From Recursion to Parallelization: Plug & Play Dynamic Programming

Jiang Long

Duke Kunshan University, Kunshan, China
jiang.long@dukekunshan.edu.cn

Abstract. Dynamic Programming (DP) is fundamental to computational science education and application, traditionally taught through tabulation methods that emphasize manual loop construction. This paper introduces a modular, systematic plug-and-play framework that greatly simplifies DP algorithm design and parallelization. Our approach begins with a recursive divide-and-conquer analysis, decomposing DP into reusable components: `Refactored Recursion (RR)`, `OrderSpec`, `TileSpec`, `dp_solve`, `dp_tile_solve`, and `dag_run`. These modules encapsulate recursive structures and facilitate seamless parallelization via dynamic Directed Acyclic Graph (DAG) scheduling. We demonstrate the versatility of this framework using three classical textbook problems: Longest Common Subsequence (LCS) highlights the plug-and-play simplicity, Matrix Chain Multiplication (MCM) employs transitive reduction for dependency clarity, and the Cut-Rod problem illustrates previously obscured tiling optimizations and parallel solutions. This new modular paradigm significantly reduces DP's learning curve, shifting educational focus from code-centric methods to intuitive, reusable patterns, bridging theoretical recursion with practical implementations and greatly enhancing computational science education.

Keywords: Dynamic Programming · Plug-and-Play Framework · Algorithm Design Patterns · Computer Science Education · Parallelization.

1 Introduction and Motivation

Dynamic programming (DP) solves complex problems by decomposing them into overlapping subproblems, widely used in optimization and bioinformatics. However, traditional teaching methods, which emphasize recursion and simple tabulation, overlook practical requirements for scalability and parallelism. Our framework addresses these challenges by offering an intuitive, modular approach that seamlessly integrates recursion with efficient parallelization.

Standard DP education begins with recursion, moves to memoization, and ends with loop-based tabulation [3]. Yet, deriving efficient loops from recursive solutions is challenging for beginners and often neglects critical real-world considerations like multi-core parallelism and data-intensive computing.

We introduce a modular framework utilizing `OrderSpec` for defining traversal order and `tileSpec` for parallel execution through dependency graphs (DAGs).

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_37

By converting recursive algorithms into modular components integrated with automated functions like `dp_solve` and `dp_tile_solve`, the approach simplifies learning. Demonstrated on problems like LCS, MCM, and an extended rod-cutting example, our approach makes DP accessible and practical.

This paper provides a condensed review of related work (Section 2), details our DP framework (Section 3), demonstrates examples (Section 4), and concludes with teaching impacts and future directions(Section 5).

2 Related Work

Dynamic Programming (DP) is a core algorithmic technique for optimization, bioinformatics, and computational science, often requiring parallelization to manage computational costs. Efforts to optimize DP range from compiler-based transformations to task-based frameworks.

OpenMP is widely used for DP parallelization due to its simplicity [4], offering directives like `#pragma omp parallel` for concurrency. However, manual dependency management can be complex for irregular DP structures. Tools like the Pluto compiler [1] and LLVM Polly [13] automate parallel code generation for affine loops, enhancing locality and parallelism. Task-based frameworks, such as Taskflow [7] and Dask [12], dynamically infer dependencies, offering flexibility but requiring task decomposition. Recent work by Maleki et al. [10] parallelizes DP via rank convergence, enabling concurrent computation of dependent stages for algorithms like Needleman-Wunsch, achieving notable speedups.

Graph-based methods capture DP dependencies effectively. Algebraic Dynamic Programming (ADP) [6] abstracts state traversal and scoring, aiding bioinformatics tasks like sequence alignment. Petri nets model dependencies as token transitions [5], automating scheduling but adding overhead.

Hardware optimizations include GenDP [8] for genome sequencing and DP-HLS [2] for high-level synthesis in bioinformatics. These improve efficiency but demand specialized expertise.

DP is critical in bioinformatics for sequence alignment [11] and RNA structure prediction [14], and physics simulation-based experiments [9], all of which demonstrates the need for efficient and parallel implementations for large-scale problems.

This paper presents a systematic approach to transform recursive DP solutions into parallel implementations via recursion, dependency analysis, tabulation, tiling, and parallelism, making DP a plug & play component once recursion is defined.

3 DP Algorithmic Framework

Dynamic Programming (DP) algorithms systematically solve optimization problems by filling tables using nested loops, exploiting subproblem dependencies. Classic examples such as CutRod, Matrix Chain Multiplication (MCM), Longest

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_37

Common Subsequence (LCS), and Optimal Binary Search Tree (BST) showcase diverse traversal patterns (Figure 1).

DP Problem	Dimension	Start Loc	End Loc	Ordering	Filter(i,j)
CutRod	1D	0	n	row-major	True
MCM	2D	(n,1)	(1,n)	wavefront/row-major	$i \leq j$
LCS	2D	(1,1)	(m,n)	grid/row-major	True
Optimal BST	2D	(n-1,0)	(0,n-1)	wavefront/row-major	$i \leq j$

Fig. 1. Traversal patterns from MIT textbook [3]

Figure 1 summarizes traversal dimensions, start and end locations, traversal ordering, and filtering conditions.

To formalize DP tabulation, our framework in Figure 2 introduces Python classes: `OrderSpec2D` and `TileSpec2D`. `OrderSpec2D` defines traversal order through starting/ending coordinates, wavefront preference, row-major selection, and cell filtering conditions (e.g., $\lambda i, j : i \leq j$). Its `gen` method generates sequences of cell indices, replacing traditional nested loops.

```
class RRO:
    def prep(self): return
    def fill(self,i,j):
        return
    def result(self): return
class OrderSpec2D:
    def __init__(self,
      start, end,
      waveFront=False,
      rowMajor=True,
      Filter=labmda i,j:True):
      ...
    def gen(): ...
class TileSpec2D:
    def __init__(self,
       start,end,tileHeight,
       tileWidth): ...
    def gen(): ...
```

```
def dp_solve(rro, orderSpec):
    rro.prep()
    for i,j in orderSpec.seqGen():
        rro.fill(i,j)
    return rro.result()
def dp_tilie_solve(rro,tileSpec,inTileOrder):
    rro.prep()
    for tileId,(x0,y0),(x1,y1) in \
                          tileSpec.gen():
        inTileOrder.start = (x0,y0)
        inTileOrder.end = (x1,y1)
        for i,j in inTileOrder.gen():
            rro.fill(i,j)
    return rro.result()
def dag_run(rro, tileSpec,
            inTileOrder, tileSchedule):
    rro.prep()
    ... # e.g. setup and run dask
    return rro.result()
```

Fig. 2. DP Algortihmic Framework Library

The `dp_solve` function encapsulates sequential computation, accepting an `OrderSpec2D` instance and a Refactored Recursion Object (RRO). The RRO modularizes recursion into preparation (`prep`), cell computation (`fill`), and result extraction (`result`). Thus, DP algorithm design becomes a straightforward process: creating RRO and `OrderSpec2D` instances.

For parallel computation, we introduce the `TileSpec2D` class, defining traversal tiles with specific coordinates, tile dimensions, and unique identifiers (`tileID`). The `dp_tile_solve` function iterates over tiles sequentially, applying `OrderSpec2D` within each tile for correctness.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_37

Parallel execution is achieved through directed acyclic graph (DAG) scheduling, explicitly modeling tile-level dependencies beyond loop boundaries, enhancing flexibility over approaches like OpenMP. We utilize existing DAG schedulers such as Dask[12] (Python) or Taskflow[7] (C++), conceptualized in the `dag_run` function. `dag_run` combines tile dependency data (`tileDep`), traversal specifications, and RROs, generating executable task DAGs.

Our framework simplifies DP algorithm design and parallelization into defining modular RROs, `OrderSpec2D`, and `TileSpec2D`. This paper demonstrates these steps through examples: LCS (2D grid), MCM (wavefront with $i \leq j$), and CutRod (1D to 2D lifted parallel traversal).

4 Case Studies

4.1 Longest Common Subsequence (LCS)

The LCS problem is a standard example of 2D dynamic programming (DP). Given two strings X and Y, the goal is to find the length of their longest common subsequence. The recursive formulation is given in Equation (1), with a corresponding Python implementation shown in Figure 3. Calling `LCS(X, Y, 0, 0)` returns the correct result.

$$LCS(i,j) = \begin{cases} 1 + LCS(i+1, j+1), & \text{if } X[i] = Y[j] \\ \max(LCS(i, j+1), LCS(i+1, j)), & \text{otherwise} \end{cases} \tag{1}$$

```
def LCS(X, Y, i, j):
    if i >= len(X): return 0
    if j >= len(Y): return 0
    if X[i]==Y[j]:
        return LCS(X,Y,i+1,j+1)
    else:
        return max(LCS(X,Y,i+1,j),
                   LCS(X,Y,i,j+1))
```

Fig. 3. LCS Recursive Formulation

```
def lcs_dep(X, Y, i, j):
    if i>= len(X): return
    if j>= len(Y): return
    gen_edge((i,j), (i+1,j))
    gen_edge((i,j), (i,j+1))
    gen_edge((i,j), (i+1,j+1))
    lcs_dep(X,Y,i+1,j)
    lcs_dep(X,Y,i,j+1)
    lcs_dep(X,Y,i+1,j+1)
```

Fig. 4. LCS Dependency Graph Generator

With a small modification (Figure 4), we generate the LCS dependency graph, where each node corresponds to `(i,j)` in the DP table, and edges represent dependencies on `LCS(i+1,j+1)`, `LCS(i+1,j)`, and `LCS(i,j+1)`.

Figure 5 shows a dependency graph for `len(X)` $= 5$ and `len(Y)` $= 4$. This informs the `OrderSpec2D` definition (Figure 6) used in our `dp_solve` framework. The RRO encapsulation appears in Figure 7, and tiling configuration with `dp_tile_solve` is shown in Figure 8.

All four configurations of tabulation orders (combinations of wavefront/grid and row-/column-major) are valid and verifiable within the framework. For tiled

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_37

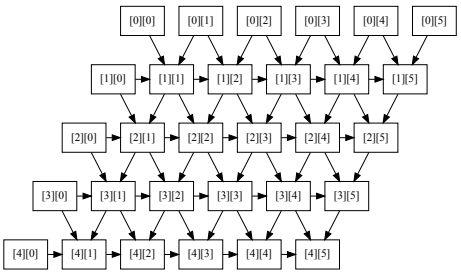

Fig. 5. Dependency Graph

```
# instantiate the LC problem
lcs = LCS_RR("...", "...")
# define ordering
orderSpec = OrderSpec2D(
    start = (lcs.m, lcs.n)
    end = (0,0),
    RowMajor=True,
    WaveFront = False )
# solve
r = dp_solve(lcs, orderSpec)
```

Fig. 6. LCS OrderSpec2D Usage

DP, combining 4 inter-tile and 4 in-tile orders yields 16 configurations. Including forward/backward traversal, a total of 32 valid orderings can compute LCS.

This variety highlights the flexibility of our framework in validating diverse traversal strategies.

```
import numpy as np
class LCS_RR(RRO):
    def __init__(self, X,Y):
        self.X,self.Y = X,Y
        self.m, self.n = len(X),len(Y)
    def prep(self): self.dp = \
        np.zeros((self.m+1, self.n+1))
    def fill(self, i, j):
        if self.X[i]  == self.Y[i]:
            self.dp[i][j] = ...
        else:
            self.dp[i][j] = max(...)
    def result(self):
        return self.dp[0][0]
```

Fig. 7. LCS RRO Encapsulation

```
rr = LCS_RR(...)
tileSpec = TileSpec(
  tileHeight = ...,
  tileWidth = ... ,
  tileOrder = OrderSpec(
     start=(rr.m,rr.n),
     end=(0,0),
     RowMajor=True,
     WaveFront = False) )
inTileOrder = OrderSpec(
  start=None, end=None,
  RowMajor=False,WaveFront=True)
r = dp_tile_solve(rr,tileSpec,
                  inTileOrder)
```

Fig. 8. LCS Tiling Configuration

4.2 MCM with Transitive Reduction

Matrix Chain Multiplication (MCM) is a classic dynamic programming optimization problem. Its recursive formulation is:

$$m(i,j) = \begin{cases} 0, & \text{if } i = j \\ \min_{k=i}^{j-1}(m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j), & \text{if } i < j \end{cases} \quad (2)$$

From this, we generate a dependency graph, shown in Figure 9.

The graph contains redundant edges (e.g., $A \to C$ if $A \to B \to C$ exists). Transitive reduction removes these, simplifying Figure 9 to Figure 10. This reduced graph clarifies start/end locations per Table 1, enabling an RRO instance

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_37

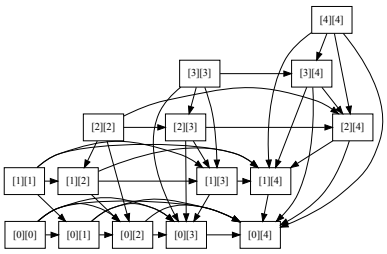

Fig. 9. Raw MCM Dependency Graph

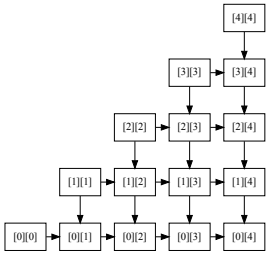

Fig. 10. Reduced Dependency Graph

for the DP framework, like LCS. We set `OrderSpec2D` with `Filter = lambda i,j: i<=j` to traverse the upper triangular table. Unlike textbook [3] wavefront ordering, all four `OrderSpec2D` configurations are valid, requiring no middle-diagonal start.

4.3 Case Study CutRod

The Rod Cutting problem is a classic dynamic programming optimization task. Its recursive formulation is:

$$R(n) = \begin{cases} 0, & \text{if } n = 0, \\ \max_{1 \le i \le n} \left(p[i] + R(n-i) \right), & \text{if } n > 0. \end{cases} \tag{3}$$

We generate a dependency graph (Figure 11), simplified via transitive reduction to Figure 12.

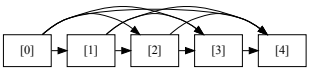

Fig. 11. Raw Dependency Graph

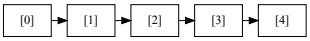

Fig. 12. Transitive Reduced Graph

A 1D DP implementation is shown in Figure 13. Viewing `r` as a 2D array, each iteration computes `r[i][j]`, with `r[i]` as `max(r[i][:i])`. This leads to a 2D version in Figure 14. The 2D version's dependency graph (Figure 15, `n=5`) is generated by instrumenting loops, unlike LCS and MCM.

This graph shows non-neighboring dependencies (e.g., `(0,0)` to `(5,0)`). Grid or wavefront ordering from `(0,0)` satisfies these, but irregular dependencies challenge OpenMP. Our DAG-based scheduling framework enables plug-and-play multicore execution, as shown with LCS and MCM.

An additional optimization is that the 2D array can revert to 1D by using Figure 13's line 5, optimizing memory while preserving correctness, since the *max* operator is communicative, associative, and monotonic.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_37

```python
def cutrod(P, n):
    r = [0] * (n+1)
    for i in range(n+1):
        for j in in range(i):
            r[i] = max(r[i],
                r[j]+P[i-j])
    return r[n]
```

Fig. 13. CutRod DP Algorithm

```python
import numpy as np
def cutrod_dp_2d(P, n):
    r = np.zeros((n+1, n+1))
    for i in range(1, n+1):
        for j in range(i):
            r[i][j] = max(
        r[j][j], r[j][j]+P[i-j])
        r[i][i]=max([r[i][:i]])
    return r[n][n]
```

Fig. 14. CutRod DP Algorithm in 2D

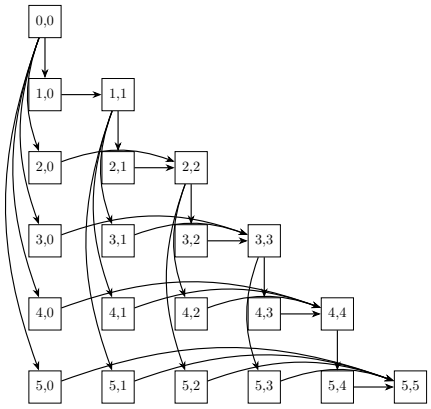

Fig. 15. Dependency Graph for Figure 14

5 Conclusion and Future Work

We presented a modular framework for dynamic programming (DP) that begins from recursion, captures dependencies explicitly, and transitions naturally into sequential and parallel implementations. Instead of teaching loop-centric DP from the outset, our approach advocates starting from the recursive formulation, generating the dependency graph, and using that to drive tabulation order and parallelism.

This method reshapes how DP is taught: students gain intuition by visualizing dependencies, then implement sequence generators as modular programming exercises. They no longer struggle to derive nested loops from recurrences—instead, they construct and test different tabulation strategies and plug them into a reusable library.

Beyond education, this framework bridges theory and practice by supporting task-based parallelization through DAG schedulers like Dask or Taskflow. It scales to real-world workloads while remaining accessible for students and researchers.

Future directions include classroom deployment with visualization tools, integrating tiled DP with GPU or distributed execution, and extending the frame-

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_37

work to support more complex problems like irregular and higher-dimensional DP, which are common in computational science and engineering.

References

1. Udayan Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. Pluto: An automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices*, 43(6):101–113, 2008. doi:10.1145/1375581.1375595.
2. Yingqi Cao, Anshu Gupta, Jason Liang, and Yatish Turakhia. DP-HLS: A high-level synthesis framework for accelerating dynamic programming algorithms in bioinformatics. *CoRR*, abs/2411.03398, 2024. URL: https://doi.org/10.48550/arXiv.2411.03398, arXiv:2411.03398, doi:10.48550/ARXIV.2411.03398.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
4. L. Dagum and R. Menon. Openmp: An industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. doi:10.1109/99.660313.
5. Javier Esparza and Jens Knoop. Petri nets for dynamic programming. In *Application and Theory of Petri Nets*, pages 210–229. Springer, 2010. doi:10.1109/HICSS.2004.1265209.
6. Robert Giegerich and Carsten Meyer. Algebraic dynamic programming. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, AMAST '02, page 349–364, Berlin, Heidelberg, 2002. Springer-Verlag.
7. Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. Cpp-taskflow: Fast task-based parallel programming using modern c++. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 974–983. IEEE, 2019. doi:10.1109/IPDPS.2019.00105.
8. Yaohua Liang, Hao Wang, Matthew Wolf, and Sheng Zhou. Gendp: A general-purpose framework for accelerating dynamic programming on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):260–274, 2016. doi:10.1145/3579371.3589060.
9. Peter N. Loxley and Ka-Wai Cheung. A dynamic programming algorithm for finding an optimal sequence of informative measurements. *Entropy*, 25(2):251, 2023. doi:10.3390/e25020251.
10. Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing dynamic programming through rank convergence. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 219–232. ACM, 2014. doi:10.1145/2555243.2555264.
11. Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. doi:10.1016/0022-2836(70)90057-4.
12. Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, pages 130–136, 2015. doi:10.25080/Majora-7b98e3ed-013.
13. Christian Lengauer Tobias Grosser, Armin Groesslinger. Polly - performing polyhedral optimizations on a low-level intermediate representation. URL: https://polly.llvm.org.
14. M. Zuker and P. Stiegler. Optimal computer folding of large rna sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):133–148, 1981. doi:10.1093/nar/9.1.133.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_37