SYSTOLIC PROCESSING FOR DYNAMIC PROGRAMMING PROBLEMS

Guo-jie Li and Benjamin W. Wah
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

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. An efficient systolic array is presented. For solving a polyadic-serial problem, the optimal architecture consists of an interconnection of $N/(\log_2 N)$ systolic arrays, where N is the number of stages in the problem. A second method to solve a polyadic-serial dynamic programming problem is to view it as the search of optimal solutions in a serial AND/OR graph. Efficient methods of mapping a regular AND/OR graph into systolic arrays are developed. Cases are studied for transforming a problem in a nonserial formulation into a serial one.

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 applicable to a large number of areas including optimal control, industrial engineering, and economics. 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 decision 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 [21,12,17]. Lastly, nonserial DP has been shown to be optimal among all nonoverlapping comparison algorithms [5,23].

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 into monadic-serial, polyadic-serial, monadic-nonserial, and polyadic-nonserial. Potential parallelism and the corresponding systolic architectures are investigated for each class. Generally, a problem can be expressed in different DP formulations, and the efficiency and costs of implementation must be compared.

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 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.

Research supported by CIDMAC, a research unit of Purdue University, sponsored by Purdue, Cincinnati Milicron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation and TRW.

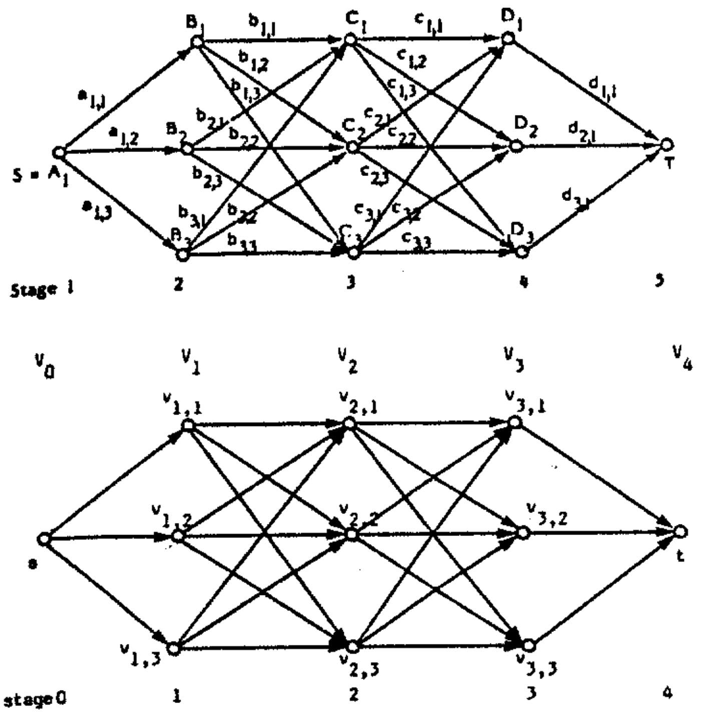

Figure 1. (a) A multistage graph with five stages and three nodes in each intermediate stage. (The cost on each edge is constant.) (b) A multistage graph with four stages and three vertices in each stage. (The cost on each edge is a function of the nodes connected.)

2. CLASSIFICATION OF DYNAMIC PROGRAMMING FORMULATIONS

DP formulation is characterized by a recursive functional equation whose left-hand side identifies a function name and whose right-hand side is an expression containing the maximization (or minimization) of values of some monotone functions. DP formulation can be classified according to the form of the functional equation.

2.1. Monadic versus Polyadic DP Formulations

For a general graph $G = (V, E)$, in which V is the set of nodes and E is the set of edges, let $c_{i,j}$ be the cost of edge $(i,j) \in E$. 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*, that is, the 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:

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

where $f_3(i,j)$ is the minimum cost of traversing from i to j. This cost function is *polyadic* because it involves more than one recursive term.

Polyadic DP formulations correspond to tree structures in which each recursive term is a subtree. 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."

2.2. Serial versus Nonserial DP Formulations

In a multistage graph, suppose $V = V_0 \cup ... \cup V_N$, where V_i, $0 \le i \le N$, is the set of nodes in Stage i, $V_0 = \{s\}$, $V_N = \{t\}$, and v_{i,j_i} is the j'th node in V_i. An example is shown in Figure 1(b). Let P be the set of all possible paths from s to t. A path $p \in P$ from s to t is of the form $s\text{-}v_{2,j_2}\text{-}v_{3,j_3}\text{-}...\text{-}t$. If the cost of edge $(v_{i,j_i}, v_{i+1,j_{i+1}})$ is $g_i(v_{i,j_i}, v_{i+1,j_{i+1}})$, then the minimum-cost path from s to t is:

$$\min_{p\in P} f(p) = \min_{p\in P} \sum_{i=0}^{N-1} g_i(v_{i,j_i}, v_{i+1,j_{i+1}}) \tag{2.4}$$

Optimization problems of the above form have terms that share one variable with its predecessor term and another one with its successor term. The problem has a serial structure and, consequently, is called a *serial optimization problem*. Many practical problems can be formulated in this way. For a traffic-control problem, V_i can be the possible times for the traffic light to be in state i, and the cost on an edge is the difference in timing. For a circuit-design problem, V_i can be the possible voltages at point i, and the cost of an edge may be the corresponding power dissipation. For a fluid-flow problem, V_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, V_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 are special cases of serial optimization problems.

A general *nonserial optimization problem* has the following form:

$$\min_V f(V) = \min_V \left[\Phi_{i=0}^{N} g_i(\Xi_i) \right] \qquad \Xi_i = \bigcup_{\substack{\ell \in L \\ L \subseteq (0,...,N)}} V_\ell \tag{2.5}$$

where Φ is a monotone function relating the g_is together. For example, the following equation is a nonserial optimization problem:

$$\min_{v_j \in V_j} \{g_1(v_1,v_2,v_4) + g_2(v_3,v_4) + g_3(v_3,v_5)\} \qquad V = \bigcup_{i=1}^{5} V_i \tag{2.6}$$

We have classified DP problems in terms of the recursive functional equations. Monadic and polyadic DP formulations are distinct approaches to representing various optimization problems, while serial and nonserial optimization problems are problems solvable by the DP formulation. We remark that the cost functions, g_i, are not restricted to be monadic, serial, and additive; however, they must be monotone in order for the Principle of Optimality to hold.

In the following sections, we investigate systolic architectures for the four classes of DP formulations. Although a given problem can be formulated in several ways, a systolic implementation favors the serial formulation, and a problem in a nonserial formulation may have to be transformed into a serial one before it can be mapped into systolic arrays.

3. SYSTOLIC ARRAY FOR MONADIC-SERIAL DP FORMULATIONS

Monadic-serial DP problems can be conveniently solved as the multiplication of a string of matrices. In this section, an efficient systolic design is presented. The proposed design does 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 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 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], Eq. (3.1) becomes:

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

Likewise, we have:

$$f(\mathbf{B}) = \mathbf{B}\cdot(\mathbf{C}\cdot\mathbf{D})$$

$$f(\mathbf{A}) = \mathbf{A}\cdot(\mathbf{B}\cdot(\mathbf{C}\cdot\mathbf{D}))$$

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 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

A linear systolic array for evaluating problems in monadic-serial DP formulations with parallel inputs is described in this section. The following scheme is based on a combination of two methods of multiplying a matrix with a vector, one of which was discussed by Kung [16]. Figure 2(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. Note that the same iteration number are carried out at different times in different processors. The iteration numbers are indicated in Figure 2(a). In the first three iterations, $\mathbf{C}\cdot\mathbf{D}$ is evaluated. The control signal FIRST is one; **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 **B** is transposed, and the i'th column of Matrix **B** is fed into P_i. The input vector, $\{f(C_i), i=1,2,3\}$, 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 2(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 2(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 matrix multiplications is odd, 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 matrix multiplications is even, ODD_i is zero, A_i is connected to the output, and the result vector is shifted. At

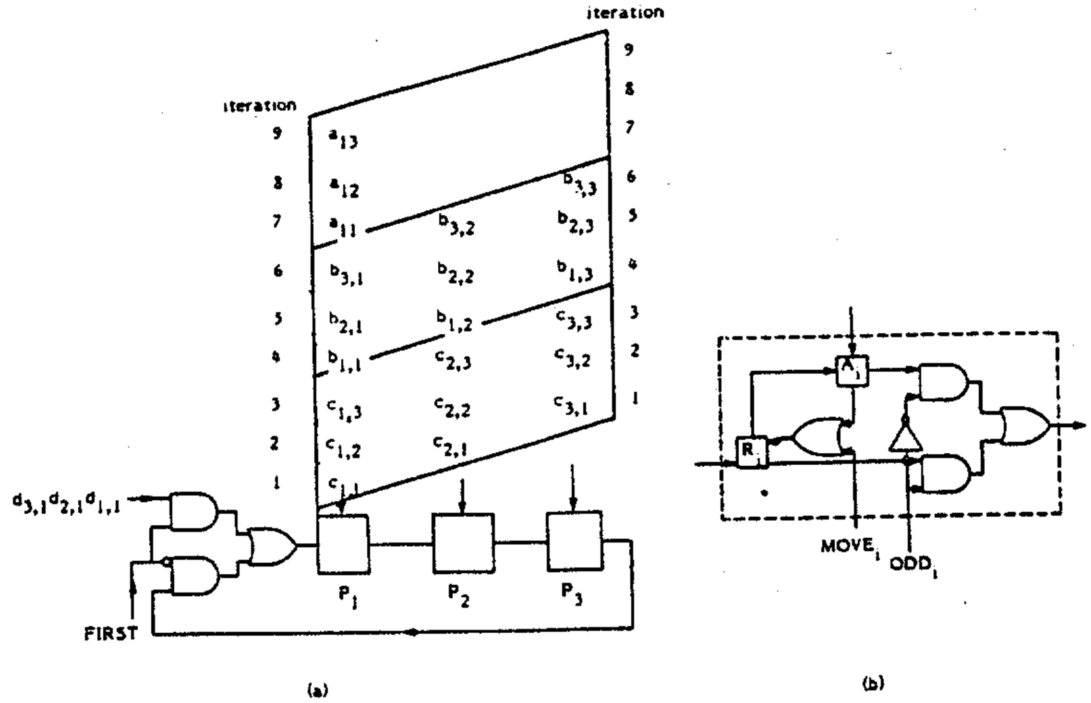

Figure 2. (a) A pipelined version of systolic array for computing a string of matrix multiplications; (b) Processor structure for P_i.

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 for 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·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:

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

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.

The degree of parallelism of the proposed scheme is 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 systolic large number of edge costs that must be fed into the systolic array. For the serial optimalization problems 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. An efficient and practical scheme with serial inputs has been developed [19], but is not shown here due to space limitation. In the improved scheme, the parallel input ports are replaced by a feedback bus, and only one serial input port is needed.

4. PARALLEL PROCESSING FOR POLYADIC-SERIAL DP FORMULATIONS

A polyadic-serial DP formulation can be solved by either a divide-and-conquer algorithm, the multiplication of a string of matrices, or the search of an AND/OR graph. The last two representations 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. Divide-and-conquer algorithms will be discussed in Section 4.1, and the last two formulations will be discussed in Section 4.2.

The most efficient representation for a given problem is usually problem dependent. 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 irregular, especially for nonserial DP problems, then the graph-search method is more beneficial.

4.1. 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.

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[1] in order to achieve the optimal performance, as measured by either the PU or the AT^2 (or (area) × (computational time)^2) criteria. If the granularity is large, then the processors can be loosely coupled; otherwise, tight coupling is necessary. In this section, the lower-bound complexity for solving polyadic-serial DP problems by parallel divide-and-conquer algorithms is derived with respect to the asymptotic PU and AT^2 criteria. The optimal numbers of processors to achieve these lower bounds are also studied.

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

$$f_3(s,t) = \min_{k \in \{v_{2,1}, v_{2,2}, v_{2,3}\}} \{f_3(s,k) + f_3(k,t)\} \tag{4.1}$$

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.

For simplicity in deriving the lower bound, all the matrices are assumed to have identical dimensions.

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 $\lceil N/2 \rceil$ matrix-multiplication systolic arrays [18]. The PU and AT^2 measures are relatively low for this approach.

It has been found that the processors are efficiently utilized if $\lceil N/(\log_2 N) \rceil$ processors are used [14,2,24]. However, the asymptotic effects of deviations from $\lceil N/(\log_2 N) \rceil$ processors have not been studied before. This is shown in 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, each of which computes a matrix multiplication. (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.)

By defining $e_\infty = \lim_{N \to \infty} \frac{k(N)}{N/\log_2 N}$, the normalized asymptotic processor utilization is:

$$\lim_{N \to \infty} PU(k,N) \begin{cases} = 0 & \text{if } e_\infty \to \infty \\ \geq \frac{1}{1+e_\infty} & \text{if } 0 < e_\infty < \infty \\ = 1 & \text{if } e_\infty = 0 \end{cases} \tag{4.3}$$

The proof is omitted due to space limitation [19]. □

As an example in applying the results of Proposition 1, suppose that there are $\sqrt{N}$ processors, it is easy to show that $\lim_{N \to \infty} \sqrt{N} \log_2 N / N = 0$. Hence $e_\infty = 0$ in Eq. (4.3), and $\lim_{N \to \infty} PU(k,N) = 1$.

[1] Processors and systolic arrays are synonymous here.

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 $k(N) = N/(\log_2 N)$.

Theorem 1: Suppose that a string of N m-by-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-by-m matrices in T_1 time units. Then $S(N)T^2(N) \geq \Theta(N \log_2 N)T_1^2$, and equality holds when $S(N)=\Theta(N/\log_2 N)$. (Θ 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 roughly be 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.

$$T(N) \geq \left[\frac{N}{S(N)} - 1 + \log_2 S(N)\right]\cdot T_1 \tag{4.4}$$

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

$$S(N)T^2(N) \geq \left[\frac{N^2}{S(N)} + 2N\log_2 S(N) + S(N)\log_2^2 S(N)\right]\cdot T_1^2 \tag{4.5}$$

To find the order-of-magnitude minimum of Eq. (4.5), 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 Eq. (4.5) is:

$$\frac{N^2}{S(N)} > \Theta(N\log_2 N) \tag{4.6}$$

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

$$S(N)\log_2^2 S(N) > \Theta(N\log_2 N) \tag{4.7}$$

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. □

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 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 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 $N=p^\ell$, where ℓ is a non-negative integer. The minimum-cost path has to pass through one and only one vertex in Stage 0, N/p, ..., p N/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 i N/p to the m vertices in Stage (i+1) N/p, $0 \leq i \leq p-1$, 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 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.[2] 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 3 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.

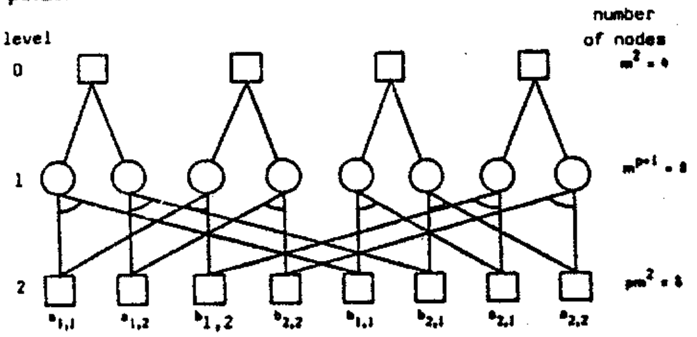

Figure 3. An AND/OR graph representation of the reduction in finding an optimal path in a 3-stage graph to a 1-stage graph. (m=2; p=2; AND nodes are represented as circles and indicate summations; and OR nodes are represented as squares and indicate comparisons. The values in the terminal nodes are edge 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 relationship between DP and graph search was investigated by Martelli and Montanari [20] 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

[2] The definitions of AND and OR nodes are taken from [20]. The roles of the AND and OR nodes are reversed in Nilsson's definition [22].

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 [20], 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 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 3 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 $N+1$ $(=p^{l}+1)$ 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:

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

$$= \frac{N-1}{p-1} m^{p+1} + \frac{N \cdot p-1}{p-1} m^2 \tag{4.8}$$

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

$$\frac{\partial u(p)}{\partial p} = \frac{(N-1)(m^{p+1}((p-1)\log_e m - 1) - m^2)}{(p-1)^2} \tag{4.9}$$

From Eq. (4.9), it is seen that $\frac{\partial u(p)}{\partial p} \geq 0$ if $N \geq 1$, $p \geq 2$, and $m \geq 3$, or $N \geq 1$, $p \geq 3$, and $m \geq 2$. Considering $\frac{\partial^2 u(p)}{\partial p^2}$, 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.

5. PARALLEL PROCESSING OF NONSERIAL DP PROBLEMS

For nonserial optimization problems, it is more efficient to transform the problem into a serial formulation before implementing it in systolic arrays. In this section, we illustrate the transformation by some examples.

5.1. Monadic-Nonserial DP Formulations

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]. An approach to solve a monadic-nonserial problem with some structural properties is to first 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 Eq. (2.5), a multistage optimization procedure can be carried out separately for each variable V_i. Of course, this optimization must be performed on all values of the independent variables that "interact" with V_i. Let $V_{i_1}, \ldots, V_{i_s}$ be the variables that are related to V_i in one or more functional terms. The cost function can be written as:

$$f(v_1, \ldots, v_n) = \min_{v_k \in V_k} \left\{ h_1(v_i, v_{i_1}, \ldots, v_{i_s}) + h_2(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \right\} \tag{5.1}$$

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

$$f(v_1, \ldots, v_n) = \min_{V-V_i} \left\{ h_{1,opt}(v_{i_1}, \ldots, v_{i_s}) + h_2(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \right\} \tag{5.2}$$

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. Eq. (5.2) can be treated as a monadic-serial form if the evaluation of $h_{1,opt}$ is done separately.

The method is illustrated by the following example. For instance, if $V = V_1 \cup \ldots \cup V_N$, and the objective function is:

$$f(V) = \min_{v_j \in V_j} \left\{ g_1(v_1, v_2, v_3) + g_2(v_2, v_3, v_4) + \ldots + g_{N-2}(v_{N-2}, v_{N-1}, v_N) \right\} \tag{5.3}$$

Let $h_1(v_2, v_3) = \min_{v_1 \in V_1} g_1(v_1, v_2, v_3)$, we have:

$$\min_{V} f(V) = \min_{V-V_1} \left\{ h_1(v_2, v_3) + \sum_{i=2}^{N-2} g_i(v_i, v_{i+1}, v_{i+2}) \right\} \tag{5.4}$$

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

$$h_k(v_{k+1}, v_{k+2}) = \min_{v_k \in V_k} \{ h_{k-1}(v_k, v_{k+1}) + g_k(v_k, v_{k+1}, v_{k+2}) \} \tag{5.5}$$

Eq. (5.5) represents the minimum of the summation of the first k terms of f(V). After eliminating k variables, $V_1, \ldots, V_k$, the remaining optimization problem becomes:

$$\min_{V} f(V) = \min_{V-(V_1 \cup \ldots \cup V_k)} \left\{ h_k(v_{k+1}, v_{k+2}) + \sum_{i=k+1}^{N-2} g_i(v_i, v_{i+1}, v_{i+2}) \right\} \tag{5.6}$$

The monadic DP procedure, thus, eliminates the variables in the order $V_1, \ldots, V_N$. If the variables V_2 and V_3 are treated as a single variable in a stage, and m_k, $1 \leq k \leq n$, quantized values are allowed for V_k, 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 V_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. (5.3) is:

$$\sum_{k=1}^{N-2}(m_k \cdot m_{k+1} \cdot m_{k+2}) + m_{N-1} \cdot m_N \qquad (5.7)$$

Although more operations are needed for monadic-nonserial DP problems than monadic-serial DP problems, the potential parallelism is higher. Further, there is no increase in delay in evaluating the transformed problem. With additional control, the linear systolic array presented earlier can be applied to evaluate monadic-nonserial DP problems.

5.3. Solving Polyadic-Nonserial DP Problems by Parallel AND/OR-Graph Search

AND/OR graphs can be sequentially searched in a breadth-first bottom-up fashion, which expands nodes by levels from the bottom up [22]. 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, so a processor can be dynamically assigned when it is free, or can be designed with a limited interconnection, so 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 search for polyadic-serial DP problems is a special case of that for 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, $1 \leq i \leq 4$, 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,

$$m_{i,j} = \min_{i \leq k < j}(m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j) \qquad (5.8)$$

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

The AND/OR graph in Figure 4 can be mapped directly into six processors connected by multiple broadcast buses. 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 $T_d(k)$, $1 \leq k \leq N$, be the time to find the optimal order of multiplying k matrices. Then,

$$T_d(k) = \begin{cases} T_d(\lceil k/2 \rceil) + \lfloor k/2 \rfloor & \text{if } k > 1 \\ 1 & \text{if } k = 1 \end{cases} \qquad (5.9)$$

This is true because, once the subproblems of size $\lceil k/2 \rceil$ are

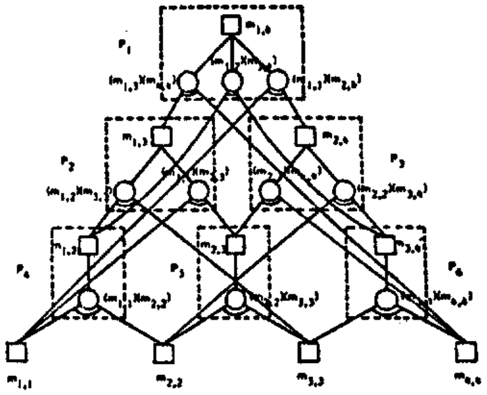

Figure 4. An AND/OR graph representation of finding the optimal order of multiplying a string of four matrices

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 Eq. (5.9) 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 buses 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 5. This pipelined design is suitable for VLSI implementation because the interconnections can be mapped into a planar structure.

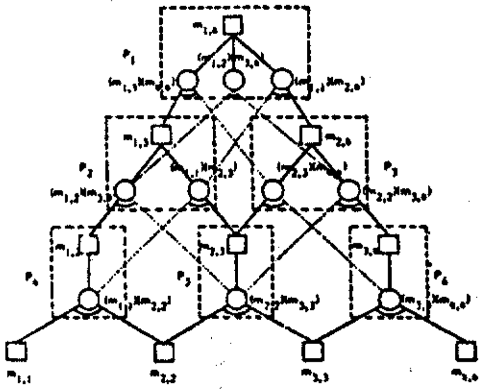

Figure 5. An AND/OR graph representation of finding the optimal order of multiplying a string of four matrices. (Communications are restricted to OR-nodes in adjacent levels of the graph and are indicated by dotted lines).

The computational time for the scheme in Figure 5 is analyzed here. Let $T_p(k)$, $1 \leq k \leq N$, 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 Eq. (5.9), it takes $\lfloor k/2 \rfloor$ steps to solve the subproblem of size k after the results of subproblems of size $\lceil k/2 \rceil$ are available. Consequently,

$$T_p(k) = T_p(\lceil k/2 \rceil) + 2\lfloor k/2 \rfloor \qquad (5.10)$$

Proposition 3: Suppose $T_p(1)=2$, then the solution to Eq. (5.10) 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, Kung and Thompson [10].

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.

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 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, 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-by-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.

We have not considered the use of branch-and-bound algorithms with dominance tests to solve dynamic programming problems. They are more suitable when a multiprocessor with a flexible interconnection or a dataflow system is used.

REFERENCES

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

[2] G. Baudet, G. 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, N.J., 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, New York, 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, 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," *JACM*, Vol. 28, No. 4, pp. 737-751, 1981.

[10] Guibas, L., H. Kung and C. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Conf. VLSI: Architecture, Design, Fabrication*, Caltech, 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," *JACM*, 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. AAAI*, pp. 182-187, 1984.

[16] H. T. Kung, "Lets Design Algorithms for VLSI Systems," *Caltech Conf. on VLSI, Jan. 1979.*

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

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

[19] G.-J. Li and B. W. Wah, *Parallel Processing for Dynamic Programming Problems*, Tech. Rep., Purdue University, Jan. 1985.

[20] A. Martelli and U. Montanari, "Additive AND/OR Graphs," *IJCAI*, pp. 1-11, 1973.

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

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

[23] A. Rosenthal, "Dynamic Programming is Optimal for Nonserial optimization problems," *SIAM J. Computing*, Vol. 11, No. 1, 1982.

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

[25] J. E. Savage, "Area-Time Tradeoffs for Matrix Multiplication and Related Problems in VLSI Models," *J. Computer and System Sciences*, Vol. 20, No. 3, pp. 230-242, 1981.

[26] P. Varman and V. Ramakrishnan, "Dynamic Programming and Transitive Closure on Linear Pipelines," *Proc. 1984 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*, June 1985.

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