CHAPTER 11

Dynamic Programming

Dynamic programming (DP) is a commonly used technique for solving a wide variety of discrete optimization problems such as scheduling, string-editing, packaging, and inventory management. More recently, it has found applications in bioinformatics in matching sequences of amino-acids and nucleotides (the Smith-Waterman algorithm). DP views a problem as a set of interdependent subproblems. It solves subproblems and uses the results to solve larger subproblems until the entire problem is solved. In contrast to divide-and-conquer, where the solution to a problem depends only on the solution to its subproblems in DP, there may be interrelationships across subproblems. In DP, the solution to a subproblem is expressed as a function of solutions to one or more subproblems at the preceding levels. We start our discussion with a simple DP algorithm for computing shortest paths in a graph.

Example 11.1 The shortest-path problem

Consider a DP formulation for the problem of finding a shortest (least-cost) path between a pair of vertices in an acyclic graph. (Refer to Section **??** for an introduction to graph terminology.) An edge connecting node i to node j has cost $c(i, j)$. If two vertices i and j are not connected then $c(i, j) = \infty$. The graph contains n nodes numbered $0, 1, \ldots, n-1$, and has an edge from node i to node j only if $i < j$. The shortest-path problem is to find a least-cost path between nodes 0 and $n-1$. Let $f(x)$ denote the cost of the least-cost path from node 0 to node x. Thus, $f(0)$ is zero, and finding $f(n-1)$ solves the problem. The DP formulation for this problem yields the following recursive equations for $f(x)$:

$$f(x) = \begin{cases} 0 & x = 0 \\ \min\limits_{0 \le j < x} \{f(j) + c(j, x)\} & 1 \le x \le n-1 \end{cases} \tag{11.1}$$

As an instance of this algorithm, consider the five-node acyclic graph shown in

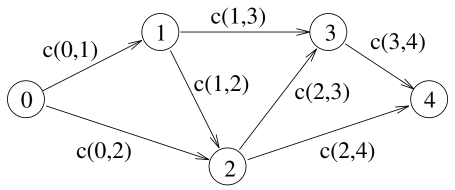

Figure 11.1 A graph for which the shortest path between nodes 0 and 4 is to be computed.

Figure 11.1. The problem is to find $f(4)$. It can be computed given $f(3)$ and $f(2)$. More precisely,

$$f(4) = \min\{f(3) + c(3,4), f(2) + c(2,4)\}.$$

Therefore, $f(2)$ and $f(3)$ are elements of the set of subproblems on which $f(4)$ depends. Similarly, $f(3)$ depends on $f(1)$ and $f(2)$, and $f(1)$ and $f(2)$ depend on $f(0)$. Since $f(0)$ is known, it is used to solve $f(1)$ and $f(2)$, which are used to solve $f(3)$. ■

In general, the solution to a DP problem is expressed as a minimum (or maximum) of possible alternate solutions. Each of these alternate solutions is constructed by composing one or more subproblems. If r represents the cost of a solution composed of subproblems $x_1, x_2, \ldots, x_l$, then r can be written as

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

The function g is called the ***c***omposition function, and its nature depends on the problem. 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. Figure 11.2 illustrates an instance of composition and minimization of solutions. The solution to problem x_8 is the minimum of the three possible solutions having costs r_1, r_2, and r_3. The cost of the first solution is determined by composing solutions to subproblems x_1 and x_3, the second solution by composing solutions to subproblems x_4 and x_5, and the third solution by composing solutions to subproblems x_2, x_6, and x_7.

DP represents the solution to an optimization problem as a recursive equation whose left side is an unknown quantity and whose right side is a minimization (or maximization) expression. Such an equation is called a ***f***unctional equation or an ***o***ptimization equation. In Equation 11.1, the composition function g is given by $f(j) + c(j,x)$. This function is additive, since it is the sum of two terms. In a general DP formulation, the cost function need not be additive. A functional equation that contains a single recursive term (for example, $f(j)$) yields a ***m***onadic DP formulation. For an arbitrary DP formulation, the cost function may contain multiple recursive terms. DP formulations whose cost function contains multiple recursive terms are called ***p***olyadic formulations.

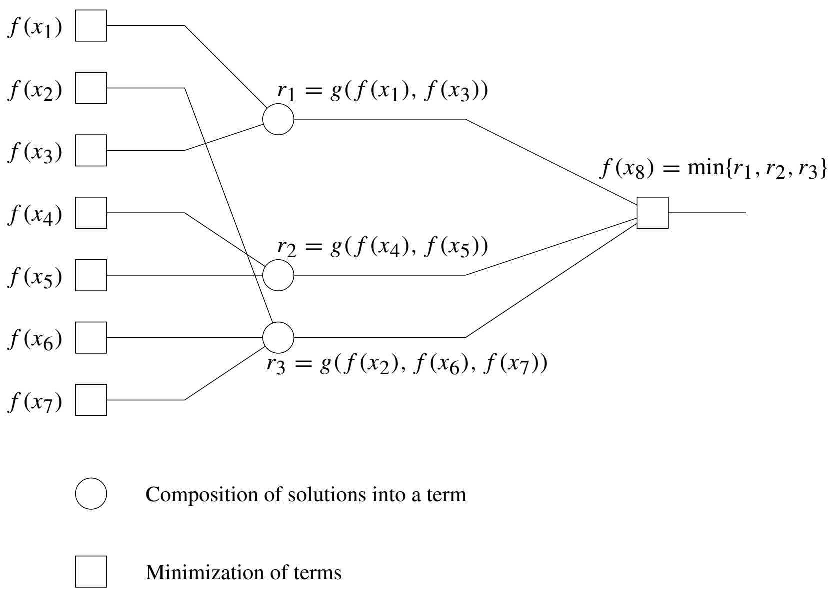

Figure 11.2 The computation and composition of subproblem solutions to solve problem $f(x_8)$.

The dependencies between subproblems in a DP formulation can be represented by a directed graph. Each node in the graph represents a subproblem. A directed edge from node i to node j indicates that the solution to the subproblem represented by node i is used to compute the solution to the subproblem represented by node j. If the graph is acyclic, then the nodes of the graph can be organized into levels such that subproblems at a particular level depend only on subproblems at previous levels. In this case, the DP formulation can be categorized as follows. If subproblems at all levels depend only on the results at the immediately preceding levels, the formulation is called a ***s***erial DP formulation; otherwise, it is called a ***n***onserial DP formulation.

Based on the preceding classification criteria, we define four classes of DP formulations: ***s***erial monadic, ***s***erial polyadic, ***n***onserial monadic, and ***n***onserial polyadic. These classes, however, are not exhaustive; some DP formulations cannot be classified into any of these categories.

Due to the wide variety of problems solved using DP, it is difficult to develop generic parallel algorithms for them. However, parallel formulations of the problems in each of the four DP categories have certain similarities. In this chapter, we discuss parallel DP formulations for sample problems in each class. These samples suggest parallel algorithms for other problems in the same class. Note, however, that not all DP problems can be parallelized as illustrated in these examples.

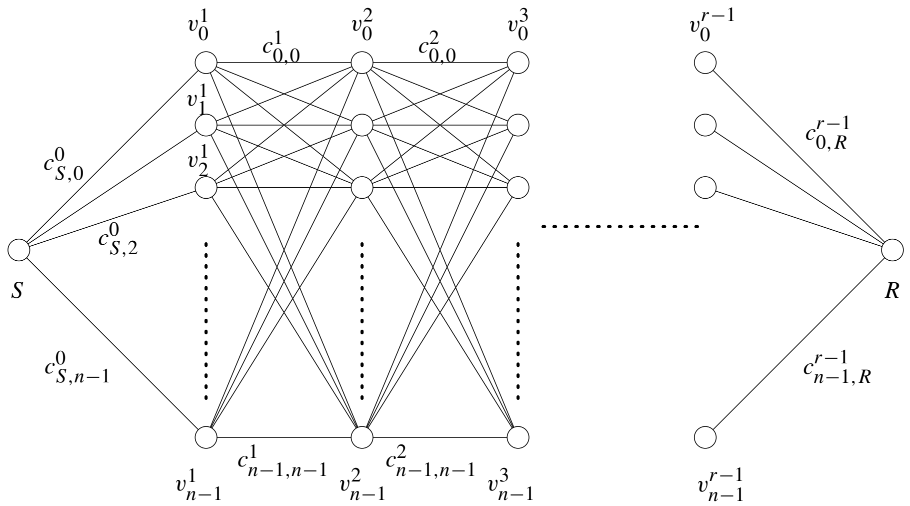

Figure 11.3 An example of a serial monadic DP formulation for finding the shortest path in a graph whose nodes can be organized into levels.

11.1 Serial Monadic DP Formulations

We can solve many problems by using serial monadic DP formulations. This section discusses the shortest-path problem for a multistage graph and the 0/1 knapsack problem. We present parallel algorithms for both and point out the specific properties that influence these parallel formulations.

11.1.1 The Shortest-Path Problem

Consider a weighted multistage graph of $r + 1$ levels, as shown in Figure 11.3. Each node at level i is connected to every node at level $i + 1$. Levels zero and r contain only one node, and every other level contains n nodes. We refer to the node at level zero as the starting node S and the node at level r as the terminating node R. The objective of this problem is to find the shortest path from S to R. The i^{th} node at level l in the graph is labeled v_i^l. 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 $\mathcal{C}^l$. The shortest-path problem reduces to computing $\mathcal{C}^0$. Since the graph has only one starting node, $\mathcal{C}^0 = [C_0^0]$. The structure of the graph is such that any path from v_i^l to R includes a node v_j^{l+1} $(0 \le j \le n - 1)$. The cost of any such path is the sum of the cost of the edge between v_i^l and v_j^{l+1} and the cost of the shortest path between v_j^{l+1} and R (which is given by C_j^{l+1}). Thus, C_i^l, the cost of the shortest path between v_i^l and R, is equal to the minimum cost over all paths through each node in level $l + 1$. Therefore,

$$C_i^l = \min \left\{ (c_{i,j}^l + C_j^{l+1}) | j \text{ is a node at level } l+1 \right\}. \tag{11.2}$$

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

$$\mathcal{C}^{r-1} = [c_{0,R}^{r-1}, c_{1,R}^{r-1}, \ldots, c_{n-1,R}^{r-1}]. \tag{11.3}$$

Because Equation 11.2 contains only one recursive term in its right-hand side, it is a monadic formulation. Note that the solution to a subproblem requires solutions to subproblems only at the immediately preceding level. Consequently, this is a serial monadic formulation.

Using this recursive formulation of the shortest-path problem, the cost of reaching the goal node R from any node at level l $(0 < l < r-1)$ is

$$\begin{array}{rcl}
C_0^l & = & \min\{(c_{0,0}^l + C_0^{l+1}), (c_{0,1}^l + C_1^{l+1}), \ldots, (c_{0,n-1}^l + C_{n-1}^{l+1})\}, \\
C_1^l & = & \min\{(c_{1,0}^l + C_0^{l+1}), (c_{1,1}^l + C_1^{l+1}), \ldots, (c_{1,n-1}^l + C_{n-1}^{l+1})\}, \\
 & \vdots & \\
C_{n-1}^l & = & \min\{(c_{n-1,0}^l + C_0^{l+1}), (c_{n-1,1}^l + C_1^{l+1}), \ldots, (c_{n-1,n-1}^l + C_{n-1}^{l+1})\}.
\end{array}$$

Now consider the operation of multiplying a matrix with a vector. In the matrix-vector product, if the addition operation is replaced by minimization and the multiplication operation is replaced by addition, the preceding set of equations is equivalent to

$$\mathcal{C}^l = M_{l,l+1} \times \mathcal{C}^{l+1}, \tag{11.4}$$

where $\mathcal{C}^l$ and $\mathcal{C}^{l+1}$ are $n \times 1$ vectors representing the cost of reaching the goal node from each node at levels l and $l+1$, and $M_{l,l+1}$ 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 $l+1$. This matrix is

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

The shortest-path problem has thus been reformulated as a sequence of matrix-vector multiplications. On a sequential computer, the DP formulation starts by computing $\mathcal{C}^{r-1}$ from Equation 11.3, and then computes $\mathcal{C}^{r-k-1}$ for $k = 1, 2, \ldots, r-2$ using Equation 11.4. Finally, $\mathcal{C}^0$ is computed using Equation 11.2.

Since there are n nodes at each level, the cost of computing each vector $\mathcal{C}^l$ is $\Theta(n^2)$. The parallel algorithm for this problem can be derived using the parallel algorithms for the matrix-vector product discussed in Section **??**. For example, $\Theta(n)$ processing elements can compute each vector $\mathcal{C}^l$ in time $\Theta(n)$ and solve the entire problem in time $\Theta(rn)$. Recall that r is the number of levels in the graph.

Many serial monadic DP formulations with dependency graphs identical to the one considered here can be parallelized using a similar parallel algorithm. For certain dependency graphs, however, this formulation is unsuitable. Consider a graph in which each node at a level can be reached from only a small fraction of nodes at the previous level. Then matrix $M_{l,l+1}$ contains many elements with value ∞. In this case, matrix M is considered to be a sparse matrix for the minimization and addition operations. This is because, for all x, $x + \infty = \infty$, and $\min\{x, \infty\} = x$. Therefore, the addition and minimization operations need not be performed for entries whose value is ∞. If we use a regular dense matrix-vector multiplication algorithm, the computational complexity of each matrix-vector multiplication becomes significantly higher than that of the corresponding sparse matrix-vector multiplication. Consequently, we must use a sparse matrix-vector multiplication algorithm to compute each vector.

11.1.2 The 0/1 Knapsack Problem

A one-dimensional 0/1 knapsack problem is defined as follows. 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. Object profits and weights are integers. 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).

A straightforward method to solve this problem 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. Here we provide a DP formulation that is faster than the simple method when $c = O(2^n/n)$. Let $F[i, x]$ be the maximum profit for a knapsack of capacity x using only objects $\{1, 2, \ldots, i\}$. Then $F[n, c]$ is the solution to the problem. The DP formulation for this problem is as follows:

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

This recursive equation yields a knapsack of maximum profit. When the current capacity of the knapsack is x, the decision to include object i can lead to one of two situations: (i) the object is not included, knapsack capacity remains x, and profit is unchanged; (ii) the object is included, knapsack capacity becomes $x - w_i$, and profit increases by p_i. The DP algorithm decides whether or not to include an object based on which choice leads to maximum profit.

The sequential algorithm for this DP formulation maintains a table F of size $n \times c$. The table is constructed in row-major order. The algorithm first determines the maximum profit by using only the first object with knapsacks of different capacities. This corresponds to filling the first row of the table. Filling entries in subsequent rows requires two entries from the previous row: one from the same column and one from the column offset by the weight of the object. Thus, the computation of an arbitrary entry $F[i, j]$ requires $F[i-1, j]$ and $F[i-1, j-w_i]$. This is illustrated in Figure 11.4. Computing each entry takes constant time; the sequential run time of this algorithm is $\Theta(nc)$.

This formulation is a serial monadic formulation. The subproblems $F[i, x]$ are organized into n levels for $i = 1, 2, \ldots, n$. Computation of problems in level i depends only on the subproblems at level $i - 1$. Hence the formulation is serial. The formulation is monadic because each of the two alternate solutions of $F[i, x]$ depends on only one subproblem. Furthermore, dependencies between levels are sparse because a problem at one level depends only on two subproblems from previous level.

Consider a parallel formulation of this algorithm on a CREW PRAM with c processing elements labeled P_0 to P_{c-1}. Processing element P_{r-1} computes the r^{th} column of matrix F. When computing $F[j, r]$ during iteration j, processing element P_{r-1} requires the values $F[j-1, r]$ and $F[j-1, r-w_j]$. Processing element P_{r-1} can read any element of matrix F in constant time, so computing $F[j, r]$ also requires constant time. Therefore, each iteration takes constant time. Since there are n iterations, the parallel run time is $\Theta(n)$. The formulation uses c processing elements, hence its processor-time product is $\Theta(nc)$. Therefore, the algorithm is cost-optimal.

Let us now consider its formulation on a distributed memory machine with c-processing elements. Table F is distributed among the processing elements so that each processing element is responsible for one column. This is illustrated in Figure 11.4. Each processing element locally stores the weights and profits of all objects. 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 from another processing element. This corresponds to the circular w_j-shift operation described in Section **??**. The time taken by this circular shift operation on p processing elements is bounded by $(t_s + t_w m) \log p$ for a message of size m on a network with adequate bandwidth. Since the size of the message is one word and we have $p = c$, this time is given by $(t_s + t_w) \log c$. If the sum and maximization operations take time t_c, then each iteration takes time $t_c + (t_s + t_w) \log c$. Since there are n such iterations, the total time is given by $O(n \log c)$. The processor-time product for this formulation is $O(nc \log c)$; therefore, the algorithm is not cost-optimal.

Let us see what happens to this formulation as we increase the number of elements per

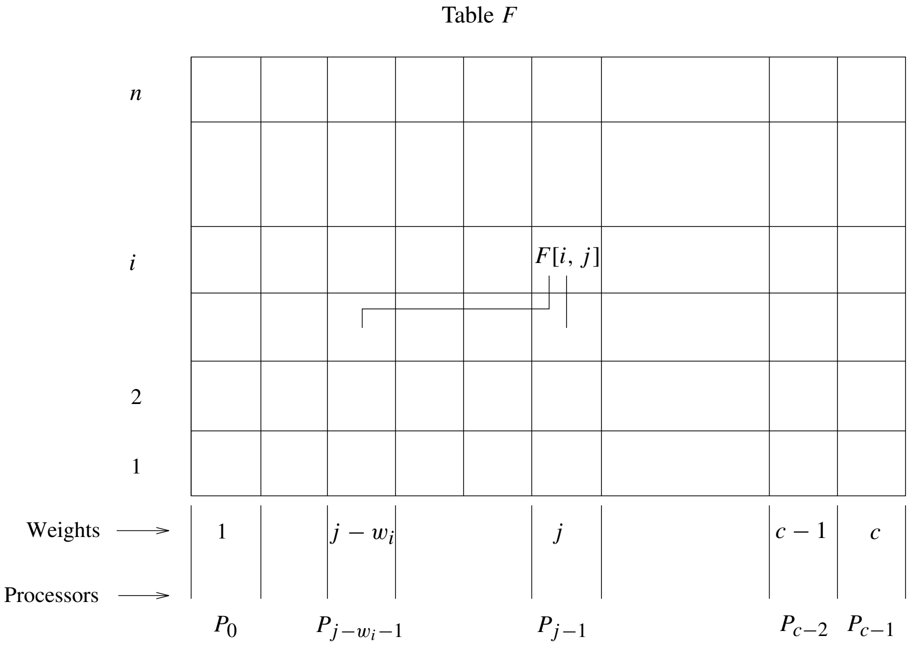

Figure 11.4 Computing entries of table F for the 0/1 knapsack problem. The computation of entry $F[i, j]$ requires communication with processing elements containing entries $F[i-1, j]$ and $F[i-1, j-w_i]$.

processor. Using p-processing elements, each processing element computes c/p elements of the table in each iteration. In the j^{th} iteration, processing element P_0 computes the values of elements $F[j, 1], \ldots, F[j, c/p]$, processing element P_1 computes values of elements $F[j, c/p+1], \ldots, F[j, 2c/p]$, and so on. Computing the value of $F[j, k]$, for any k, requires values $F[j-1, k]$ and $F[j-1, k-w_j]$. Required values of the F table can be fetched from remote processing elements by performing a circular shift. Depending on the values of w_j and p, the required nonlocal values may be available from one or two processing elements. Note that the total number of words communicated via these messages is c/p irrespective of whether they come from one or two processing elements. The time for this operation is at most $(2t_s + t_w c/p)$ assuming that c/p is large and the network has enough bandwidth (Section **??**). Since each processing element computes c/p such elements, the total time for each iteration is $t_c c/p + 2t_s + t_w c/p$. Therefore, the parallel run time of the algorithm for n iterations is $n(t_c c/p + 2t_s + t_w c/p)$. In asymptotic terms, this algorithm's parallel run time is $O(nc/p)$. Its processor-time product is $O(nc)$, which is cost-optimal.

There is an upper bound on the efficiency of this formulation because the amount of data that needs to be communicated is of the same order as the amount of computation. This upper bound is determined by the values of t_w and t_c (Problem 11.1).

11.2 Nonserial Monadic DP Formulations

The DP algorithm for determining the longest common subsequence of two given sequences can be formulated as a nonserial monadic DP formulation.

11.2.1 The Longest-Common-Subsequence Problem

Given a sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, a subsequence of A can be formed by deleting some entries from A. For example, $\langle a, b, z \rangle$ is a subsequence of $\langle c, a, d, b, r, z \rangle$, but $\langle a, c, z \rangle$ and $\langle a, d, l \rangle$ are not. The *l*ongest-common-subsequence (LCS) problem can be stated as follows. Given two sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$, find the longest sequence that is a subsequence of both A and B. For example, if $A = \langle c, a, d, b, r, z \rangle$ and $B = \langle a, s, b, z \rangle$, the longest common subsequence of A and B is $\langle a, b, z \rangle$.

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 determine $F[n, m]$. The DP formulation for this problem expresses $F[i, j]$ in terms of $F[i-1, j-1]$, $F[i, j-1]$, and $F[i-1, j]$ as follows:

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

Given sequences A and B, consider two pointers pointing to the start of the sequences. If the entries pointed to by the two pointers are identical, then they form components of the longest common subsequence. Therefore, both pointers can be advanced to the next entry of the respective sequences and the length of the longest common subsequence can be incremented by one. If the entries are not identical then two situations arise: the longest common subsequence may be obtained from the longest subsequence of A and the sequence obtained by advancing the pointer to the next entry of B; or the longest subsequence may be obtained from the longest subsequence of B and the sequence obtained by advancing the pointer to the next entry of A. Since we want to determine the longest subsequence, the maximum of these two must be selected.

The sequential implementation of this DP formulation computes the values in table F in row-major order. Since there is a constant amount of computation at each entry in the table, the overall complexity of this algorithm is $\Theta(nm)$. This DP formulation is nonserial monadic, as illustrated in Figure 11.5(a). 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 earlier. The formulation is monadic because a solution to any subproblem at a level is a function of only one of the solutions at preceding levels. (Note that, for the third case in Equation 11.5, both $F[i, j-1]$ and $F[i-1, j]$ are possible solutions to $F[i, j]$, and the optimal solution to $F[i, j]$ is the maximum of the two.) Figure 11.5 shows that this problem has a very regular structure.

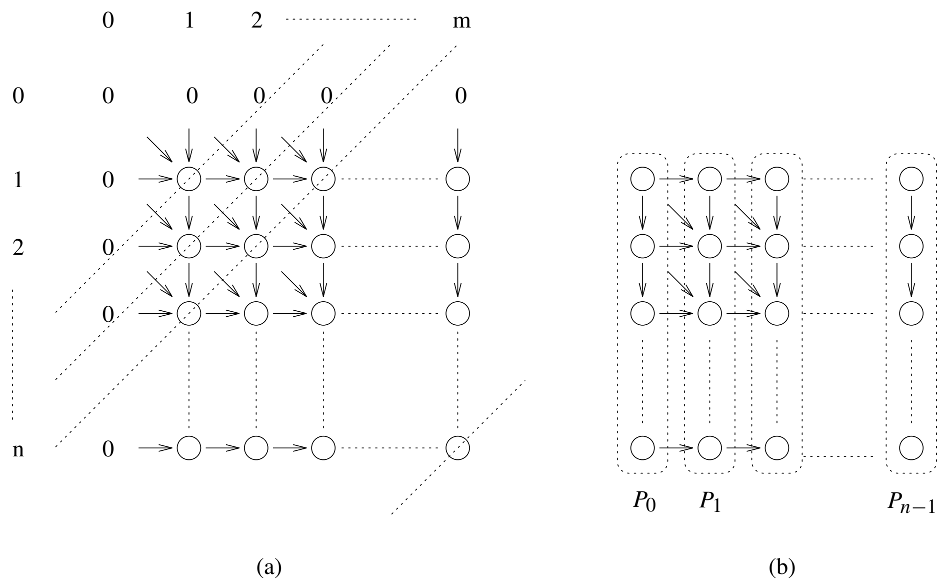

Figure 11.5 (a) Computing entries of table F for the longest-common-subsequence problem. Computation proceeds along the dotted diagonal lines. (b) Mapping elements of the table to processing elements.

Example 11.2 Computing LCS of two amino-acid sequences
Let us consider the LCS of two amino-acid sequences `H E A G A W G H E E` and `P A W H E A E`. 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 table of F entries for these two sequences is shown in Figure 11.6. The LCS of the two sequences, as determined by tracing back from the maximum score and enumerating all the matches, is `A W H E E`.

■

To simplify the discussion, we discuss parallel formulation only for the case in which $n = m$. Consider a parallel formulation of this algorithm on a CREW PRAM with n processing elements. Each processing element P_i computes the i^{th} column of table F. Table entries are computed in a diagonal sweep from the top-left to the bottom-right corner. Since there are n processing elements, and each processing element can access any entry in table F, the elements of each diagonal are computed in constant time (the diagonal can contain at most n elements). Since there are $2n - 1$ such diagonals, the algorithm requires $\Theta(n)$ iterations. Thus, the parallel run time is $\Theta(n)$. The algorithm is cost-optimal, since its $\Theta(n^2)$ processor-time product equals the sequential complexity.

This algorithm can be adapted to run on a logical linear array of n processing elements by distributing table F among different processing elements. Note that this logical topol-

		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1	1	1	1
W	0	0	0	1	1	1	2	2	2	2	2
H	0	1	1	1	1	1	2	2	3	3	3
E	0	1	2	2	2	2	2	2	3	4	4
A	0	1	2	3	3	3	3	3	3	4	4
E	0	1	2	3	3	3	3	3	3	4	5

Figure 11.6 The F table for computing the LCS of sequences `H E A G A W G H E E` and `P A W H E A E`.

ogy can be mapped to a variety of physical architectures using embedding techniques in Section **??**. Processing element P_i stores the $(i+1)$th column of the table. Entries in table F are assigned to processing elements as illustrated in Figure 11.5(b). When computing the value of $F[i, j]$, processing element P_{j-1} may need either the value of $F[i-1, j-1]$ or the value of $F[i, j-1]$ from the processing element to its left. It takes time $t_s + t_w$ to communicate a single word from a neighboring processing element. To compute each entry in the table, a processing element needs a single value from its immediate neighbor, followed by the actual computation, which takes time t_c. Since each processing element computes a single entry on the diagonal, each iteration takes time $(t_s + t_w + t_c)$. The algorithm makes $(2n-1)$ diagonal sweeps (iterations) across the table; thus, the total parallel run time is

$$T_P = (2n-1)(t_s + t_w + t_c).$$

Since the sequential run time is $n^2 t_c$, the efficiency of this algorithm is

$$E = \frac{n^2 t_c}{n(2n-1)(t_s + t_w + t_c)}.$$

A careful examination of this expression reveals that it is not possible to obtain efficiencies above a certain threshold. To compute this threshold, assume it is possible to communicate values between processing elements instantaneously; that is, $t_s = t_w = 0$. In this case, the efficiency of the parallel algorithm is

$$E_{max} = \frac{1}{2 - 1/n}. \tag{11.5}$$

Thus, the efficiency is bounded above by 0.5. This upper bound holds even if multiple columns are mapped to a processing element. Higher efficiencies are possible using alternate mappings (Problem 11.3).

Note that the basic characteristic that allows efficient parallel formulations of this algorithm is that table F can be partitioned so computing each element requires data only from neighboring processing elements. In other words, the algorithm exhibits locality of data access.

11.3 Serial Polyadic DP Formulations

Floyd's algorithm for determining the shortest paths between all pairs of nodes in a graph can be reformulated as a serial polyadic DP formulation.

11.3.1 Floyd's All-Pairs Shortest-Paths Algorithm

Consider a weighted graph G, which consists of a set of nodes V and a set of edges E. An edge from node i to node j in E has a weight $c_{i,j}$. Floyd's algorithm determines the cost $d_{i,j}$ of the shortest path between each pair of nodes (i, j) in V (Section **??**). The cost of a path is the sum of the weights of the edges in the path.

Let $d^k_{i,j}$ be the minimum cost of a path from node i to node j, using only nodes $v_0, v_1, \ldots, v_{k-1}$. The functional equation of the DP formulation for this problem is

$$d^k_{i,j} = \begin{cases} c_{i,j} & k = 0 \\ \min \{d^{k-1}_{i,j}, (d^{k-1}_{i,k} + d^{k-1}_{k,j})\} & 0 \le k \le n-1 \end{cases}. \tag{11.6}$$

Since $d^n_{i,j}$ is the shortest path from node i to node j using all n nodes, it is also the cost of the overall shortest path between nodes i and j. The sequential formulation of this algorithm requires n iterations, and each iteration requires time $\Theta(n^2)$. Thus, the overall run time of the sequential algorithm is $\Theta(n^3)$.

Equation 11.6 is a serial polyadic formulation. Nodes $d^k_{i,j}$ can be partitioned into n levels, one for each value of k. Elements at level $k+1$ depend only on elements at level k. Hence, the formulation is serial. The formulation is polyadic since one of the solutions to $d^k_{i,j}$ requires a composition of solutions to two subproblems $d^{k-1}_{i,k}$ and $d^{k-1}_{k,j}$ from the previous level. Furthermore, the dependencies between levels are sparse because the computation of each element in $d^{k+1}_{i,j}$ requires only three results from the preceding level (out of n^2).

A simple CREW PRAM formulation of this algorithm uses n^2 processing elements. Processing elements are organized into a logical two-dimensional array in which processing element $P_{i,j}$ computes the value of $d^k_{i,j}$ for $k = 1, 2, \ldots, n$. In each iteration k, processing element $P_{i,j}$ requires the values $d^{k-1}_{i,j}$, $d^{k-1}_{i,k}$, and $d^{k-1}_{k,j}$. Given these values, it computes the value of $d^k_{i,j}$ in constant time. Therefore, the PRAM formulation has a parallel run time of $\Theta(n)$. This formulation is cost-optimal because its processor-time product is the same as the sequential run time of $\Theta(n^3)$. This algorithm can be adapted to various practical architectures to yield efficient parallel formulations (Section **??**).

As with serial monadic formulations, data locality is of prime importance in serial polyadic formulations since many such formulations have sparse connectivity between levels.

11.4 Nonserial Polyadic DP Formulations

In nonserial polyadic DP formulations, in addition to processing subproblems at a level in parallel, computation can also be pipelined to increase efficiency. We illustrate this with the optimal matrix-parenthesization problem.

11.4.1 The Optimal Matrix-Parenthesization Problem

Consider the problem of multiplying n matrices, $A_1, A_2, \ldots, A_n$, where each A_i is a matrix with r_{i-1} rows and r_i columns. The order in which the matrices are multiplied has a significant impact on the total number of operations required to evaluate the product.

Example 11.3 Optimal matrix parenthesization
Consider three matrices A_1, A_2, and A_3 of dimensions 10×20, 20×30, and 30×40, 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 10×30. 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. Clearly, the first parenthesization is desirable. ■

The objective of the parenthesization problem is to determine a parenthesization that minimizes the number of operations. Enumerating all possible parenthesizations is not feasible since there are exponentially many of them.

Let $C[i, j]$ be the optimal cost of multiplying the matrices $A_i, \ldots, A_j$. This 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 $r_{i-1} r_k r_j$. Hence, the cost of the parenthesization $(A_i, A_{i+1}, \ldots, A_k)(A_{k+1}, \ldots, A_j)$ is given by $C[i, k] + C[k+1, j] + r_{i-1} r_k r_j$. This gives rise to the following recurrence relation for the parenthesization problem:

$$C[i, j] = \begin{cases} \min\limits_{i \le k < j} \{C[i, k] + C[k+1, j] + r_{i-1} r_k r_j\} & 1 \le i < j \le n \\ 0 & j = i, 0 < i \le n \end{cases} \tag{11.7}$$

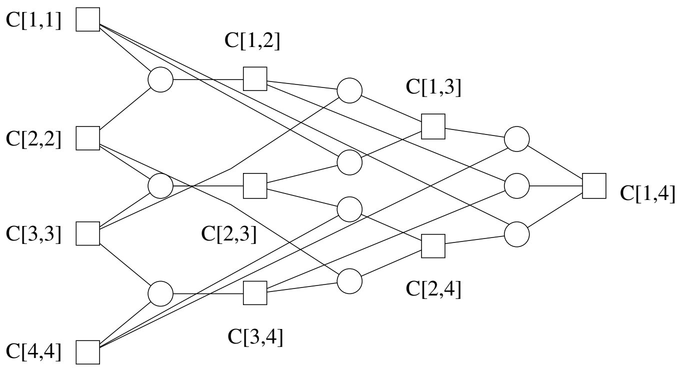

Figure 11.7 A nonserial polyadic DP formulation for finding an optimal matrix parenthesization for a chain of four matrices. A square node represents the optimal cost of multiplying a matrix chain. A circle node represents a possible parenthesization.

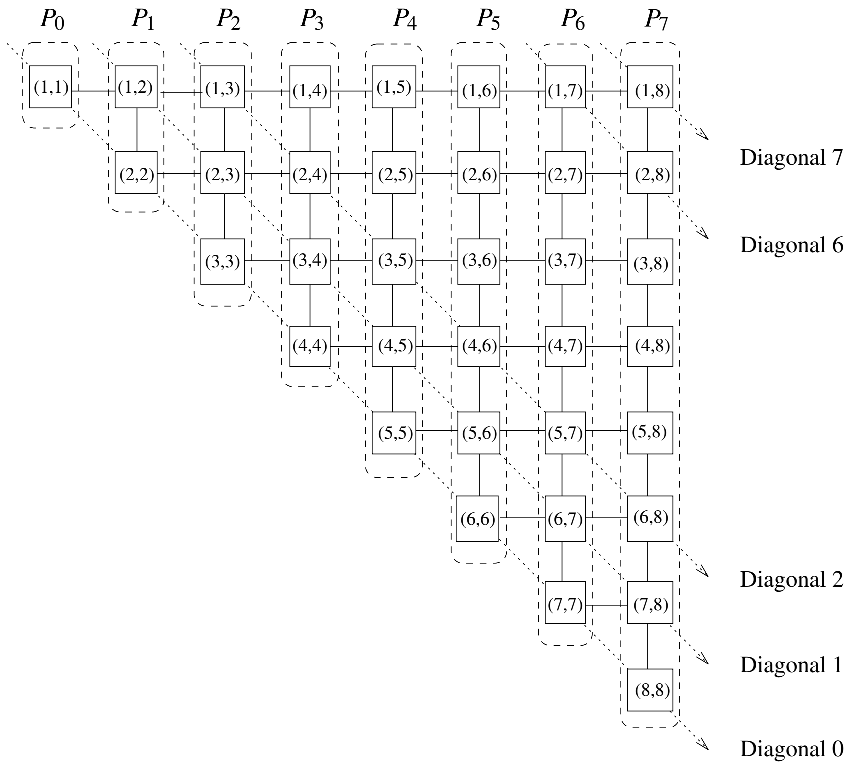

Figure 11.8 The diagonal order of computation for the optimal matrix-parenthesization problem.

Given Equation 11.7, the problem reduces to finding the value of $C[1, n]$. The composition of costs of matrix chains is shown in Figure 11.7. Equation 11.7 can be solved if we use a bottom-up approach for constructing the table C that stores the values $C[i, j]$. The algorithm fills table C in an order corresponding to solving the parenthesization problem on matrix chains of increasing length. Visualize this by thinking of filling in the table diagonally (Figure 11.8). Entries in diagonal l corresponds to the cost of multiplying matrix chains of length $l+1$. From Equation 11.7, we can see that the value of $C[i, j]$ is computed as $\min\{C[i, k]+C[k+1, j]+r_{i-1}r_k r_j\}$, where k can take values from i to $j-1$. Therefore, 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 $(j-i)t_c$. Thus, each entry in diagonal l can be computed in time lt_c.

In computing the cost of the optimal parenthesization sequence, the algorithm computes $(n-1)$ chains of length two. This takes time $(n-1)t_c$. Similarly, 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. This takes time $(n-1)t_c$. Thus, the sequential run time of this algorithm is

$$
\begin{aligned}
T_S &= (n-1)t_c + (n-2)2t_c + \cdots + 1(n-1)t_c, \\
&= \sum_{i=1}^{n-1}(n-i)i t_c, \\
&\simeq (n^3/6)t_c. \qquad (11.8)
\end{aligned}
$$

The sequential complexity of the algorithm is $\Theta(n^3)$.

Consider the parallel formulation of this algorithm on a logical ring of n processing elements. In step l, each processing element computes a single element belonging to the l^{th} diagonal. Processing element P_i computes the $(i+1)$th column of Table C. Figure 11.8 illustrates the partitioning of the table among different processing elements. After computing the assigned value of the element in table C, each processing element sends its value to all other processing elements using an all-to-all broadcast (Section **??**). Therefore, the assigned value in the next iteration can be computed locally. Computing an entry in table C during iteration l takes time lt_c because it corresponds to the cost of multiplying a chain of length $l+1$. An all-to-all broadcast of a single word on n processing elements takes time $t_s \log n + t_w(n-1)$ (Section **??**). The total time required to compute the entries along diagonal l is $lt_c + t_s \log n + t_w(n-1)$. The parallel run time is the sum of the time taken over computation of $n-1$ diagonals.

$$
\begin{aligned}
T_P &= \sum_{l=1}^{n-1}(lt_c + t_s \log n + t_w(n-1)), \\
&= \frac{(n-1)(n)}{2}t_c + t_s(n-1)\log n + t_w(n-1)^2.
\end{aligned}
$$

The parallel run time of this algorithm is $\Theta(n^2)$. Since the processor-time product is $\Theta(n^3)$, which is the same as the sequential complexity, this algorithm is cost-optimal.

When using p processing elements ($1 \leq p \leq n$) organized in a logical ring, if there are n nodes in a diagonal, each processing element stores n/p nodes. Each processing element computes the cost $C[i, j]$ of the entries assigned to it. After computation, an all-to-all broadcast sends the solution costs of the subproblems for the most recently computed diagonal to all the other processing elements. Because each processing element has complete information about subproblem costs at preceding diagonals, no other communication is required. The time taken for all-to-all broadcast of n/p words is $t_s \log p + t_w n(p-1)/p \approx t_s \log p + t_w n$. The time to compute n/p entries of the table in the l^{th} diagonal is $l t_c n/p$. The parallel run time is

$$
\begin{aligned}
T_P &= \sum_{l=1}^{n-1} (l t_c n/p + t_s \log p + t_w n), \\
&= \frac{n^2(n-1)}{2p} t_c + t_s (n-1) \log p + t_w n(n-1).
\end{aligned}
$$

In order terms, $T_P = \Theta(n^3/p) + \Theta(n^2)$. Here, $\Theta(n^3/p)$ is the computation time, and $\Theta(n^2)$ the communication time. If n is sufficiently large with respect to p, communication time can be made an arbitrarily small fraction of computation time, yielding linear speedup.

This formulation can use at most $\Theta(n)$ processing elements to accomplish the task in time $\Theta(n^2)$. This time can be improved by pipelining the computation of the cost $C[i, j]$ on $n(n+1)/2$ processing elements. Each processing element computes a single entry $c(i, j)$ of matrix C. Pipelining works due to the nonserial nature of the problem. Computation of an entry on a diagonal t does not depend only on the entries on diagonal $t-1$ but also on all the earlier diagonals. Hence work on diagonal t can start even before work on diagonal $t-1$ is completed.

11.5 Summary and Discussion

This chapter provides a framework for deriving parallel algorithms that use dynamic programming. It identifies possible sources of parallelism, and indicates under what conditions they can be utilized effectively.

By representing computation as a graph, we identify three sources of parallelism. First, the computation of the cost of a single subproblem (a node in a level) can be parallelized. For example, for computing the shortest path in the multistage graph shown in Figure 11.3, node computation can be parallelized because the complexity of node computation is itself $\Theta(n)$. For many problems, however, node computation complexity is lower, limiting available parallelism.

Second, subproblems at each level can be solved in parallel. This provides a viable method for extracting parallelism from a large class of problems (including all the problems in this chapter).

The first two sources of parallelism are available to both serial and nonserial formulations. Nonserial formulations allow a third source of parallelism: pipelining of computations among different levels. Pipelining makes it possible to start solving a problem as soon as the subproblems it depends on are solved. This form of parallelism is used in the parenthesization problem.

Note that pipelining was also applied to the parallel formulation of Floyd's all-pairs shortest-paths algorithm in Section **??**. As discussed in Section 11.3, this algorithm corresponds to a serial DP formulation. The nature of pipelining in this algorithm is different from the one in nonserial DP formulation. In the pipelined version of Floyd's algorithm, computation in a stage is pipelined with the communication among earlier stages. If communication cost is zero (as in a PRAM), then Floyd's algorithm does not benefit from pipelining.

Throughout the chapter, we have seen the importance of data locality. If the solution to a problem requires results from other subproblems, the cost of communicating those results must be less than the cost of solving the problem. In some problems (the 0/1 knapsack problem, for example) the degree of locality is much smaller than in other problems such as the longest-common-subsequence problem and Floyd's all-pairs shortest-paths algorithm.

11.6 Bibliographic Remarks

Dynamic programming was originally presented by Bellman [2] for solving multistage decision problems. Various formal models have since been developed for DP [10, 15, 12]. Several textbooks and articles present sequential DP formulations of the longest-common-subsequence problem, the matrix chain multiplication problem, the 0/1 knapsack problem, and the shortest-path problem [4, 7, 16, 3].

Li and Wah [14, 20] show that monadic serial DP formulations can be solved in parallel on systolic arrays as matrix-vector products. They further present a more concurrent but non-cost-optimal formulation by formulating the problem as a matrix-matrix product. Ranka and Sahni [17] present a polyadic serial formulation for the string editing problem and derive a parallel formulation based on a checkerboard partitioning.

The DP formulation of a large class of optimization problems is similar to that of the optimal matrix-parenthesization problem. Some examples of these problems are optimal triangularization of polygons, optimal binary search trees [4], and CYK parsing [1]. The serial complexity of the standard DP formulation for all these problems is $\Theta(n^3)$. Several parallel formulations have been proposed by Ibarra *et al.* [9] that use $\Theta(n)$ processing elements on a hypercube and that solve the problem in time $\Theta(n^2)$. Guibas, Kung, and Thompson [6] present a systolic algorithm that uses $\Theta(n^2)$ processing cells and solves the problem in time $\Theta(n)$. Karypis and Kumar [11] analyze three distinct mappings of the systolic algorithm presented by Guibas *et al.* [6] and experimentally evaluate them by using the matrix-multiplication parenthesization problem. They show that a straightforward mapping of this algorithm to a mesh architecture has an upper bound on efficiency of $1/12$.

They also present a better mapping without this drawback, and show near-linear speedup on a mesh embedded into a 256-processor hypercube for the optimal matrix-parenthesization problem.

Many faster parallel algorithms for solving the parenthesization problem have been proposed, but they are not cost-optimal and are applicable only to theoretical models such as the PRAM. For example, a generalized method for parallelizing such programs is described by Valiant *et al.* [19] that leads directly to formulations that run in time $O(\log^2 n)$ on $O(n^9)$ processing elements. Rytter [18] uses the parallel pebble game on trees to reduce the number of processing elements to $O(n^6/\log n)$ for a CREW PRAM and $O(n^6)$ for a hypercube, yet solves this problem in time $O(\log^2 n)$. Huang *et al.* [8] present a similar algorithm for CREW PRAM models that run in time $O(\sqrt{n}\log n)$ on $O(n^{3.5}\log n)$ processing elements. DeMello *et al.* [5] use vectorized formulations of DP for the Cray to solve optimal control problems.

As we have seen, the serial polyadic formulation of the 0/1 knapsack problem is difficult to parallelize due to lack of communication locality. Lee *et al.* [13] use specific characteristics of the knapsack problem and derive a divide-and-conquer strategy for parallelizing the DP algorithm for the 0/1 knapsack problem on a MIMD message-passing computer (Problem 11.2). Lee *et al.* demonstrate experimentally that it is possible to obtain linear speedup for large instances of the problem on a hypercube.

Problems

11.1 Consider the parallel algorithm for solving the 0/1 knapsack problem in Section 11.1.2. Derive the speedup and efficiency for this algorithm. Show that the efficiency of this algorithm cannot be increased beyond a certain value by increasing the problem size for a fixed number of processing elements. What is the upper bound on efficiency for this formulation as a function of t_w and t_c?

11.2 **[13]** In the parallel formulation of the 0/1 knapsack problem presented in Section 11.1.2, the degree of concurrency is proportional to c, the knapsack capacity. Also this algorithm has limited data locality, as the amount of data to be communicated is of the same order of magnitude as the computation at each processing element. Lee *et al.* present another formulation in which the degree of concurrency is proportional to n, the number of weights. This formulation also has much more data locality. In this formulation, the set of weights is partitioned among processing elements. Each processing element computes the maximum profit it can achieve from its local weights for knapsacks of various sizes up to c. This information is expressed as lists that are merged to yield the global solution.

Compute the parallel run time, speedup, and efficiency of this formulation. Compare the performance of this algorithm with that in Section 11.1.2.

11.3 We noticed that the parallel formulation of the longest-common-subsequence problem has an upper bound of 0.5 on its efficiency. It is possible to use an alternate mapping to achieve higher efficiency for this problem. Derive a formulation that does not suffer from this upper bound, and give the run time of this formulation. ***H*int:** Consider the blocked-cyclic mapping discussed in Section **??**.

11.4 **[7]** The traveling salesman problem (TSP) is defined as follows: Given a set of cities and the distance between each pair of cities, determine a tour through all cities of minimum length. A tour of all cities is a trip visiting each city once and returning to the starting point. Its length is the sum of distances traveled.

This problem can be solved using a DP formulation. View the cities as vertices in a graph $G(V, E)$. Let the set of cities V be represented by $\{v_1, v_2, \ldots, v_n\}$ and let $S \subseteq \{v_2, v_3, \ldots, v_n\}$. Furthermore, let $c_{i,j}$ be the distance between cities i and j. If $f(S, k)$ represents the cost of starting at city v_1, passing through all the cities in set S, and terminating in city k, then the following recursive equations can be used to compute $f(S, k)$:

$$f(S,k) = \begin{cases} c_{1,k} & S = \{k\} \\ \min\limits_{m \epsilon S-\{k\}} \{f(S-\{k\}), m\} + c_{m,k}\} & S \neq \{k\} \end{cases} \tag{11.9}$$

Based on Equation 11.9, derive a parallel formulation. Compute the parallel run time and the speedup. Is this parallel formulation cost-optimal?

11.5 **[7]** Consider the problem of merging two sorted files containing $O(n)$ and $O(m)$ records. These files can be merged into a sorted file in time $O(m + n)$. Given r such files, the problem of merging them into a single file can be formulated as a sequence of merge operations performed on pairs of files. The overall cost of the merge operation is a function of the sequence in which they are merged. The optimal merge order can be formulated as a greedy problem and its parallel formulations derived using principles illustrated in this chapter.

Write down the recursive equations for this problem. Derive a parallel formulation for merging files using p processing elements. Compute the parallel run time and speedup, and determine whether your parallel formulation is cost-optimal.

11.6 **[7]** Consider the problem of designing a fault-tolerant circuit containing n devices connected in series, as shown in Figure 11.9(a). If the probability of failure of each of these devices is given by f_i, the overall probability of failure of the circuit is given by $1 - \prod(1 - f_i)$. Here, $\prod$ represents a product of specified terms. The reliability of this circuit can be improved by connecting multiple functional devices in parallel at each stage, as shown in Figure 11.9(b). If stage i in the circuit has r_i duplicate functional units, each with a probability of failure given by f_i, then the overall probability of failure of this stage is reduced to $r_i^{m_i}$ and the overall probability of failure of the circuit is given by $1 - \prod(1 - r_i^{m_i})$. In general, for physical reasons, the probability of failure at a particular level may not be $r_i^{m_i}$, but some function $\phi_i(r_i, m_i)$. The objective of the problem is to minimize the overall probability of failure of the circuit, $1 - \prod(1 - \phi_i(r_i, m_i))$.

Construction cost adds a new dimension to this problem. If each of the functional

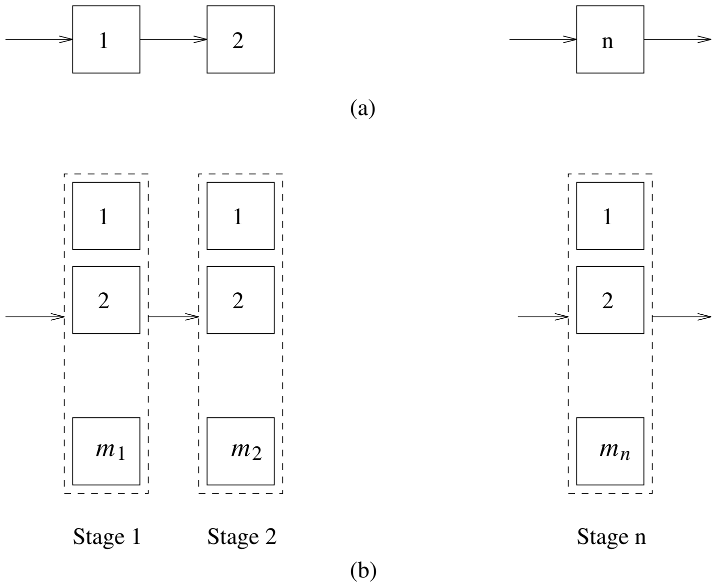

Figure 11.9 (a) n devices connected in a series within a circuit. (b) Each stage in the circuit now has m_i functional units. There are n such stages connected in the series.

units used at stage i costs c_i then due to cost constraints, the overall cost $\sum c_i m_i$ should be less than a fixed quantity c.

The problem can be formally defined as

$$\text{Minimize } 1 - \prod(1 - \phi_i(r_i, m_i)), \text{ such that } \sum c_i m_i < c$$

where $m_i > 0$ and $0 < i \leq n$.

Let $f_i(x)$ represent the reliability of a system with i stages of cost x. The optimal solution is given by $f_n(c)$. The recursive equation for $f_i(x)$ is as follows:

$$f_i(x) = \begin{cases} 1 & i = 0 \\ \max\limits_{i \leq m_i \leq u_i} \{\phi_i(m_i) f_{i-1}(c - c_i m_i)\} & i \geq 1 \end{cases} \tag{11.10}$$

Classify this formulation into one of the four DP categories, and derive a parallel formulation for this algorithm. Determine its parallel run time, speedup, and isoefficiency function.

11.7 **[4]** Consider the simplified optimal polygon-triangulation problem. This problem can be defined as follows. Given a simple polygon, break the polygon into a set of triangles by connecting nodes of the polygon with chords. This process is illustrated in Figure 11.10. The cost of constructing a triangle with nodes v_i, v_j, and v_k is defined by a function $f(v_i, v_j, v_k)$. For this problem, let the cost be the total length of the edges of the triangle (using Euclidean distance). The optimal polygon-triangulation problem breaks up a polygon into a set of triangles such that

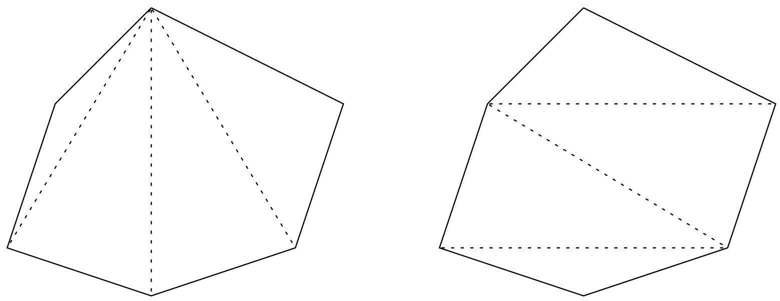

Figure 11.10 Two possible triangulations of a regular polygon.

the total length of each triangle (the sum of the individual lengths) is minimized. Give a DP formulation for this problem. Classify it into one of the four categories and derive a parallel formulation for p processing elements. Determine its parallel run time, speedup, and isoefficiency function.
Hint: This problem is similar to the optimal matrix-parenthesization problem.

Bibliography

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling: Volume 1, Parsing*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[2] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[3] K. Brown. Dynamic programming in computer science. Technical Report CMU-CS-79-106, Carnegie Mellon University, Pittsburgh, PA, 1979.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, New York, NY, 1990.

[5] J. D. DeMello, J. L. Calvet, and J. M. Garcia. Vectorization and multitasking of dynamic programming in control: experiments on a CRAY-2. *Parallel Computing*, 13:261–269, 1990.

[6] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI Implementation of Combinatorial Algorithms. In *Proceedings of Conference on Very Large Scale Integration, California Institute of Technology*, pages 509–525, 1979.

[7] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, Maryland, 1978.

[8] S. H. S. Huang, H. Liu, and V. Vishwanathan. A sub-linear parallel algorithm for some dynamic programming problems. In *Proceedings of 1990 International Conference on Parallel Processing*, pages III–261–III–264, 1990.

[9] Oscar H. Ibarra, T. C. Pong, and Stephen M. Sohn. Parallel recognition and parsing on the hypercube. *IEEE Transactions on Computers*, 40(6):764–770, June 1991.

[10] R. M. Karp and M. H. Held. Finite state processes and dynamic programming. *SIAM Journal of Applied Math*, 15:693–718, 1967.

[11] George Karypis and Vipin Kumar. Efficient Parallel Mappings of a Dynamic Programming Algorithm. Technical Report 92–59, University of Minnesota, Computer Science Departement, October 1992.

[12] Vipin Kumar and Laveen Kanal. The cdp: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In Laveen Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence*. Springer-Verlag, New York, 1988.

[13] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing*, (5):438–456, 1988.

[14] Guo-Jie Li and Benjamin W. Wah. Parallel processing of serial dynamic programming problems. In *Proceedings of COMPSAC 85*, pages 81–89, 1985.

[15] A. Martelli and U. Montanari. From dynamic programming to search algorithms with functional costs. In *Proceedings of the International Joint Conference on Artifi cial Intelligence*, pages 345–349, 1973.

[16] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[17] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*. Springer-Verlag, New York, 1990.

[18] W. Rytter. Efficient parallel computations for dynamic programming. *Theoretical Computer Science*, 59:297–307, 1988.

[19] L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM Journal of Computing*, 12(4):641–644, 1983.

[20] B. W. Wah and G.-J. Li. Systolic processing for dynamic programming problems. *Circuits, Systems, and Signal Processing*, 7(2):119–149, 1988.