Dynamic Programming

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text "Introduction to Parallel Computing", Addison Wesley, 2003
Topic Overview

• Overview of Serial Dynamic Programming
• Serial Monadic DP Formulations
• Nonserial Monadic DP Formulations
• Serial Polyadic DP Formulations
• Nonserial Polyadic DP Formulations
Overview of Serial Dynamic Programming

• *Dynamic programming* (DP) is used to solve a wide variety of *discrete optimization problems* such as scheduling, string-editing, packaging, and inventory management.

• **Break problems into subproblems** and **combine their solutions into solutions** to larger problems.

• In contrast to divide-and-conquer, there may be relationships across subproblems.
Dynamic Programming: Example

- Consider the problem of finding a **shortest path between a pair of vertices** in an **acyclic graph**.
- An edge connecting node $i$ to node $j$ has cost $c(i,j)$.
- The graph contains $n$ nodes numbered $0, 1, \ldots, n-1$, and has an **edge from node $i$ to node $j$ only if $i < j$**. Node 0 is **source** and node $n-1$ is the **destination**.
- Let $f(x)$ be the **cost of the shortest path** from node 0 to node $x$.

$$f(x) = \begin{cases} 0 & x = 0 \\ \min_{0 \leq j < x} \{ f(j) + c(j, x) \} & 1 \leq x \leq n - 1 \end{cases}$$
Dynamic Programming: Example

- A graph for which the shortest path between nodes 0 and 4 is to be computed.

\[ f(4) = \min \{ f(3) + c(3, 4), f(2) + c(2, 4) \} . \]
Dynamic Programming

• The solution to a DP problem is typically expressed as a minimum (or maximum) of possible alternate solutions.
• If \( r \) represents the cost of a solution composed of subproblems \( x_1, x_2, \ldots, x_l \), then \( r \) can be written as

\[
r = g(f(x_1), f(x_2), \ldots, f(x_l)).
\]

Here, \( g \) is the composition function.
• If the optimal solution to each problem is determined by composing optimal solutions to the subproblems and selecting the minimum (or maximum), the formulation is said to be a DP formulation.
Dynamic Programming: Example

The computation and composition of subproblem solutions to solve problem $f(x_8)$. 
Dynamic Programming

• The recursive DP equation is also called the *functional equation* or *optimization equation*.

• In the equation for the *shortest path problem* the composition function is $f(j) + c(j,x)$. This contains a single recursive term ($f(j)$). Such a formulation is called *monadic*.

• If the RHS has multiple recursive terms, the DP formulation is called *polyadic*.
Dynamic Programming

• The **dependencies between subproblems** can be expressed as a graph.
• If the **graph can be levelized** (i.e., solutions to problems at a level depend only on solutions to problems at the previous level), the formulation is called **serial**, else it is called **non-serial**.
• Based on these two criteria, we can classify DP formulations into **four categories** - **serial-monadic**, **serial-polyadic**, **non-serial-monadic**, **non-serial-polyadic**.
• This classification is useful since it **identifies concurrency** and dependencies that guide parallel formulations.
Serial Monadic DP Formulations

• It is difficult to derive canonical parallel formulations for the entire class of formulations.
• For this reason, we select two representative examples, the shortest-path problem for a multistage graph and the 0/1 knapsack problem.
• We derive parallel formulations for these problems and identify common principles guiding design within the class.
Shortest-Path Problem

- **Special class of shortest path problem** where the graph is a weighted multistage graph of \( r + 1 \) levels.
- **Each level is assumed to have \( n \) nodes** and every node at level \( i \) is connected to every node at level \( i + 1 \).
- Levels zero and \( r \) contain only one node, the **source and destination nodes**, respectively.
- The objective of this problem is to **find the shortest path from \( S \) to \( R \).**
An example of a serial monadic DP formulation for finding the shortest path in a graph whose nodes can be organized into levels.
Shortest-Path Problem

• The \( j^{th} \) node at level \( l \) in the graph is labeled \( v_i^l \) and the cost of an edge connecting \( v_i^l \) to node \( v_j^{l+1} \) is labeled \( c_{i,j}^l \).

• The **cost of reaching the goal node** \( R \) from any node \( v_i^l \) is represented by \( C_i^l \).

• If there are \( n \) nodes at **level** \( l \), the vector 
  \[
  [C_0^l, C_1^l, \ldots, C_{n-1}^l]^T
  \]
  is referred to as \( C^l \). Note that \( C_0 = [C_0^0] \).

• We have \( C_i^l = \min \left\{ (c_{i,j}^l + C_j^{l+1}) \mid j \text{ is a node at level } l + 1 \right\} \)
Shortest-Path Problem

• Since all nodes $v_{r-1}^j$ have only one edge connecting them to the goal node $R$ at level $r$, the cost $C_{r-1}^j$ is equal to $c_{r-1, R}^j$.

• We have:

$$C_{r-1} = \left[ c_{0, R}^{r-1}, c_{1, R}^{r-1}, \cdots, c_{n-1, R}^{r-1} \right].$$

Notice that this problem is serial and monadic.
Shortest-Path Problem

• The cost of reaching the goal node $R$ from any node at level $l$ is $(0 < l < r - 1)$ is

\[
C^l_0 = \min\{(c^l_{0,0} + C^{l+1}_0), (c^l_{0,1} + C^{l+1}_1), \ldots, (c^l_{0,n-1} + C^{l+1}_{n-1})\},
\]

\[
C^l_1 = \min\{(c^l_{1,0} + C^{l+1}_0), (c^l_{1,1} + C^{l+1}_1), \ldots, (c^l_{1,n-1} + C^{l+1}_{n-1})\},
\]

\vdots

\[
C^l_{n-1} = \min\{(c^l_{n-1,0} + C^{l+1}_0), (c^l_{n-1,1} + C^{l+1}_1), \ldots, (c^l_{n-1,n-1} + C^{l+1}_{n-1})\}.
\]
Shortest-Path Problem

• We can express the solution to the problem as a modified sequence of matrix-vector products.

• Replacing the addition operation by minimization and the multiplication operation by addition, the preceding set of equations becomes:

\[ C^l = M_{l,l+1} \times C^{l+1}, \]

where \( C^l \) and \( C^{l+1} \) are \( n \times 1 \) vectors representing the cost of reaching the goal node from each node at levels \( l \) and \( l+1 \).
Shortest-Path Problem

• Matrix $M_{l,l+1}$ is an $n \times n$ matrix in which entry $(i, j)$ stores the cost of the edge connecting node $i$ at level $l$ to node $j$ at level $l+1$.

$$M_{l,l+1} = \begin{bmatrix}
  c_{0,0}^l & c_{0,1}^l & \cdots & c_{0,n-1}^l \\
  c_{1,0}^l & c_{1,1}^l & \cdots & c_{1,n-1}^l \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{n-1,0}^l & c_{n-1,1}^l & \cdots & c_{n-1,n-1}^l
\end{bmatrix}.$$ 

• The shortest path problem has been formulated as a sequence of $r$ matrix-vector products.
Parallel Shortest-Path

- We can parallelize this algorithm using the parallel algorithms for the matrix-vector product.
- \( \Theta(n) \) processing elements can compute each vector \( C' \) in time \( \Theta(n) \) and solve the entire problem in time \( \Theta(rn) \).
- In many instances of this problem, the matrix \( M \) may be sparse. For such problems, it is highly desirable to use sparse matrix techniques.
0/1 Knapsack Problem

- We are given a **knapsack of capacity** $c$ and a set of $n$ **objects** numbered $1, 2, \ldots, n$. Each object $i$ has **weight** $w_i$ and **profit** $p_i$.
- Let $v = [v_1, v_2, \ldots, v_n]$ be a **solution vector** in which $v_i = 0$ if object $i$ is not in the knapsack, and $v_i = 1$ if it is in the knapsack.
- The goal is to **find a subset of objects** to put into the knapsack so that
  \[
  \sum_{i=1}^{n} w_i v_i \leq c
  \]
  (that is, the **objects fit into the knapsack**) and
  \[
  \sum_{i=1}^{n} p_i v_i
  \]
  is maximized (that is, the **profit is maximized**).
0/1 Knapsack Problem

- The **naive method** is to consider all $2^n$ possible **subsets** of the $n$ objects and choose the one that fits into the knapsack and maximizes the profit.

- Let $F[i,x]$ be the **maximum profit for a knapsack of capacity** $x$ using only objects $\{1,2,\ldots,i\}$. The DP formulation is:

$$F[i, x] = \begin{cases} 
0 & x \geq 0, i = 0 \\
-\infty & x < 0, i = 0 \\
\max\{F[i - 1, x], (F[i - 1, x - w_i] + p_i)\} & 1 \leq i \leq n
\end{cases}$$
0/1 Knapsack Problem

- Construct a table $F$ of size $n \times c$ in row-major order.
- Filling an entry in a row requires two entries from the previous row: one from the same column and one from the column offset by the weight of the object corresponding to the row.
- Computing each entry takes constant time; the sequential run time of this algorithm is $\Theta(nc)$.
- The formulation is serial-monadic.
Computing entries of table $F$ for the 0/1 knapsack problem. The computation of entry $F[i,j]$ requires communication with processing elements containing entries $F[i-1,j]$ and $F[i-1,j-w_i]$.2
0/1 Knapsack Problem

- Using **c processors** in a **PRAM**, we can derive a simple parallel algorithm that runs in **$O(n)$** time by partitioning the columns across processors.

- In a **distributed memory machine**, in the $j^{th}$ iteration, for computing $F[j,r]$ at processing element $P_{r-1}$, $F[j-1,r]$ is available locally but $F[j-1,r-w_j]$ must be fetched.

- The communication operation is a **circular shift** and the time is given by $(t_s + t_w) \log c$. The **total time** is therefore $t_c + (t_s + t_w) \log c$.

- Across all $n$ iterations (rows), the parallel time is $O(n \log c)$. Note that this is **not cost optimal**.
Nonserial Monadic DP Formulations: Longest-Common-Subsequence

• Given a sequence $A = <a_1, a_2, \ldots, a_n>$, a subsequence of $A$ can be formed by deleting some entries from $A$.

• Given two sequences $A = <a_1, a_2, \ldots, a_n>$ and $B = <b_1, b_2, \ldots, b_m>$, find the longest sequence that is a subsequence of both $A$ and $B$.

• If $A = <c,a,d,b,r,z>$ and $B = <a,s,b,z>$, the longest common subsequence of $A$ and $B$ is $<a,b,z>$. 
Longest-Common-Subsequence Problem

- Let $F[i,j]$ denote the **length of the longest common subsequence** of the first $i$ elements of $A$ and the first $j$ elements of $B$. The objective of the LCS problem is to find $F[n,m]$.

- We can write:

$$F[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
F[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\
\max \{F[i, j - 1], F[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j
\end{cases}$$
Longest-Common-Subsequence Problem

• The algorithm computes the **two-dimensional F table** in a row- or column-major fashion. **The complexity is** $\Theta(nm)$.
• Treating nodes along a diagonal as belonging to one level, **each node depends on two subproblems at the preceding level and one subproblem two levels prior**.
• This DP formulation is **nonserial monadic**.
Longest-Common-Subsequence Problem

(a) Computing entries of table for the longest-common-subsequence problem. Computation proceeds along the dotted diagonal lines. (b) Mapping elements of the table to processing elements.
Consider the LCS of two amino-acid sequences **HEAGAWGHEE** and **PAWHHEA**. For the interested reader, the names of the corresponding amino-acids are A: Alanine, E: Glutamic acid, G: Glycine, H: Histidine, P: Proline, and W: Tryptophan.

The $F$ table for computing the LCS of the sequences. The LCS is **A W H E E**.
Parallel Longest-Common-Subsequence

- Table entries are **computed in a diagonal sweep** from the top-left to the bottom-right corner.
- Using *n processors* in a **PRAM**, each entry in a **diagonal** can be computed in **constant time**.
- For two sequences of length *n*, there are **2n-1 diagonals**.

- The parallel **run time is \( \Theta(n) \)** and the **algorithm is cost-optimal**.
Parallel Longest-Common-Subsequence

- Consider a (logical) **linear array of processors**. Processing element $P_i$ is responsible for the $(i+1)^{th}$ column of the table.
- To compute $F[i,j]$, processing element $P_{j-1}$ may need either $F[i-1,j-1]$ or $F[i,j-1]$ from the processing element to its left. This **communication takes time** $t_s + t_w$.
- The computation takes constant time ($t_c$).
- We have:
  $$T_P = (2n - 1)(t_s + t_w + t_c).$$
- Note that this **formulation is cost-optimal**, however, its efficiency is upper-bounded by 0.5!
Serial Polyadic DP Formulation: Floyd's All-Pairs Shortest Path

• Given **weighted graph** $G(V,E)$, Floyd's algorithm determines the cost $d_{i,j}$ of the **shortest path between each pair of nodes in** $V$.

• Let $d_{i,j}^{k}$ be the minimum cost of a path from node $i$ to node $j$, using only nodes $v_0, v_1, \ldots, v_{k-1}$.

• We have:

$$
d_{i,j}^{k} = \begin{cases} 
c_{i,j} & k = 0 \\
\min \{ d_{i,j}^{k-1}, (d_{i,k}^{k-1} + d_{k,j}^{k-1}) \} & 0 \leq k \leq n - 1
\end{cases}
$$

• **Each iteration requires time** $\Theta(n^2)$ and the overall run time of the **sequential algorithm is** $\Theta(n^3)$. 
Serial Polyadic DP Formulation: Floyd's All-Pairs Shortest Path

• A PRAM formulation of this algorithm uses \( n^2 \) processors in a logical 2D mesh. Processor \( P_{i,j} \) computes the value of \( d_{i,j}^k \) for \( k=1,2,\ldots,n \) in constant time.

• The parallel runtime is \( \Theta(n) \) and it is cost-optimal.
Nonserial Polyadic DP Formulation: Optimal Matrix-Parenthesization Problem

- When **multiplying a sequence of matrices**, the order of multiplication significantly impacts **operation count**.
- Let $C[i,j]$ be the optimal cost of multiplying the matrices $A_i, \ldots A_j$.
- The chain of matrices can be expressed as a **product of two smaller chains**:
  $A_i, A_{i+1}, \ldots, A_k$ and $A_{k+1}, \ldots, A_j$.

- The chain $A_i, A_{i+1}, \ldots, A_k$ results in a matrix of dimensions $r_{i-1} \times r_k$, and the chain $A_{k+1}, \ldots, A_j$ results in a matrix of dimensions $r_k \times r_j$.
- The **cost of multiplying these two matrices** is $r_{i-1}r_k r_j$. 
Optimal Matrix-Parenthesization Problem - Example

- Consider three matrices $A_1$, $A_2$, and $A_3$ of dimensions 10x20, 20x30, and 30x40, respectively.
- The product of these matrices can be computed as $(A_1 \times A_2) \times A_3$ or as $A_1 \times (A_2 \times A_3)$.
- In $(A_1 \times A_2) \times A_3$, computing $(A_1 \times A_2)$ requires $10 \times 20 \times 30$ operations and yields a matrix of dimensions $10 \times 30$. Multiplying this by $A_3$ requires $10 \times 30 \times 40$ additional operations. Therefore the total number of operations is $10 \times 20 \times 30 + 10 \times 30 \times 40 = 18000$.
- Similarly, computing $A_1 \times (A_2 \times A_3)$ requires $20 \times 30 \times 40 + 10 \times 20 \times 40 = 32000$ operations.
- The first parenthesization is desirable.
Optimal Matrix-Parenthesization Problem

• We have:

\[ C[i, j] = \begin{cases} 
\min_{i \leq k < j} \{ C[i, k] + C[k + 1, j] + r_{i-1}r_kr_j \} & 1 \leq i < j \leq n \\
0 & j = i, 0 < i \leq n 
\end{cases} \]
A nonserial polyadic DP formulation for finding an optimal matrix parenthesization for a chain of four matrices. A square node represents the optimal cost of multiplying a matrix chain. A circle node represents a possible parenthesization.
Optimal Matrix-Parenthesization Problem

- The goal of finding $C[1,n]$ is accomplished in a bottom-up fashion.
- Visualize this by thinking of filling in the $C$ table diagonally. Entries in diagonal $l$ corresponds to the cost of multiplying matrix chains of length $l+1$.
- The value of $C[i,j]$ is computed as $\min\{C[i,k] + C[k+1,j] + r_{i-1}r_kr_j\}$, where $k$ can take values from $i$ to $j-1$.
- Computing $C[i,j]$ requires that we evaluate $(j-i)$ terms and select their minimum.
- The computation of each term takes time $t_c$, and the computation of $C[i,j]$ takes time $(j-i)t_c$. Each entry in diagonal $l$ can be computed in time $lt_c$. 
Optimal Matrix-Parenthesization Problem

- The algorithm computes \((n-1)\) chains of length two. This takes time \((n-1)t_c\); computing \(n-2\) chains of length three takes time \((n-2)2t_c\). In the final step, the algorithm computes one chain of length \(n\) in time \(1(n-1)t_c\).

- It follows that the serial time is \(\Theta(n^3)\).
Optimal Matrix-Parenthesization Problem

The diagonal order of computation for the optimal matrix-parenthesization problem.
Parallel Optimal Matrix-Parenthesization Problem

- Consider a **logical ring of processors**. In step $l$, each processor computes a single element belonging to the $l^{th}$ diagonal.
- On computing the assigned value of the element in table $C$, each processor sends its value to all other processors using an all-to-all broadcast.
- The next value can then be computed locally.
- The total time required to compute the entries along diagonal $l$ is $lt_c + ts \log n + tw(n-1)$.
- The corresponding parallel time is given by:

$$T_P = \sum_{l=1}^{n-1} (lt_c + ts \log n + tw(n - 1)),$$

$$= \frac{(n - 1)(n)}{2} t_c + ts(n - 1) \log n + tw(n - 1)^2.$$