Final Report

Group members:
Jiachun Li <jiachunl>
Jianan Ji <jji2>

1 SUMMARY

We developed ParallelDP, a high-level parallel dynamic programming library for multi-core CPUs. Users define DP problems through a simple DSL, while the backend automatically applies parallelization techniques such as the Cordon Algorithm. We implemented two sequential baselines (a naïve unoptimized DP and an optimized work-efficient DP) and three parallel versions using OpenMP, OpenCilk, and Parlay. We evaluated all implementations on the Pittsburgh Supercomputing Center (PSC) machines.

2 BACKGROUND

Dynamic programming (DP) is a fundamental algorithmic technique that solves problems by breaking them down into overlapping subproblems. Despite its wide applicability, efficiently parallelizing DP algorithms remains challenging due to dependencies between states. Recent research has shown that many DP problems can be parallelized efficiently using techniques like the Cordon Algorithm, which identifies sets of states that can be computed concurrently.

Many common DP problems share underlying structure:

- **States** represented as one or more dimensions (e.g., indices, positions)
- **Transitions** defining how states depend on other states
- **Optimization functions** (e.g., min, max) to select the best result
- **Base cases** as initial conditions

Traditionally, developers must manually implement both the problem definition and parallelization strategy, requiring expertise in parallel systems and specific DP optimizations. Our library will bridge this gap by providing a unified interface for defining DP problems and automatically applying appropriate parallelization techniques.

2.1 DP Problem

There are mainly three kinds of DP problems we have implemented: Longest Increasing Subsequence (LIS), Longest Common Subsequence (LCS) and convex Generalized Least Weight Subsequence (GLWS). To evaluate the convex GLWS solver, we used a variant of the Post Office problem. In this version, there is no limit on the number of offices that can be established. Each time a new office is built, it incurs a fixed setup cost in addition to the transportation costs. The overall objective is to minimize the total cost, which consists of the sum of the distances from each resident to their nearest office plus the cumulative setup cost for all established offices. This problem naturally fits into the convex GLWS framework due to its optimal substructure and the convexity of the associated cost functions.

2.2 Segment Tree

A segment tree is a fully-balanced binary search structure that overlays an ordinary array in order to answer interval aggregates quickly. The construction procedure recursively divides the array into halves: the root node represents the whole range $[0, n-1]$; each internal node represents the left or right half of its parent; and the leaves correspond one-to-one with the elements of the base array. At every node we store the value of an associative operator applied to the elements covered by that node. Because each element participates in $O(logn)$ nodes, both memory consumption and build time remain linear in the problem size.

During the build phase, the left and right halves of every node can be processed independently; with a work-stealing runtime such as OpenCilk the aggregate work remains $\Theta(n)$ while the critical-path depth is only $\Theta(logn)$. To avoid wasting time on very small sub-trees we introduce a granularity threshold: only segments larger than this cutoff are spawned as separate tasks, whereas smaller segments are handled sequentially in their parent's stack frame. The same idea appears inside the prefix-minimum routine, where a leaf either performs a short linear scan or switches to binary search depending on the local list length, again trading constant factors for asymptotic efficiency.

In the context of this project the segment tree is not merely a textbook data structure but the backbone of a million-scale LCS solver. Its logarithmic query cost, its ability to accommodate both parallel construction and fine-grained dynamic updates make it the most appropriate choice for sustaining high throughput on modern multicore processors.

2.3 Cordon Algorithm

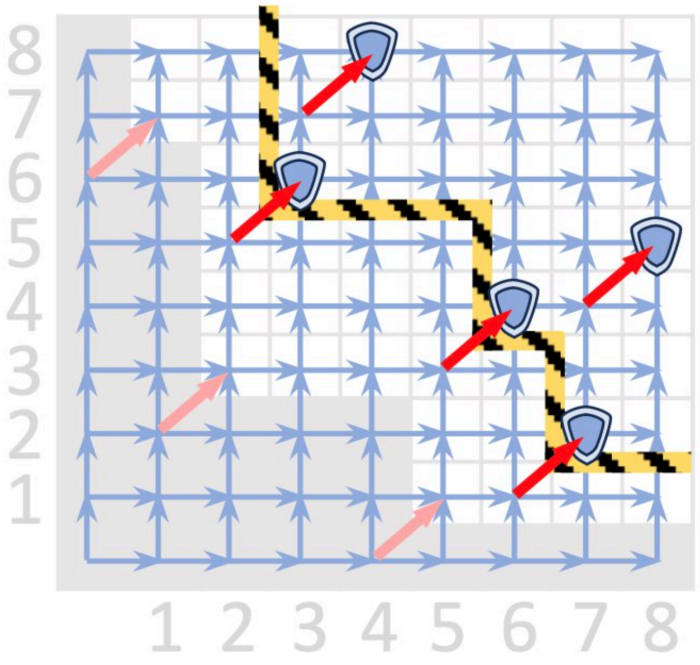
The Cordon Algorithm offers a unified framework to partition computation of dynamic programming problems into independent units that can be processed in parallel.

The algorithm proceeds in the following steps:

1. Initialization: All states are marked as tentative, and boundary condition states are immediately finalized with known values.
2. Placing Sentinels: For each tentative state, if another state can provide a better update, a sentinel is placed on the target state to indicate that it is not yet ready.
3. Identifying the Frontier: All states not blocked by any sentinels form the frontier, which can be safely processed in parallel.
4. Parallel Updates: The frontier states are finalized in parallel, computing their definitive dynamic programming values.

5. Iteration: If any tentative states remain, the process repeats from step 2 until all states are finalized.

The algorithm builds a cordon, a dynamic boundary that separates finalized or ready-to-compute states from states that still depend on unresolved computations. Each round advances the computation frontier while ensuring no dependency violations occur.



2.4 Prefix Doubling

As for LIS and LCS, each round of frontier is just finding prefix-min elements and is straightforward with the segment tree. Taking LIS as an example, a prefix-minimal element is one that has no smaller number preceding it in the sequence. Since no earlier element can extend it, it must initiate a new increasing subsequence by itself.

Prefix doubling is used in GLWS to efficiently find the frontier in each round. GLWS requires verifying whether the best decision dependencies of tentative states have already been finalized. To avoid the high cost of exhaustively checking all states, prefix doubling gradually expands the range of exploration, doubling the batch size at each step. This ensures that even if some extra work is done, the overall work remains proportional to the size of the frontier, thus preserving work-efficiency while enabling parallel frontier discovery.

For example, suppose we have tentative states $D[1]$ to $D[8]$ with only $D[0]$ finalized at the beginning. Using prefix doubling, the algorithm first checks a batch containing $D[1]$ and $D[2]$. If both states are ready, it then expands to check a batch containing $D[3]$, $D[4]$, $D[5]$, and $D[6]$. During this second expansion, if $D[5]$ is found to be blocked due to unresolved dependencies, the expansion halts immediately. As a result, $D[1]$, $D[2]$, $D[3]$, and $D[4]$ are finalized together in parallel, while $D[5]$, $D[6]$, $D[7]$, and $D[8]$ are deferred to future rounds. This controlled expansion minimizes unnecessary exploration and ensures that the work in each round scales with the actual size of the frontier.

In contrast, if the algorithm were to use exhaustive scanning without prefix doubling, it would need to examine all tentative states from $D[1]$ to $D[8]$ in every round. Even if $D[5]$ is blocked early, states like $D[6]$, $D[7]$, and $D[8]$ would still be unnecessarily checked. While this might seem manageable in a sequential setting where scanning could stop early upon encountering a blocked state, parallel execution requires dispatching batches of work ahead of time and does not allow individual threads to be interrupted once started. As a result, a naive exhaustive approach would lead to significant wasted computation in parallel environments, undermining the efficiency of the algorithm. Prefix doubling overcomes this limitation by cautiously expanding the batch size, stopping exploration as soon as a conflict is detected, and ensuring that all scheduled parallel work contributes meaningfully to progress.

2.5 Compressed Array

In dynamic programming problems such as LIS and LCS, the dependency structure between states is inherently localized and simple. In LIS, each state $D[i]$ independently finds its best decision by scanning all earlier indices $j < i$ where $A[j] < A[i]$, and selecting the one that yields the maximum subsequence length $D[j]$. This process is performed individually for each state without requiring coordination across different states. Once a state $D[i]$ determines its best predecessor, this decision remains fixed and will not change based on future computations.

In comparison, the dependency structure in GLWS, where each state $D[i]$ minimizes over earlier states through the expression $D[j] + w(j,i)$, exhibits significantly more global behavior. When the weight function $w(j,i)$ satisfies convexity or concavity properties, the optimal best decision tends to remain stable over long contiguous intervals of i. In such cases, a single predecessor j^* can be the optimal choice for many consecutive i-values until the accumulated cost forces a switch to a new predecessor $j\prime$.

For example, suppose we have tentative states $D[1]$ to $D[8]$ with a convex weight function $w(j,i) = (i-j)^2$ and varying initial costs $D[j]$. It is possible that for $i = 3$ to $i = 7$, the best predecessor remains $j = 2$, and only at $i = 8$ does the optimal decision switch to $j = 3$. In such settings, managing best decisions individually for each state would be prohibitively costly, especially in parallel environments where redundant work amplifies inefficiencies. Therefore, GLWS employs compressed array structures to group consecutive states sharing the same best decision into intervals. Each entry in the compressed array is represented in the form $(\text{range start}, \text{range end}, \text{best decision})$, indicating that all states within the specified range share the same optimal predecessor. This enables efficient updates, keeping the overall work proportional to the number of decision changes rather than the total number of states.

3 APPROACH

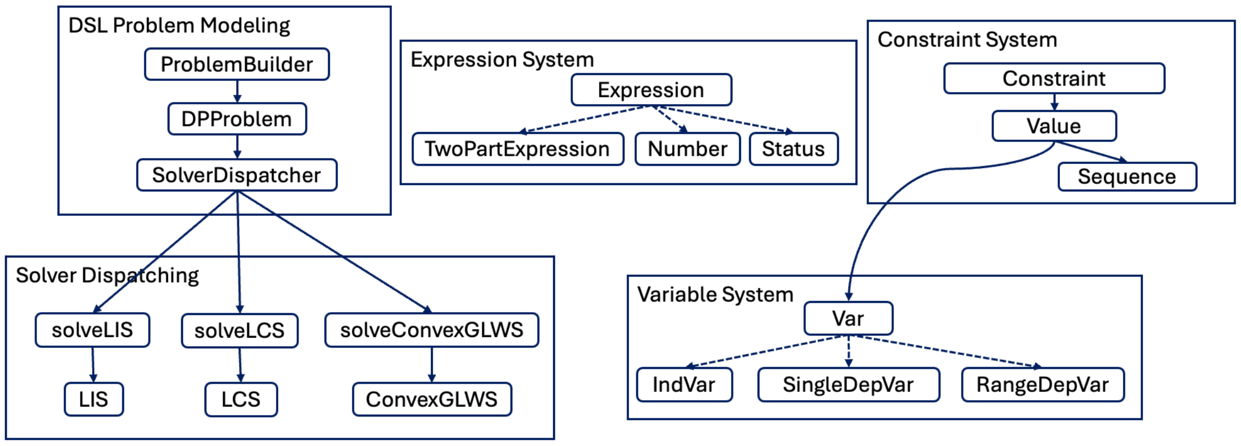
3.1 Technology

Language: C++

Library: OpenMP, OpenCilk, Parlay

Targeted Machine: We conducted our experiments on machines at the Pittsburgh Supercomputing Center (PSC), equipped with AMD EPYC 7742 processors. Each machine features 128 CPU cores without hyperthreading, distributed across 2 processors with 64 cores per processor. The system has 251 GB of main memory, along with an additional 699 GB of swap space. Architecturally, the machine consists of 2 NUMA nodes, and each processor provides 16 MB of L3 cache.

3.2 DSL



As one of the main goals of this project, we propose and implement a domain-specific language (DSL) tailored for the declarative specification and efficient parallel solution of dynamic programming (DP) problems. The DSL enables users to define problems such as Longest Increasing Subsequence (LIS), Longest Common Subsequence (LCS), and Convex Generalized Least Weight Subsequence (Convex GLWS) through a high-level, concise API, abstracting away the low-level procedural details and emphasizing semantic clarity.

3.2.1 Design Philosophy

The core design of the DSL is motivated by three key principles: declarativeness, problem-type recognition, and automatic solver dispatching. By elevating the level of abstraction, users are relieved from the burden of manually encoding recurrence relations, while the backend ensures that highly optimized parallel solvers are automatically selected and invoked based on the problem structure.

The DSL follows a two-stage architecture:

1. Frontend: Users specify the problem declaratively using a set of combinators to define state variables, sequences, constraints, objectives, and recurrence relationships.

2. Backend: The framework performs static analysis on the user input to infer the underlying DP problem type and dispatches the problem to a specialized solver with appropriate parallel optimizations.

3.2.2 Core Abstractions

Variables and State Space

The DSL introduces a flexible abstraction for representing DP state variables:

- Independent Variables (`IndVar`): These represent standard iteration indices with explicit ranges.
- Single-Dependent Variables (`SingleDepVar`): Variables defined as an offset from an independent variable, supporting common DP formulations involving neighboring states.
- Range-Dependent Variables (`RangeDepVar`): Variables dependent on dynamic intervals, enabling the expression of more complex DP formulations such as those seen in convex optimization.

This typed hierarchy facilitates sophisticated pattern matching during problem recognition while ensuring type safety at the DSL level.

Sequences and Values

Input data is encapsulated using the `Sequence` abstraction, which allows indexed access via state variables. Moreover, scalars or auxiliary constants are managed through a flexible `Value` system based on variant types, supporting heterogeneous problem specifications without runtime type ambiguity.

Constraints and Objectives

Constraints between variables or sequence elements are expressed via a unified `Constraint` object, supporting relational operations (`<`, `>`, `==`, `!=`). These constraints are integral to the recognition engine, helping the system differentiate between problem types such as LIS (strictly increasing subsequences) and LCS (matching subsequences).

Users can also specify the optimization objective, either `MAXIMIZE` or `MINIMIZE`, guiding the selection of the backend solver and influencing cost function design in complex problems.

Expressions and Recurrence Relations

The DSL provides a lightweight intermediate representation for recurrence expressions, including:

- Status: Captures the DP state based on current variables.
- Arithmetic Operations: Supports constant offsets on states.
- Composite Expressions: Such as `max` and `min`, enabling higher-level recurrence relations without manual indexing.

Users may optionally define custom recurrence functions, but for common patterns, the system can deduce standard recurrences automatically.

3.2.3 Problem Recognition and Solver Dispatch

A key contribution of our work is the automatic problem recognition engine. Based on the structure of variables, sequences, constraints, and objective, the framework can classify problems into one of the known types. If no known pattern is detected, the problem is labeled as `UNKNOWN`, providing clear feedback to the user. Upon successful recognition, the `SolverDispatcher` transparently routes the problem to the corresponding specialized backend solvers. This design ensures that users benefit from highly optimized parallel performance without any manual intervention.

3.2.4 DSL Usability and Extensibility

To facilitate rapid prototyping and enforce best practices, we also implement a Builder Pattern interface. The `ProblemBuilder` class allows fluent chaining of configuration methods (`withVar`, `withCondition`, `withObjective`, etc.), reducing boilerplate and improving code readability.

Moreover, the modular structure of variables, constraints, and dispatching logic ensures that extending the DSL to new classes of DP problems requires minimal additional effort. Future work can enrich the DSL with support for problems involving cyclic dependencies, multi-objective optimization, or stochastic states.

3.2.5 Illustrative Examples

To demonstrate the expressiveness and automation capabilities of our DSL, we provide an illustrative examples, the Longest Increasing Subsequence (LIS) problem.

```
auto Seq = new Sequence<int>({3, 1, 4, 2, 7, 5, 8, 6, 9, 10});
auto I = new IndVar(0, 10);
auto J = new RangeDepVar(0, I-1);

auto problem1 = ProblemBuilder<int>::create()
    .withVar(I)
    .withVar(J)
    .withSequence(Seq)
    .withCondition(dp_dsl::max(Status(J) + 1, Status(I)))
    .build();
ProblemType type2 = problem2.getProblemType();
```

In this example, the user defines a single independent variable `I` representing the current index and a dependent variable `J` ranging over all previous indices (`0` to `I-1`). The recurrence relation states that for each position `I`, the optimal subsequence length is the maximum of maintaining the current value (`Status(I)`) or extending a subsequence ending at a valid previous index (`Status(J) + 1`). This declarative specification captures the essence of the LIS problem without explicit loops or recursion. Upon calling `problem1.getProblemType()`, the system correctly recognizes the problem as `LIS`, showcasing the DSL's ability to infer classical DP patterns solely from structural information.

3.3 Problem Mapping

We mapped our dynamic programming problems to the PSC machines by distributing frontier states and compressed array updates across multiple cores. During each round of the Cordon Algorithm, frontier states are partitioned among threads, allowing each thread to compute updates independently. For LIS and LCS, we used a segment tree structure to efficiently find prefix-minimal elements, with threads working together to find the minimum values in parallel. For GLWS, compressed array updates are parallelized by assigning non-overlapping intervals to different threads, enabling efficient modifications without conflicts.

To better adapt to parallel execution, we modified the original serial algorithms. For LIS and LCS, we introduced segment-tree based parallel reductions to accelerate frontier selection. For GLWS, we introduced prefix doubling for controlled parallel exploration and redesigned best-decision tracking into compressed array updates. These changes ensure efficient parallelism while maintaining work-efficiency.

4 Optimization Process

4.1 LIS Optimization

In optimizing the parallel dynamic programming (DP) solver for the Longest Increasing Subsequence (LIS) problem, we underwent several iterations of refinement to improve performance and eliminate bottlenecks. Our initial implementation was functionally correct but exhibited suboptimal speed. Throughout the optimization process, we carefully analyzed synchronization costs, memory access patterns, and thread contention, leading to a series of targeted improvements.

Initial Implementation: Critical Section Bottleneck

Our first parallel version employed OpenMP with dynamic scheduling and used a `#pragma omp critical` block to guard updates to the `dp` array:

```
#pragma omp parallel for schedule(dynamic) firstprivate(dp_cordon, data_cordon)
for (int i = cordonIdx + 1; i < n; i++) {
    if (!finalized[i] && cmp(data_cordon, data[i])) {
        #pragma omp critical
        {
            dp[i] = std::max(dp[i], dp[cordonIdx] + 1);
        }
    }
}
```

However, profiling results indicated extremely poor scalability, with a parallel execution time of **10469 ms** for a granularity of 1000.

Upon inspection, we realized that the critical section serialized all updates, creating significant thread contention—even though logically, different threads were updating different entries in the `dp` array independently. The critical protection was unnecessarily conservative.

Removing Over-synchronization

Recognizing that concurrent writes to different memory locations are inherently safe in OpenMP, we removed the `#pragma omp critical` section altogether. Additionally, we noticed that since each thread only needs to read `dp[cordonIdx]` once per iteration, it could use a local cached copy (`dp_cordon`) rather than reloading from memory repeatedly.

The improved code was:

```
#pragma omp parallel for schedule(dynamic) firstprivate(dp_cordon, data_cordon)
for (int i = cordonIdx + 1; i < n; i++) {
    if (!finalized[i] && cmp(data_cordon, data[i])) {
        dp[i] = std::max(dp[i], dp_cordon + 1);
    }
}
```

This optimization led to a significant speedup, reducing the parallel runtime to **5455 ms** under the same conditions—a nearly **2×** performance improvement.

Addressing False Sharing

Further analysis, however, revealed that despite the removal of critical sections, scalability was still limited.

Although different threads were logically updating independent entries of the dp array, physically adjacent memory addresses might map to the same CPU cache line (typically 64 bytes). When multiple threads modify different variables within the same cache line, cache coherency protocols force the entire line to be invalidated and reloaded across cores, even if the actual memory cells being updated are disjoint. This unintended invalidation and synchronization across threads lead to substantial hidden overhead, effectively serializing what should be parallel updates.

To mitigate this, we firstly checked the cache line size of the machine using:

```
cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
```

Then we adjusted the scheduling granularity by aligning the chunk size with the size of a cache line:

```
const int CACHE_LINE = 64 / sizeof(int);
#pragma omp parallel for schedule(dynamic, CACHE_LINE) firstprivate(dp_cordon, data_cordo
n)
for (int i = cordonIdx + 1; i < n; i++) {
    if (!finalized[i] && cmp(data_cordon, data[i])) {
        dp[i] = std::max(dp[i], dp_cordon + 1);
    }
}
```

By ensuring that each thread processes a contiguous block of elements fitting exactly into one or more cache lines, we reduced inter-thread cache conflicts and minimized false sharing. This optimization allows each thread to work more independently, achieving much better cache locality and avoiding frequent cache line ping-ponging between cores. This change yields an additional **~14%** performance improvement compared to the previous version.

4.2 LCS Optimization

4.2.1 Comparison of Different Library

Our segment tree is deep, irregular, and dynamically pruned: Each recursive call explores *only the branches whose current minimum ≤ pre*, so the shape of the DAG depends on the data, not on the tree height. What's more, leaf work is tiny (a few comparisons and, occasionally, a binary-search). Also, The write-back (`tree= min(...)`) must execute bottom-up, but no global barrier is needed between siblings. These characteristics have direct consequences for the choice of run-time: A loop-parallel construct cannot express your irregular traversal without materialising a work list, which would destroy locality and introduce an O(n) pre-processing cost. Therefore OpenMP must use tasks—and these cost ~0.4–2 µs each on today's Clang/GCC run-times, drowning the < 0.1 µs leaf computations. But adding a depth cut-off and reducing taskwaits to a single taskgroup, the heap-allocated task descriptors, shared queues and atomics still impose roughly 3× the spawn overhead of Cilk. The run-time also keeps all team threads alive, so cache contention grows with core count.

OpenCilk matches the algorithm's shape. Continuation frames live on the parent's stack and are < 100 B, so spawning adds only ~30 ns. Help-first work stealing keeps the hot leftmost path in the same worker's cache; thieves start at large unexplored sub-trees, which amortises steal overhead. The adaptive worker pool allows only as many threads as there is ready to work. These properties line up perfectly with a branch-and-bound segment tree.

Therefore we tried OpenCilk verison of the LCS solver. Both versions implement the same recursive prefix-minimum segment tree used inside; they differ only in the parallel-task substrate:

Variant	Parallel API	Spawn	Wait
OpenMP -task	`#pragma omp task`	heap-allocated task-descriptor (`kmp_taskdata_t`)	implicit `taskwait` at each internal node
OpenCilk	`cilk_spawn`	stack-resident continuation frame	single `cilk_sync` at subtree root

We used `perf stat` to collect the counters shown in Table below.

Metric	OpenMP	Cilk	Δ
Kernel wall-time	988 ms	**718 ms**	-27 %
Total elapsed	4.63 s	**4.17 s**	-10 %

Cycles	4.99×10^{11}	**1.01×10^{11}**	-80 %
Instructions	3.35×10^{11}	**1.01×10^{11}**	-70 %
IPC	0.67	**1.00**	+50 %
L1-load misses	2.78×10^{8} (15 %)	**1.42×10^{8} (2 %)**	-49 %
task-clock	172 288 ms	**37 488 ms**	-78 %
Avg. workers	37.2	**9.0**	–

Micro-architectural Interpretation

Instruction pressure. Every `#pragma omp task` allocates and initialises a 300 + B descriptor that must be pushed onto a shared queue. Dynamic-instruction count therefore triples and branch instructions quadruple relative to the Cilk run time, which records only a few words for its continuation frame.

Cache locality. Cilk's *help-first* work-stealing keeps the parent on-stack and executes the child immediately; only when a thief succeeds is the saved continuation migrated. Thus each worker's deque is private and almost never invalidated. OpenMP, in contrast, manages a shared task list protected by atomics, inflating the L1 miss-rate from 2 % to 15%.

Synchronisation latency. The OpenMP version enters a full thread-team at every call, and each internal node executes an implicit barrier (`taskwait`). This balloons *task-clock* to 172 s, with 28 idle threads spinning per barrier. Cilk wakes new workers on demand and uses a single `cilk_sync`, explaining the much lower 37 s CPU time.

Pipeline utilisation. The combined effect of fewer misses and less synchronisation allows Cilk to sustain IPC ≈ 1.0; OpenMP stalls at 0.67 IPC because the core frequently waits for memory and locks.

Replacing OpenMP-tasks with Cilk in the LCS prefix-minimum kernel yields a 27 % kernel-time and 10 % end-to-end speed-up. Perf counters reveal that Cilk executes 70 % fewer instructions, suffers 7.5× fewer cache misses, and consumes only one-fifth of the CPU time. These gains stem from Cilk's lighter continuation frames, lock-free private deques and demand-driven worker activation—properties already predicted by work-stealing theory and confirmed by multiple independent studies.

4.2.2 Granularity of works

Grain-size control turns a theoretically unbounded recursion into a cost-aware parallel execution tree whose leaves are at least an order of magnitude more expensive than the spawn operation itself. The `granularity` acts as an application-level cut-off expressed here over the segment-range length (r–l).

Because the same cut-off is evaluated in both the Cilk and OpenMP variants, the 27 % kernel-time advantage we measured can be ascribed to the run-time's task overhead. Nevertheless, exposing `granularity` as a tunable knob makes the code portable: if ported to GPUs or many-light-core NUMA machines, the threshold can be scaled (e.g., proportional to L1 latency) without rewriting the recursion.

4.2.3 Memory allocation and reading

The raw `perf stat` profile of our original Cilk implementation exposes three problems. First, the algorithm stores its arrow lists in a `vector<vector<T>>`. Because only the outer vector is contiguous, every inner-vector access dereferences a heap pointer; the resulting pointer-chasing drives many cache references and L1 misses. Prior work notes that nested vectors destroy spatial locality and aggravate allocator contention on modern caches. Second, Cilk's work-stealing runtime receives a torrent of very small tasks. Intel's own guide shows that when the grain size falls below roughly $n/(8P)$, the fixed cost of `spawn` comes to dominate execution time. In our trace, spawn overhead inflates the task-clock to 44365 ms while wall time is only 4.3 s, meaning most cycles are spent in scheduler bookkeeping rather than user work. Consequently the program sustains barely 10 cores, and the pipeline retires only 0.83 instructions per cycle. Finally, the nested vectors trigger thousands of reallocations that must acquire the global `ptmalloc` lock. All three issues are aggravated on NUMA hardware because random heap pages migrate to remote nodes, adding yet more latency to every miss.

Aware of these bottlenecks, we studied ParlayLib's data-structure layer. Its central container, `parlay::sequence`, is still a contiguous dynamic array but it embeds two key optimisations. For trivial element types that fit in fifteen bytes, the entire array is stored inline in the sequence header, eliminating any heap traffic. When the array must grow, Parlay parallelises the move with a single `memcpy` per worker and uses a thread-local pool allocator, avoiding the global malloc lock and keeping cache lines hot. Together, the contiguous layout, SSO, and pool allocator promise exactly the cache locality and allocation scalability the DSL backend needs.

We therefore introduce additional parlay::sequence based backend, replacing the nested STL vectors. The resulting back-end was evaluated on the same 10 M × 10 M LCS instance. Wall-clock time dropped more than 4X, and every low-level metric moved in the expected direction: the task-clock fell by a factor of 2.3 because workers now spend far less time idle, the average number of active cores doubled to 20.9, IPC more than doubled to 1.89, and cache references as well as L1 misses shrank by roughly 60 %. The table below summarises the comparison and underscores that the 4.8× speed-up comes almost entirely from solving the memory-system problems unearthed in the first profile.

Metric	Parlay	Cilk	Speed-up	Comment
Wall-time	0.909 s	4.323 s	4.8×	headline result
task-clock	18 949 ms	44 365 ms	2.3×	fewer idle cycles
Avg. active CPUs	20.9	10.3	~2×	better load balance
IPC	1.89	0.83	2.3×	fewer pipeline stalls
cache references	125 M	308 M	2.46×	contiguous access
L1 misses	60 M	161 M	2.7×	locality + SSO
branch-mispredict	0.54 %	0.05 %	–	speculation reclaimed
cycles	54 G	131 G	2.4×	less time stalled

5 RESULT

5.1 Performance Measurement Criteria

We measured performance primarily using wall-clock time. For each dynamic programming problem variant, we recorded the total execution time from the start of the DP computation to the completion of all finalizations. In addition to absolute runtime, we also computed speedup by comparing the sequential baseline with the corresponding parallel implementations. Speedup was defined as the ratio of the sequential execution time to the parallel execution time. All measurements were conducted on PSC machines under controlled conditions to ensure consistency.

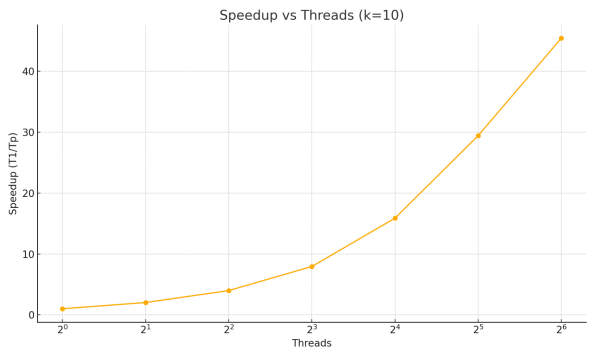
5.2 Experimental Setup

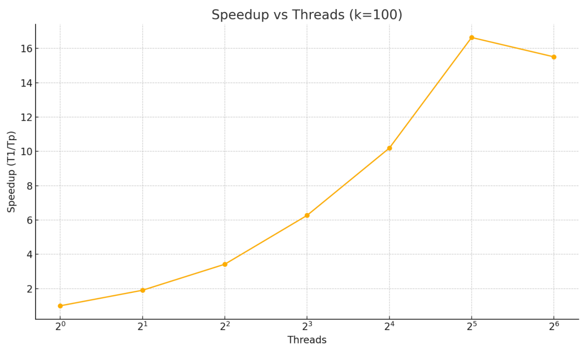
Our experiments were conducted on PSC machines equipped with AMD EPYC 7742 processors.

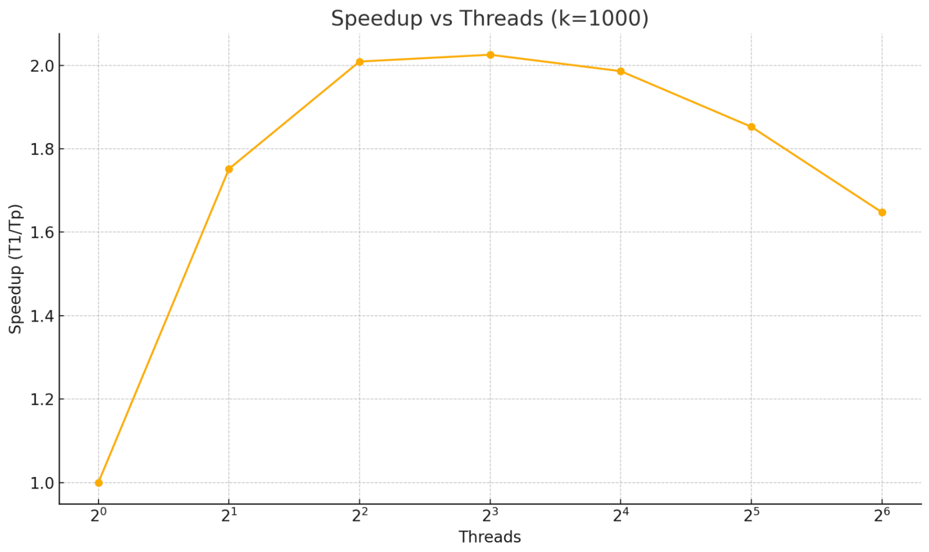
For LIS and LCS, we generated input sequences of $[10^6, 10^9]$. Within each sequence, the number of valid parts ranged $[1, 10^6]$, allowing us to control sparsity and variation across different experiments.

5.3 Speedup

We tested LCS problem on different answer lengths (k) and different CPU kernel numbers with input sequence at 10^7.







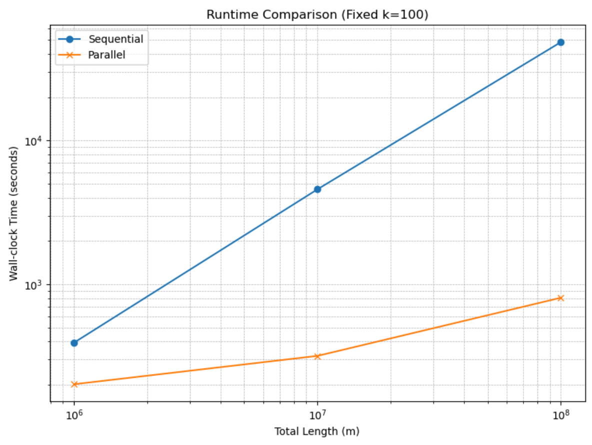
When k=10, speed-up is nearly linear: $\times 8$ on 8 threads, $\times 29$ on 32, and $\times 46$ on 64 (≈ 72 % of ideal). Because *k* is small, every outer-loop iteration inspects only a handful of “arrows”, so the algorithm becomes compute-light but branchy. Parallelism is abundant (≈ $n log n$ independent subtrees) and each task’s working-set easily fits into L1, hiding spawn latency all the way to 64 cores until memory-controller contention finally appears.

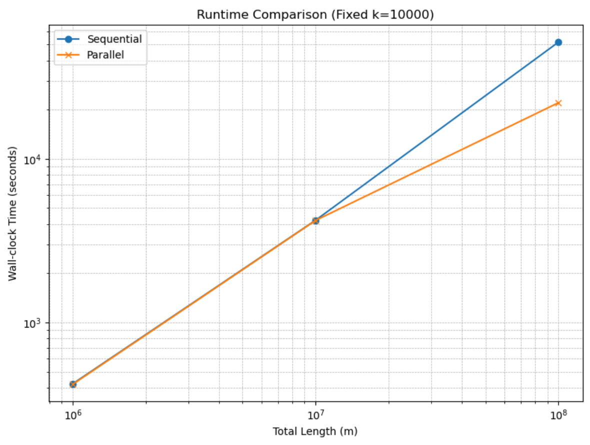
When k=100, scaling is good up to 16 threads ($\times 10$) and flattens thereafter (peak $\times 17$ at 32T; slight drop at 64T). Longer arrow lists inflate per-leaf work. The work–span ratio is still high, but critical sections (prefix path updates) share cache lines more often, making steals costlier.

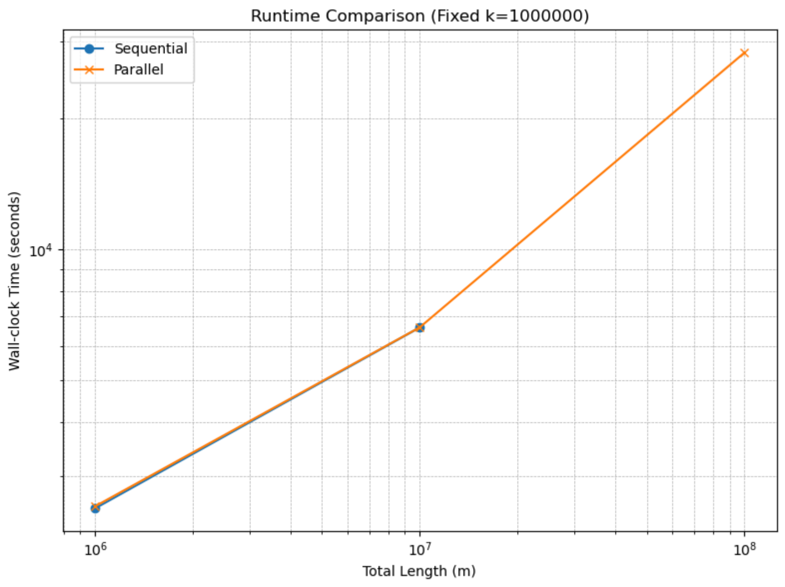
When k=1000, speed-up saturates at $\times 2$, because sequential work outgrows the spawn cutoff. Even though 64 workers are available, only ≈ 2 parallel tasks are ready on average. Furthermore, the fixed granularity = 5000 is not suitable for most sub-trees never spawn at all.

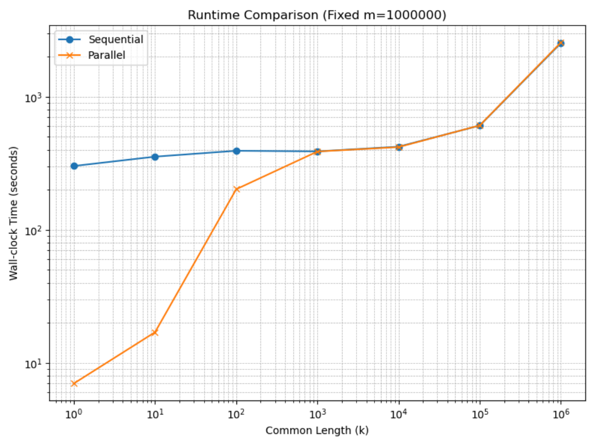
5.4 Problem Size Analysis (on LCS)

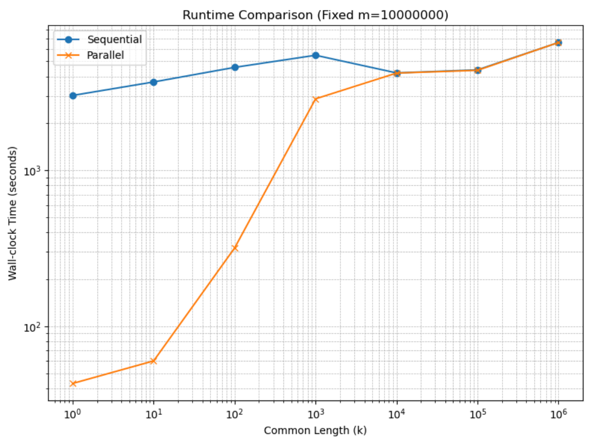
We tested LCS problem on different answer lengths (k) and different input sequence (m) with same CPU kernel numbers at 64.

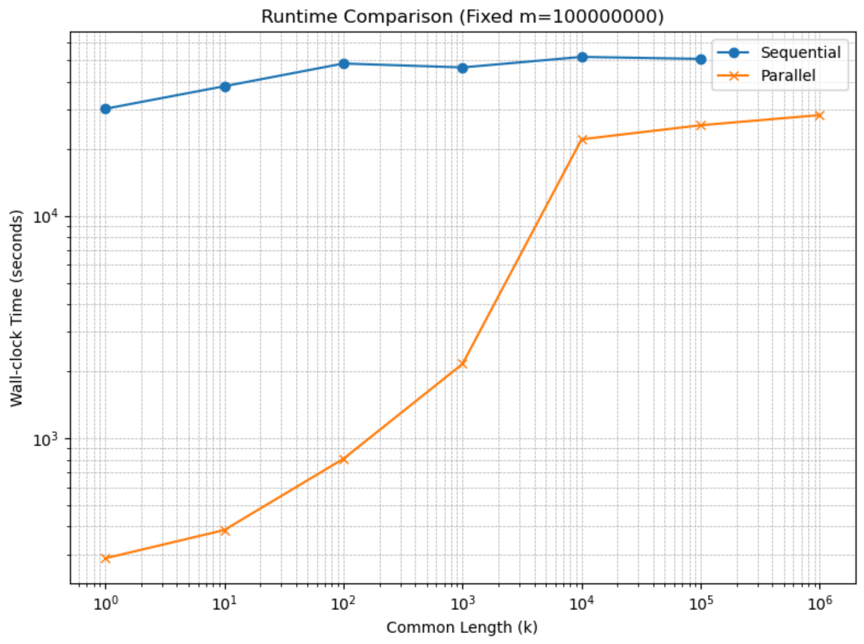

Runtime Comparison (Fixed k=100)
Sequential
Parallel
Wall-clock Time (seconds)
Total Length (m)
10^3
10^4
10^6
10^7
10^8


Runtime Comparison (Fixed k=10000)
Sequential
Parallel
Wall-clock Time (seconds)
Total Length (m)
10^3
10^4
10^6
10^7
10^8


Runtime Comparison (Fixed k=1000000)
Sequential
Parallel
Wall-clock Time (seconds)
Total Length (m)
10^4
10^6
10^7
10^8


Runtime Comparison (Fixed m=1000000)
Sequential
Parallel
Wall-clock Time (seconds)
Common Length (k)
10^1
10^2
10^3
10^0
10^1
10^2
10^3
10^4
10^5
10^6


Runtime Comparison (Fixed m=10000000)
Sequential
Parallel
Wall-clock Time (seconds)
Common Length (k)
10^2
10^3
10^0
10^1
10^2
10^3
10^4
10^5
10^6



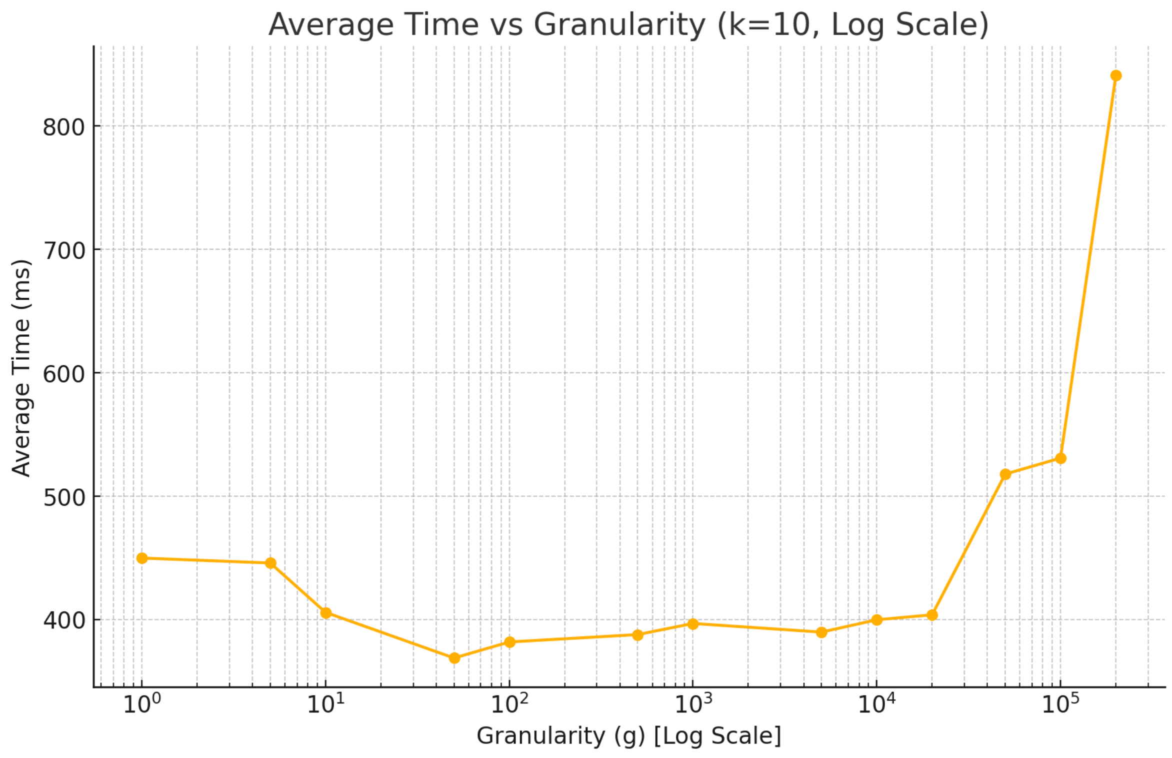
Across all three plots where the total input length m is fixed and the common subsequence length k varies, we observe clear performance trends. For small k values, the parallel implementations achieve substantial speedups compared to the sequential baselines, sometimes reducing wall-clock time by an order of magnitude. This is expected, as sparse matching leads to lower computational dependencies, allowing parallel threads to finalize frontier states with minimal synchronization overhead.

As k increases, the amount of active data and the number of effective dependencies grow significantly. Consequently, the parallel curves gradually approach the sequential ones, and the speedup diminishes. In particular, when k becomes large (e.g., $k = 10^6$), the amount of work required for each frontier and the need for coordination between threads both increase, limiting the benefit of parallelism.

Additionally, as m grows larger, the overall runtime for both sequential and parallel versions increases accordingly, but the trend remains consistent: parallelism provides the greatest advantages when the problem is sparse and frontier sizes are small relative to m. These results validate the effectiveness of the Cordon Algorithm framework in handling sparse dynamic programming problems in parallel while maintaining efficiency even as problem density changes.

5.5 Granularity Comparison

We tested LCS problem on different granularities with same input length at 100000000, k at 10 and CPU kernel numbers at 64.



At small granularity each leaf represents only a few dozen arithmetic/logical instructions, but the run-time still pays ~30–50 ns to create a continuation frame and push it onto the deque. With tens of millions of such micro-tasks, the aggregate cost inflates the critical path and starves the memory hierarchy. Beyond most suitable granularity the traversal becomes mostly sequential along the paths. Workers sleep rather than steal, and wall-time rises gradually. Past $g \approx 50000$ the kernel degenerates to serial execution and runtime explodes to >800 ms.

5.6 Target Machine

Our choice of using a multi-core CPU platform was sound for this project. The Cordon Algorithm and its associated techniques, such as prefix doubling and compressed arrays, are designed to minimize work overhead and exploit fine-grained parallelism across states with complex dynamic dependencies. These characteristics align well with the CPU multicore model, where each thread can independently access shared memory and perform dynamic synchronization efficiently. Given that our algorithms involve frequent irregular memory accesses, dynamic task creation, and non-uniform computation across frontier states, CPUs are better suited than GPUs, which prefer uniform workloads and structured memory access patterns. Furthermore, the Cordon framework is designed to achieve (nearly) work-efficient execution with minimal redundant computation, which maps naturally to CPU thread-based parallelism without requiring GPU-specific batch scheduling. Therefore, using CPUs allowed us to fully leverage the algorithmic design while achieving strong practical performance.

REFERENCES

[1] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5*. November 2015. Available at https://www.openmp.org.

[2] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. *The Implementation of the Cilk-5 Multithreaded Language*. Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98), pages 212–223.

[3] Xiangyun Ding, Yan Gu, and Yihan Sun. *Parallel and (Nearly) Work-Efficient Dynamic Programming*. arXiv preprint arXiv:2404.16314v2, 2024.

[4] Xiangyun Ding, Yan Gu, and Yihan Sun. 2024. *Source Code for Parallel and (Nearly) Work-Efficient Dynamic Programming*. Available at https://github.com/ucrparlay/Parallel-Work-Efficient-Dynamic-Programming.

[5] Yihan Sun, Guy Blelloch, and Jeremy Fineman. *The Parlay Library: Parallel Algorithms and Data Structures*. Available at https://github.com/cmuparlay/parlaylib.

[6] Intel Corporation. *Introduction to Intel Cilk*. Available at https://www.intel.com/content/dam/www/public/apac/xa/en/pdfs/ssg/Introduction_to_Intel_Cilk.pdf

LIST OF WORK

- Design DSL interface and core abstractions (Jiachun Li, Jianan Ji)
- Implement naive DP solver for LIS, LCS, and GLWS (Jiachun Li)
- Set up testing infrastructure (Jiachun Li, Jianan Ji)
- Implement Cordon Algorithm for LIS (Jiachun Li)
- Implement Cordon Algorithm for LCS (Jiachun Li, Jianan Ji)
- Implement segment tree for LIS and LCS (Jianan Ji)
- Optimize segment tree for LIS and LCS (Jiachun Li, Jianan Ji)
- Compare performance of different parallel libraries in segment tree (Jianan Ji)
- Implement Cordon Algorithm for convex GLWS (Jiachun Li)
- Add prefix-doubling and compressed array optimization for convex GLWS (Jiachun Li)
- Complete documentation (Jiachun Li, Jianan Ji)
- Verify correctness of LIS, LCS, and convex GLWS (Jiachun Li)
- Test LIS, LCS and GLWS performance (Jianan Ji)
- Complete final report and poster (Jiachun Li, Jianan Ji)

DISTRIBUTION OF TOTAL CREDIT

Jiachun Li - Jianan Ji: 50% - 50%