CIRCUITS SYSTEMS SIGNAL PROCESS
VOL. 7, NO. 2, 1988

SYSTOLIC PROCESSING FOR DYNAMIC PROGRAMMING PROBLEMS*

Benjamin W. Wah[1] *and Guo-jie Li*[2]

Abstract. In this paper we investigate systolic processing for problems formulated in dynamic programming. These problems are classified as monadic-serial, polyadic-serial, monadic-nonserial, and polyadic-nonserial. Problems in serial formulations can be implemented easily in systolic arrays; however, nonserial problems may have to be transformed into a serial one before an efficient implementation can be found. A monadic-serial dynamic programming problem can be solved as the search of an optimal path in a multistage graph and can be computed as a string of matrix multiplications. Three efficient systolic-array designs are presented. A polyadic-serial dynamic programming problem can be solved by either a divide-and-conquer algorithm or the search of optimal solutions in a serial AND/OR-graph. We have evaluated the asymptotically optimal architecture for divide-and-conquer algorithms and have developed efficient methods of mapping a regular AND/OR-graph into systolic arrays. Cases are studied for transforming a problem in a nonserial formulation into a serial one.

1. Introduction

Dynamic programming (DP) is a powerful optimization methodology that is widely applicable to a large number of areas including optimal control, industrial engineering, economics, and artificial intelligence [3], [5], [9], [23], [29]. Many practical problems involving a sequence of interrelated decisions can be solved efficiently by DP. Bellman has characterized DP through the *Principle of Optimality*, which states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence

* Received February 2, 1987; revised April 14, 1987. This research was supported by National Science Foundation Grant DMC 85-19649 and Joint Services Electronics Program Contract N000014-84-C-0149.

[1] Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, USA.

[2] Institute of Computer Technology, Academia Sinica, Beijing, People's Republic of China.

with regard to the state resulting from the first decision [3]. Subsequently, numerous efforts have been devoted to the rigorous mathematical framework and effective evaluation of DP problems [14], [12], [29].

In general, DP is an approach that yields a transformation of the problem into a more suitable form for optimization but is not an algorithm for optimizing the objective function. Moreover, DP can be interpreted differently depending on the computational approach, and efficient implementations are based on the corresponding representations. Tradeoffs must be performed on the efficiency and costs of implementation of the alternate representations.

Bellman, Dreyfus, White, and many others viewed DP as a multistage optimization technique, that is, reducing a single N-dimensional problem into a sequence of N one-dimensional problems [3], [29]. The decisions that transform an initial state into a final state must be ordered in terms of stages, and functional equations relate state values in successive stages. The use of monotone sequential processes has been proved by Karp and Held to correspond naturally to DP [14] and has been further developed by Ibaraki [12] and Kumar [16]. On the other hand, Gensi and Montanari have shown that formulating a DP problem in terms of polyadic functional equations is equivalent to searching a minimum-cost solution tree in an AND/OR-graph with monotone cost function [10]. DP can also be formulated as a special case of the branch-and-bound algorithm, which is a general top-down OR-tree search procedure with dominance tests [22], [13], [18]. Lastly, nonserial DP has been shown to be optimal among all nonoverlapping comparison algorithms [5], [25].

Although DP has long been recognized as a powerful approach to solving a wide spectrum of optimization problems, its applicability has been somewhat limited due to the large computational requirements. Recent advances in very-large-scale integration (VLSI) and multiprocessor technologies have provided feasible means of implementation. Casti *et al.* have studied parallelism in DP [7]. Guibas *et al.* have proposed a VLSI algorithm for solving the optimal parenthesization problem [11]. Linear pipelines for DP have been described recently [27]. Clarke and Dyer have designed a systolic array for curve and line detection in terms of nonserial DP [9]. Wah *et al.* have proposed parallel processing for branch-and-bound algorithms with dominance tests [28]. Chen has studied a synthesis method for transforming a class of DP problems into VLSI architectures [8]. However, these studies were directed toward the implementation of a few special cases of DP formulations.

In this paper we classify DP problems into monadic-serial, polyadic-serial, monadic-nonserial, and polyadic-nonserial representations. Potential parallelism and the corresponding systolic architectures are investigated for each class. DP problems can be solved as the search of an optimal path in a multistage graph or as the search for an optimal solution in an AND/OR-

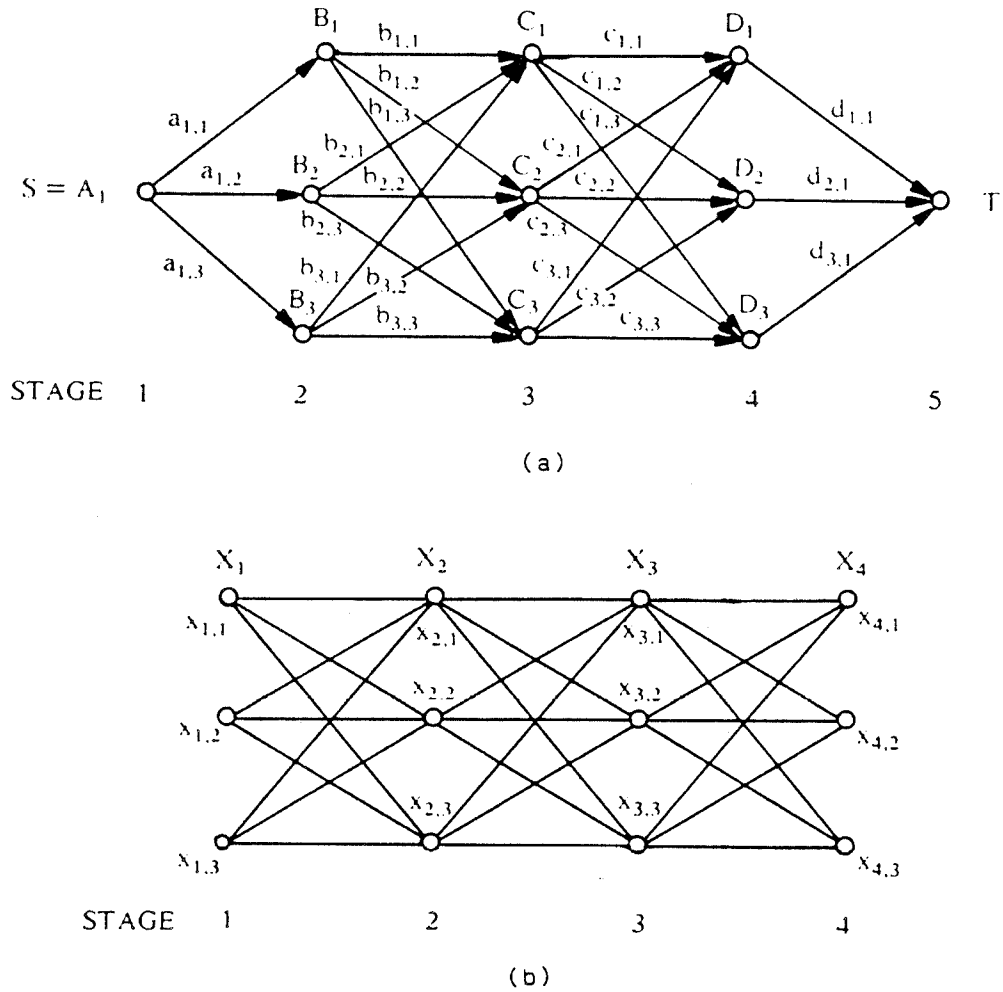

Figure 1. Examples of multistage graphs. (a) A multistage graph with one source and one sink. (b) A multistage graph with multiple sources and sinks.

graph. We will adopt the graph search as a paradigm to illustrate the various approaches of DP. To take advantage of the regular and limited interconnections of systolic arrays, the graph should have a regular structure. For DP problems in serial formulations, the corresponding graph representations are serial; however, for nonserial problems, they must be converted into serial formulations before efficient implementations can be found. A type of regular graphs of special interest is the multistage graph in which nodes are decomposed into stages, and nodes in one stage are connected to nodes in adjacent stages only. Figure 1 depicts two examples of multistage graphs.

2. Classification of dynamic programming formulations

A DP formulation is represented in a recursive functional equation whose left-hand side identifies a function name and whose right-hand side is an

expression involving the maximization (or minimization) of values of some cost functions. Note that the cost functions are neither restricted to be monadic nor additive; however, they must be monotone in order for the Principle of Optimality to hold. DP formulations were traditionally classified according to either the form of the functional equations [21] or the nature of recursion [5]. Both attributes are important in designing a systolic array to implement the recurrences, hence, we can classify these recurrences into four classes as follows.

2.1. *Monadic versus polyadic formulations*

A DP formulation is called *monadic* if its cost function involves only one recursive term, otherwise it is called *polyadic*. The distinction is illustrated by an example in finding the minimum-cost path in a multistage graph. For a multistage graph, let $c_{i,j}$ be the cost of an edge. The cost of a path from source, s, to sink, t, is the sum of costs on the edges of the path. Define $f_1(i)$ as the minimum cost of a path from i to t. Thus, the cost of a path from i to t via a neighbor j is $c_{i,j}+f_1(j)$. To find $f_1(i)$, paths through all possible neighbors must be compared. Hence,

$$f_1(i) = \min_j [c_{i,j} + f_1(j)]. \tag{1}$$

This equation is termed a *forward functional equation*. Similarly, if $f_2(i)$ is defined as the minimum cost of a path from s to i, then the functional equation becomes

$$f_2(i) = \min_j [f_2(j) + c_{j,i}]. \tag{2}$$

This equation is termed a *backward functional equation*. The formulations in equations (1) and (2) are *monadic* since each cost function involves one recursive term only.

Equations (1) and (2) can be generalized to find the optimal path from any vertex i to any other vertex j. This is equivalent to finding an intermediate vertex k such that the shortest paths from i to k and from k to j constitute the shortest path from i to j. The functional equation is

$$f_3(i, j) = \min_k [f_3(i, k) + f_3(k j)], \tag{3}$$

where $f_3(i, j)$ is the minimum cost of a path from i to j. This cost function is *polyadic* because it involves more than one recursive term. Examples of problems expressed in polyadic formulations include finding the optimal binary search tree and computing the minimum-cost order of multiplying a string of matrices.

For polyadic DP formulations, Bellman's Principle of Optimality must be generalized to include the statement that "all subsequences of an optimal policy are also optimal" [12]. For instance, according to equation (3), if it

is found that the minimum-cost path from i to j passes through k, then the subpath from i to k of this optimal path must be optimal over all subpaths from i to k; so is the subpath from k to j.

2.2. *Serial versus nonserial formulations*

The distinction between serial and nonserial formulations is based on both the form of their objective functions and the nature of recursion. From the objective function, an optimization problem is said to be serial if all functional terms of its objective function share one variable with its predecessor functional term (except for the first functional term) and another one with its successor functional term (except for the last functional term); otherwise, it is said to be nonserial. In an interaction graph to represent the problem, vertices stand for variables, and an edge exists between two vertices if and only if the two variables belong to a functional term of the objective function [5]. It is obvious that a serial optimization problem has a corresponding interaction graph with a serial structure.

An example of a serial optimization problem is depicted in Figure 1(b). In this multistage graph each stage, $\mathbf{X}_i$, $1 \le i \le N = 4$, stands for a discrete variable, and node x_{i,j_i} stands for the jth value taken by variable $\mathbf{X}_i$. Bold characters are used to denote vectors and matrices, and variables here can be considered as vectors of defined values. If the cost of edge $(x_{i,j_i}, x_{i+1,j_{i+1}})$ is $g_i(x_{i,j_i}, x_{i+1,j_{i+1}})$, then the minimum-cost path from any node in stage 1 to any node in stage N is

$$\min_{\mathbf{X}} f(\mathbf{X}) = \min_{\mathbf{X}} \sum_{i=1}^{N-1} g_i(\mathbf{X}_i, \mathbf{X}_{i+1}), \tag{4}$$

where $\mathbf{X}$ is the set of discrete variables $\{\mathbf{X}_i, \ldots, \mathbf{X}_N\}$. In equation (4) every functional term of the objective function has two variables that only interact with variables in two other functional terms, and all the interactions are in a serial fashion. For instance, $g_i(\mathbf{X}_i, \mathbf{X}_{i+1})$ interacts only with the functional terms $g_{i-1}(\mathbf{X}_{i-1}, \mathbf{X}_i)$ through $\mathbf{X}_i$ and $g_{i+1}(\mathbf{X}_{i+1}, \mathbf{X}_{i+2})$ through $\mathbf{X}_{i+1}$. As a result, equation (4) is a serial optimization problem.

Many practical DP problems can be represented in a serial formulation. For a traffic-control problem, $\mathbf{X}_i$ can be the possible times for the traffic light to be in state i, and the cost on an edge of the graph representation is the difference in timings. For a circuit-design problem, $\mathbf{X}_i$ can be the possible voltages at point i, and the cost of an edge of the graph representation may be the corresponding power dissipation. For a fluid-flow problem, $\mathbf{X}_i$ can be the possible pressure values in the ith pump, and function f may be the flow rate for a given pressure. For a scheduling problem, $\mathbf{X}_i$ can be the possible task service times for the ith task, and the edge cost reflects the delay. Note that the optimal-path problem in multistage graphs is a special case of serial optimization problems.

In contrast, the objective function of a general *nonserial optimization problem* has the following form:

$$f(\mathbf{X}) = \Phi_{i=1}^{N} g_i(\mathbf{X}^i), \tag{5}$$

where $\mathbf{X} = \{\mathbf{X}_1, \ldots, \mathbf{X}_N\}$ is a set of discrete variables, $\mathbf{X}^i \subseteq \mathbf{X}$, and Φ is a monotone function relating the g_i's together. For example, the following equation is a nonserial optimization problem:

$$\min_{\mathbf{X}}\{g_1(\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_4) + g_2(\mathbf{X}_3, \mathbf{X}_4) + g_3(\mathbf{X}_2, \mathbf{X}_5)\},$$

where $\mathbf{X} = \{\mathbf{X}_1, \ldots, \mathbf{X}_5\}$.

From the viewpoint of recursion, a DP problem can be represented as a folded AND/OR-tree (or AND/OR-graph) in which the nodes are classified into levels or stages [21]. An AND-node represents a (sub)problem that is solved only if all its children have been solved, whereas an OR-node represents a (sub)problem that is solved if any one of its children can be solved. If this AND/OR-graph has a serial structure such that arcs only exist between nodes in adjacent levels, then the corresponding DP problem has a serial formulation. For nonserial DP problems, the dependency between states is not restricted to successive stages but may exist between states in arbitrary stages. In the corresponding AND/OR-graphs the arcs are not restricted to successive levels, but may run between any two arbitrary levels.

As an example, consider the problem of finding the optimal order of multiplying a string of four matrices:

$$M = M_1 \times M_2 \times M_3 \times M_4,$$

where M_i, $1 \le i \le 4$, is a matrix with r_{i-1} rows and r_i columns. Let $m_{i,j}$ be the minimum cost of computing $M_i \times \cdots \times M_j$. Clearly,

$$m_{i,j} = \begin{cases} 0 & \text{if } j = i, \\ \min\limits_{i \le k < j} (m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j) & \text{if } j > i. \end{cases} \tag{6}$$

The solution to be found is $m_{1,4}$. This formulation is polyadic-nonserial and can be represented as the search of an AND/OR-graph in Figure 2, where AND-nodes denote multiplications and OR-nodes denote comparisons. In Figure 2 the topmost node represents the original problem of multiplying four matrices. This can be achieved in three ways:

(1) $(M_1 \times M_2 \times M_3) \times M_4$;

(2) $(M_1 \times M_2) \times (M_3 \times M_4)$; or

(3) $M_1 \times (M_2 \times M_3 \times M_4)$.

These three alternatives are represented by the three AND-nodes in the second level. Note that the first AND-node in the second level, which denotes $m_{1,3} \cdot m_{4,4}$, is connected to the node representing $m_{4,4}$ in the

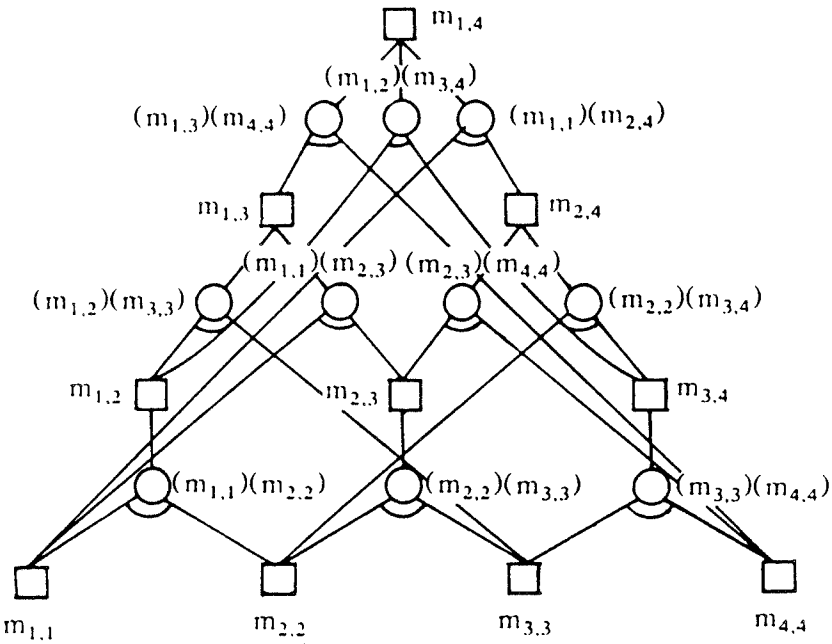

Figure 2. An AND/OR-graph representation of finding the optimal order of multiplying a string of matrices.

bottommost level. Similarly, the third node in the second level is connected to the node representing $m_{1,1}$ in the bottommost level. Regardless of the rearrangements of nodes in this graph, it is impossible to have arcs that only connect nodes in adjacent levels, hence, the formulation in equation (6) is polyadic-nonserial.

We have classified DP problems in terms of their recursive functional equations and objective functions. Monadic and polyadic DP formulations are distinct approaches to representing various optimization problems, while serial and nonserial optimization problems are problems solvable by the corresponding DP formulations.

3. Systolic arrays for monadic-serial DP formulations

Monadic-serial DP problems can be conveniently solved as the multiplication of a string of matrices. In this section three efficient systolic designs are presented. The proposed designs do not exploit all potential parallelism of solving a given problem, especially when the number of stages is large. Other parallel designs using different formulations may allow a higher degree of parallelism and will be discussed later.

3.1. *Solving a monadic-serial DP problem as multiplying a string of matrices*

We have shown that the search for a solution of a problem in a monadic-serial DP formulation can be viewed as finding a path in a multistage graph.

For the multistage graph in Figure 1(a) and from equation (2), $f(C_1)$, the minimum cost from C_1 to t, is

$$f(C_1) = \min\{c_{1,1} + d_{1,1}, c_{1,2} + d_{2,1}, c_{1,3} + d_{3,1}\}. \tag{7}$$

$f(C_2)$ and $f(C_3)$ are obtained similarly.

Equation (7) is similar to an inner-product operation. If we define matrix multiplication in terms of a closed semiring $(R, \text{MIN}, +, +\infty, 0)$, in which "MIN" corresponds to addition and "+" corresponds to multiplication in conventional matrix multiplications [1], equation (7) becomes

$$\mathbf{f(C)} = \mathbf{C} \cdot \mathbf{D} = \begin{bmatrix} f(C_1) \\ f(C_2) \\ f(C_3) \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \cdot \begin{bmatrix} d_{1,1} \\ d_{2,1} \\ d_{3,1} \end{bmatrix}. \tag{8a}$$

Likewise, we have

$$\mathbf{f(B)} = \mathbf{B} \cdot (\mathbf{C} \cdot \mathbf{D}), \tag{8b}$$

$$\mathbf{f(A)} = \mathbf{A} \cdot (\mathbf{B} \cdot (\mathbf{C} \cdot \mathbf{D})). \tag{8c}$$

Thus, solving the multistage-graph problem with a forward monadic DP formulation is equivalent to multiplying a string of matrices. The order of multiplications is reversed in backward monadic DP formulations.

For a multistage graph with N stages and m vertices in each stage, the computational complexity is $O(m^2N)$. For single-source, single-sink problems, the first and last matrices degenerate into row and column vectors, respectively.

3.2. *Systolic arrays for multiplying a string of matrices*

Three linear systolic arrays for evaluating problems in monadic-serial DP formulations are described in this section. The following scheme is based on a combination of two methods of multiplying a two-dimensional matrix with a vector, one of which was discussed by Kung [17]. Figure 3(a) depicts a scheme for computing $(\mathbf{A} \cdot (\mathbf{B} \cdot (\mathbf{C} \cdot \mathbf{D})))$ for the multistage graph in Figure 1(a). An *iteration* is defined as a shift-multiply-accumulate operation with respect to the time at which a row or column of the input matrix enters a given processor, and the iteration numbers are indicated in Figure 3(a). Note that the same iteration number is carried out at different times in different processors. In the first three iterations, $\mathbf{C} \cdot \mathbf{D}$ is evaluated. The control signal FIRST is one; $\mathbf{D}$, the input vector, is serially shifted into the systolic array; and the result vector, $\{f(C_i), i = 1, 2, 3\}$, remains stationary. At the end of the third iteration, FIRST is set to zero. In the following three iterations, $\mathbf{B} \cdot (\mathbf{C} \cdot \mathbf{D})$ is computed. Note that matrix $\mathbf{B}$ is transposed, and the ith column of matrix $\mathbf{B}$ is fed into P_i. The input vector, $\{f(C_i), i = 1, 2, 3\}$,

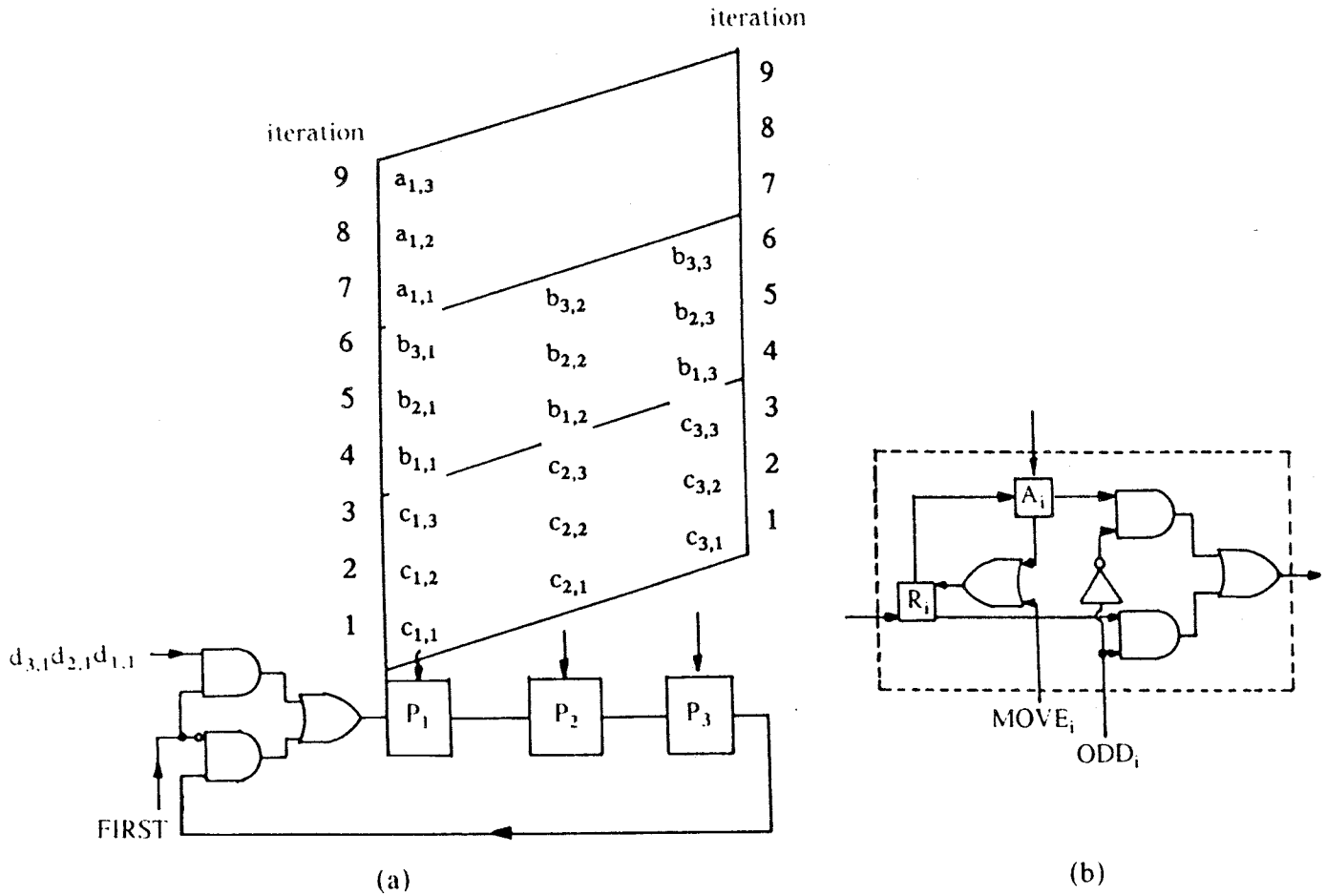

Figure 3. A pipelined version of a systolic array for computing a string of matrix multiplications. (a) Architecture of the systolic array. (b) Architecture of the *i*th processing element.

remains stationary, while the result vector, $\{f(B_i), i = 1, 2, 3\}$, is shifted. At the end of the sixth iteration, the output vector $\{f(B_i), i = 1, 2, 3\}$ is formed. In the last three iterations, input vectors **A** and $\{f(B_i), i = 1, 2, 3\}$ are shifted into P_1 to form the final result.

For the systolic array in Figure 3(a), the data shifted alternate between the input vector and the result vector every three iterations. This alternation can be controlled by the processor structure of P_i depicted in Figure 3(b). R_i is a register that stores an element of the input vector, and A_i is the accumulator that stores the temporary result of an element of the result vector. The data paths are controlled by control signals ODD_i and $MOVE_i$. When the number of matrices multiplied is odd (not including the initial input vector), ODD_i is one, hence, R_i is connected to the output, and the input vector is shifted along the pipeline. When the number of matrices multiplied is even, ODD_i is zero, A_i is connected to the output, and the result vector is shifted. At the end of a matrix multiplication, the result vector generated becomes the input vector in the next iteration and is moved by the control signal $MOVE_i$ from A_i to R_i. Note that there is a one-cycle delay between switching the control signals in P_{i+1} and P_i.

To search a multistage graph with $(N+1)$ stages and m nodes in each intermediate stage (the first and last stages have one node each), it takes $N \cdot m$ iterations with m processors. There is no delay between feeding successive input matrices into the systolic array, and the processors are kept busy most of the time. In contrast, it takes $(N-2)m^2+m$ iterations to solve the problem with a single processor. Define PU, the *processor utilization*, as the ratio of the number of serial iterations to the product of the number of parallel iterations and the number of processors. *PU* for the above systolic array is

$$PU = \frac{(N-2)m^2+m}{N \cdot m \cdot m} = \frac{N-2}{N} + \frac{1}{N \cdot m}. \tag{9}$$

When N and m are large, *PU* is very close to 1.

Although the proposed systolic array is designed for matrices in which each element is a single constant, it can be extended to many practical sequentially controlled systems, such as Kalman filtering, inventory systems, and multistage production processes, in which each matrix element is a vector with many quantized values. In this case the potential parallelism could be very large.

If broadcast is allowed, the above scheme can be simplified. In what follows, a linear systolic array with parallel inputs and broadcasting is described. Figure 4(a) depicts a scheme for computing $(\mathbf{A} \cdot (\mathbf{B} \cdot (\mathbf{C} \cdot \mathbf{D})))$ for the multistage graph depicted in Figure 1(a). In this scheme all input matrices are fed into the systolic array in the same format. In the first three iterations. $\mathbf{C} \cdot \mathbf{D}$ is evaluated. The control signal FIRST is one; **D**, the input vector, is broadcast to all processing elements (PEs); and the intermediate

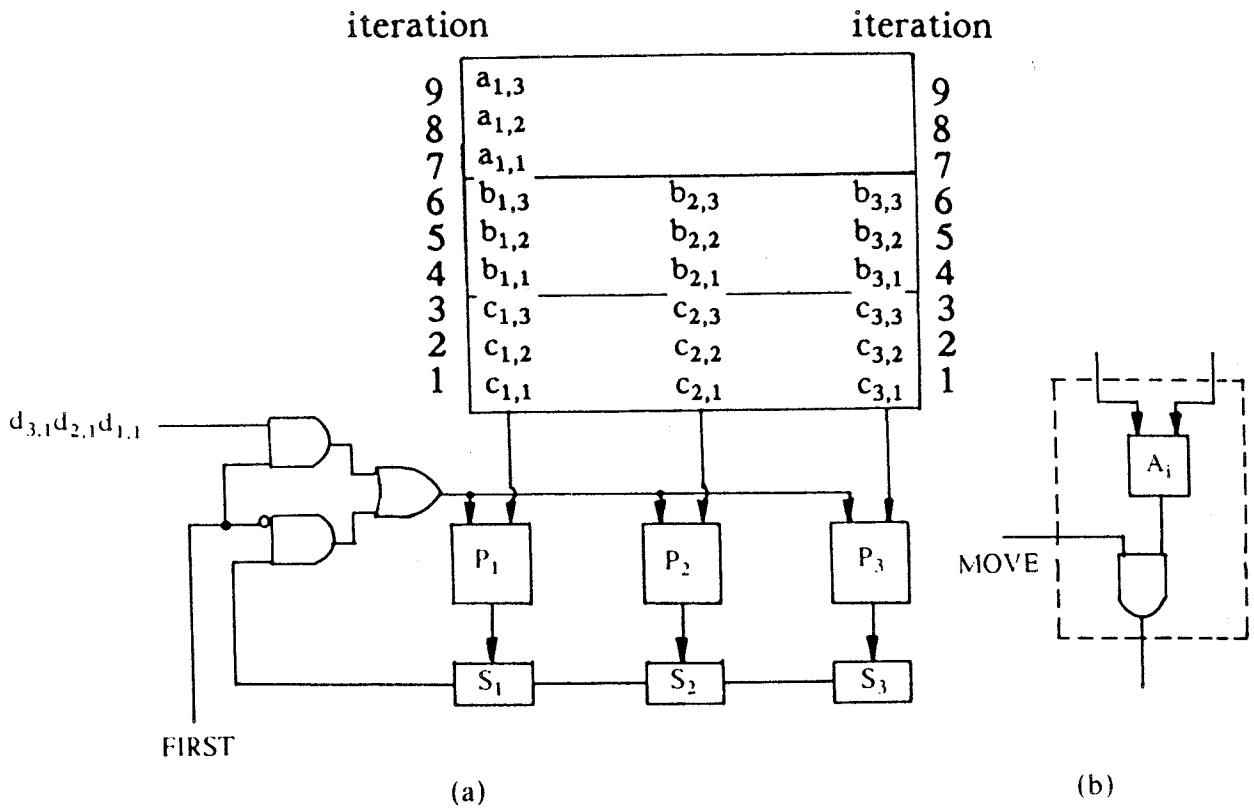

Figure 4. A systolic array with broadcasts for computing a string of matrix multiplications. (a) Architecture of the systolic array. (b) Architecture of the *i*th processing element.

results of $f(C_i)$, $i = 1, 2, 3$, remain stationary. At the end of the third iteration, the result vector is gated into registers S_1, S_2, S_3, by the control signal MOVE (see Figure 4(b)), and FIRST is set to zero. Since FIRST is zero, $f(C_i)$, $i = 1, 2, 3$, are fed back and broadcast as new inputs. In the following three iterations, $\mathbf{B} \cdot \mathbf{f}(\mathbf{C}) = \mathbf{B} \cdot (\mathbf{C} \cdot \mathbf{D})$ is computed. At the end of the sixth iteration, the output vector $\{f(B_i), i = 1, 2, 3\}$ is formed. In the last three iterations, input vectors $\mathbf{A}$ and $\{f(B_i), i = 1, 2, 3\}$ enter PE P_1 to form the final result. *PU* for this systolic array is again given by equation (9).

In the two schemes proposed above, the degree of parallelism may be restricted by the limited number of input/output ports in a VLSI chip and the fact that the ratio of the computational overhead to the input/output overhead is relatively low in matrix-vector multiplications. The input/output bottleneck is due to the large number of edge costs that must be fed into the systolic array.

The input/output bottleneck can be relieved if the serial optimalization problem is formulated by equation (4). The search for an optimal assignment of $\mathbf{X}_i$'s in equation (4) corresponds to the search for the shortest path in a multistage graph, where nodes in each stage represent values that can be assigned to a variable. An example graph with four variables, each of which can take on three quantized values, is shown in Figure 1(b). Here, the edge costs are expressed as functions of the nodes connected, and, hence, only the values of the nodes rather than the values on the edges have to be input. This results in an order-of-magnitude reduction in the input overhead. Systolic processing is suitable in this case when the number of quantized values in each stage is constant, and the f_i's, the functions to compute edges costs, are independent of i.

To solve equation (4) the variables can be eliminated one by one. First, $\mathbf{X}_1$ is considered. Since only one term, $\mathbf{f}(\mathbf{X}_1, \mathbf{X}_2)$, is affected by $\mathbf{X}_1$, it is sufficient to compute

$$\mathbf{h}(\mathbf{X}_2) = \min_{\mathbf{X}_1} \mathbf{f}(\mathbf{X}_1, \mathbf{X}_2). \tag{10}$$

In other words,

$$h(x_{2,j_2}) = \min_{x_{1,j_1} \in \mathbf{X}_1} f(x_{1,j_1}, x_{2,j_2}), \qquad x_{2,j_2} \in \mathbf{X}_2.$$

The optimization problem then becomes

$$\min_{\mathbf{X}} \mathbf{f}(\mathbf{X}) = \min_{\mathbf{X}-\{\mathbf{X}_1\}} \left\{ \mathbf{h}(\mathbf{X}_2) + \sum_{i=2}^{N-1} \mathbf{f}(\mathbf{X}_i, \mathbf{X}_{i+1}) \right\}. \tag{11}$$

If $\mathbf{h}(\mathbf{X}_k)$ is defined as

$$\mathbf{h}(\mathbf{X}_k) = \min_{\mathbf{X}_{k-1}} \{\mathbf{h}(\mathbf{X}_{k-1}) + \mathbf{f}(\mathbf{X}_{k-1}, \mathbf{X}_k)\}, \qquad 2 \le k \le N, \tag{12}$$

or

$$h(x_{k,j_k}) = \min_{x_{k-1,j_{k-1}} \in \mathbf{X}_{k-1}} \{h(x_{k-1,j_{k-1}}) + f(x_{k-1,j_{k-1}}, x_{k,j_k})\}, \quad x_{k,j_k} \in \mathbf{X}_k, \quad 2 \le k \le N,$$

then $h(x_{k,j_k})$ represents the shortest path from any vertex in stage 1 to x_{k,j_k}. After eliminating $k-1$ variables, $\mathbf{X}_1, \ldots, \mathbf{X}_{k-1}$, the remaining optimization problem becomes

$$\min_{\mathbf{X}} \mathbf{f}(\mathbf{X}) = \min_{\mathbf{X} - \{\mathbf{X}_1, \ldots, \mathbf{X}_{k-1}\}} \left\{ \mathbf{h}(\mathbf{X}_k) + \sum_{i=k}^{N-1} \mathbf{f}(\mathbf{X}_i, \mathbf{X}_{i+1}) \right\}. \tag{13}$$

Finally, we get $\mathbf{h}(\mathbf{X}_N)$, each element of which represents the shortest path from any vertex in stage 1 to a node in stage N. The problem is solved by comparing the m elements of $\mathbf{h}(\mathbf{X}_N)$.

Figure 5 shows a systolic array with three PEs that performs the search of the graph in Figure 1(b). PE P_i consists of three registers, R_i, K_i, H_i, and three operation components, F_i, A_i, C_i. Input data pass through R_i in a pipelined fashion. Feedback data are maintained in K_i and H_i until new data replace them. The operation components, F_i, A_i, C_i, are used to compute function f, and perform additions and comparisons, respectively. For simplicity, function f is assumed to be independent of i, and, hence, the subscripts in F_i, A_i, and C_i will be dropped. The connections of the registers and operation components are shown in Figure 5(b).

The systolic array is initialized by zeroing all registers, H_i's and K_i's and by sequentially loading input data in $\mathbf{X}_1, x_{1,m}, \ldots, x_{1,1}$, to PEs $P_1, \ldots, P_m$. As the intermediate results are shifted from P_3, the feedback controller feeds them back in a round-robin fashion. Referring to Figure 5(a), when $x_{2,1}$ enters P_1, $x_{1,1}$ and $h(x_{1,1})$ (equals 0) leave P_3 and are fed back to P_1 through the feedback controller. $f(x_{1,1}, x_{2,1})$ and $h^1(x_{2,1}) = \min(0, h(x_{1,1}) + f(x_{1,1}, x_{2,1}))$ are then computed in P_1. In the next iteration, $x_{2,2}$ enters P_1, $x_{2,1}$ and $h^1(x_{2,1})$ are shifted to P_2, and $x_{1,2}$ and $h(x_{1,2})$ (equals 0) are fed back by the feedback controller to P_2. In P_2, $f(x_{1,2}, x_{2,1})$ and $h^2(x_{2,1}) = \min(h^1(x_{2,1}), h(x_{1,2}) + f(x_{1,2}, x_{2,1}))$ are computed. In P_1, $f(x_{1,1}, x_{2,2})$ and $h^1(x_{2,2}) = \min(0, h(x_{1,1}) + f(x_{1,1}, x_{2,2}))$ are computed. When $x_{2,1}$ and $h^2(x_{2,1})$ arrive at P_3, $h(x_{2,1}) = h^3(x_{2,1})$ is evaluated, and $x_{2,1}$ and $h(x_{2,1})$ are fed back to P_1 at the end of this iteration. Input data are continuously shifted into the pipeline, and the process is repeated. For the graph in Figure 1(b), the process is completed in 15 iterations.

In general, to evaluate the optimal path for an N-stage graph, each with m quantized values, a pipeline with m PEs is needed. Between the $((k-1)m+1)$th and $(k \cdot m)$th iterations, $2 \le k \le N$, $x_{k,1}, \ldots, x_{k,m}$ enter the R-pipeline; $x_{k-1,i}$ and $h(x_{k-1,i})$, $1 \le i \le m$, are fed back to registers K_i and H_i in the $((k-1)m+i)$th iteration; and $h^m(x_{k,1}), h^{m-1}(x_{k,2}), \ldots, h^1(x_{k,m})$ are obtained at the end of the $(k \cdot m)$th iteration in $P_m, \ldots, P_1$. After $N \cdot m$ iterations, $h^m(x_{N,1}), \ldots, h^1(x_{N,m})$ are obtained in $P_m, \ldots, P_1$, and the final

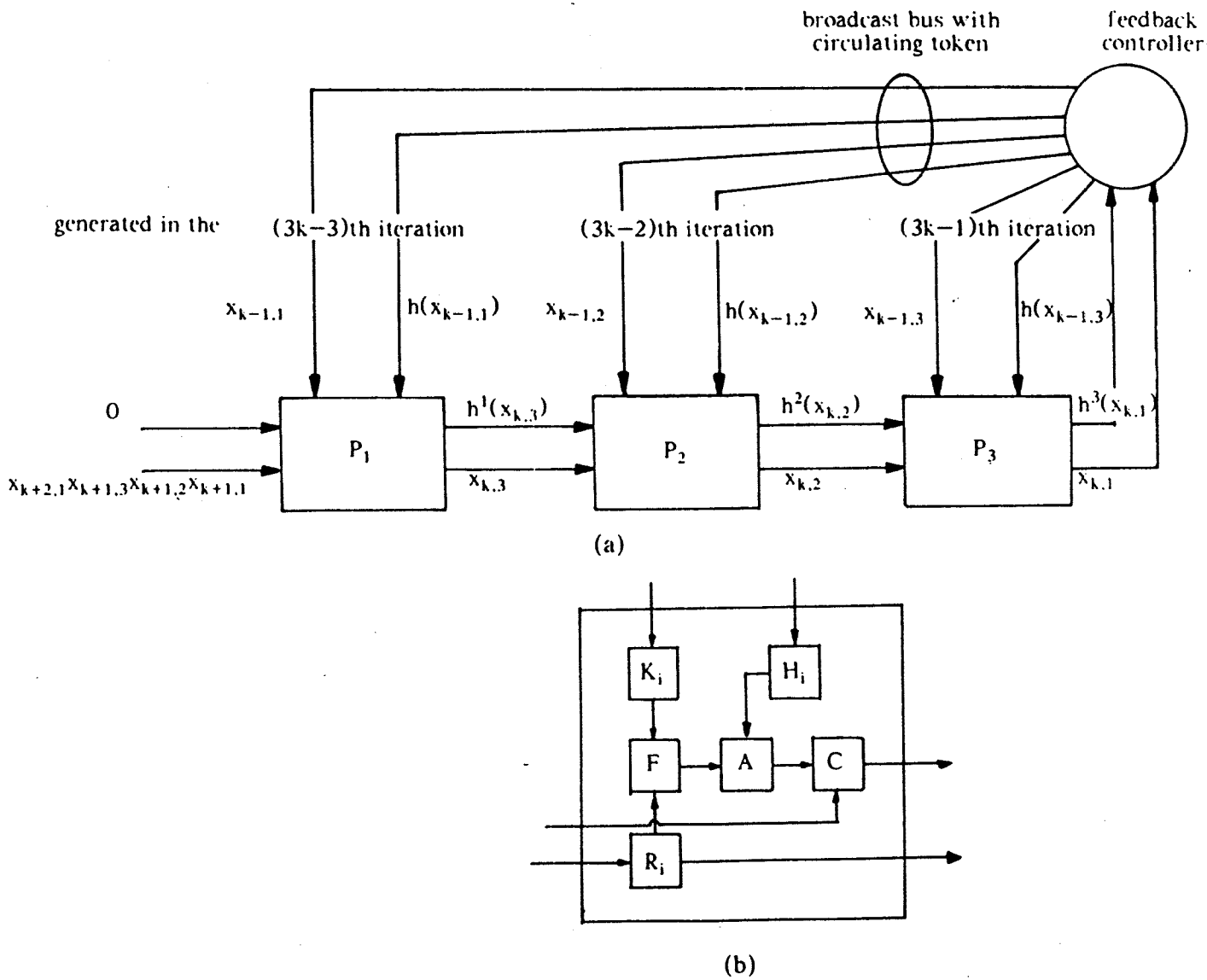

Figure 5. A systolic array with serial inputs and outputs to solve a monadic-serial DP problem. (a) Architecture of the systolic array. (b) Architecture of the *i*th processing element.

solution is obtained in P_m by comparing $h(x_{N,1}), \ldots, h(x_{N,m})$. This is done by setting $F = 0$ in the last m iterations and circulating the values of $h(x_{N,i})$, $1 \le i \le m$, through the pipeline. Therefore, the total computational time is $(N+1)m$ iterations, each of which includes the time for the computation of function f, one addition, and one comparison. *PU* for this scheme is $((N-1)m^2+m)/((N+1)m \cdot m) \approx 1$.

Although distinct feedback lines are shown in Figure 5(a), only one of the feedback lines is used in any iteration. Hence a single broadcast bus suffices, and the station to pick up the data from the bus is controlled by a circulating token.

If the optimal path in addition to the optimal cost value is desired, N path registers, each of which can store m indices, are needed in P_m. In the computation of $h^m(x_{k+1,i}) = \min_j\{h(x_{k,j}) + f(x_{k,j}, x_{k+1,i})\}$, $1 \le k \le N-1$, index j', $1 \le j' \le m$, of the edge $(x_{k,j'}, x_{k+1,i})$ belonging to the optimal path from any vertex in stage 1 to vertex $x_{k+1,i}$ must be propagated in the pipeline and is known to P_m. Index j' is stored in the ith word of the kth path register. The pointers stored in the path registers are used to trace the optimal path at completion.

4. Solving polyadic-serial DP problems by divide-and-conquer algorithms

Recall that a serial optimization problem can be solved as the multiplication of a string of matrices. However, a problem expressed in a monadic-serial formulation does not exploit all the potential parallelism because the order of matrix multiplications is fixed. On the other hand, there is more flexibility for parallelism when the problem is formulated in a polyadic equation because the matrices can be multiplied recursively by a divide-and-conquer algorithm.

Consider the polyadic-serial DP formulation in equation (3) for the multistage shortest-path problem in Figure 1(b):

$$f_3(s, t) = \min_{k \in \{x_{2,1}, x_{2,2}, x_{2,3}\}} [f_3(s, k) + f_3(k, t)], \tag{14}$$

where $f_3(i, j)$ is the cost of the optimal path from i to j, s is a node in stage 1 of the graph, k is a node in stage 2, and t is a node in stage 4. In matrix notations, let $f_3(\mathbf{V}_i, \mathbf{V}_j)$ be a cost matrix, each element of which denotes the cost of the optimal path from a vertex in stage i to a vertex in stage j. It is easy to see, for an intermediate stage k between i and j, that

$$f_3(\mathbf{V}_i, \mathbf{V}_j) = f_3(\mathbf{V}_i, \mathbf{V}_k) \cdot f_3(\mathbf{V}_k, \mathbf{V}_j). \tag{15}$$

This formulation allows a string of matrix multiplications to be reduced to two smaller strings of matrix multiplications. For simplicity in deriving the lower bound, all the matrices are assumed to have identical dimensions.

One important issue in parallel divide-and-conquer algorithms is the *granularity of parallelism* [28]. This is the minimum size of a subproblem that is evaluated by a processor (processors and systolic arrays are synonymous here) in order to achieve the optimal performance, as measured by the *PU*, or the AT^2, or the KT^2 criteria, where A, K, and T are the area of a VLSI implementation, the number of processors, and the computational time, respectively. If the granularity is large, then the processors can be loosely coupled; otherwise, tight coupling is necessary.

Parallel divide-and-conquer algorithms is a parallel AND-tree search that can roughly be divided into three phases: start-up, computation, and wind-down. In the start-up phase, the problem is split, and the tasks are distributed. During the computation phase, all processors are kept busy until the number of tasks in the system is less than the number of processors. In the wind-down phase, the results are combined together, and some processors may be idle. *PU* depends on the ratio between the amount of time spent in the computation phase and that of the other phases. The time complexity of searching a binary AND-tree of N leaves can be formulated in the following recursive equation:

$$T(N)=\begin{cases} S(N)+2T\left(\dfrac{N}{2}\right)+C(N), & N>1, \\ O(1), & N=1, \end{cases} \tag{16}$$

hwere $S(N)$ and $C(N)$ are the time complexities of the start-up and wind-down phases. The granularity that results in the optimal *PU* is related to the complexity of $S(N)$ and $C(N)$. In finding the sum or the maximum of N numbers, $S(N)+C(N)=O(1)$, and using $O(N/(\log_2 N))$ processors will achieve the maximum *PU* [15], [2], [26]. In sorting N numbers, $S(N)+C(N)=O(N)$, and $\log_2 N$ processors should be used to maximize the *PU*.

The fastest way to multiply N $m\times m$ matrices is to locate the matrices in the leaves of a complete binary tree of height $\lceil \log_2 N \rceil$. The N-stage graph problem can be solved in $O(m\cdot \lceil \log_2 N \rceil)$ time units with $\lceil N/2 \rceil$ matrix-multiplication systolic arrays [19]. The *PU* of the systolic arrays is relatively low in this approach. In this case, *PU* does not measure the utilization of the processing elements in a systolic array but measures the average fraction of time that a systolic array is used in matrix multiplications. Similar to the search of the maximum, using $\lceil N/(\log_2 N) \rceil$ systolic arrays will result in the maximum *PU*. However, the asymptotic effects of deviations from the optimum have not been studied before. This is addressed by the following proposition.

Proposition 1. *Let $PU(k, N)$ be the processor utilization of multiplying a string of N matrices using $k(N)$ synchronous systolic arrays. Assume the time to multiply two matrices by a systolic array to be constant. By defining*

$c_\infty = \lim_{N\to\infty}(k(N)/(N/\log_2 N))$, *the normalized asymptotic processor utilization is*

$$\lim_{N\to\infty} PU(k, N) = \begin{cases} 0 & \text{if } c_\infty \to \infty, \\ \dfrac{1}{1+c_\infty} & \text{if } 0 < c_\infty < \infty, \\ 1 & \text{if } c_\infty = 0. \end{cases} \tag{17}$$

Proof. Note that there is no start-up phase in this problem as the matrices to be multiplied are known initially. The number of nodes evaluated in the first iteration of the wind-down phase is at least $k/2$ and at most $k-1$. Hence, the total number of nodes evaluated in the wind-down phase is at least $k-1$ and at most $2k-3$. As the total number of nonterminal nodes in the binary AND-tree is $N-1$, the total number of nonterminal nodes evaluated in the computation phase is at least $(N-1)-(2k-3) = N+2-2k$, and at most $(N-1)-(k-1) = N-k$. I_a, the number of iterations in the computation phase, is bounded by

$$\left\lceil \frac{N+2}{k} - 2 \right\rceil \le I_a \le \left\lceil \frac{N}{k} - 1 \right\rceil. \tag{18}$$

According to the bound of tree-height reduction, I_w, the number of iterations in the wind-down phase, is bounded by

$$\log_2(2k-3) \le I_w \le \log_2(k-1). \tag{19}$$

By the definition of PU we have

$$\lim_{N\to\infty} PU(k, N) = \lim_{N\to\infty} \frac{N-1}{k(I_a + I_w)}. \tag{20}$$

We now discuss three separate cases.

(a) $c_\infty \to \infty$. From equations (18)-(20) we get

$$\begin{aligned} \lim_{N\to\infty} PU(k, N) &\le \lim_{N\to\infty} \frac{N-1}{k[(N+2)/k - 2 + \log_2(2k-3)]} \\ &\le \lim_{N\to\infty} \frac{N-1}{N + k\cdot(\log_2(2k-3)-2)} \\ &\le \frac{1}{1 + c_\infty \lim\limits_{N\to\infty} \dfrac{\log_2((2k-3)-2)}{\log_2 N}}. \end{aligned} \tag{21}$$

Since $c_\infty \to \infty$, we can assume that $k > N/\log_2 N$ for $N > N_0$ and

$$\log_2((2k-3)-2) > \log_2 k > [\log_2 N - \log_2 \log_2 N].$$

This means that $\lim_{N\to\infty}((\log_2((2k-3)-2))/\log_2 N)>1$, implying that $PU(k, N)\le 0$. As $PU(k, N)$ must be nonnegative, the first case of equation (17) is proved.

(b) $c_\infty = 0$. From equations (18)-(20) we get

$$\lim_{N\to\infty} PU(k, N) \ge \lim_{N\to\infty} \frac{N-1}{k(N/k-1+\log_2(k-1))]}$$

$$\ge \lim_{N\to\infty} \frac{N-1}{N+k\cdot(\log_2(k-1)-1)}. \tag{22}$$

Since $c_\infty \to 0$, we can assume that $k < N/\log_2 N$ for $N > N_0$, and

$$\log_2((k-1)-1) < \log_2 k < [\log_2 N - \log_2\log_2 N].$$

This means that $\lim_{N\to\infty}((\log_2((k-1)-1))/\log_2 N)<1$, implying that $PU(k, N)\ge 1$. As $PU(k, N)\le 1$, the third case of equation (17) is proved.

(c) $0 < c_\infty < \infty$. From the definition of c_∞ we have $\lim_{N\to\infty} k = \lim_{N\to\infty}(c_\infty N/\log_2 N)$ and

$$\lim_{N\to\infty}\frac{\log_2 k}{\log_2 N} = \lim_{N\to\infty}\frac{\log_2 c_\infty + \log_2 N - \log_2\log_2 N}{\log_2 N} = 1. \tag{23}$$

From equations (21)-(23) it is easy to see that

$$\frac{1}{1+c_\infty} \le \frac{1}{1+c_\infty \lim_{N\to\infty}\dfrac{\log_2((k-1)-1)}{\log_2 N}} \le PU(k, N)$$

$$\le \frac{1}{1+c_\infty \lim_{N\to\infty}\dfrac{\log_2((2k-3)-2)}{\log_2 N}} \le \frac{1}{1+c_\infty}. \tag{24}$$

This proves the second case of equation (17). □

As an example in applying equation (17), suppose that there are $\sqrt{N}$ processors. It is easy to show that $c_\infty = \lim_{N\to\infty}(\sqrt{N} * \log_2 N/N) = 0$. Hence $PU(k, N) = 1$. Note that the above proposition holds in the case in which each node is evaluated in constant time, as the constant complexity does not affect the limits in equations (21) and (22).

Since $PU(k, N)$ increases monotonically with decreasing k, $PU(k, N)$ alone is not adequate to measure the effects of parallel processing. A popular measure in VLSI complexity theory is the AT^2 criterion. The following theorem proves the lower-bound AT^2 complexity of divide-and-conquer algorithms for solving polyadic-serial DP problems. This lower bound is attained when $k(N) = N/(\log_2 N)$. Results on the scheduling algorithms to achieve the optimal granularity of divide-and-conquer algorithms have been presented elsewhere [20].

Theorem 1. *Suppose that a string of N $m \times m$ matrices are multiplied by $S(N)$ processors in time $T(N)$ using a divide-and-conquer algorithm, and that each processor performs a multiplication of a pair of $m \times m$ matrices in T_1 time units. Then $S(N)T^2(N) \geq \Theta(N \cdot \log_2 N)T_1^2$, and equality holds when $S(N) = \Theta(N/\log_2 N)$. ($\Theta$ indicates the set of functions of the same order.)*

Proof. The multiplication of a string of M matrices by a divide-and-conquer algorithm can be represented as a complete binary tree with M terminals. The number of matrix-multiplications, or the number of nonterminals, is $M-1$. A parallel divide-and-conquer algorithm for multiplying a string of N, $N > M$, matrices can be roughly divided into two phases: computation and wind-down. During the computation phase, all processors are kept busy until half of the number of intermediate matrices to be multiplied is less than the number of processors. There are $(N-1)-(S(N)-1) = N-S(N)$ nonterminals to be evaluated, and at least $(N/S(N)-1) \cdot T_1$ time units are required. In the wind-down phase the results are combined together, and some processors would be idle. According to the data dependence, at least $\log_2 S(N) \cdot T_1$ time units are required in this phase. Therefore, the following lower bound of time complexity holds for any parallel divide-and-conquer algorithm:

$$T(N) \geq \left(\frac{N}{S(N)} - 1 + \log_2 S(N)\right) \cdot T_1, \tag{25}$$

where $1 \leq S(N) \leq N$. For simplicity, the constant term in equation (25) can be ignored without affecting the validity of the following proof. The AT^2 lower bound is derived as

$$S(N)T^2(N) \geq \left(\frac{N^2}{S(N)} + 2N \log_2 S(N) + S(N) \log_2^2 S(N)\right) \cdot T_1^2. \tag{26}$$

To find the order-of-magnitude minimum of equation (26), it is necessary to compare the following three cases. When $S(N) = \Theta(N/\log_2 N)$, $S(N)T^2(N) = \Theta((N \log_2 N)T_1^2)$. In contrast, when $S(N) < \Theta(N/\log_2 N)$, the first term on the right-hand side of equation (26) is

$$\frac{N^2}{S(N)} > \Theta(N \log_2 N). \tag{27}$$

When $S(N) > \Theta(N/\log_2 N)$, the third term on the right-hand side of equation (26) is

$$S(N) \log_2^2 S(N) > \Theta(N \log_2 N), \tag{28}$$

since $\log_2^2 S(N) \geq \Theta(\log_2^2 N)$. The above analysis shows that the AT^2 complexity is $\Omega((N \log_2 N)T_1^2)$, and that $\Theta(N/\log_2 N)$ is the optimal granularity to achieve this lower bound. □

To investigate the relationship between K and KT^2, the exact time required to multiply N $m \times m$ matrices using K processors is derived. The total time required is

$$T = T_c + T_w$$
$$= \left\lfloor \frac{N-1}{K} \right\rfloor \cdot T_1 + \left\lfloor \log_2 \left(N + K - 1 - K \cdot \left\lfloor \frac{N-1}{K} \right\rfloor \right) \right\rfloor \cdot T_1, \quad (29)$$

where T_c and T_w represent the times in the computation and wind-down phases, respectively. The numerical evaluations of equation (29) for $N = 4096$ is shown in Figure 6, in which KT^2 is minimum when 431 or 465 processors are used. Notice that the curve is not smooth because the time needed in the wind-down phase is decreased by 1 whenever N is divisible by K, and this affects KT^2 significantly, especially when K is large. The simulation results for different values of N verify that the optimal granularity is close to $N/\log_2 N$.

When N is large and $\lfloor (N-1)/K \rfloor$ is approximately equal to $(N-1)/K$, $T_c \approx (N/K) - 1$, and $T_w \approx \log_2 K$,

$$T \approx \frac{N}{K} - 1 + \log_2 K. \quad (30)$$

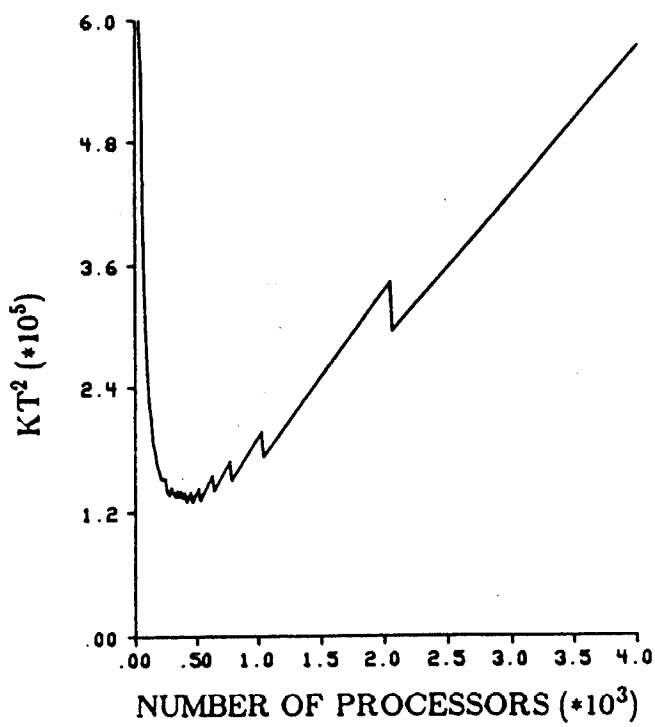

Figure 6. Simulation results of finding the optimal granularity of parallel divide-and-conquer algorithms ($N = 4096$).

KT^2 will achieve the minimum value when $T_c = T_w$. This can be shown by differentiating $KT^2(K)$ with respect to K and relaxing the constraint that K is an integer:

$$\frac{\partial KT^2(K)}{\partial K} = T^2 + 2KT\frac{\partial T}{\partial K}. \tag{31}$$

From equations (30) and (31) we get $\partial KT^2(K)/\partial K = 0$ if $(N/K - 1) = \log_2 K$, i.e., $T_c = T_w$. This means KT^2 for solving polyadic-serial DP problems by parallel divide-and-conquer algorithms approaches minimum if N is large and the times needed in the computation and wind-down phases are approximately equal.

So far the matrices are assumed to have identical dimensions. When this is not true, the order in which the matrices are multiplied together has a significant effect on the total number of operations. Finding the optimal order of multiplying a string of matrices with different dimensions is itself a polyadic-nonserial DP problem, the so-called secondary optimization problem [6], [4]. Guibas *et al.* have proposed a systolic array to solve the optimal parenthesization problem, which can be used to compute the minimum-cost order of multiplying a string of matrices [11]. Once the optimal order is found, the processors can be assigned to evaluate the matrix multiplications in the defined order and in an asynchronous fashion. In this sense, the tree of matrix multiplications can be treated as a dataflow graph.

5. Solving polyadic-serial DP problems by searching folded AND/OR trees

In this section we discuss the evaluation of polyadic-serial DP problems as AND/OR-graph searches. AND/OR-graphs are naturally obtained by representing the DP problem using a problem-reduction method. The mapping of a regular AND/OR-graph onto a systolic array is straightforward and will be illustrated in the next section.

Polyadic-serial problems are discussed with respect to the search of a multistage graph as formulated by equation (4). Suppose an $(N+1)$-stage graph, with stages from 0 to N and m nodes in each stage, is divided into p subgraphs, each of which contains $N/p+1$ consecutive stages. For simplicity, assume that $N = p^Q$, where Q is a nonnegative integer. The minimum-cost path has to pass through one and only one vertex in stage 0, $N/p, \ldots, pN/p$ in the segmented graph. The cost of a path equals the sum of costs of the p subpaths. If all the m^2 subpaths from the m vertices in stage iN/p to the m vertices in stage $(i+1)N/p$, $0 \le i \le p-1$, have been optimized, there are m^{p+1} possible combinations of subpaths from stage 0 to stage N that must be considered for the optimal path. Using a divide-and-conquer algorithm, each subgraph with $N/p+1$ stages is further divided

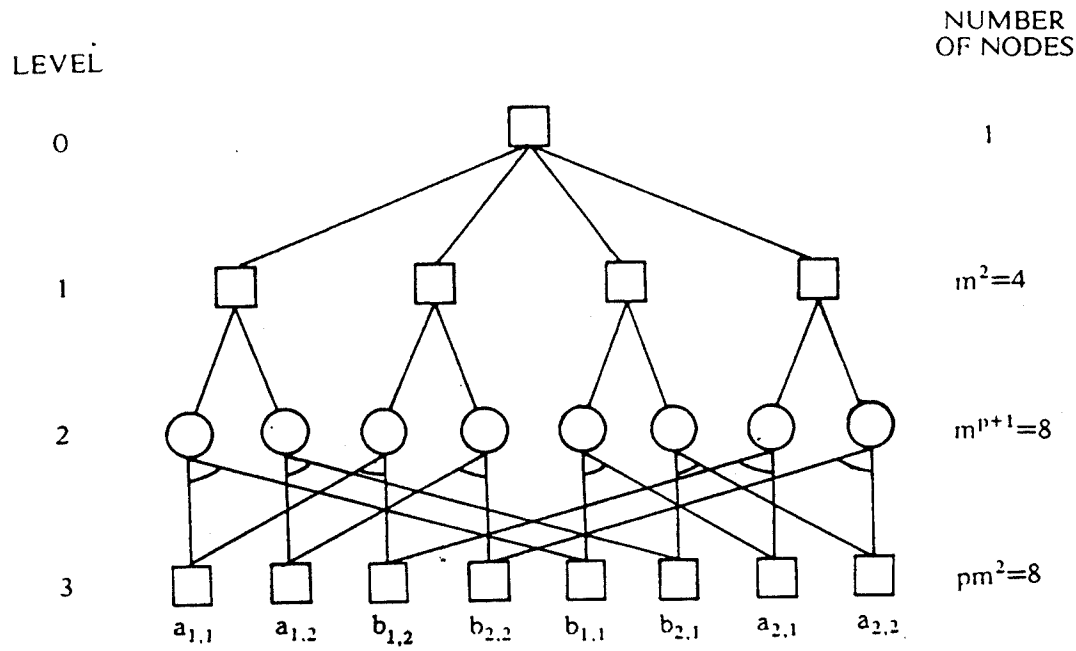

Figure 7. An AND/OR-graph representation of the reduction in finding an optimal path in a three-stage graph to a one-stage graph.

into p smaller subgraphs. This partitioning process continues until each subgraph has one stage.

The partitioning process can be conveniently represented as an AND/OR graph, in which an AND-node corresponds to a subproblem sum, and an OR-node corresponds to alternative selections or comparisons. In this case we have a regular AND/OR-graph of height $2 \cdot \log_p N$, whose AND-nodes have p branches (p-arc nodes) and whose OR-nodes have m^{p-1} branches (m^{p-1}-arc nodes). Figure 7 shows an AND/OR-graph that represents the reduction of the multistage-graph problem with $m = 2$ and $p = 2$ from three stages to one stage. The four nodes at the top of the AND/OR-graph represents the four possible alternate paths in the reduced single-stage graph. The shortest path is obtained by a single comparison of these paths.

The relationship between DP and graph search was investigated by Martelli and Montanari [21] who showed that, in the case of polyadic cost functions, the solution of a DP problem can be obtained by finding a minimal-cost solution tree in an AND/OR-graph. This equivalence allows various graph searching techniques to be translated into techniques for solving DP problems. For those acyclic AND/OR-graphs with positive arc costs, Martelli and Montanari have named them as additive [21], and have proposed top-down and bottom-up search algorithms. A similar algorithm, called AO*, for searching hypergraphs was dicussed by Nilsson [24].

The above AND/OR-graph representation of a polyadic DP problem can be considered as a folded AND/OR-tree. It is easy to see that the efficiency of solving a DP problem by searching an AND/OR-graph depends

on the graph structure (parameter p). The following theorem analyzes the optimal structure.

Theorem 2. *If a serial* DP *problem is solved by searching a regular* AND/OR-*graph, the binary partition, namely, using* 2-*arc* AND-*nodes, is optimal in the sense of minimizing the total number of nodes in the* AND/OR-*graph.*

Proof. The total number of nodes in the AND/OR-graph is derived here. First, consider the reduction from $p+1$ stages to one stage (see Figure 7 for $p=2$). The number of nodes in the bottom level is $p \cdot m^2$, as there are $p \cdot m^2$ cost values for all pairs of vertices between neighboring stages. Similarly, there are m^2 OR-nodes in level 1 and $m^2 \cdot m^{p-1}$ AND-nodes in level 2.

For the reduction from $N+1$ $(=p^Q+1)$ stages to one stage, there are $\log_p N$ levels of AND-nodes at odd levels, with m^{p+1}, $p \cdot m^{p+1}, \ldots, p^{(\log_p N)-1} m^{p+1}$ nodes, respectively. Similarly, there are $(\log_p N)+1$ levels of OR-nodes at even levels with $m^2, p \cdot m^2, \ldots, p^{\log_p N} m^2$ nodes, respectively. Let $u(p)$ be the total number of nodes in the AND/OR-graph:

$$u(p) = \sum_{i=0}^{(\log_p N)-1} p^i \cdot m^{p+1} + \sum_{j=0}^{\log_p N} p^j m^2$$

$$= \frac{N-1}{p-1} m^{p+1} + \frac{N \cdot p - 1}{p-1} m^2. \tag{32}$$

To find the minimum $u(p)$ we relax the restriction that p is an integer and evaluate the differential of $u(p)$:

$$\frac{\partial u(p)}{\partial p} = \frac{(N-1)(m^{p+1}((p-1)\log_e m - 1) - m^2)}{(p-1)^2}. \tag{33}$$

From equation (33), it is seen that $\partial u(p)/\partial p \geq 0$ if $N \geq 1$, $p \geq 2$, and $m \geq 3$, or $N \geq 1$, $p \geq 3$, and $m \geq 2$. Considering $\partial^2 u(p)/\partial p^2$, we conclude that $u(p)$ increases monotonically when $N \geq 1$, $p \geq 2$, and $m \geq 3$. In other words, the binary partition with $p=2$ is optimal for solving regular multistage-graph problems in the sense of minimizing the total number of nodes in the corresponding AND/OR-graph. □

For an AND/OR-graph, the larger the value of p is, the less the Principle of Optimality is applied. In the extreme case, $p=N$, the corresponding AND/OR-graph search becomes a brute-force search, and the Principle of Optimality is never used. For irregular multistage-graph problems, the number of nodes in the AND/OR-graph depends on the ordering of stage reduction. However, it is not difficult to demonstrate that binary partitioning is optimal. Assume that stages $i_1, \ldots, i_4$ with $m_1, \ldots, m_4$ nodes are to be reduced to two stages i_1 and i_4. If 3-arc AND-nodes are used,

$m_1 \cdot m_2 \cdot m_3 \cdot m_4$ comparisons are needed to eliminate stages 2 and 3. However, when 2-arc AND-nodes are used, $m_1 \cdot m_3 \cdot (m_2 + m_4)$ comparisons are needed if stage 2 is eliminated first, and $m_2 \cdot m_4 \cdot (m_1 + m_3)$ comparisons are needed if stage 3 is eliminated first. It is easy to see that using 3-arc AND-nodes requires more comparisons as long as $m_i \geq 2$, $1 \leq i \leq 4$. Furthermore, binary partitioning requires less additions since one addition is needed for each AND-node.

6. Parallel processing of nonserial DP problems

It has been shown that unrestricted nonserial optimization problems are *NP*-hard, but problems with a favorable pattern of term interactions may be solved efficiently [5]. The key of DP is to break a complex optimization problem into a sequence of easier subproblems. In serial optimization problems variables are shared by successive terms in the objective function and hence can be dealt with one by one. This serial structure allows efficient parallel processing, especially systolic processing. On the other hand, nonserial DP problems may have to be transformed into the corresponding serial formulations before they are implemented by systolic processing. This transformation is possible if the nonserial problems have some special structures.

Generally speaking, there are two ways to convert a nonserial formulation into a serial one. One way is to combine several primary variables into a new variable. Another way is to transform an irregular AND/OR-graph into regular one by adding dummy nodes. The former one is suited for solving monadic-nonserial problems, and the latter one is usually applied to solving polyadic-nonserial problems. These methods are illustrated by some examples in the following sections.

6.1. *Solving monadic-serial problems*

An approach to solving a monadic-nonserial problem with some structural properties is first to convert it into a monadic-serial problem, such as a multistage graph-search problem, and to map the serial formulation into systolic arrays.

For the nonserial formulation in equation (5), a multistage optimization procedure can be carried out separately for each variable $\mathbf{V}_i$. Of course, this optimization must be performed on all values of the independent variables that "interact" with $\mathbf{V}_i$. Let $\mathbf{V}_{i_1}, \ldots, \mathbf{V}_{i_s}$ be the variables that are related to $\mathbf{V}_i$ in one or more functional terms. The cost function can be written as

$$f(v_1, \ldots, v_n) = \min_{v_k \in \mathbf{V}_k} \{h_1(v_i, v_{i_1}, \ldots, v_{i_s}) + h_2(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)\}, \tag{34}$$

where h_2 is a function independent of $\mathbf{V}_i$. By denoting $h_{1,\text{opt}}$ as $\min_{v_i \in \mathbf{V}_i} h_1(v_i, v_{i_1}, \ldots, v_{i_s})$, the cost function can be rewritten as

$$f(v_1, \ldots, v_n) = \min_{\mathbf{V}-\mathbf{V}_i} \{h_{1,\text{opt}}(v_{i_1}, \ldots, v_{i_s}) + h_2(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)\}. \tag{35}$$

A multistage optimization process is, therefore, a step-by-step elimination of all variables. The computational time and storage depend on the number of elements in the domain of function h_1. Equation (35) can be treated as a monadic-serial form if the evaluation of $h_{1,\text{opt}}$ is done separately.

The method is illustrated by the following example. For instance, if $\mathbf{V} = \mathbf{V}_1 \cup \cdots \cup \mathbf{V}_N$, then the objective function is

$$f(\mathbf{V}) = \min_{v_i \in \mathbf{V}_i} \{g_1(v_1, v_2, v_3) + g_2(v_2, v_3, v_4) + \cdots + g_{N-2}(v_{N-2}, v_{N-1}, v_N)\}. \tag{36}$$

Let $h_1(v_2, v_3) = \min_{v_1 \in \mathbf{V}_1} g_1(v_1, v_2, v_3)$, we have

$$\min_{\mathbf{V}} f(\mathbf{V}) = \min_{\mathbf{V}-\mathbf{V}_1} \left\{ h_1(v_2, v_3) + \sum_{i=2}^{N-2} g_i(v_i, v_{i+1}, v_{i+2}) \right\} \tag{37}$$

if $h_k(v_{k+1}, v_{k+2})$ is defined as

$$h_k(v_{k+1}, v_{k+2}) = \min_{v_k \in \mathbf{V}_k} \{h_{k-1}(v_k, v_{k+1}) + g_k(v_k, v_{k+1}, v_{k+2})\}. \tag{38}$$

Equation (38) represents the minimum of the summation of the first k terms of $f(\mathbf{V})$. After eliminating k variables, $\mathbf{V}_1, \ldots, \mathbf{V}_k$, the remaining optimization problem becomes

$$\min_{\mathbf{V}} f(\mathbf{V}) = \min_{\mathbf{V}-(\mathbf{V}_1 \cup \cdots \cup \mathbf{V}_k)} \left\{ h_k(v_{k+1}, v_{k+2}) + \sum_{i=k+1}^{N-2} g_i(v_i, v_{i+1}, v_{i+2}) \right\}. \tag{39}$$

The monadic DP procedure, thus, eliminates the variables in the order $\mathbf{V}_1, \ldots, \mathbf{V}_N$. If the variables $\mathbf{V}_2$ and $\mathbf{V}_3$ are treated as a single variable in a stage, and m_k, $1 \le k \le n$, quantized values are allowed for $\mathbf{V}_k$, then there would be $m_2 \cdot m_3$ states in this stage, and $m_1 \cdot m_2 \cdot m_3$ steps are required to eliminate $\mathbf{V}_1$. Here, a step consists of a computation of function f, an addition, and a comparison operation. The process of eliminating the remaining variables is repeated until $\mathbf{V}_{N-1}$ and $\mathbf{V}_N$ remain. The optimal solution is obtained by comparing all values of $h_{N-2}(v_{N-1}, v_N)$. The total number of steps required to compute equation (36) is

$$\sum_{k=1}^{N-2} (m_k \cdot m_{k+1} \cdot m_{k+2}) + m_{N-1} \cdot m_N. \tag{40}$$

In short, the monadic-nonserial problem in equation (36) is solved from the following serial problem:

$$\min_{\mathbf{V}} f(\mathbf{V}) = \min_{\mathbf{V}} \{g'_1(\mathbf{V}'_1, \mathbf{V}'_2) + g'_2(\mathbf{V}'_2, \mathbf{V}'_3) + \cdots + g'_{N-2}(\mathbf{V}'_{N-2}, \mathbf{V}'_{N-1})\}, \tag{41}$$

where the new variable $\mathbf{V}'_i$ is combined from $\mathbf{V}_i$ and $\mathbf{V}_{i+1}$. From this example it is observed that more operations are needed for evaluating monadic-nonserial DP problems than that of monadic-serial DP problems, but the potential parallelism is higher. With additional control, the linear systolic array presented earlier can be applied to evaluate monadic-nonserial DP problems.

6.2. *Solving polyadic-nonserial DP problems by searching AND/OR-graphs*

AND/OR-graphs can be sequentially searched in a breadth-first bottom-up fashion, which expands nodes by levels from the bottom up [24]. Since an acyclic AND/OR-graph can be viewed as a folded tree, searching the AND/OR-graph can be accomplished by searching the corresponding AND/OR-tree. In a parallel AND/OR-tree search, the nodes in the tree are evaluated in parallel in a bottom-up fashion. The parallel architecture can be designed with a flexible interconnection, such that a processor can be dynamically assigned when it is free, or can be designed with a limited interconnection, such that a static evaluation order is maintained for a given problem. A dataflow processor is an example of the first alternative. We will investigate the second alternative here.

Parallel AND/OR-tree search of polyadic-serial DP problems is a special case of that of polyadic-nonserial DP problems. For nonserial DP problems, the dependency between states is not restricted to successive stages but may exist between states in arbitrary stages. In the corresponding AND/OR-graphs the arcs are not restricted to successive levels but may run between any two arbitrary levels. It may be difficult to map an irregular AND/OR-graph to a systolic array with a regular interconnection structure. The nonserial AND/OR-graph may have to be transformed into a serial one before the mapping is done.

The strategy is illustrated by the problem of finding the optimal order of multiplying a string of matrices. For simplicity, consider the evaluation of the product of four matrices,

$$M = M_1 \times M_2 \times M_3 \times M_4,$$

where M_i, $1 \le i \le 4$, is a matrix with r_{i-1} rows and r_i columns, $r_j > 0$, $0 \le j \le 4$.

Let $m_{i,j}$ be the minimum cost of computing $M_i \times \cdots \times M_j$. Clearly,

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} (m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j) & \text{if } j > i. \end{cases}$$

The solution to be found is $m_{1,4}$. This problem was described in Section 2 and can be represented as a search of an AND/OR-graph, where the AND-nodes denote additions and the OR-nodes denote comparisons (see Figure 2).

The AND/OR-graph in Figure 2 can be mapped directly into six processors connected by multiple broadcast busses. Each processor evaluates an OR-node and its immediate descendent AND-node(s). The broadcast structure is necessary because a processor has to communicate with multiple processors and not its neighbors alone. Let $T_d(k)$, $1 \le k \le N$, be the time to find the optimal order of multiplying k matrices. Then,

$$T_d(k) = \begin{cases} T_d(\lceil k/2 \rceil) + \lfloor k/2 \rfloor & \text{if } k > 1, \\ 1 & \text{if } k = 1. \end{cases} \tag{42}$$

Equation (42) is true because, once the subproblems of size $\lceil k/2 \rceil$ are completed, the results can be used as inputs to subproblems of size larger than $\lceil k/2 \rceil$. In the following step, only subproblems of size $\lceil k/2 \rceil + 1$ can be completed, and the results will be available as inputs to subproblems of size larger than $\lceil k/2 \rceil + 1$. Thus it takes $\lfloor k/2 \rfloor$ steps to solve a subproblem of size k. In each step, two additions and two comparisons are performed.

Proposition 2. *The solution to equation* (42) *is* $T_d(N) = N$.

Proof. The proposition can be proved by induction. □

Although the above scheme is fast, it requires a large number of broadcast busses and may be difficult to implement when the problem size is large. To overcome this we can transform the nonserial problem into a serial one, that is, convert the general AND/OR-graph into a simpler graph in which all arcs connect nodes in successive levels. Suppose that an OR-node and its immediate parent are not located in adjacent levels, then the OR-node is connected to its parent via other intermediate nodes in adjacent levels. The additional connections are represented as dotted lines in Figure 8. This pipelined design is suitable for VLSI implementation because the interconnections can be mapped into a planar structure.

The computational time for the scheme in Figure 8 is analyzed here. Let $T_p(k)$, $1 \le k \le N$, be the time to find the optimal order of multiplying k matrices. Once a subproblem of size $\lceil k/2 \rceil$ is solved, it takes $\lfloor k/2 \rfloor$ time units to transfer the result into the processor that evaluates the subproblem of size k. Analogous to the explanation for equation (42), it takes $\lfloor k/2 \rfloor$

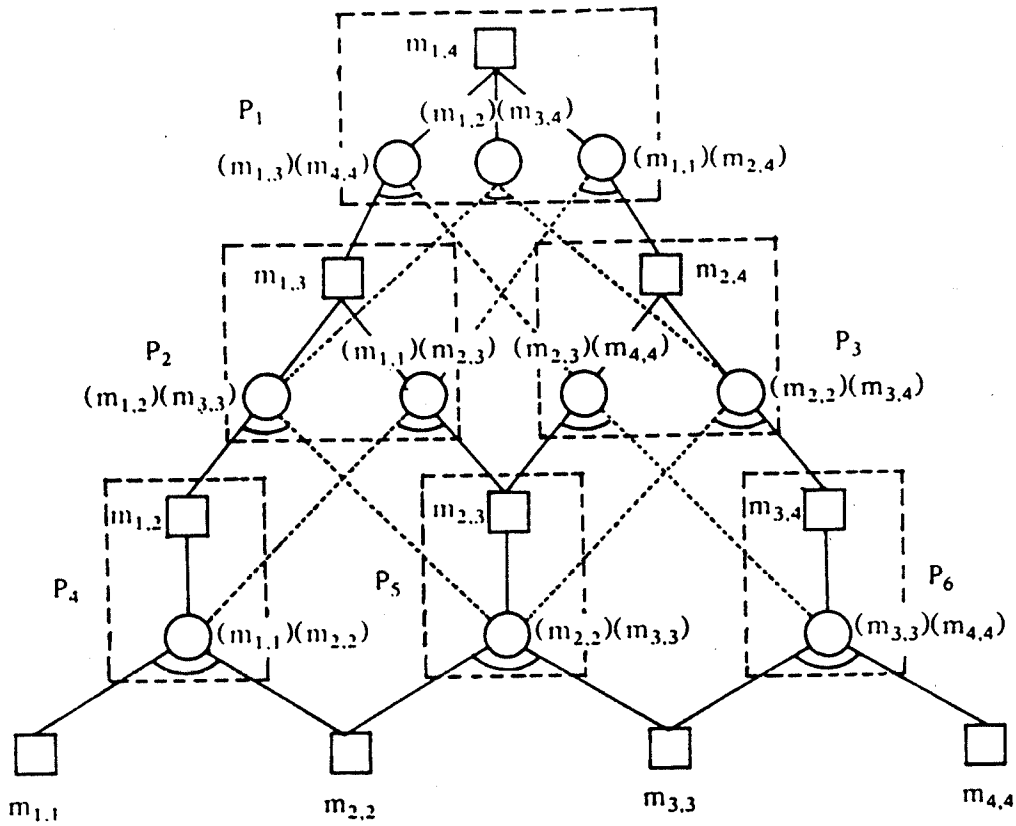

Figure 8. A structural AND/OR-graph representation of finding the optimal order of multiplying a string of four matrices.

steps to solve the subproblem of size k after the results of subproblems of size $\lceil k/2 \rceil$ are available. Consequently,

$$T_p(k) = T_p(\lceil k/2 \rceil) + 2\lfloor k/2 \rfloor. \tag{43}$$

Proposition 3. *Suppose* $T_p(1) = 2$, *then the solution to equation* (43) *is* $T_p(N) = 2N$.

Proof. The proposition can be proved by induction. □

A systolic array usually demands that all operands for an operation arrive at a processor simultaneously and that the computations are carried out in a pipelined fashion. Recall from Theorem 2 that the optimal branching factors for AND- and OR-nodes are two and m (≥ 2), respectively. Hence, it is necessary for two data items to arrive at an AND-node simultaneously, and that the OR-nodes are evaluated sequentially. Keeping the timing and Proposition 3 in mind, it is not difficult to design a systolic algorithm for this problem. In fact, the derived structure is the same as that proposed by Guibas *et al.* [11].

The above example demonstrates the relationship between an AND/OR-graph representation of polyadic DP problems and the corresponding systolic design. In general, starting from an AND/OR-graph, a systolic array with planar interconnections can be designed by first serializing links that connect nodes not in adjacent levels in the AND/OR-graph, and by designing the appropriate control signals. As shown in the examples, the transformation may introduce additional delay and redundant hardware in the implementation.

7. Conclusions

Dynamic programming is a good example to show the variant architectures required by multiple formulations. In this paper DP formulations have been classified according to the objective functions and the structure of the corresponding AND/OR-graphs. A given DP problem can usually be formulated in multiple ways, such as a folded OR-tree (multistage graph), an AND-tree, a folded AND/OR-tree, and an AND/OR-graph. Hence, it is

Table 1. Summary of results in parallel processing of dynamic programming problems.

Formulation		Problem characteristic	Suitable method	Functional requirements
Serial	Monadic	Many states or quantized values in each stage	Solve as string of matrix multiplications	Systolic processing
	Polyadic	Many stages	Solve by divide-and-conquer algorithms, or search AND/OR-trees	Loose coupling for fine grain; tight coupling for coarse grain
Nonserial	Monadic	Variables can be eliminated one by one	Transform into monadic-serial representation (by grouping variables)	Systolic processing
	Polyadic	Unstructured problems	Search AND/OR-graphs; transform into serial AND/OR-graphs	Dataflow or systolic processing

important to compare the alternative implementations. Systolic processing is most applicable when the formulation is serial.

Many sequential decision problems have serial formulations that can be considered as searching a multistage graph. If there are a large number of states and/or quantized values in each stage, then a monadic formulation is more appropriate, and the problem is efficiently solved as a serial string of matrix multiplications. On the other hand, if the number of stages is large, then the problem should be put into a polyadic formulation. The matrices are grouped into a binary tree and multiplied by a divide-and-conquer algorithm. We have found the AT^2 lower bound for multiplying a string of N $m \times m$ matrices and have proved that dividing the string into $O(N/\log_2 N)$ groups and multiplying each by a systolic array is optimal in the sense of achieving this lower bound.

When the formulation is nonserial, it may be necessary to transform the problem into a serial formulation before an efficient implementation can be found. A monadic-nonserial formulation can be transformed into a monadic-serial one by grouping state variables. A problem in a polyadic-nonserial formulation can be represented as the search of an optimal solution in an AND/OR-graph, which can be transformed into an AND/OR-graph for a serial problem by adding dummy nodes. The transformed AND/OR-graph can be mapped directly into a planar systolic array by using appropriate control signals. The additional hardware and delay introduced is problem dependent. A summary of variant architectures required by the multiple formulations of dynamic programming problems are shown in Table 1.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] G. Baudet and D. Stevenson, Optimal Sorting Algorithms for Parallel Computers, *IEEE Trans. Comput.* **27** (1), 84-87, 1978.

[3] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ, 1962.

[4] U. Bertele and F. Brioschi, A New Algorithm for the Solution of the Secondary Optimization in Nonserial Dynamic Programming. *J. Math. Anal. Appl.*, 27, 565-574, 1969.

[5] U. Bertele and F. Brioschi, *Nonserial Dynamic Programming*, Academic Press, New York, 1972.

[6] F. Brioschi and S. Even, Minimizing the Number of Operations in Certain Discrete-Variable Optimization Problems, *Oper. Res.*, **18**, 67-81, 1970.

[7] J. Casti, M. Richardson, and R. Larson, Dynamic Programming and Parallel Computers, *J. Optim. Theory Appl.*, **12** (4), 423-438, 1973

[8] M. C. Chen, Synthesizing VLSI Architectures: Dynamic Programming Solver, *Proc. Int. Conf. on Parallel Processing*, pp. 776-784, IEEE, 1986.

[9] M. Clarke and C. Dyer, Systolic Array for a Dynamic Programming Application, *Proc. 12th Workshop on Applied Imagery Pattern Recognition*, 1983.

[10] S. Gensi, U. Montanari, and A. Martelli, Dynamic Programming as Graph Searching: An Algebraic Approach, *J. Assoc. Comput. Mach.*, **28** (4), 737-751, 1981.

[11] L. J. Guibas, H. T. Kung, and C. D. Thompson, Direct VLSI Implementation of Combinatorial Algorithms, *Proc. Caltech Conf. on VLSI*, pp. 509-525, Caltech, Pasadena, CA, 1979.

[12] T. Ibaraki, Solvable Classes of Discrete Dynamic Programming, *J. Math. Anal. Appl.*, **43**, 642-693, 1973.

[13] T. Ibaraki, The Power of Dominance Relations in Branch-and-Bound Algorithms, *J. Assoc. Comput. Mach.*, **24** (2), 264-279, 1977.

[14] R. Karp and M. Held, Finite State Processes and Dynamic Programming, *SIAM J. Appl. Math.*, **15**, 693-718, 1967.

[15] D. J. Kuck, A Survey of Parallel Machine Organization and Programming, *Comput. Surveys*, **9** (1), 29-59, 1977.

[16] V. Kumar, A General Bottom-up Procedure for Searching AND/OR-Graphs, *Proc. National Conf. on Artificial Intelligence*, pp. 182-187, AAAI, 1984.

[17] H. T. Kung, Let's Design Algorithms for VLSI Systems, *Proc. Caltech. Conf. on VLSI*, pp. 65-90, Caltech, Pasadena, CA, 1979.

[18] G.-J. Li and B. W. Wah, Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms, *Proc. Int. Conf. on Parallel Processing*, pp. 473-480, IEEE, 1984.

[19] G.-J. Li and B. W. Wah, Design of Optimal Systolic Arrays, *IEEE Trans. Comput.*, **35** (1), 66-77, 1985.

[20] G.-J. Li and B. W. Wah, Optimal Granularity of Parallel Evaluation of AND-Trees, *Proc. 1986 Fall Joint Computer Conf.*, ACM-IEEE, 1986.

[21] A. Martelli and U. Montanari, Additive AND/OR-Graphs, *Proc. Int. Joint Conf. on AI, IJCAI*, 1973.

[22] T. Morin and R. Marsten, Branch-and-Bound Strategies for Dynamic Programming, *Oper. Res.*, **24** (4), 611-627, 1976.

[23] H. Ney, Dynamic Programming as a Technique for Pattern Recognition, *Proc. 6th Int. Conf. Pattern Recognition*, pp. 1119-1125, 1982.

[24] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

[25] A. Rosenthal, Dynamic Programming is Optimal for Nonserial Optimization Problems, *SIAM J. Comput.*, **11** (1), 1982.

[26] C. Savage, Parallel Algorithms for Graph Theoretic Problems, Ph.D. Dissertation, University of Illinois, Urbana, IL, 1978.

[27] P. Varman and V. Ramakrishnan, Dynamic Programming and Transitive Closure on Linear Pipeline, *Proc. Int. Conf. on Parallel Processing*, pp. 359-364, IEEE, 1984.

[28] B. W. Wah, G.-J. Li, and C. F. Yu, Multiprocessing of Combinatorial Search Problems, *Computer*, **18** (6), 93-108, 1985.

[29] D. White, *Dynamic Programming*, Oliver and Boyd, Edinburgh, 1969.