JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 12, 213–222 (1991)
Vol. 12, No. 3, July 1991

A Highly Parallel Algorithm for Multistage Optimization Problems and Shortest Path Problems

JOHN K. ANTONIO

School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907

WEI K. TSAI

Department of Electrical Engineering, University of California, Irvine, Irvine, California 92717

AND

G. M. HUANG*

Department of Electrical Engineering, Texas A&M University, College Station, Texas 77843

It appears that all of the known algorithms for solving multistage optimization problems are based explicitly on standard dynamic programming concepts. Such algorithms are inherently serial in the sense that computation must be completed at the current stage before meaningful computation can begin at the next stage. In this paper we present a technique which recursively divides the original problem into a set of smaller problems which can be solved in parallel. This technique is based on the recursive application of a simple aggregation procedure. For a multistage process with n stages, we show that our new algorithm can achieve a time complexity of $O(\log n)$. In contrast, competing algorithms based exclusively on the standard dynamic programming technique can only achieve a time complexity of $\Theta(n)$. The new algorithm is designed to operate on a tightly coupled parallel computer. As some important applications, it is shown that our algorithm can serve as a fast and efficient means of decoding convolutional codes, solving shortest path problems, and determining minimum-fuel flight paths.

1. INTRODUCTION

The objective of the multistage optimization problem is to determine the control policy (i.e., decision) at each stage of a multistage process so as to yield a minimum overall cost. The overall cost is defined to be the sum of all costs associated with decisions made at each intermediate stage. The key aspect of such problems is that decisions at a given stage cannot be viewed completely independently of decisions of other stages, since a balance must be struck between the desire to obtain a low cost at the present stage and the possibility of high future costs that may be unavoidable.

Determining the minimum-fuel altitude path between two locations (a problem faced by the airline industry) is an example application which requires the solution of a multistage optimization problem. It is well known that substantial savings in fuel cost can be obtained by using strong winds and other factors efficiently. For simplicity, assume that only the altitude of an otherwise fixed flight path is to be controlled (i.e., the other parameters such as speed and direction are fixed). The flight path is finitely discretized into n stages or "checkpoints." It is assumed that the fuel cost associated with flying from stage j to stage $j+1$ is known, for all possible initial and final altitudes, and for all $j \in \{1, \ldots, n-1\}$. By using a weighted and directed graph, as depicted in Fig. 1, the minimum-fuel altitude path is determined by finding the shortest path from stage 1 to stage n, where the cost associated with each transition is represented by the weight of the appropriate edge. For simplicity, we have assumed in Fig. 1 that there are at most three possible choices for the control at any stage: (i) increase by one unit of altitude, (ii) remain at the same altitude, or (iii) decrease by one unit of altitude.

The classical approach to solving the multistage optimization problem is to employ the (backward) dynamic programming technique [1, 2, 5], whereby decisions at each stage are made so as to minimize the estimated cost to go. The computation involved in this classical approach is inherently serial in the sense that decisions at the current stage must be made before decisions at the next stage can be determined. A parallel version of Moore's algorithm (a variant of Dijkstra's algorithm, which is again a variant of the dynamic programming technique) was proposed and tested by Deo *et al.* [9]. However, this algorithm has limitations on

* Supported in part by Texas Advanced Research Program 4659 and NSF ECS 8900499.

0743-7315/91 $3.00

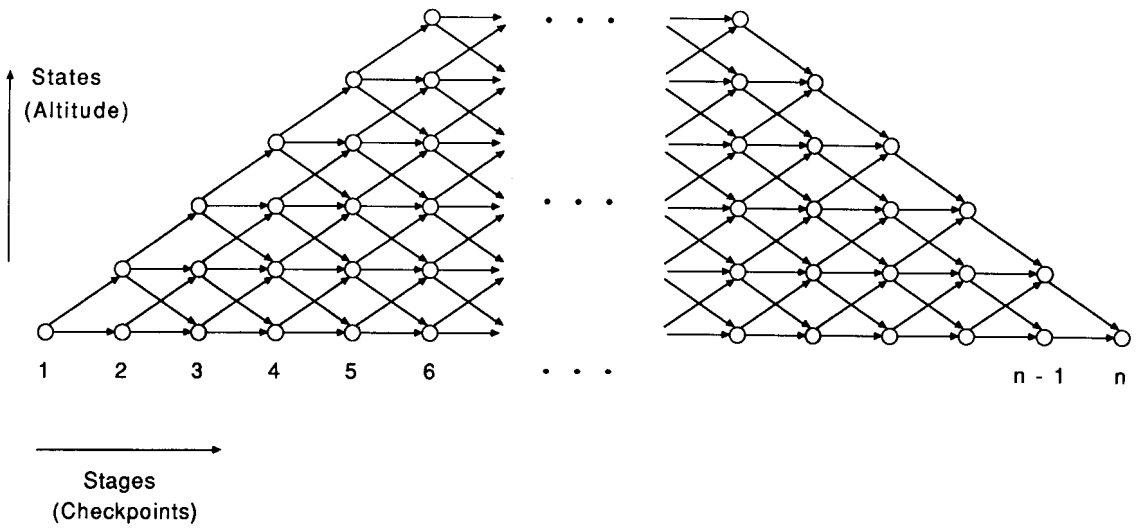

FIG. 1. An example application in which the minimum-fuel flight path is determined by solving a multistage optimization problem. The minimum-fuel flight path corresponds to the shortest path from stage 1 to stage n.

the number of processors which can be efficiently utilized due to an inherent problem of data contention—which leads to software lockout as the number of processors is increased. Also, a general time complexity result for the algorithm is not derived. Other schemes which simply parallelize the computation required at each stage of the standard dynamic programming technique have been proposed; see, for example, [7]. However, this scheme still has the property that the computation of the different stages must be computed sequentially (as is the case in the standard serial version) and therefore cannot achieve a time complexity better than $\Theta(n)$.[1]

In this paper we present a new parallel algorithm which solves the multistage optimization problem with less computational time complexity than any of the standard dynamic programming-type algorithms. Our proposed algorithm employs the concepts of divide and conquer, recursion, and aggregation. The original multistage optimization problem is first partitioned into a collection of small problems which can be solved in parallel. The solutions to these small subproblems are used to aggregate the number of required stages by a constant factor. This basic idea is used recursively to achieve a computation time complexity of $O(\log n)$ (assuming an n-stage process having $O(1)$ states per stage).

The remainder of the paper is organized in the following manner. In Section 2, the multistage optimization problem is formulated and the standard dynamic programming type algorithms for its solution are reviewed. In Section 3, we present our parallel algorithm for solving the multistage optimization problem. Section 4 gives some practical examples in which the new algorithm has advantages over the standard dynamic programming type algorithms.

[1] $\Theta(\cdot)$ is defined as follows: We say that $T(n) = \Theta(g(n))$, if there exists positive constants κ_1, κ_2, and n_0, such that $\kappa_1|g(n)| \leq T(n) \leq \kappa_2|g(n)|$, for all $n \geq n_0$.

2. THE MULTISTAGE OPTIMIZATION PROBLEM

A. *Formulation of the Multistage Optimization Problem*

Consider a multistage process in which each stage may assume one of several possible states. The process itself is sequential in the sense that proceeding to stage $j+1$ is possible only by passing through stage j. The cost associated with advancing from stage j to stage $j+1$ depends on the initial state (at stage j) and the final state (at stage $j+1$). A multistage process is represented with a directed acyclic graph, called a multistage process graph, in which states at stage j are connected to states at stage $j+1$ via directed edges whose weights represent the cost associated with each transition; see Fig. 2. The multistage optimization problem involves determining shortest paths from initial states to final states.

Consider an n-stage process where the number of possible states at stage j is given by m_j. We let $\bar{m}$ denote the maximum number of states contained in any single stage. Thus, $\bar{m}$ is defined as

$$\bar{m} = \max_{j \in \{1,\ldots,n\}} m_j. \tag{2.1}$$

We use the notation x_j^i to denote the ith state at stage j. For convenience, we often refer to x_j^i generically as a node. In order to simplify the formal statement of the multistage optimization problem and the description of our new algorithm, we introduce the following notation for characterizing all nodes associated with a common stage. We let X_j denote the

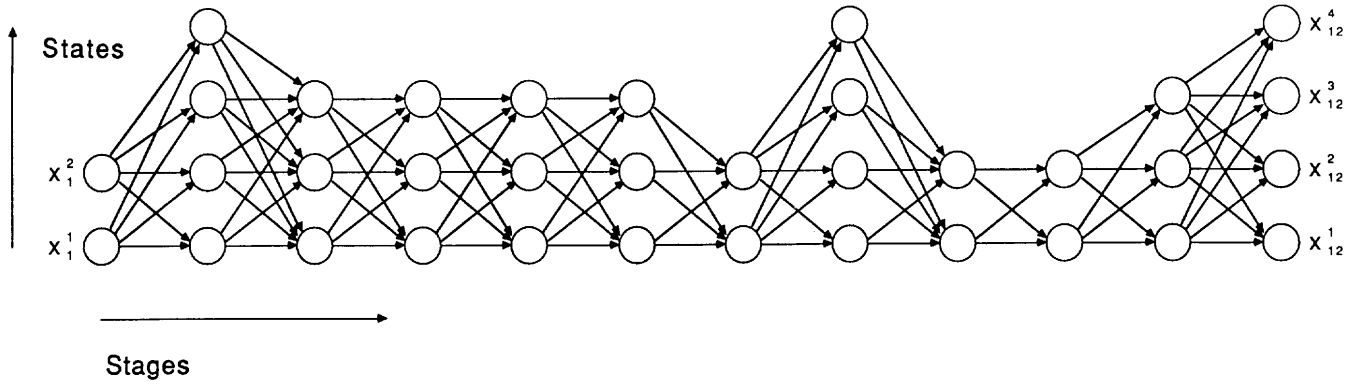

FIG. 2. An example of a 12-stage MPG in which the maximum number of states per stage is 4.

set of all nodes belonging to stage j. So, according to our notation thus far, we have

$$X_j = \{x_j^1, \ldots, x_j^{m_j}\}, \quad \text{for all} \quad j \in \{1, \ldots, n\}. \quad (2.2)$$

An important characteristic of a multistage process graph (MPG) is that the nodes must be interconnected in an acyclic manner. In particular, edges originating at node x_j^i must terminate at some node x_{j+1}^k, for each $j \in \{1, \ldots, n-1\}$ and for some $k \in \{1, \ldots, m^{j+1}\}$. We now formally state the multistage optimization problem.

The Multistage Optimization Problem (MOP). Find the shortest distances and associated shortest paths from each node in X_1 to a given node in X_n.

B. *Standard Parallel Algorithms for Multistage Optimization Problems*

Let $w_j^{i,k}$ denote the weight of the edge connecting node x_j^i to node x_{j+1}^k, defined for all $j \in \{1, \ldots, n-1\}$, $i \in \{1, \ldots, m_j\}$, and $k \in \{1, \ldots, m_{j+1}\}$. (Note: Without loss of generality we define $w_j^{i,k} = \infty$ if there is not an edge connecting x_j^i to x_{j+1}^k.) For definiteness, assume we are interested in finding shortest paths to node x_n^1. Let d_j^i denote the shortest distance from node x_j^i to node x_n^1. Below is a standard dynamic programming (DP)-based algorithm for MOPs.

Standard DP *Algorithm for* MOPS

0. *Initialize distances*
1. $d_n^1 \leftarrow 0$
2. for $i \leftarrow 2$ to m_n
3. $\quad d_n^i \leftarrow \infty$
4. end
5. *Update distances*
6. for $j \leftarrow n-1$ down to 1
7. $\quad d_j^i = \min_{\substack{i \in \{1,\ldots,m_j\} \\ k \in \{1,\ldots,m_{j+1}\}}} \{w_j^{i,k} + d_{j+1}^k\}$
8. end

It is clear that a single processor can execute the above algorithm with a time complexity bounded by $\bar{m}^2(n-1)$. Now, if $\bar{m}$ processors are available, then one processor can be assigned to each of the (at most) $\bar{m}$ values of i and achieve a complexity of $\bar{m}(n-1)$. Similarly, if $\bar{m}^2$ processors are available, then the (at most) $\bar{m}$ comparisons associated with the k indices (for each value of i) can be done with $\bar{m}$ processors in $\log \bar{m}$ time, thus yielding an overall complexity bounded by $\log \bar{m}(n-1)$.

The types of parallel algorithms described above simply parallelize the computation required by each of the $n-1$ main steps of the standard DP algorithm. Therefore, even in the best case (i.e., whenever $\bar{m} = O(1)$) their complexities are $\Theta(n)$. In the next section we employ the concepts of recursion and divide and conquer to develop a parallel algorithm which can solve the MOP with a time complexity of $O(\log n)$.

3. THE NEW PARALLEL ALGORITHM FOR MULTISTAGE OPTIMIZATION PROBLEMS

The new algorithm operates by recursively partitioning (i.e., clustering) the original n-stage MPG into independent c-stage subproblems which are solved in parallel (where c is an integer and is assumed to be independent of n). The solutions to the subproblems are then interpreted as the weights of aggregate edges associated with each cluster. Therefore, at the first level of operation, the original n-stage problem is reduced to an equivalent problem having only $O(n/c)$ stages. By recursively applying this procedure, the algorithm obtains the solution to the original problem after $O(\log_c n)$ such iterations. Since the complexity of each iteration depends only on c (which is assumed to be independent of n), the entire complexity of the algorithm is shown to be $O(\log n)$.

The recursive procedure of aggregation is easily conceptualized with the hierarchical structure of a balanced tree [10]; see Fig. 3. Note that if we let the number of leaves of a balanced tree represent the number of stages associated with the original MOP, then the number of nodes belonging to, say, level j of the tree represents the size of the aggregate

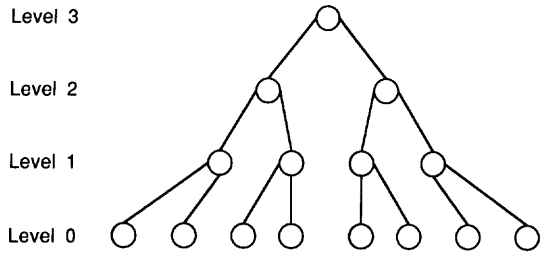

FIG. 3. The hierarchical nature of the recursive aggregation procedure (of the new algorithm) is conceptualized by the structure of a balanced tree.

problem after the recursive application of j aggregation steps. The operation of disaggregation takes place at the end of the algorithm (i.e., at the top of the tree) when the actual edges which make up the aggregate edges are retrieved from memory. We should note that a similar type of procedure was used in Ref. [12] to develop a distributed shortest path algorithm for a class of hierarchically clustered data networks.

Each iteration of the algorithm involves clusterizing the problem at hand into a number of independent c-stage subproblems. The clustering scheme we use is balanced in the sense that each cluster contains an equal number of stages, with the exception that the final cluster may contain fewer stages than the rest. The clusters are chosen so that each cluster shares its boundary stages with its neighboring clusters; see Fig. 4. The idea of "overlapping" the clusters in this manner was inspired by the textured model [14]. As we show later, this balanced overlapping clustered (BOC) partitioning actually has certain advantages (in terms of time complexity) over clustering schemes in which the clusters are not balanced in size and/or do not overlap. We note that recursively aggregating the clusters and reclusterizing according to the BOC partitioning is actually equivalent to characterizing the original MPG with a special type of BHC topology. (The BHC topology was introduced in Ref. [12] as a means of characterizing large hierarchically clustered data networks.)

In order to precisely define the BOC partitioning, we need the following preliminary definitions. The first of these definitions is related to the cluster size, and the second defines the member nodes of each cluster.

DEFINITION 3.1. The cluster size, denoted c, is an integer $c \geqslant 3$ which is assumed to be independent of n.

DEFINITION 3.2. For a given cluster size c, the ith cluster set, denoted C_i, is defined for all

$$i \in \left\{1, \ldots, \left\lceil \frac{n-1}{c-1} \right\rceil\right\}$$

as follows:

$$C_i = \begin{cases} \bigcup_{j=(i-1)(c-1)+1}^{i(c-1)+1} X_j & \text{if } i \in \left\{1, \ldots, \left\lceil \frac{n-1}{c-1} \right\rceil - 1\right\}; \\ \bigcup_{j=(i-1)(c-1)+1}^{n} X_j & \text{if } i = \left\lceil \frac{n-1}{c-1} \right\rceil. \end{cases}$$

In order to clarify the above definitions, recall the MPG depicted in Fig. 2, which has $n = 12$ stages and a value of $\bar{m} = 4$. Figure 4 shows how the MPG of Fig. 2 is clustered according to a BOC partitioning. Note that for this example we chose $c = 3$; therefore, the number of cluster sets is given by $\lceil (12-1)/(3-1) \rceil = 6$.

A. *Description of the New Algorithm*

The number of parallel processors needed by our new algorithm depends both on the dimensions of the MOP (i.e., n and $\bar{m}$) and on the choice of the cluster size c. For now, we assume that c is given—in the next subsection the optimal choice of c is derived. The number of processors required, denoted as $P(n)$, is given by

$$P(n) = \bar{m}^3 \left\lceil \frac{n-1}{c-1} \right\rceil. \tag{3.1}$$

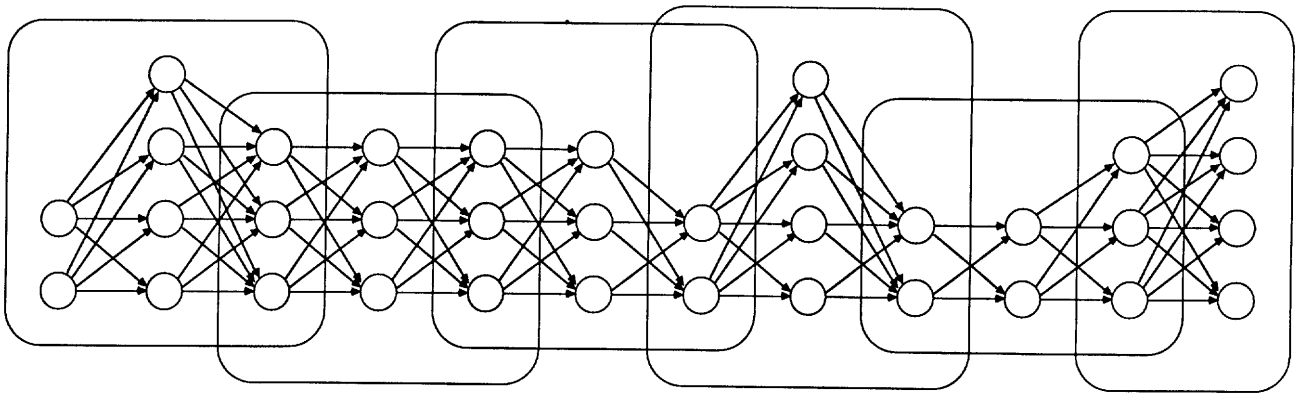

FIG. 4. Characterization of the MPG of Fig. 2 with a BOC structure.

From the above requirement, note that $\bar{m}^3$ processors can be alotted to each cluster of a multistage process graph, since a BOC partitioning of a given MPG contains exactly $\lceil (n-1)/(c-1) \rceil$ clusters.

The main idea of the algorithm is as follows. First, the n-stage MPG is characterized with the BOC partitioning. Each of the clusters then solve versions of their own sub-MOP[2] problem in parallel with all other clusters. The solution of the clusters' sub-MOPs are then used to create a natural aggregate representation of the original n-stage MPG, by using only $\lceil (n-1)/(c-1) \rceil + 1$ stages. This aggregate representation is obtained as follows: A cluster first determines the shortest distance from the first stage of the cluster to the last stage of the cluster, for all possible initial and final states. These distances are then viewed as the weights of associated virtual edges which connect all possible initial states to all possible final states for each cluster. This aggregation procedure is recursively applied to the current aggregate representation until an aggregate MPG which has only two stages is reached. The final aggregate two-stage MPG obviously contains the solution to the original n-stage problem. A detailed description of the algorithm, which includes the time complexities associated with each step, follows. The notation n_j is used to denote the number of stages of the aggregate MPG (after completion of the jth iteration).

Detailed Description.

(0) $n_0 \leftarrow n$

DO $j \leftarrow 1$ to $\lceil \log_{c-1}(n-1) \rceil$

(1) Characterize the (n_{j-1})-stage MPG with the BOC partitioning. Each cluster solves at most $\bar{m}$ versions of the associated sub-MOP—one version for each final state—in parallel with every other cluster set, by utilizing the $\bar{m}^3$ processors allotted for each cluster. The problems associated with each of the c-stage sub-MOPs are solved in parallel at each cluster, by using standard dynamic programming techniques, requiring a time complexity of $(c-2)\lceil \log_2(\bar{m}) \rceil$. This time complexity is obtained by executing at most $\bar{m}$ versions of the standard DP algorithm in parallel—one for each of the at most $\bar{m}$ destination nodes per cluster. Since each cluster contains at most c stages, and each of the $\bar{m}$ versions (per cluster) is allotted $\bar{m}^2$ processors, a total of $(c-2)$ iterations per cluster is required, having a computational time complexity of at most $\log_2(\bar{m})$ per iteration. (The $\log_2(\bar{m})$ complexity per iteration comes from the parallel comparison of at most $\bar{m}$ edges per node.)

(2) Next, the (n_{j-1})-stage MOP is collapsed into an n_j $(=\lceil (n_{j-1}-1)/(c-1) \rceil + 1)$-stage MOP, by using the shortest path information computed by each cluster set in step (1); see Fig. 5. We should note that this step is purely conceptual in the sense that it does not have an associated computational time complexity. The purpose of this step is to define the aggregate data to be accessed at the beginning of the next iteration.

END DO

The overall computational time complexity is the product of the number of iterations and the complexity per iteration. Thus, letting $T(n)$ denote the overall time complexity, we have

$$T(n) \leqslant (c-2)\lceil \log_2(\bar{m}) \rceil \lceil \log_{c-1}(n-1) \rceil. \quad (3.2)$$

Note that if we assume that $\bar{m}$ is independent of n, which is the case for many practical applications, then

$$T(n) = O(\log n). \quad (3.3)$$

Proof of Correctness. The correctness of the algorithm is based on the fact that an intermediate segment, from stage j to stage $j+c$, taken from a shortest path from stage 1 to stage n, must also be a shortest path from the associated origin node of stage j to the destination node of stage $j+c$ (by the optimality principle). By using this fact recursively, each iteration of the algorithm is obviously correct.

Next, we show that after $\lceil \log_{c-1}(n-1) \rceil$ iterations, a two-stage aggregate MPG is obtained. Recall that the number of stages of the aggregate MPG, after the completion of iteration j, is given by

$$n_j = \left\lceil \frac{n_{j-1}-1}{c-1} \right\rceil + 1, \quad (3.4)$$

which can be expressed as

$$n_j - 1 = \left\lceil \frac{n_{j-1}-1}{c-1} \right\rceil. \quad (3.5)$$

By using Eq. (3.5), we can determine the number of iterations required so that $n_j - 1 = 1$. Clearly, if $n_0 - 1 = (c-1)^k$ for some integer k, then the number of required iterations is given by $k = \log_{c-1}(n_0 - 1) = \log_{c-1}(n-1)$. Now, if $n_0 - 1 = (c-1)^y$, where y is not an integer, then the number of iterations needed will clearly be less than or equal to the number needed for an $\bar{n}\,(=(c-1)^{\lceil y \rceil}+1)$-stage MPG. Therefore, since the number of iterations needed for an $\bar{n}$-stage MPG is $\lceil y \rceil = \lceil \log_{c-1}(n_0 - 1) \rceil$, the result is proven. Q.E.D.

Several slightly modified versions of the above algorithm are possible when the number of processors available is less than that assumed in Eq. (3.1). For instance, by assuming

[2] By sub-MOP, we mean a c-stage MOP associated with the number nodes of each cluster.

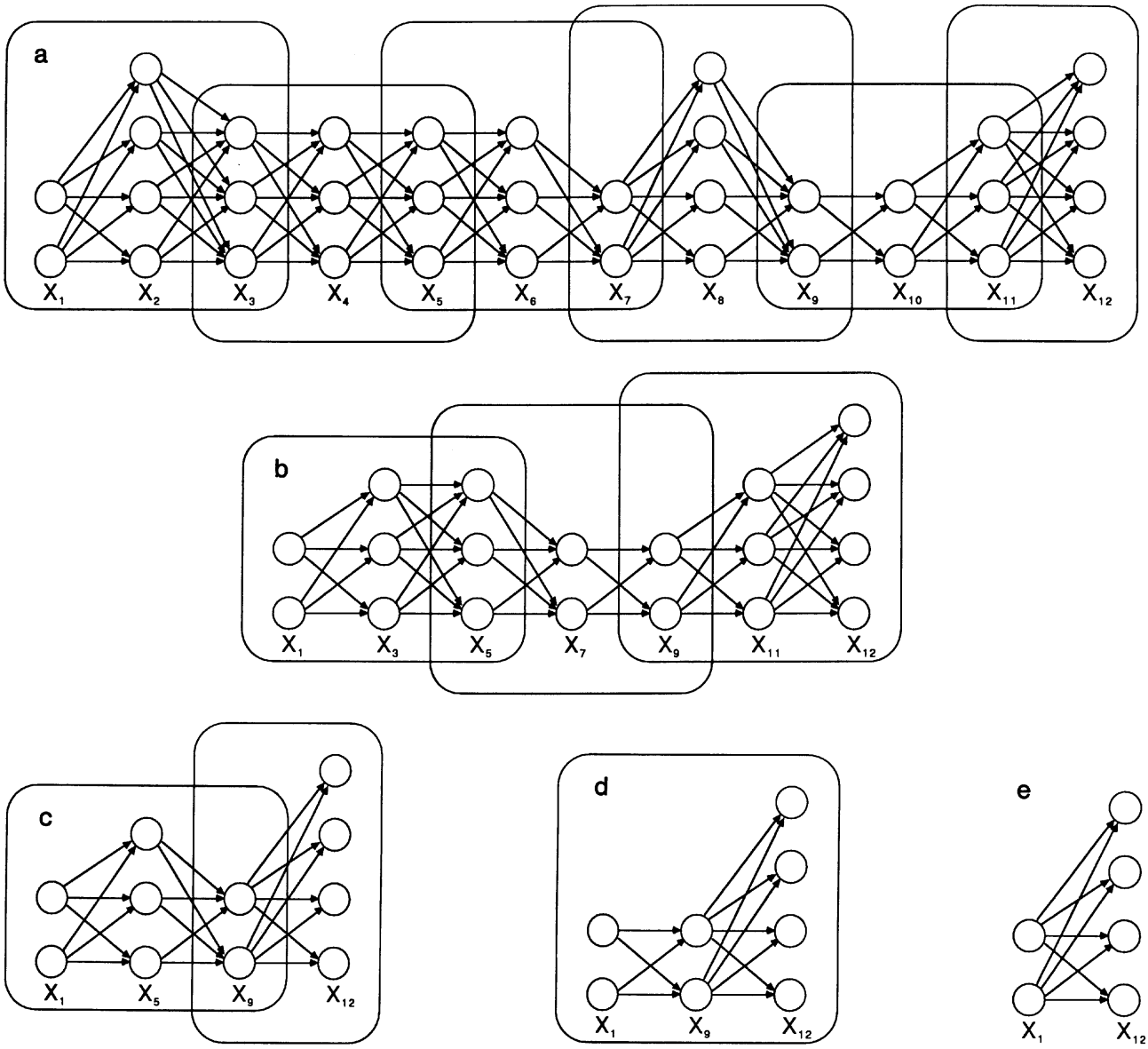

FIG. 5. (a) The original MPG—to be used at the first iteration of the algorithm. (b) The aggregate MPG produced after the first iteration—to be used at the second iteration. (c) The aggregate MPG produced after the second iteration—to be used at the third iteration. (d) The aggregate MPG produced after the third iteration—to be used at the fourth iteration. (e) The aggregate MPG produced after the fourth iteration—this MPG represents the solution to the original MOP.

$$P(n) = \bar{m}^2 \left\lceil \frac{n-1}{c-1} \right\rceil, \tag{3.6}$$

a straightforward modification (i.e., assign $\bar{m}$ more tasks to each processor) to the above algorithm yields a time complexity of

$$T(n) \leqslant \bar{m}(c-2)\lceil \log_{c-1}(n-1) \rceil. \tag{3.7}$$

Similarly, if $P(n) = \bar{m}\lceil (n-1)/(c-1) \rceil$, the time complexity is $T(n) \leqslant \bar{m}^2(c-2)\lceil \log_{c-1}(n-1) \rceil$, and if $P(n) = \lceil (n-1)/(c-1) \rceil$, then a time complexity of $T(n) \leqslant \bar{m}^3(c-2)\lceil \log_{c-1}(n-1) \rceil$ can be achieved.

In general, if

$$P(n) = \frac{\bar{m}^3}{k} \left\lceil \frac{n-1}{c-1} \right\rceil, \quad \text{for some } k, \tag{3.8}$$

then the time complexity is bounded by

$$T(n) \leqslant k(c-2)\lceil \log_{c-1}(n-1) \rceil. \tag{3.9}$$

B. *Optimal Cluster Size*

We now turn our attention toward determining the cluster size which minimizes the bound for the time complexity of the proposed algorithm. Note first that any choice for c which is independent of n can be used to obtain a time complexity of $O(\log n)$, assuming that $\bar{m}$ is independent of n. By noting that c must be an integer which satisfies $c \geq 3$, it can be shown with straightforward analysis that the value $c = 3$ minimizes the upper bound given in Eq. (3.2). Note that using a cluster size of $c = 2$ does not make sense; i.e., $\log_{c-1}(n-1)$ is not defined for $c = 2$.

Figure 6 demonstrates the effect that cluster size has on the normalized time complexity, defined in Eq. (3.10). Note that $c = 3$ does in fact minimize this function, regardless of the size of n. The normalized time complexity plotted in Fig. 6 is defined by dividing the upper bound given in (3.2) by $\lceil \log_2(\bar{m}) \rceil$, which yields

$$T_{\text{norm}}(n) = (c-2)\lceil \log_{c-1}(n-1) \rceil. \tag{3.10}$$

We stress that the optimal cluster size of $c = 3$ does not have to be used in order to reap the benefits of the new algorithm. Recall from Eq. (3.1) that the number of processors required is inversely proportional to $c - 1$. Thus, there is an inherent trade-off between the time complexity and the number of processors needed. That is, smaller values of c achieve a better time complexity at the expense of requiring more processors. If only a specified number of processors is available, then for fixed values of n and m, Eq. (3.1) can be used to determine an appropriate value of c. On the other hand, if ample processors are available, then a cluster size of $c = 3$ should be used in order to achieve the best possible time complexity.

Other clustered partitionings (besides the BOC partitioning) may be used to develop algorithms based on the same ideas presented in this section. However, from our studies it appears that the BOC partitioning actually has an added advantage over clustered structures which do not "overlap." For instance, if we use a nonoverlapping clustered structure in which the clusters are of size $\tilde{c}$, it can be shown that the associated bound for the time complexity is

$$\tilde{T}(n) \leq (\tilde{c}-2)\lceil \log_2(\bar{m}) \rceil \lceil \log_{\tilde{c}/2}(n) \rceil. \tag{3.11}$$

It can be verified that for a given value of n and $\bar{m}$, the optimal (i.e., smallest) value of the above bound for $\tilde{T}(n)$ is larger than the optimal bound of $T(n)$.

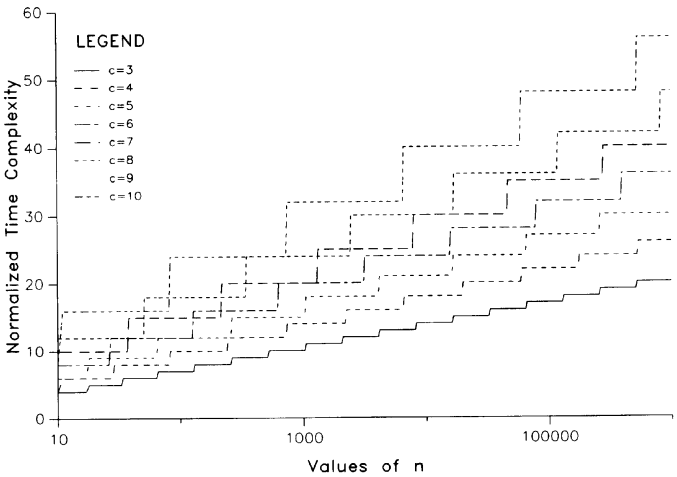

FIG. 6. The effect of cluster size on the normalized time complexity.

4. APPLICATIONS FOR THE NEW ALGORITHM

A. *Decoding Convolutional Codes*

The following formulation of convolutional coding follows the description and notation used in Ref. [6]. Convolutional coding is widely used as a means of improving the reliability associated with sending binary data over a noisy communication channel. This method of coding converts a source generated binary sequence

$$\{w_1, w_2, \dots\}, \quad w_k \in \{0, 1\}, \quad k \in \{1, 2, \dots\}, \tag{4.1}$$

into a coded sequence $\{y_1, y_2, \dots\}$, where each y_k, $k \in \{1, 2, \dots\}$, is an l-dimensional vector with binary coordinates,

$$y_k = (y_k^1, \dots, y_k^l)^{\mathrm{T}}, \quad y_k^i \in \{0, 1\}, \quad i \in \{1, \dots, l\}. \tag{4.2}$$

The vectors y_k are related to w_k via equations of the form

$$x_k = Ax_{k-1} + bw_k \tag{4.3}$$

$$y_k = Cx_{k-1} + dw_k, \tag{4.4}$$

$$x_0\text{: given,}$$

where x_k is a p-dimensional vector (called the state vector) with binary coordinates and A, b, C, d are $p \times p$, $p \times 1$, $l \times p$, and $l \times 1$ matrices, respectively, with binary coordinates. Since there are only two possible values for w_k (i.e., zero and one) and there are 2^p different states, the evolution of the system of equations of (4.3) and (4.4) can be represented with a transition diagram as shown in Fig. 7.

The coded vectors y are transmitted over a noisy communication channel where they are corrupted and therefore received as z with some known probability $p(z|y)$. We use the notation

$$Z_n = \{z_1, z_2, \dots, z_n\} \tag{4.5}$$

to denote the sequence received when the transmitted sequence is

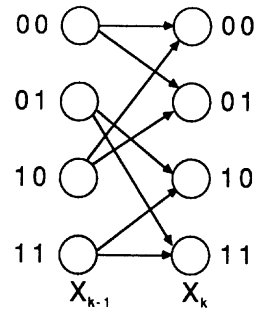

FIG. 7. An example of a transition diagram having four possible states per stage. Note that the two outgoing edges at each node correspond to the possible values of the binary input.

$$Y_n = \{y_1, y_2, \ldots, y_n\}. \tag{4.6}$$

By assuming that errors are independent we have that

$$p(Z_n | Y_n) = \prod_{k=1}^{n} p(z_k | y_k). \tag{4.7}$$

A maximum likelihood decoder converts a received sequence Z_n into a sequence

$$\hat{Y}_n = \{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n\} \tag{4.8}$$

such that

$$p(Z_n | \hat{Y}_n) = \max_{Y_n} \{p(Z_n | Y_n)\}. \tag{4.9}$$

Viterbi [4] was the first to develop a shortest path scheme that implements the maximum likelihood decoder.[3] To view the maximum likelihood decoder problem as a shortest path problem, first note that by using (4.7), the problem of (4.9) is equivalently transformed to the problem

$$\min_{Y_n} \sum_{k=1}^{n} -\log[p(z_k | y_k)]. \tag{4.10}$$

Next, assign to each arc on the state transition diagram the weight $-\log[p(z_k | y_k)]$. Finally, concatenate n versions of the transition diagram together and connect artificial nodes u and v with zero weight edges to states x_0 and x_{n-1}, respectively (see Fig. 8). The solution to the problem of (4.10) is then obtained by finding the shortest path from node u to v.

In the Viterbi algorithm, decoding is done sequentially in time (as new coded vectors are received), by employing dynamic programming techniques at each stage of the associated MOP. By using this type of approach, the maximum rate at which coded vectors can be received obviously depends on the computation time required to solve a single stage of the dynamic programming algorithm. Let R_{rec} denote the rate at which the vectors are received, measured in vectors per second, and let R_{sol} denote the rate at which stages of the standard dynamic programming algorithm can be solved, measured in stages per second. (For instance, if $\bar{m}^2$ parallel processors are available, then $1/R_{\text{sol}} = O(\log_2 \bar{m})$; similarly, with $\bar{m}$ parallel processors, we have $1/R_{\text{sol}} = O(\bar{m})$, and with one processor, $1/R_{\text{sol}} = O(\bar{m}^2)$.) Therefore, for the Viterbi algorithm, the rate at which the vectors may be received is bounded by R_{sol}. Obviously, if $R_{\text{rec}} > R_{\text{sol}}$ then the decoder cannot keep up with the incoming vectors.

Now, assume we have a tightly coupled parallel computer having $m\lceil (n-1)/(c-1) \rceil$ parallel versions of the processor(s) used to achieve the given value of R_{sol} (where n denotes the length of the sequence to be decoded, $\bar{m}$ is the maximum number of states per stage, and c is the cluster size). The proposed procedure is then described as follows. First, an n-length sequence of coded vectors is received and stored in memory. The conditional probabilities are then computed in parallel for each stored vector, and the results are stored as the associated edge weight for the n-stage MPG. By using the parallel algorithm (PA) of Section 3, the n-stage MOP is solved with a time complexity

$$T_{\text{PA}}(n) \leqslant \frac{(c-2)}{R_{\text{sol}}} \lceil \log_{c-1}(n-1) \rceil. \tag{4.11}$$

In contrast, note that the time complexity required for the Viterbi algorithm (VA) to decode a sequence of n vectors is given by

$$T_{\text{VA}}(n) \leqslant \frac{n}{R_{\text{sol}}}. \tag{4.12}$$

Therefore, the speedup in the rate at which vectors may be received, when one is decoding an n-length sequence, is given by

$$S(n) \geqslant \frac{n}{(c-2)\lceil \log_{c-1}(n-1) \rceil}. \tag{4.13}$$

By way of illustration, consider the case where $n = 512$; then by letting $c = 3$, a speedup more than 56 is obtained. If n

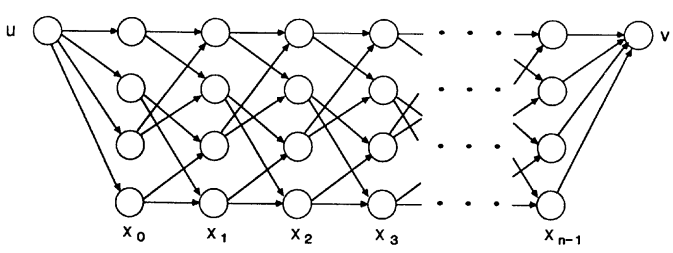

FIG. 8. Viewing the maximum likelihood decoder problem as one of finding the shortest path between nodes u and v.

[3] The Viterbi algorithm is essentially a forward dynamic programming technique, as observed by Omura [15].

= 2048, then with $c = 3$, a speedup in excess of 186 can be achieved.

Employing this new proposed procedure does require slightly more delay for decoding a sequence of n vectors than does the Viterbi algorithm. Note that for a fixed value of R_{rec}, the Viterbi algorithm has a delay of

$$D_{\text{VA}}(n) = \frac{1}{R_{\text{rec}}}. \tag{4.14}$$

On the other hand, our new procedure can receive vectors at a rate of $S(n)R_{\text{rec}}$, so its associated delay is

$$D_{\text{PA}}(n) \leq \frac{n}{S(n)R_{\text{rec}}} = \frac{(c-2)\lceil \log_{c-1}(n-1)\rceil}{R_{\text{rec}}}. \tag{4.15}$$

For many practical applications, the value of $(c-2) \times \lceil \log_{c-1}(n-1)\rceil$ is relatively small; thus the actual increase in the delay may be acceptable.

B. *The General All-Pairs Shortest Path Problem*

Consider a connected and directed graph $G = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ denotes the set of nodes and $\mathcal{E}$ the set of interconnecting edges, each of which is assigned a positive weight. The all-pairs shortest path problem involves finding shortest paths from every node in the graph to every other node. The fastest known all-pairs shortest path algorithm is the so-called doubling algorithm, proposed by Dekel *et al.* [16]. For a general n-node graph G, the doubling algorithm achieves a time complexity of

$$T(n) = O[(\log n)^2], \tag{4.16}$$

by using $P(n) = n^3$ processors.

It turns out that our new algorithm can also be used to solve the all-pairs shortest path problem with the same time complexity as the doubling algorithm. This is accomplished by transforming the original graph G into an appropriate multistage process graph. The solution of this transformed MOP represents the solution to the original all-pairs shortest path problem.

The all-pairs shortest path problem for a general n-node graph is transformed into an n-stage MPG having n states per stage (see Fig. 9). Each node at each stage corresponds to an actual network node (from the original graph G) and is connected to nodes of the next stage according to the edge weights of the original network graph. Also, node i at stage j is connected to node i at stage $j+1$ with an edge of weight zero. Clearly, solving this MOP is equivalent to solving the all-pairs shortest path problem of the original network graph. Note from Fig. 9 that each stage of the n-stage MPG has $\bar{m} = n$ states. Furthermore, the connectivity pattern and outgoing edge weights at each stage are identical. So, if we apply our algorithm to this type of MPG, we note that the intermediate solutions to all clusters are identical (for a given iteration); thus, we can reduce the number of processors required by a factor of $\lceil (n-1)/(c-1)\rceil$. Therefore, since $\bar{m} = n$, the number of processors required is

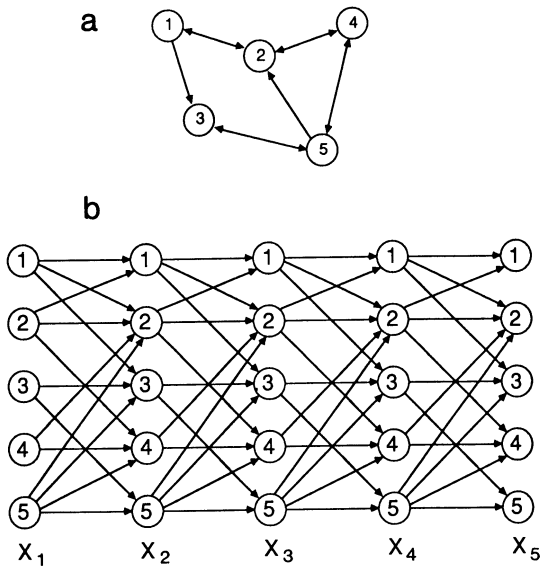

FIG. 9. (a) An example of a 5-node directed network graph. (b) Solving the all-pairs shortest path problem for the network graph of (a) is equivalent to solving the MOP for the associated MPG shown in (b).

$$P(n) = \bar{m}^3 = O(n^3). \tag{4.17}$$

Also, from (3.2), the time complexity is bounded by

$$T(n) \leq (c-2)\lceil \log_2(n)\rceil \lceil \log_{c-1}(n-1)\rceil \tag{4.18}$$

or

$$T(n) = O[(\log n)^2]. \tag{4.19}$$

The requirement of $O(n^3)$ processors (by both the doubling algorithm and our new algorithm) may not be practical for very large values of n. However, the fact that our new algorithm achieves a time complexity of $O[(\log n)^2]$ is theoretically interesting since this result matches the best known time complexity for solving the all-pairs shortest path problem for general graphs.

Our new algorithm is a generalization of the standard doubling algorithm in the following sense: When applied to the all-pairs shortest path problem for general graphs, our new algorithm and the doubling algorithm are equivalent. However, in solving an n-stage MOP in which $\bar{m} = O(1)$, our algorithm achieves a complexity of $O(\log n)$, while the standard doubling algorithm achieves only $O[(\log n)^2]$.

5. CONCLUSIONS

In this paper we have presented a parallel algorithm for solving the MOP which combines the dynamic programming

technique with the concepts of recursion and aggregation to achieve a time complexity which is generically better than that of any other known algorithm. Many practical problems require solving a MOP, and thus our fast algorithm should serve well for numerous applications. Two such applications were discussed, in which the computational complexity associated with solving a MOP dictates the rate at which the overall problem can be solved. By applying our algorithm to the problem of decoding convolutional codes, we obtain a speedup of $O(n/\log n)$ in the rate at which vectors may be received, while increasing the decoding delay by a factor of only $O(\log n)$. Also, it is shown that our new algorithm can be used to solve the general all-pairs shortest path problem with a time complexity of $O[(\log n)^2]$.

Due to the remarkable strides made in the area of VLSI technology, we feel that our algorithm has the potential to be used in numerous applications. Also, the natural modularity of the algorithm should make the actual programming on existing parallel machines a fairly straightforward task.

REFERENCES

1. Bellman, R. E. *Dynamic Programming.* Princeton Univ. Press, Princeton, NJ, 1957.
2. Bryson, A. E., and Ho, Y.-C. *Applied Optimal Control.* Hempshire, Washington, DC, 1975.
3. Viterbi, A. J., and Omura, J. K. *Principles of Digital Communication and Coding.* McGraw-Hill, New York, 1979.
4. Viterbi, A. J. Error bounds for convolutional codes and an asymptotic optimum decoding algorithm. *IEEE Trans. Inform. Theory* **IT-13** (1967), 238–242.
5. Bertsekas, D., and Gallager, R. *Data Networks.* Prentice-Hall, Englewood Cliffs, NJ, 1987.
6. Bertsekas, D. P. *A Course in Dynamic Programming and Stochastic Control.* Prentice-Hall, Englewood Cliffs, NJ, 1988.
7. Larson, R. E., and Tse, E. Parallel processing algorithms for the optimal control of nonlinear dynamic systems. *IEEE Trans. Comput.* **C-22** (Aug. 1973), 777–786.
8. Dijkstra, E. A note on two problems in connexion with graphs. *Numer. Math.* **1** (1959), 269–271.
9. Deo, N., Pang, C., and Lord, R. E. Two parallel algorithms for shortest path problems. *Proc. IEEE International Conference on Parallel Processing.* IEEE Computer Society, Silver Spring, MD, Aug. 1980, pp. 244–253.
10. Aho, A. V., Hoptcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.
11. Tsai, W. K., Huang, G. M., Antonio, J. K., and Tsai, W. T. Distributed iterative aggregation algorithms for box-constrained minimization problems and optimal routing in data networks. *IEEE Trans. Automat. Control.* **34** (Jan. 1989), 34–46.
12. Antonio, J. K., Huang, G. M., and Tsai, W. K. A fast distributed shortest path algorithm for a class of hierarchically clustered data networks. *IEEE Trans. Comput.*, in press.
13. Schwartz, M., and Stern, T. E. Routing techniques used in computer communication networks. *IEEE Trans. Comm.* **COM-28** (Apr. 1980), 539–552.
14. Huang, G., Lu, K. W., and Zaborszky, J. A textured model/algorithm for computationally efficient dispatch and control on the power system. *Proc. 25th IEEE Conference on Decision and Control,* Athens, Greece, Dec. 1986, pp. 1224–1227.
15. Omura, J. K. On the Viterbi decoding algorithm. *IEEE Trans. Inform. Theory* **IT-15** (1969), 177–179.
16. Dekel, E., Nassimi, D., and Sahni, S. Parallel matrix and graph algorithms. *SIAM J. Comput.* (Nov. 1981), 657–675.

Received April 6, 1990; accepted February 27, 1991

JOHN K. ANTONIO was born in Forth Worth, Texas, in 1961. He received the B.S. degree (with honors), the M.S. degree, and the Ph.D. degree, all in electrical engineering, from Texas A&M University, College Station, in 1984, 1986, and 1989, respectively. Dr. Antonio worked with the Digital Flight Control Group, General Dynamics/Fort Worth Division, during the summers of 1983 and 1984. From 1984 to 1986, he was both a teaching assistant and a research assistant, while working on his Master's degree. During the summer of 1986, he was employed by General Motors Research Laboratories, Warren, Michigan, as a Research Associate. From 1986 until 1989, he held the position of lecturer in the Electrical Engineering Department at Texas A&M University, while pursuing his Ph.D. After graduation, he accepted employment at Purdue University, where he currently holds the position of assistant professor of electrical engineering. His research interests include analysis and design of fast distributed algorithms for high-speed networks, optimal routing in data networks, computational aspects of control and optimization, and control theory. He is a member of the IEEE computer society, control systems society and communications society. Also, he is a member of Tau Beta Pi, Eta Kappa Nu, and Phi Kappa Phi. In 1989 he received the Ruth and Joel Spira "Outstanding Teaching Award" from the School of Electrical Engineering at Purdue University.

WEI K. TSAI received the B.Sc., M.Sc., and Ph.D. degrees, all in electrical engineering, from the Massachusetts Institute of Technology, Cambridge, in 1979, 1982, and 1986, respectively. In the summer of 1982, he was on the engineering staff of the New England Electric Power Service Co., Westboro. From September 1986 through October 1989, he was an assistant professor in the Department of Electrical Engineering at Texas A&M University, College Station. In September 1989, he joined University of California at Irvine, Department of Electrical and Computer Engineering, as an assistant professor. His research interests include data communication networks, artificial neural networks, parallel and distributed algorithms, parallel computer architectures, and control systems. Dr. Tsai is a member of IEEE, SIAM, Sigma Xi, and Eta Kappa Nu.

G. M. HUANG received his B.S. and M.S. degrees in electrical engineering from National Chiao Tung University, Hsinchu, Taiwan, Republic of China, in 1975 and 1977, respectively. He received his doctorate degree in systems science and mathematics from Washington University, St. Louis, in 1980, and taught there until 1984. In 1984 he joined Texas A&M University, Department of Electrical Engineering, where he is currently an associate professor. He has worked on many funded research projects, such as Emergency Control of Large Interconnected Power System, HVDC Systems, Restoration of Large Scale Power Systems, Online Detection of System Instabilities and Online Stabilization of Large Power System, fast parallel/distributed textured algorithms, and fast parallel textured algorithms for large power systems. His current interest is in large-scale systems theory, large-scale parallel and distributed computing and control, and their applications. Dr. Huang is a senior member of the IEEE, a registered professional engineer of Texas, the Committee Chairman of the Energy System Control Committee of the IEEE Transactions on Automatic Control Society, and a member of the Systems Engineering Committee of the IEEE PAS Society. Dr. Huang has published more than 80 papers and reports in the areas of nonlinear, distributed control systems and parallel computing and their applications to power systems, data networks, and flexible structures.