PARALLEL PROCESSING OF SERIAL DYNAMIC PROGRAMMING PROBLEMS

Guo-jie Li and Benjamin W. Wah
Department of Electrical and Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

ABSTRACT

Dynamic programming problems are classified into monadic-serial, polyadic-serial, monadic-nonserial, and polyadic-nonserial formulations. Problems in serial formulations can be implemented easily in systolic arrays, while problems in nonserial formulations may have to be transformed into serial ones before efficient implementations can be found. This paper presents parallel architectures for problems formulated in (monadic- or polyadic-) serial dynamic-programming formulations. 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. Two 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. The optimal granularity of parallel divide-and-bound algorithms is analyzed and simulated. Some transformational techniques to convert nonserial problems into serial ones are briefly discussed.

INDEX TERMS: AND/OR graph, dynamic programming, matrix multiplication, monadic, multistage graph, nonserial, parallel processing, polyadic, serial, systolic arrays.

1. INTRODUCTION

Dynamic Programming (DP) is a powerful optimization methodology that is widely applied to a large number of areas including optimal control, industrial engineering, economics, and artificial intelligence [3,5,8,21,28]. Many practical problems involving a sequence of interrelated decisions can be solved by DP efficiently. 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 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 [13,11,28].

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. Bellman, Dreyfus, White, and many others viewed DP as a multistage optimization technique, that is, reducing a single N-dimensional problem to a sequence of N one-dimensional problems [3,28]. 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 [13] and has been further developed by Ibaraki [11] and Kumar [15]. On the other hand, Gensi and Montanari have shown that formulating a DP problem in terms of polyadic functional equations is equivalent to searching for a minimum-cost solution tree in an AND/OR graph with monotone cost function [9]. 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 [20,12,16]. Lastly, nonserial DP has been shown to be optimal among all nonoverlapping comparison algorithms [5,24].

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, Kung and Thompson have proposed a VLSI algorithm for solving the optimal parenthesization problem [10]. Linear pipelines for DP have been described recently [26]. Clarke and Dyer have designed a systolic array for curve and line detection in terms of nonserial DP [8]. Wah, et al., have proposed parallel processing for branch-and-bound algorithms with dominance tests [27]. However, these studies were directed towards 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 formulations. Potential parallelism and the corresponding parallel architectures for solving serial DP problems are investigated. Generally, a problem can be expressed in both monadic and polyadic formulations, and the efficiency and costs of implementations must be compared. It has been shown that unrestricted nonserial optimization problems are NP-hard, but that problems with a favorable pattern of term interactions may be solved efficiently [5]. An approach to solve a nonserial DP problem with some structural properties is to first convert it into a serial DP problem, and to map the serial formulation into an appropriate parallel architecture. Some methods for the conversion are proposed in this paper.

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

Research supported partially by the National Science Foundation Grant ECS-80-16580 and by CIDMAC, a research unit of Purdue University, sponsored by Purdue, Cincinnati Milicron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation, and TRW.

Computer Software and Applications Conference, 1985.

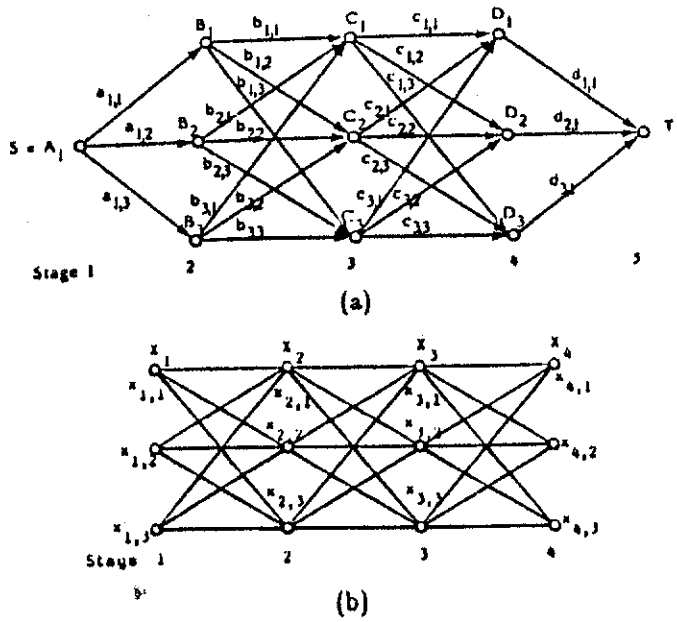

Figure 1. (a) A multistage graph with five stages and three nodes in each intermediate stage. (The cost on each edge is constant.) (b) A multistage graph with four stages (variables) and three vertices (quantized values) in each stage. (The cost on each edge is a function of the nodes connected.)

nor additive; however, they must be monotone in order for the Principle of Optimality to hold. DP formulations are classified according to the form of the functional equations and the nature of recursion.

DP problems can be solved as either the search of an optimal path in a multistage graph or as the search for an optimal solution in an AND/OR graph. We will adopt the graph search as a paradigm to illustrate the various approaches of DP. For DP problems in serial formulations, the corresponding graph representations have serial structures, and hence can be implemented easily in systolic arrays. A type of serial 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.

2.1. 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 of 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

$$f_1(i) = \min_j [c_{i,j} + f_1(j)] \tag{2.1}$$

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

$$f_2(i) = \min_j [f_2(j) + c_{j,i}] \tag{2.2}$$

This equation is termed a *backward functional equation*. The formulations in Eq's (2.1) and (2.2) are *monadic* since each cost function involves one recursive term only.

Eq's (2.1) and (2.2) can be generalized to find the optimal path from any vertex i to any other vertex j. The functional equation is

$$f_3(i,j) = \min_{k \in V_m} [f_3(i,k) + f_3(k,j)] \tag{2.3}$$

where $f_3(i,j)$ is the minimum cost of a path traversing from i to j and passing through a node in Stage V_m. This cost function is *polyadic* because it involves more than one recursive term. Examples of this kind of problems 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." For instance, according to Eq. (2.3), 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 optimization problems 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 terms of its objective function share one variable with its predecessor term (except for the first term) and another one with its successor term (except for the last term); otherwise, it is said to be nonserial. The name "serial" refers to the interaction graph to represent the problem, in which vertices stand for variables, and an edge exist between two vertices if and only if two variables belong to a 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 $x_{i,j}$ stands for the j'th 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

$$\min_{\mathbf{X}} f(\mathbf{X}) = \min_{\mathbf{X}} \sum_{i=1}^{N-1} g_i(\mathbf{X}_i, \mathbf{X}_{i+1}) \tag{2.4}$$

where $\mathbf{X}$ is the set of discrete variables $\{\mathbf{X}_1, ..., \mathbf{X}_N\}$. In Eq. (2.4), every term of the objective function has two variables that only interact with variables in the neighboring terms. Therefore, Eq. (2.4) 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 timing. 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 i'th 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 i'th 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.

$$f(\mathbf{X}) = \Phi_{i=1}^{N} g_i(\mathbf{X}') \tag{2.5}$$

where $\mathbf{X} = (\mathbf{X}_1, ..., \mathbf{X}_N)$ is a set of discrete variables, $\mathbf{X}' \subseteq \mathbf{X}$, and Φ is a monotone function relating the g_is together. For

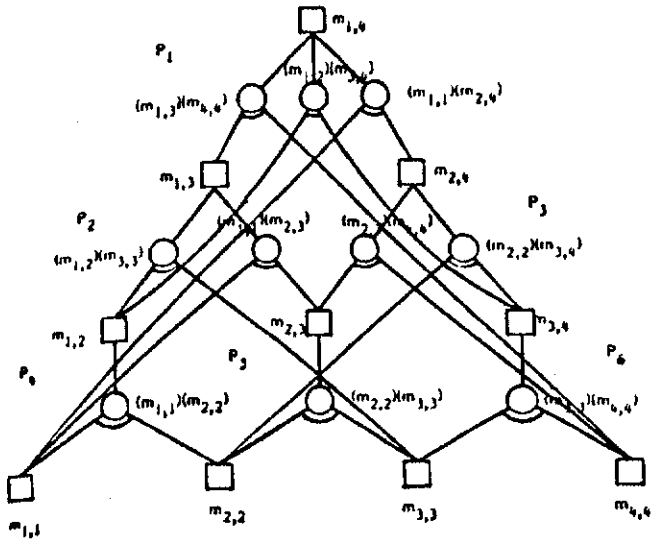

Figure 2. An AND/OR graph representation of finding the optimal order of multiplying a string of four matrices. (AND nodes are represented as circles and indicate multiplications; and OR nodes are represented as squares and indicate comparisons.)

example, the following equation is a nonserial optimization problem.

$$\min_{X} \{g_1(X_1,X_2,X_4) + g_2(X_3,X_4) + g_3(X_2,X_5)\} \qquad (2.6)$$

where $X = \{X_1, \ldots X_5\}$.

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 [19]. If this AND/OR graph has a serial structure such that arcs only exist between 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 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. Let $m_{i,j}$ be the minimum cost of computing $M_i \times \ldots \times M_j$. Clearly,

$$m_{i,j} = \min_{i \le k < j} (m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j) \qquad (2.7)$$

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 as shown in Figure 2, where the AND-nodes* denote multiplications and the OR-nodes denote comparisons. In Figure 2, the topmost node represents the original problem of multiplying four matrices. There 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 is connected to the node representing $m_{4,4}$ in the bottommost level. Similarly, the third node is connected to the node representing $m_{1,1}$ in the bottommost level. These arcs do not connect nodes in adjacent levels, hence the formulation in Eq. (2.7) 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 FORMULATIONS

Monadic-serial DP problems can be conveniently solved as the multiplication of a string of matrices. In this section, two 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 Monadic-Serial DP Problems as Strings of Matrix Multiplications

Recall that the search for a solution of a monadic-serial problem can be viewed as finding a path in a multistage graph. For the multistage graph in Figure 1(a) and from Eq. (2.2), $f(C_1)$, the minimum cost from C_1 to t, is

$$f(C_1) = \min \{c_{1,1} + d_{1,1}, c_{1,2} + d_{2,1}, c_{1,3} + d_{3,1}\} \qquad (3.1)$$

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

Eq. (3.1) is similar to an inner-product operation. If we define matrix multiplication in terms of a closed semi-ring (R, MIN, +, $+\infty$, 0) in which 'MIN' corresponds to addition and '+' corresponds to multiplication in conventional matrix multiplications [1], then Eq. (3.1) becomes

$$f(C) = C \cdot D = \begin{bmatrix} f(C_1) \\ f(C_2) \\ f(C_3) \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \begin{bmatrix} d_{1,1} \\ d_{2,1} \\ d_{3,1} \end{bmatrix} \qquad (3.2)$$

Likewise, we have

$$f(B) = B \cdot (C \cdot D)$$

$$f(A) = A \cdot (B \cdot (C \cdot D))$$

Thus solving the multistage-graph problem with a backward monadic DP formulation is equivalent to multiplying a string of matrices. The order of multiplications is reversed in forward 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 and single-sink problems, the first and last matrices degenerate into row and column vectors, respectively.

3.2. Systolic Array for String of Matrix Multiplications with Broadcasting

A linear systolic array with parallel inputs and broadcasting for evaluating monadic-serial DP problems is described in this section. The following scheme is based on multiplying a matrix with a vector. Figure 3(a) depicts a scheme for computing $(A \cdot (B \cdot (C \cdot D)))$ for the multistage graph in Figure 1(a). An *iteration* is defined as a time unit during which data are shifted or broadcast into the processing elements (PEs) and a multiply-accumulate operation is carried out in each. The iteration numbers are indicated in Figure 3(a). All input matrices are fed into the systolic array in the same format. In the first three iterations, $C \cdot D$ is evaluated. The control signal FIRST is one; D, the input vector, is broadcast to all PEs; and the intermediate results of $f(C_i)$, i=1,2,3, remain stationary.

* The definitions of AND and OR nodes used are due to Martelli and Montanari [19]. The roles of the AND and OR nodes are reversed in Nilsson's definition [22].

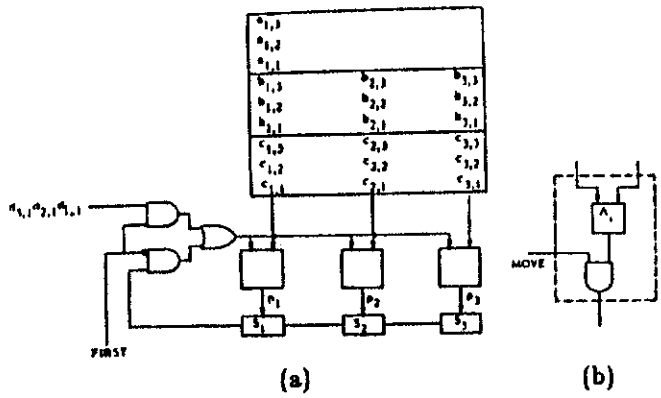

Figure 3. (a) A systolic array with broadcasts for computing a string of matrix multiplications. (b) Structure of PE P_i.

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 3(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, $B \cdot f(C) = B \cdot (C \cdot D)$ 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 A and $(f(B_i), i=1, 2, 3)$ enter PE P_1 to form the final result.

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 PEs. There is no delay between feeding successive input matrices into the systolic array, and the PEs are kept busy most of the time. In contrast, it takes $(N-2)m^2+m$ iterations to solve the problem with a single PE. 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 PEs. PU for the above systolic array is

$$PU = \frac{(N-2)m^2+m}{N \cdot m \cdot m} = \frac{N-2}{N} + \frac{1}{N \cdot m} \quad . \quad (3.3)$$

When N and m are large, PU is very close to 1. A non-broadcast version of the above design is shown elsewhere [18].

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.

3.3. Systolic Arrays for String of Matrix Multiplications with Serial Inputs

The degree of parallelism of the proposed scheme in the last section is restricted by the limited number of I/O ports in a VLSI chip and the fact that the ratio of the computational overhead to the I/O overhead is relatively low in matrix-vector multiplications. The I/O bottleneck is due to the large number of edge costs that must be fed into the systolic array. For the serial problem formulated by Eq. (2.4) and illustrated in Figure 1(b), the edge costs are expressed as functions of the nodes connected, and hence only the values of the nodes have to be input. This results in an order-of-magnitude reduction in the input overhead. In this section, we develop an efficient and practical design for this type of problems.

The search for an optimal assignment of X_is in Eq. (2.4) 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 1b. There are multiple sources and sinks, and all possible paths from any vertex in Stage 1 to any vertex in Stage N must be compared. Systolic processing is suitable when the number of quantized values in each stage is constant, and the f_is, the functions to compute edges costs, are independent of i.

To solve Eq. (2.4), the variables can be eliminated one by one. First, X_1 is considered. Since only one term, $f(X_1, X_2)$, is affected by X_1, it is sufficient to compute

$$h(X_2) = \min_{X_1} f(X_1, X_2) \quad (3.4)$$

In other words,

$$h(x_{2,j_2}) = \min_{x_{1,j_1} \in X_1} f(x_{1,j_1}, x_{2,j_2}) \qquad x_{2,j_2} \in X_2$$

The optimization problem then becomes

$$\min_X f(X) = \min_{X-\{X_1\}} \left\{ h(X_2) + \sum_{i=2}^{N-1} f(X_i, X_{i+1}) \right\} \quad (3.5)$$

If $h(X_k)$ is defined as

$$h(X_k) = \min_{X_{k-1}} \{h(X_{k-1}) + f(X_{k-1}, X_k)\} \qquad 2 \le k \le N \quad (3.6)$$

or

$$h(x_{k,j_k}) = \min_{x_{k-1,j_{k-1}} \in X_{k-1}} \{h(x_{k-1,j_{k-1}}) + f(x_{k-1,j_{k-1}}, x_{k,j_k})\}$$

$$x_{k,j_k} \in X_k, \; 2 \le k \le N$$

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, $X_1, \ldots, X_{k-1}$, the remaining optimization problem becomes

$$\min_X f(X) = \min_{X-\{X_1,\ldots,X_{k-1}\}} \left\{ h(X_k) + \sum_{i=k}^{N-1} f(X_i, X_{i+1}) \right\} \quad (3.7)$$

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

Figure 4 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 4(b).

The systolic array is initialized by zeroing all registers, H_is and K_is, and by sequentially loading input data in X_1, $x_{1,m}, \ldots, x_{1,1}$, to PEs $P_1, \ldots, P_m$. As the intermediate results are shifted out from P_3, the feedback controller feeds them back in a round-robin fashion. Referring to Figure 4(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

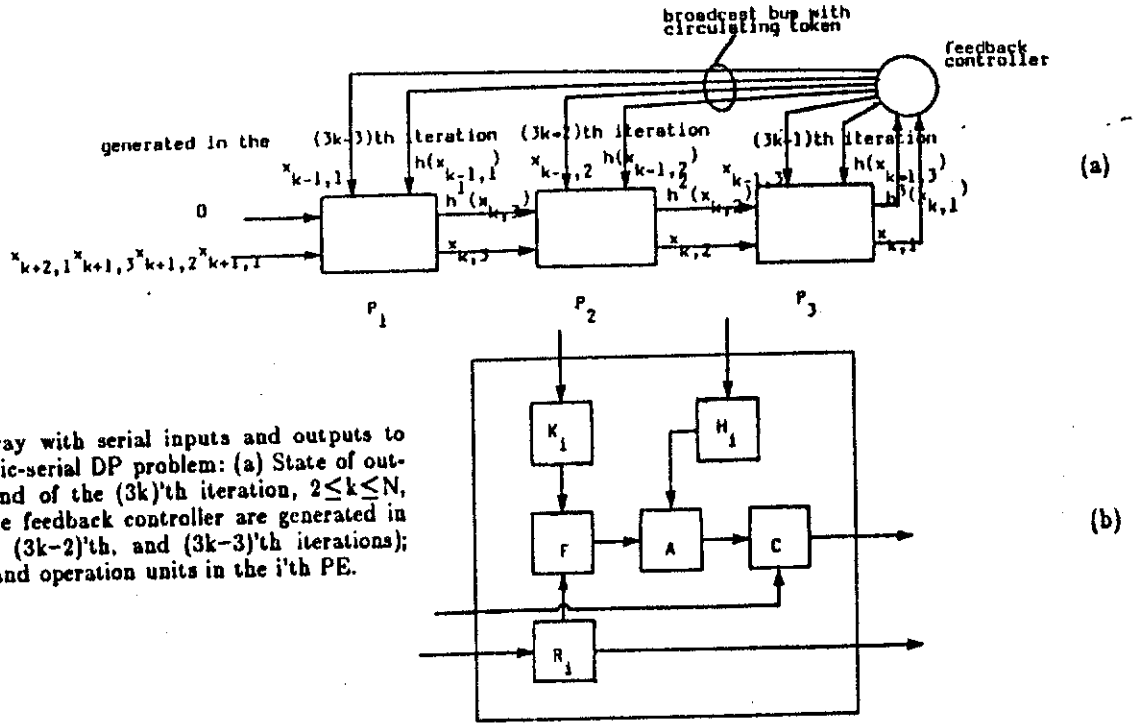

Figure 4. A systolic array with serial inputs and outputs to solve a monadic-serial DP problem: (a) State of outputs at the end of the (3k)'th iteration, $2 \le k \le N$, (outputs of the feedback controller are generated in the (3k−1)'th, (3k−2)'th, and (3k−3)'th iterations); (b) Registers and operation units in the i'th PE.

graph in Figure 1(b), the process is completed in fifteen 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 $((k-1)m+1)$'st and $(k*m)$'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 $(k*m)$'th iteration in $P_m, \ldots, P_1$. After $N*m$ iterations, $h^m(x_{N,1}), \ldots, h^1(x_{N,m})$ are obtained in $P_m, \ldots, P_1$, and the final 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 4(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 i'th word of the k'th path register. The pointers stored in the path registers are used to trace the optimal path at the end of the computation.

4. PARALLEL PROCESSING OF POLYADIC-SERIAL DP PROBLEMS

We have shown 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 in a more flexible order.

A polyadic-serial DP formulation can be solved by a divide-and-conquer algorithm, the multiplication of a string of matrices, or the search of an AND/OR graph. The last two alternatives are related since the evaluation of a set of AND-nodes and their common parent is equivalent to computing an item in a matrix multiplication, that is, an inter-product of a row vector and a column vector. In general, if the number of states in each stage is large and constant, then the matrix-multiplication method is preferable to an AND/OR-graph search, as more potential parallelism can be exploited. When the AND/OR graph is nonserial and irregular, the graph-search method is more beneficial.

4.1. Solving Polyadic-Serial DP Problems by Divide-and-Conquer Algorithms

Consider the polyadic-serial DP formulation in Eq. (2.3) for the multistage-path problem in Figure 1(a).

$$f_3(s,t) = \min_{k \in \{c1,c2,c3\}} \{f_3(s,k) + f_3(k,t)\} \tag{4.1}$$

where $f_3(i,j)$ is the cost of the optimal path from i to j, and k is a node in Stage 2 of the graph. In matrix notations, let $f_3(V_i,V_j)$ 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

$$f_3(V_i,V_j) = f_3(V_i,V_k) \cdot f_3(V_k,V_j) \tag{4.2}$$

This formulation allows a string of matrix multiplications to be reduced to two smaller strings of matrix multiplications. The substring of matrices can partitioned further until each substring contains only one matrix.

The fastest way to multiply N m-by-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

$O\left(m\lceil\log_2 N\rceil\right)$ time units with $\lceil N/2\rceil$ processors** or matrix-multiplication systolic arrays [17]. PU*** for this approach is relatively low due to the large number of idle processors.

One important issue in parallel divide-and-conquer algorithms is the *granularity of parallelism* [27]. This is the minimum size of a subproblem that is evaluated by a processor in order to achieve the optimal performance, as measured by the PU, the AT^2, or the KT^2 criteria, where A, K, T is the area of a VLSI implementation, the number of processors, and the computational time, respectively.

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 diffuse through the network. 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:

$$T(N) = \begin{cases} S(N) + 2T\left(\frac{N}{2}\right) + C(N) & N>1 \\ O(1) & N=1 \end{cases} \qquad (4.3)$$

where 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 [14,2,25]. In sorting N numbers, $S(N)+C(N)=O(N)$, and $\log_2 N$ processors should be used to maximize the PU. We have studied the *asymptotic PU* and found that $N/\log_2 N$ is a threshold when $S(N)+C(N)=O(1)$ [18]. For k (a function of N) processors, PU is one, between zero and one, or zero when the limiting ratio of k and $N/\log_2 N$ is zero, greater than zero, or approaching infinity, respectively.

Since PU increases monotonically with decreasing number of processors, it is not adequate to measure the effects of parallel processing. Another appropriate measure is the KT^2 criterion, which considers both PU and computational time. The following theorem proves the lower-bound KT^2 complexity of divide-and-conquer algorithms for solving polyadic-serial DP problems. This lower bound is attained when k(N) is $\Theta(N/(\log_2 N))$.

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

Proof: The multiplication of a string of N matrices by a divide-and-conquer algorithm can be represented in a complete binary tree with N terminals. The number of matrix-multiplications, or the number of nonterminals, is N−1. The operations are roughly divided into two phases: computation and wind-down (no start-up phase in this case). 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 $(N-1)-(K(N)-1)=N-K(N)$ matrix multiplications to be evaluated, and at least $(N/K(N)-1)\cdot T_1$ time units are needed. In the wind-down phase, the results are combined together, and some processors would be idle. According to the data dependence, at least $\log_2 K(N)\cdot T_1$ time units are required in this phase. Therefore, the following lower bound of time complexity holds.

$$T(N) \geq \left(\frac{N}{K(N)} - 1 + \log_2 K(N)\right)\cdot T_1 \qquad (4.4)$$

where $1 \leq K(N) \leq N$. For simplicity, the constant term in Eq. (4.4) can be ignored without affecting the validity of the following proof. The KT^2 lower bound is derived as

$$K(N)T^2(N) \geq \left(\frac{N^2}{K(N)} + 2N\cdot\log_2 K(N) + K(N)\cdot\log_2^2 K(N)\right)\cdot T_1^2 \qquad (4.5)$$

To find the order-of-magnitude minimum of Eq. (4.5), it is necessary to compare the following three cases. When $K(N)=\Theta(N/\log_2 N)$, $K(N)T^2(N)=\Theta((N\cdot\log_2 N)T_1^2)$. In contrast, when $K(N) < \Theta(N/\log_2 N)$, the first term on the right-hand side of Eq. (4.5) is

$$\frac{N^2}{K(N)} > \Theta(N\cdot\log_2 N) \qquad (4.6)$$

When $K(N) > \Theta(N/\log_2 N)$, the third term on the right-hand side of Eq. (4.5) is

$$K(N)\log_2^2 K(N) > \Theta(N\cdot\log_2 N) \qquad (4.7)$$

since $\log_2^2 K(N) \geq \Theta(\log_2^2 N)$. The above analysis shows that the KT^2 complexity is $\Omega((N\cdot\log_2 N)T_1^2)$, 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-by-m matrices using K processors is derived. The total time required is

$$T = T_c + T_w = \left(\left\lfloor\frac{N-1}{K}\right\rfloor + \left\lceil\log_2\left(N + K - 1 - K\cdot\left\lfloor\frac{N-1}{K}\right\rfloor\right)\right\rceil\right)\cdot T_1 \qquad (4.8)$$

where T_c and T_w represent the times in the computation and wind-down phases, respectively. The numerical evaluations of Eq. (4.8) for N=4096 is shown in Figure 5, 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 $T_w \approx \log_2 K$.

$$T \approx \frac{N}{K} - 1 + \log_2 K. \qquad (4.9)$$

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.

$$\frac{\partial KT^2(K)}{\partial K} = T^2 + 2KT\frac{\partial T}{\partial K} \qquad (4.10)$$

From Eq's (4.9) and (4.10), we get $\frac{\partial KT^2(K)}{\partial K} = 0$ *if* $(N/K-1) = \log_2 K$, i.e., $T_c = T_w$. This means KT^2 for solving

** Processors and systolic arrays are synonymous here.

*** PU refers to the utilization of the all the processors or matrix-multiplication systolic arrays and does not refer to the utilization of the PEs within a systolic array as in Section 3.

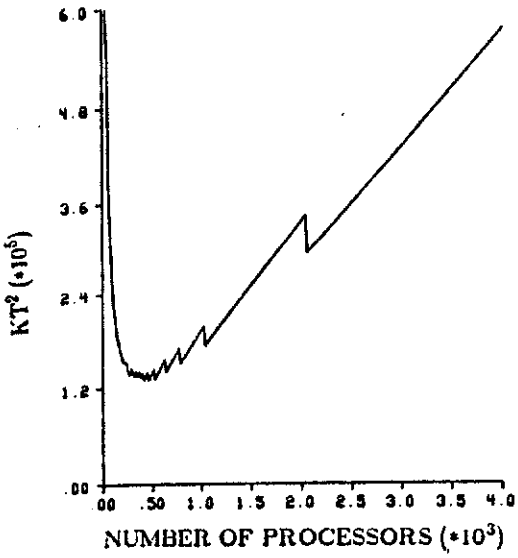

Figure 5. Simulation results of finding the optimal granularity of parallel divide-and-conquer algorithms (N = 4096).

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, Kung, and Thompson 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 [10]. 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.

4.2. Solving Polyadic-Serial DP Problems by AND/OR Graph Searches

In this section, we discuss the evaluation of polyadic-serial DP problems as AND/OR graph searches. AND/OR graphs are naturally obtained by using a problem-reduction method to represent the DP problem. The mapping of a regular and serial AND/OR graph to a systolic array is straightforward and will not be illustrated here [18].

Polyadic-serial problems are discussed with respect to the search of a multistage graph as formulated by Eq. (2.4). 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 simplity, assume that $N=p^q$, where q is a non-negative integer. The minimum cost path has to pass through one and only one vertex in Stages 0, N/p, 2N/p, ..., N in the segmented graph. The cost of a path is equal to the sum of costs of the p-subpaths. There are m^{p+1} possible combinations of subpaths from Stage 0 to Stage N that must be considered for the optimal path. 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, then the $(N+1)$-stage graph is reduced to a $(p+1)$-stage graph. By using a divide-and-conquer algorithm, each subgraph with $(N/p)+1$ stages is further divided into p smaller subgraphs. This partitioning process continues until each subgraph has only two consecutive stages.

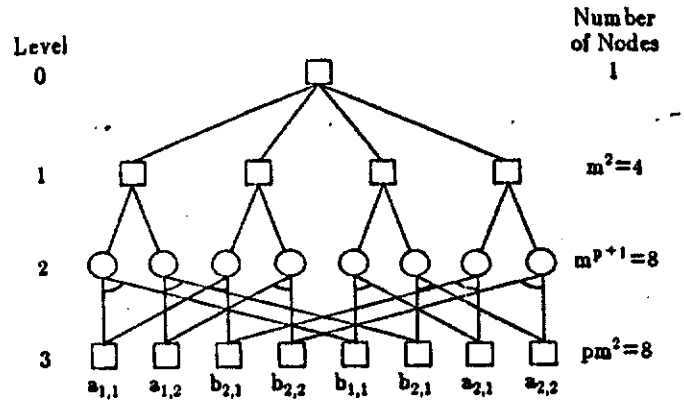

Figure 6. An AND/OR graph representation of the reduction in finding an optimal path in a 3-stage graph to a 1-stage graph. (m=2; p=2; AND nodes are represented as circles and indicate summations; and OR nodes are represented as squares and indicate comparisons. The values in the terminal nodes are edge costs: costs between Stages 1 and 2 are $a_{i,j}$, $i,j \in \{1,2\}$; costs between Stages 2 and 3 are $b_{i,j}$.)

The partitioning process can be conveniently represented in an AND/OR graph, in which an AND-node corresponds to a summation, and an OR-node corresponds to alternative selections or comparisons. In this case, we have a regular and serial 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 6 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 represent the four possible alternate paths in the reduced two-stage graph. The shortest path is obtained by comparing the costs of these paths.

The relationship between DP and graph search was investigated by Martelli and Montanari [19] 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 [23]. For those acyclic AND/OR graphs with positive arc costs, Martelli and Montanari have named them as additive [19] and have proposed top-down and bottom-up search algorithms. A similar algorithm, called AO*, for searching hypergraphs was discussed by Nilsson [22].

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 serial and regular AND/OR graph, then binary partitioning, namely, using 2-arc AND nodes, is optimal in the sense of minimizing the total number of nodes in the AND/OR graph.

The proof is omitted here and can be found elsewhere [18]. The reasonableness of this theorem can be interpreted intuitively. 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. In contrast, in the case of binary partitioning, unnecessary computations are pruned by comparisons in the AND/OR graph.

For multistage-graph problems represented in an irregular but serial AND/OR graph, the number of nodes in the graph depends on the ordering of stage reduction. However, it is not difficult to demonstrate that binary partitioning is also optimal in this case. Assume that Stages $i_1,...,i_4$ with $m_1,...,m_4$ nodes are to be reduced to two stages i_1 and i_4. If 3-arc AND-nodes are used, then $m_1 \cdot m_2 \cdot m_3 \cdot m_4$ 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 only one addition is needed for each AND-node.

5. SOLVING NONSERIAL DP PROBLEMS

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, to implement nonserial DP problems by systolic processing, they may have to be transformed into the corresponding serial formulations before they are implemented. This transformation is possible if the nonserial problems have some special structures.

One way to convert a nonserial formulation into a serial one is to combine several primary variables into a new variable. The method is illustrated by the following example on monadic-nonserial problems. For instance, let $V = \{V_1,...,V_N\}$ be a set of discrete variables and the objective function be

$$f(V) = g_1(V_1,V_2,V_3) + g_2(V_2,V_3,V_4) + ... + g_{N-2}(V_{N-2},V_{N-1},V_N) \quad (5.1)$$

To transform this into a serial formulation, the variables are eliminated one by one as follows. Let $h_1(V_2,V_3) = \min_{V_1} g_1(V_1,V_2,V_3)$. Then

$$\min_V f(V) = \min_{V-\{V_1\}} \left\{ h_1(V_2,V_3) + \sum_{i=2}^{N-2} g_i(V_i,V_{i+1},V_{i+2}) \right\} \quad (5.2)$$

If $h_k(V_{k+1},V_{k+2})$ is defined as

$$h_k(V_{k+1},V_{k+2}) = \min_{V_k} \{ h_{k-1}(V_k,V_{k+1}) + g_k(V_k,V_{k+1},V_{k+2}) \} \quad (5.3)$$

then it represents the minimum of the summation of the first k terms of $f(V)$. After eliminating k variables, $V_1,...,V_k$, the remaining optimization problem becomes

$$\min_V f(V) = \min_{V-\{V_1,...,V_k\}} \left\{ h_k(V_{k+1},V_{k+2}) + \sum_{i=k+1}^{N-2} g_i(V_i,V_{i+1},V_{i+2}) \right\} \quad (5.4)$$

The variables are, therefore, eliminated in the order $V_1,...,V_N$. Here, variables V_i and V_{i+1} are treated as a single variable in a combined stage V_i'. If m_k, $1 \leq k \leq n$, quantized values are allowed for V_k, then there would be $m_i \cdot m_{i+1}$ quantized values in the combined stage V_i', and $m_{i-1} \cdot m_i \cdot m_{i+1}$ steps are required to eliminate V_{i-1}, in which 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 V_{N-1} and 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 Eq. (5.1) is

$$\sum_{k=1}^{N-2} (m_k \cdot m_{k+1} \cdot m_{k+2}) + m_{N-1} \cdot m_N \quad (5.5)$$

In short, the monadic-nonserial problem in Eq. (5.1) is solved from the following serial problem.

$$\min_V f(V) = \min_V \left\{ g_1'(V_1',V_2') + g_2'(V_2',V_3') + ... + g_{N-2}'(V_{N-2}',V_{N-1}') \right\} \quad (5.6)$$

where the new variable V_i' is combined from V_i and 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. However, the potential parallelism is higher, and there is no increase in delay in processing the transformed problem.

Another way to solve nonserial problems is to convert the nonserial AND/OR graph into a serial graph by adding dummy nodes such that all arcs connect nodes in adjacent stages. In this case, the nonserial DP problems are characterized by high-order recursive functions. The method of adding dummy nodes essentially adds new functions to the problem, which converts the high-order recursion to a linear recursive function. For instance, in Figure 2, the bottommost OR-nodes can be connected to their parents via other intermediate AND-nodes in adjacent levels. This transformed AND/OR graph is suitable for VLSI implementation since the interconnections can be mapped into a planar structure. However, the transformation may introduce additional delay and redundant hardware. A systolic array for finding the optimal order of multiplying a string of matrices is designed by the systematic method mentioned above [18].

6. CONCLUSIONS

Dynamic programming formulations have been classified according to the structure of the functional equations. A given problem can usually be formulated in multiple ways, hence it is important to compare the alternative implementations. The applicability of systolic processing is most suitable when the formulation is serial.

Many sequential decision problems have serial formulations that can be considered as searching multistage graphs. 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. Two efficient and practical systolic arrays have been developed. 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 KT^2 lower bound for multiplying a string of N m-by-m matrices, where K is the number of processors. It is shown that dividing the string into $\Theta(N/\log_2 N)$ groups is optimal in the sense of achieving this lower bound.

When the formulation is nonserial, it might have to be transformed into a serial one before an efficient design can be found. A monadic-nonserial formulation can be transformed into a monadic-serial one by grouping state variables. A polyadic-nonserial problem 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.

REFERENCES

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

[2] G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. on Computers*, Vol. C-27, No. 1, pp 84-87, Jan. 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 Non-serial Dynamic Programming," *J. of Math. Analysis and Applications*, Vol. 27, pp. 565-574, 1969.

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

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

[7] J. Casti, M. Richardson, and R. Larson, "Dynamic Programming and Parallel Computers," *J. of Optimization Theory and Applications*, Vol. 12, No. 4, pp. 423-438, 1973.

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

[9] S. Gensi, U. Montanari, and A. Martelli, "Dynamic Programming as Graph Searching: An Algebraic Approach," *J. of the ACM*, Vol. 28, No. 4, pp. 737-751, 1981.

[10] L. Guibas, H. T. Kung, and C. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Caltech Conf. VLSI: Architecture, Design, Fabrication*, Caltech, CA, pp. 509-525, 1979.

[11] T. Ibaraki, "Solvable Classes of Discrete Dynamic Programming," *J. of Mathematical Analysis and Applications*, Vol. 43, pp. 642-693, 1973.

[12] T. Ibaraki, "The Power of Dominance Relations in Branch-and-Bound Algorithms," *J. of the ACM*, Vol. 24, No. 2, pp. 264-279, 1977.

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

[14] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys*, Vol. 9, No. 1, pp. 29-59, March 1977.

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

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

[17] G.-J. Li and B. W. Wah, "The Design of Optimal Systolic Algorithms," *IEEE Trans. on Computers*, Vol. C-34, No. 1, pp. 66-77, Jan. 1985.

[18] G.-J. Li and B. W. Wah, "Parallel Processing for Dynamic Programming Problems," *Proc. Int'l Conf. on Parallel Processing*, 1985.

[19] A. Martelli and U. Montanari, "Additive AND/OR Graphs," *Proc. Int'l Conf. on Artificial Intelligence*, pp. 1-11, 1973.

[20] T. Morin and R. Marsten, "Branch-and-Bound Strategies for Dynamic Programming," *Operations Research*, Vol. 24, No. 4, pp. 611-627, 1976.

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

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

[23] J. Pearl, *Heuristics: Intelligent Search Strategy for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.

[24] A. Rosenthal, "Dynamic Programming is Optimal for Nonserial Optimization Problems," *SIAM J. Computing*, Vol. 11, No. 1, 1982.

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

[26] P. Varman and V. Ramakrishnan, "Dynamic Programming and Transitive Closure on Linear Pipelines," *Proc. Int'l Conf. on Parallel Processing*, pp. 359-364, 1984.

[27] B. W. Wah, G.-J. Li, and C. F. Yu, "Multiprocessing of Combinatorial Search Problems", *IEEE Computer*, pp. 93-108, June 1985.

[28] D. White, *Dynamic Programming*, Oliver & Boyd, Edinburgh, England, 1969.