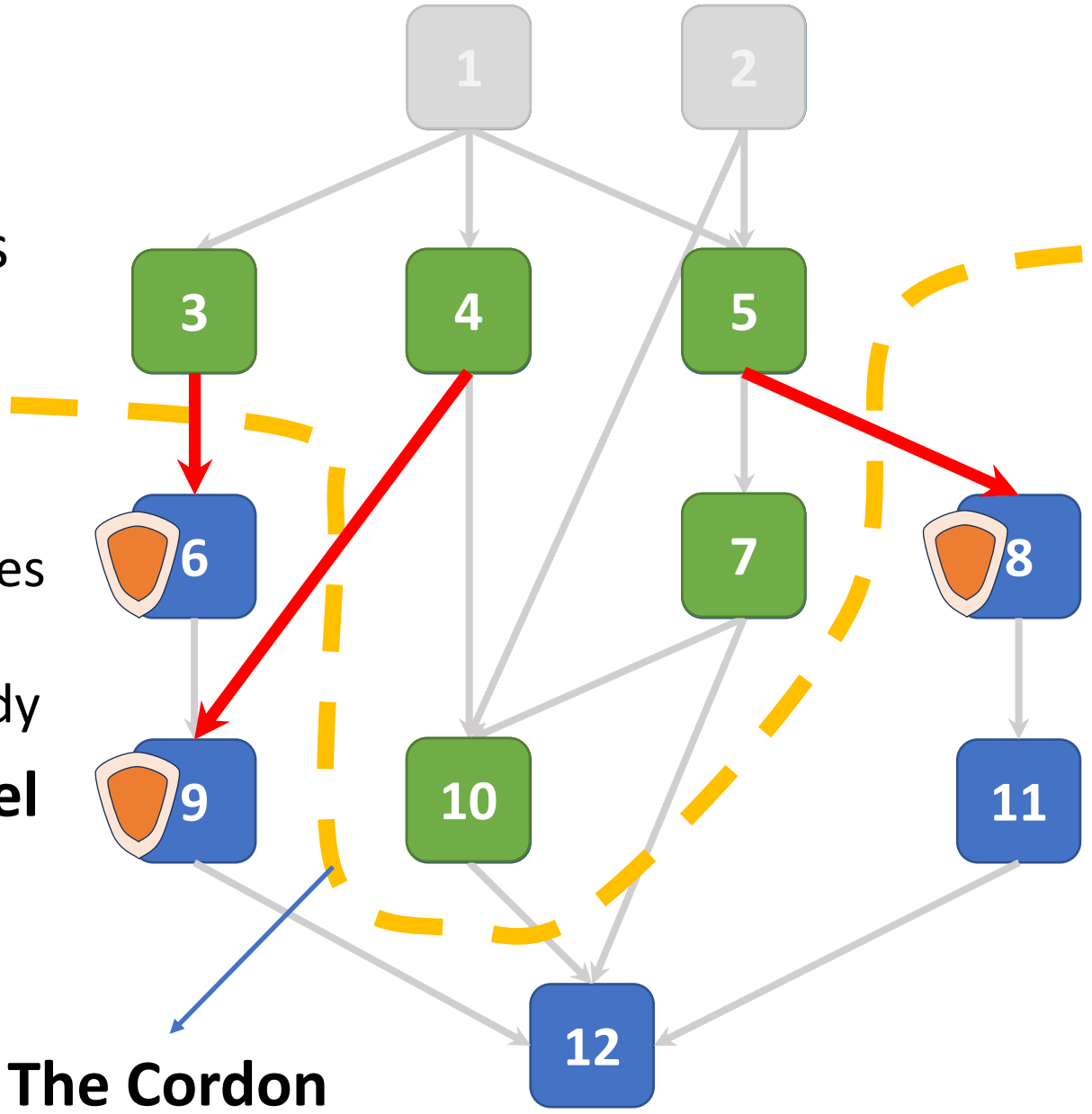
Background

- However, in an optimized algorithm, **some edges can be skipped**
- This can give us **more ready states** in this round
- Question: how to exactly find all ready states?



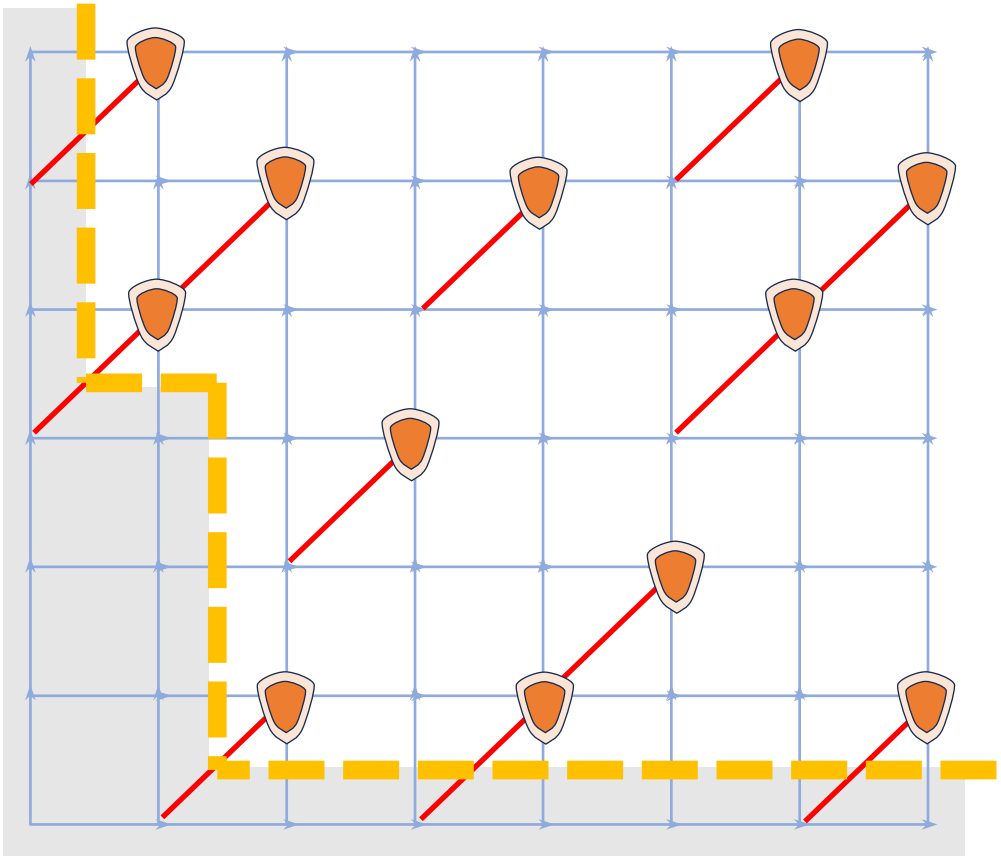
The Cordon Algorithm

- Idea: find all the **successful relaxations** between tentative states
- If a tentative state j can successfully relax tentative state i, we **put a sentinel** at state i
  - Then state i and all its descendant states are un-ready.
  - Example: states 6, 9, and 12 are unready
- **A state is ready if there is no sentinel on any of its ancestors (inclusive).**



Parallel LCS

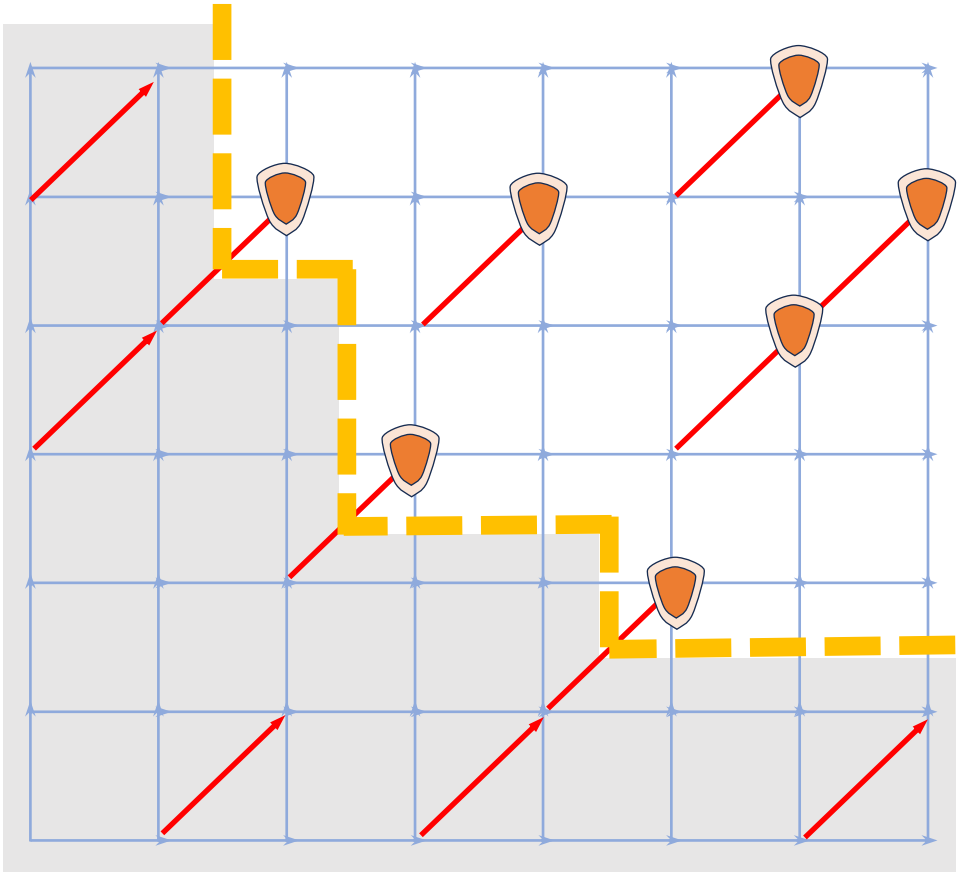
- Maintain a global counter: $k = 0$
  - Current DP value for tentative states
- Only the red arrows (where $A[i]$ and $B[j]$ match) are successful relaxations
- Put sentinels at the end of red arrows
- The cordon is a **stair-case**
- The finalized states have DP value 0
- The tentative values are updated to 1



- $$D[i,j] = \begin{cases} D[i-1,j-1]+1 & \text{if } A[i] = B[j] \\ \max\{D[i-1,j], D[i,j-1]\} & \text{otherwise.} \end{cases}$$

Parallel LCS

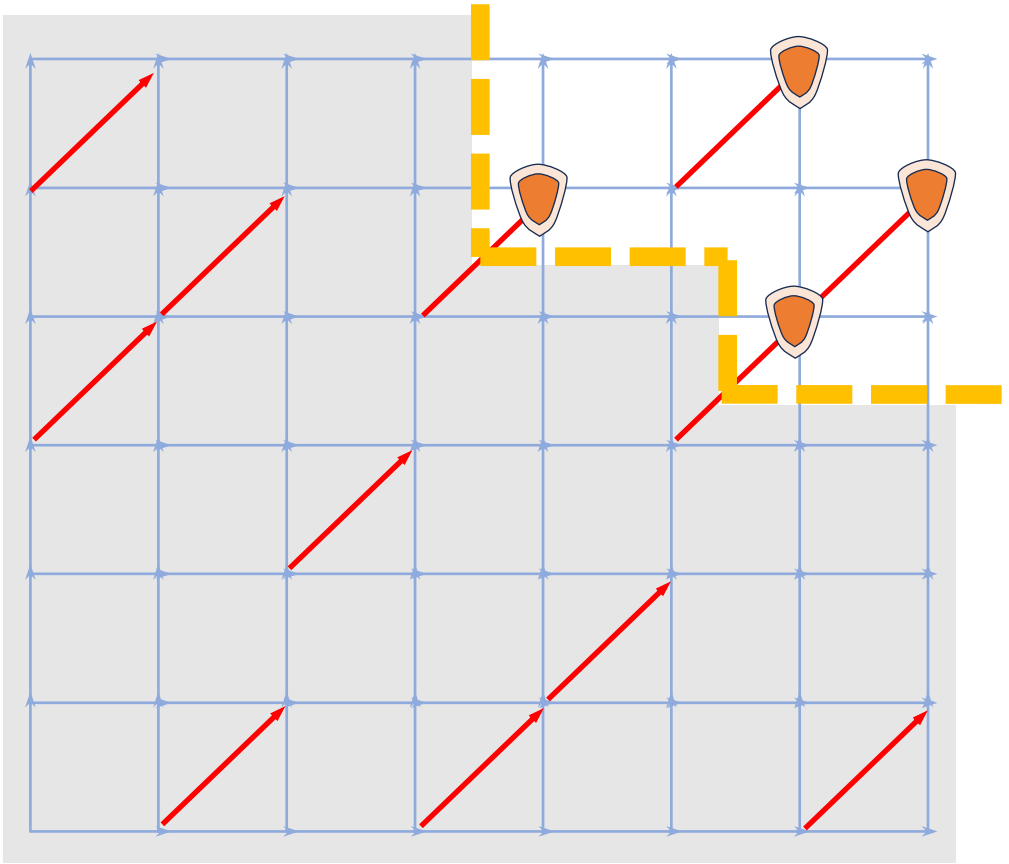
- In the next round, increase the counter: $k = 1$
- Repeat the steps and push the stair-case region
- The new finalized states have DP value 1



- $D[i,j] = \begin{cases} D[i-1,j-1]+1 & \text{if } A[i] = B[j] \\ \max\{D[i-1,j], D[i,j-1]\} & \text{otherwise.} \end{cases}$

Parallel LCS

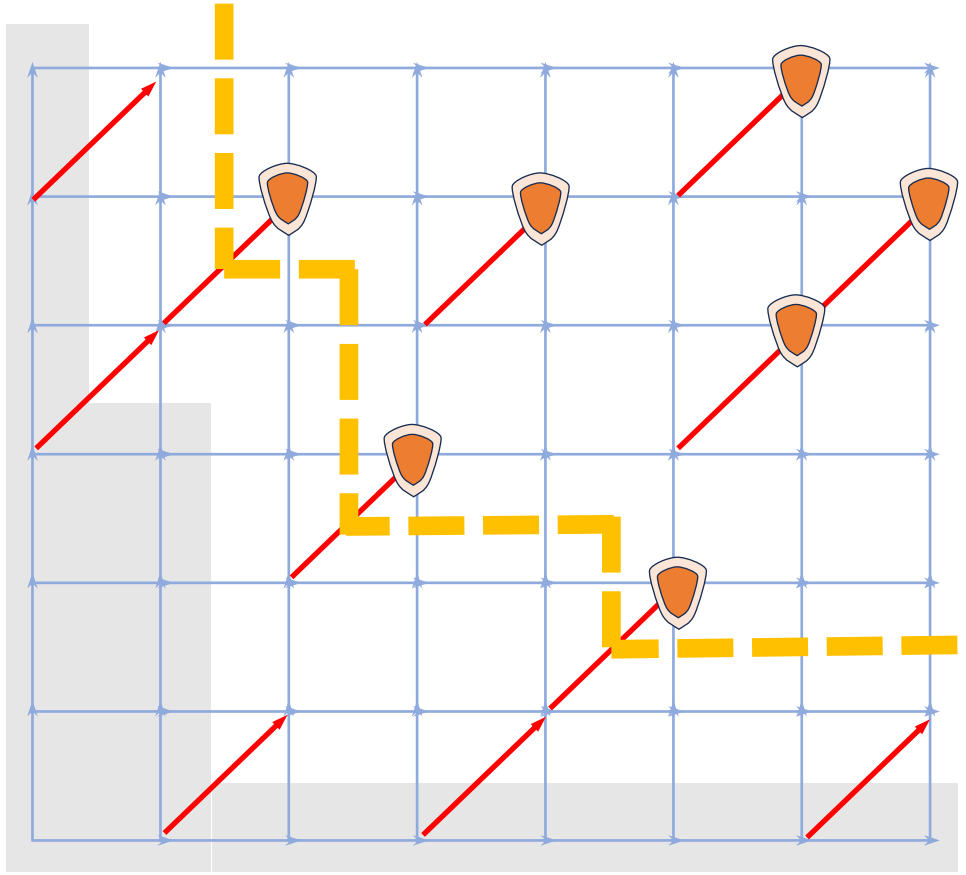
- In the next round, increase the counter: $k = 2$
- Repeat the steps until there is no tentative states
- Finally we have k is the LCS length



- $$D[i,j] = \begin{cases} D[i-1,j-1]+1 & \text{if } A[i] = B[j] \\ \max\{D[i-1,j], D[i,j-1]\} & \text{otherwise.} \end{cases}$$

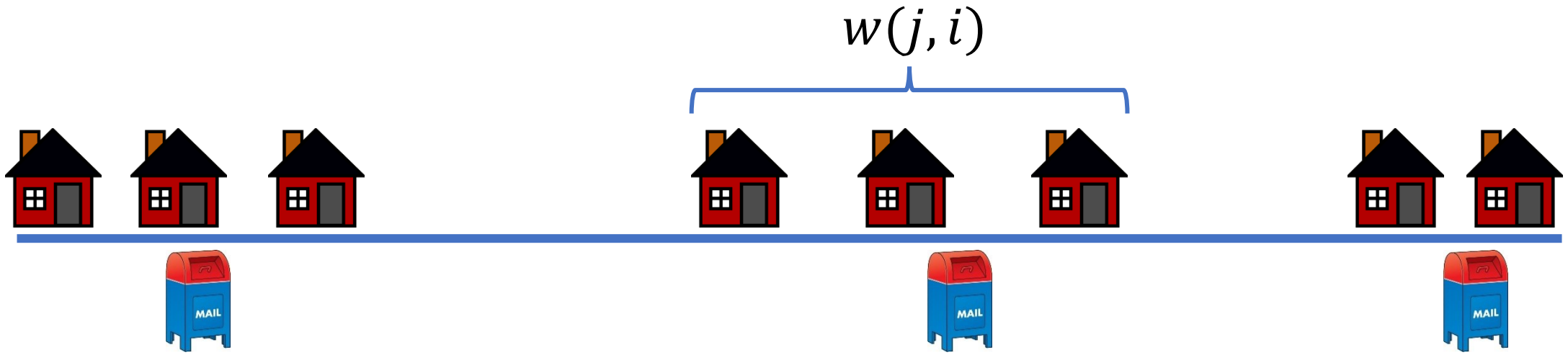
Parallel LCS

- The cordon algorithm identifies the set of ready states
- Question: how to find the red arrows **efficiently**?
  - Parallel data structures!
- Use a parallel winning tree [Gu et al. 2023] to maintain the location of red arrows
- Work: $O(L \log n)$
  - L is the number of pairs (i, j) such that $A[i] = B[j]$
- Span: $O(k \log n)$
  - k is the LCS length



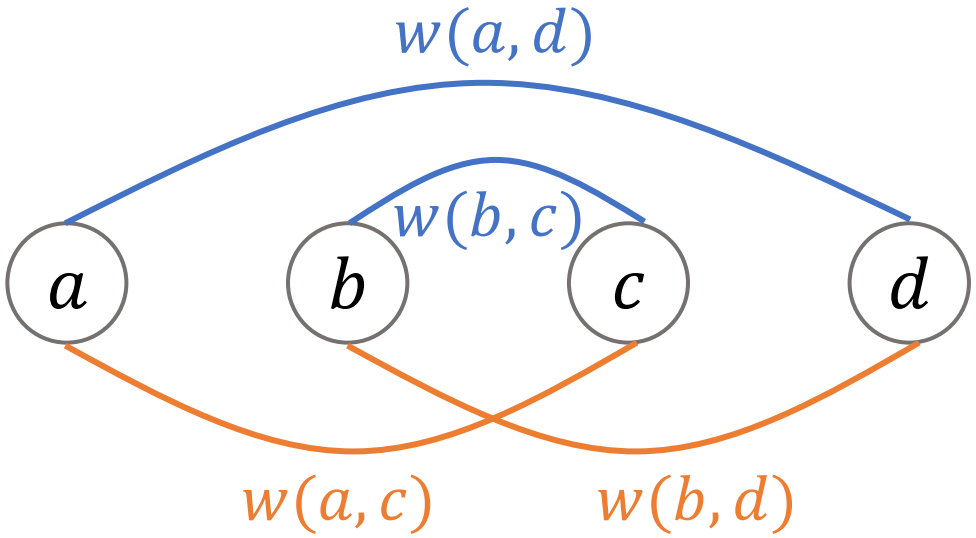
LWS with Decision Monotonicity

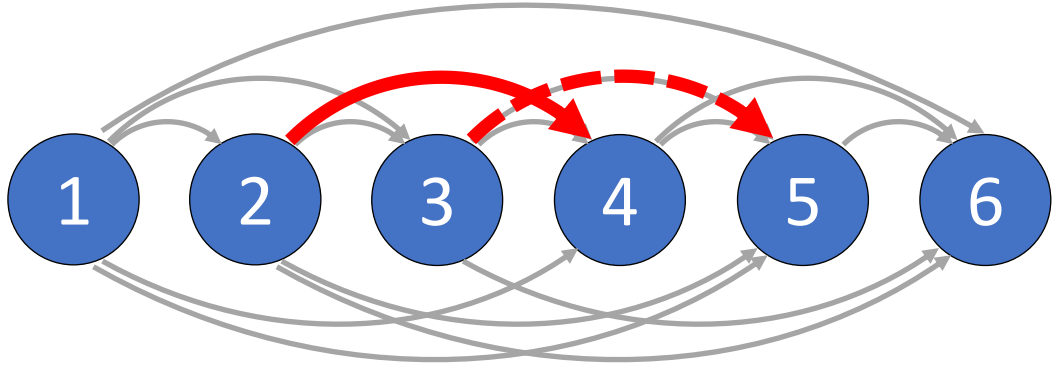
- DP recurrence: $D[i] = \min_{j<i} D[j] + w(j, i)$
- Example: the mailbox problem
  - There are n villages in a row, and we want to build some mailboxes
  - $w(j, i)$ is the cost to build one mailbox for villages from $j + 1$ to i



Decision Monotonicity (DM)

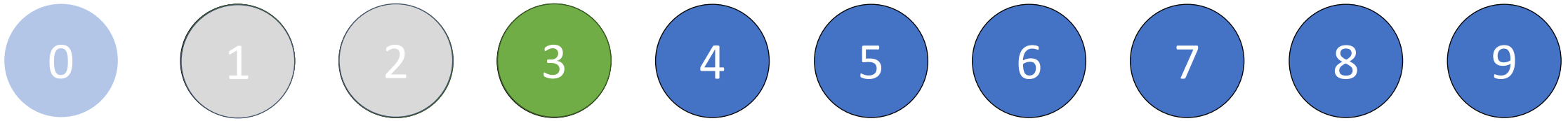
- DP recurrence: $D[i] = \min_{j<i} D[j] + w(j,i)$
- **Convexity**: $w(a,c) + w(b,d) \leq w(b,c) + w(a,d)$
  - Also known as **Quadrangle Inequality**
- If $w(j,i)$ is convex, then we have decision monotonicity (DM)
- With DM, Sequential algorithms can have $\tilde{O}(n)$ work





Review: Sequential Convex LWS

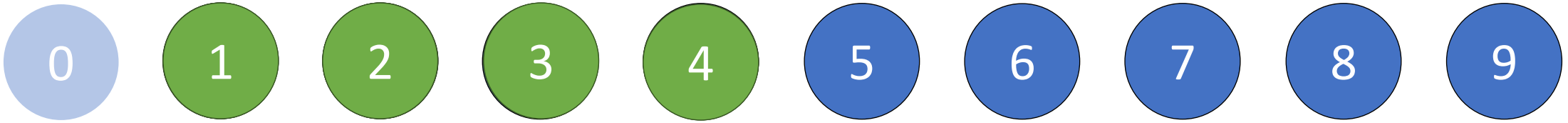
- DP recurrence: $D[i] = \min_{j<i} D[j] + w(j, i)$
- Key idea: maintain the **best decisions** for all states
- After computing $D[i]$, we use it to update the best decisions of $i + 1 \ldots n$
- By decision monotonicity, $D[i]$ **dominates a consecutive range at the end**
- Use data structure to maintain the best decision array



Best Decisions	0	0	0	1	3	3	3	3	3

Parallel Convex LWS

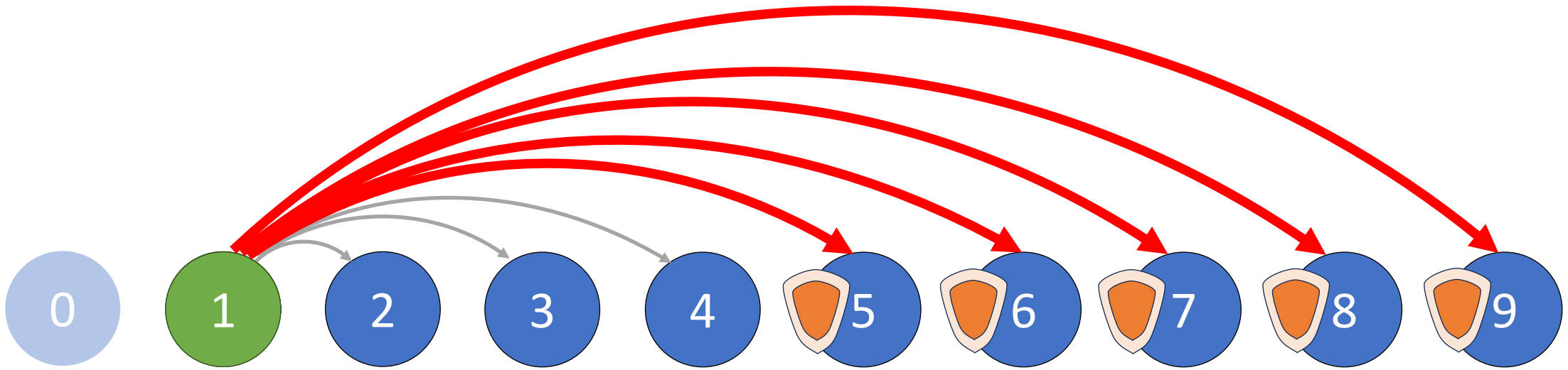
- DP recurrence: $D[i] = \min_{j<i} D[j] + w(j,i)$
- We want to process as many states as possible in parallel
- Challenges:
  - States can interact with each other when updating the best decisions
  - Achieving high parallelism without sacrificing work
    - Should not check unnecessary states and transitions



Best Decisions	0	0	0	0	1	3	3	4	4

Parallel Convex LWS

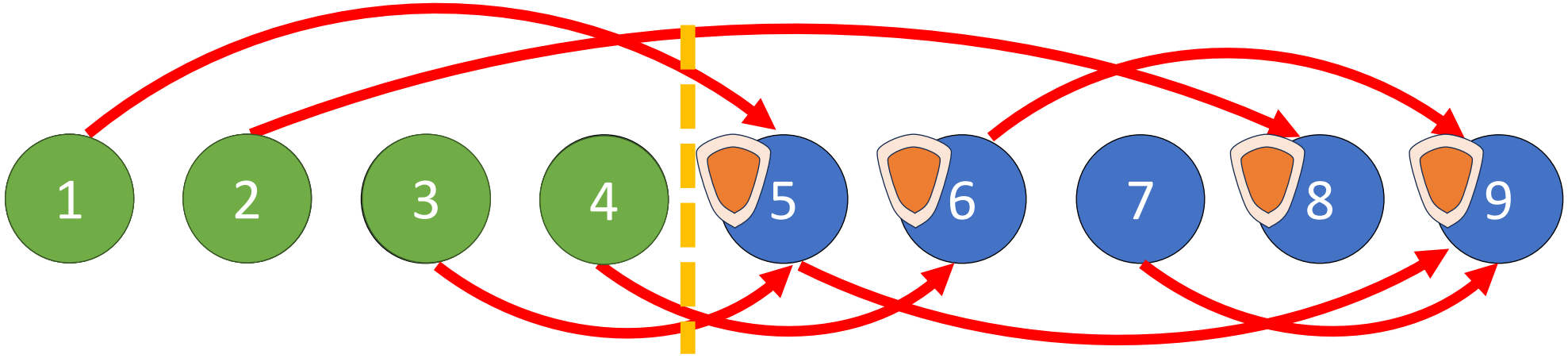
- Apply the cordon algorithm!
  - Find all successful relaxations between tentative states
  - Put sentinels at the states that can be updated
  - As the DAG flows from left to right, only the **left-most** sentinel matters!



Best Decisions	0	0	0	0	1	1	1	1	1

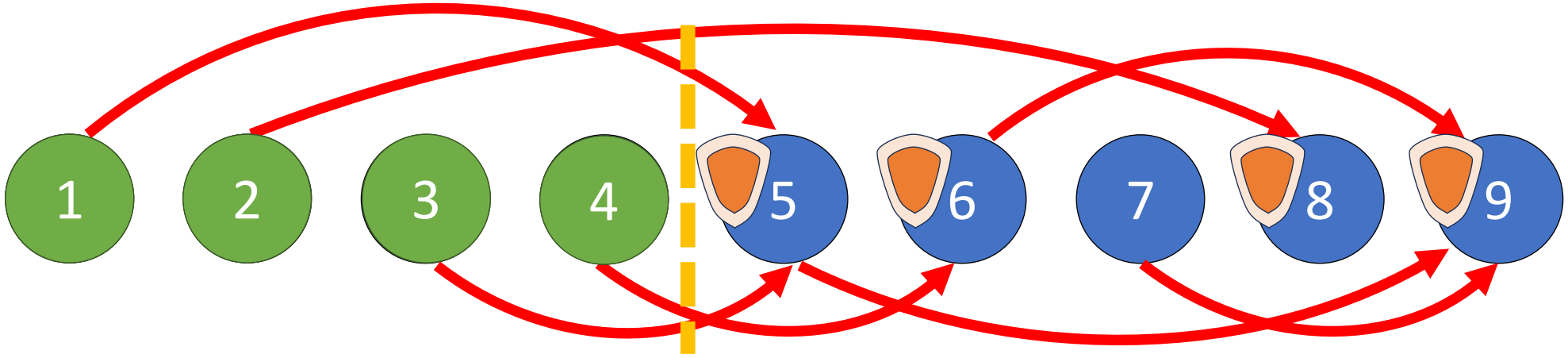
Parallel Convex LWS

- For each state, only the left-most sentinel matters
- For each tentative state j, we find the left-most state s_j that it can successfully relax (the red arrows)
- After we put all sentinels, the cordon is at the left-most sentinel



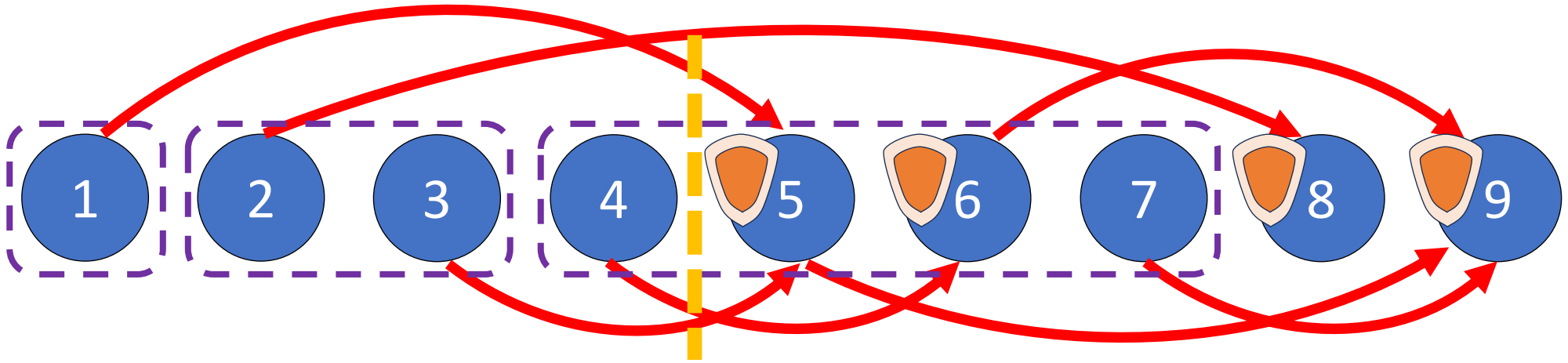
Parallel Convex LWS

- The cordon algorithm tells us which states are ready
- However, how to **efficiently** find them?
  - In each round we may need to find $O(n)$ red arrows
  - This gives us $\Omega(kn)$ work, not perfect



Parallel Convex LWS

- Our solution: prefix-doubling
  - Doubling the checked states by 1, 2, 4, 8
  - Break if we exceed the left-most sentinel
- The number of states we checked is no more than twice of the number of ready states



Parallel Convex LWS

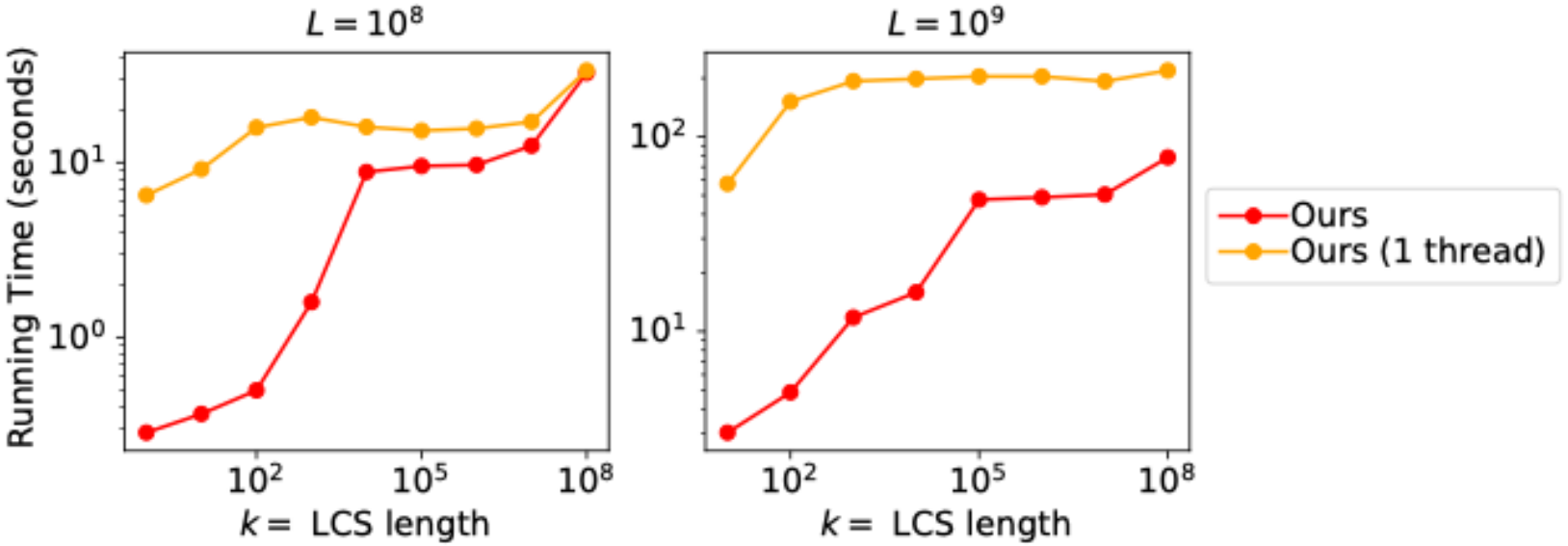
- How to maintain the best decisions in parallel?
  - A queue storing $[l, r, i]$ tuples
  - By decision monotonicity, after each round the queue is totally refreshed by the new finalized states
- Many details here
  - How to find s_j (the left-most state that j can successfully relax)
    - Binary search $D[j] + w(j, s_j) < D\left[\text{best}[s_j]\right] + w(\text{best}[s_j], s_j)$
  - How to read/write the best decisions in the queue
  - How to rebuild the best decision queue in parallel

Summary for Parallel DP with Decision Monotonicity

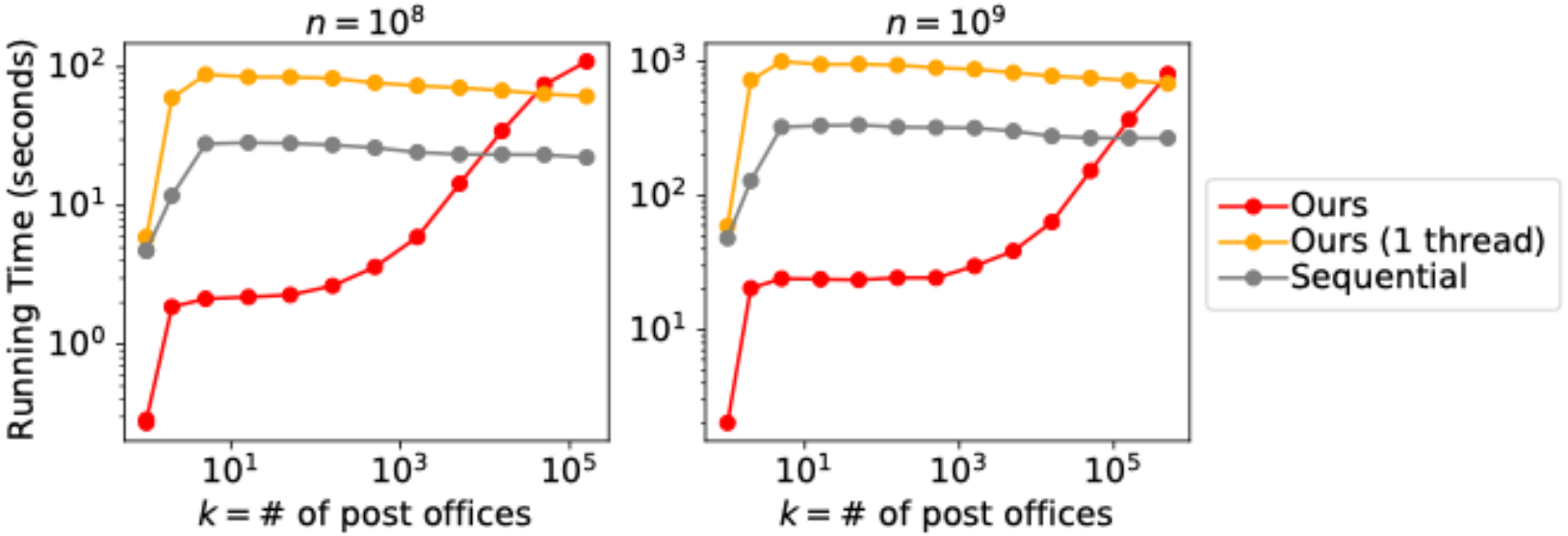
- Parallel Convex LWS: $O(n\log n)$ work and $O(k\log^2 n)$ span, where k is the longest path following the best decisions
  - Perfect parallelization: if we follow the DP DAG, the span is at least $\tilde{O}(k)$
  - In the mailbox problem, k is the number of mailboxes in the optimal solution
- Many other parallel algorithms discussed in the paper, e.g.,
  - Parallel concave LWS
  - Optimal alphabetic tree (OAT)
    - $O(n\log^2 n)$ work and $\text{polylog}(n)$ span for positive integer weights
    - Partially answers the open problem in [Larmore et al, SPAA 1993]
  - GAP edit distance problem
    - DP on a 2D grid with decision monotonicity
  - LWS on trees

Our Algorithms are Pratical!

Parallel LCS



Parallel
Convex LWS



Summary

- Motivation
  - Bridge the gap between sequential and parallel DP algorithms
- Contributions
  - The Cordon Algorithm framework: detect parallelism for DP optimizations
  - Techniques for work-efficiency
    - Prefix doubling
    - Parallel data structures
  - New parallel algorithms
    - Sparse LCS, convex/concave LWS, OAT, GAP edit distance, etc.
- Full version paper: https://arxiv.org/abs/2404.16314
- Code: https://github.com/ucrparlay/Parallel-Work-Efficient-Dynamic-Programming