JOURNAL OF OPTIMIZATION THEORY AND APPLICATIONS: Vol. 12, No. 4, 1973

Dynamic Programming and Parallel Computers[1]

J. CASTI,[2] M. RICHARDSON,[3] AND R. LARSON[4]

Communicated by R. E. Kalaba

Abstract. The computational theory of dynamic programming is examined from the viewpoint of parallel computation. A discussion of various forms of parallelism, the corresponding parallel algorithms, the applicability of the algorithms to different types of optimization problems, and their advantages over serial computation is presented. In addition, parallel aspects of various dimensionality reduction techniques such as state increment dynamic programming, successive approximations, and shift vectors are also given.

1. Introduction

It is well known that dynamic programming is a powerful technique for solving large classes of optimization problems under very general conditions. However, from a computational point of view, the method is somewhat limited by the large computational requirements of the algorithm used on present-day serial machines.

The parallel processing computer, which is presently under development in this country (Ref. 1), could greatly reduce the computer time and memory required for solving large-scale optimization problems by dynamic programming, since there are a number of parallel operations that occur in the evaluation of the dynamic programming recursive formula.

A discussion of various forms of parallelism, the corresponding parallel algorithms, the algorithms' applicability to different types of problems, and their advantages over serial methods is given in this paper. In addition, discussions of parallelism for state increment dynamic

[1] This research was supported by the Air Force Office of Scientific Research, Contract No. F44620-70-C-0084.

[2] Senior Research Mathematician, Systems Control, Palo Alto, California.

[3] Research Engineer, Systems Control, Palo Alto, California.

[4] Vice-President, Technical Operations, Systems Control, Palo Alto, California.

© 1973 Plenum Publishing Corporation, 227 West 17th Street, New York, N.Y. 10011.

programming (Ref. 2), for the shift vector method (Ref. 3), and for a successive approximations method (Ref. 4) are also given.

We are interested in solving the following N-stage decision problem whose constituents are: (a) an interval over which the decisions are to be made, each decision at stage i; (b) a state variable x_i characterizing the system at any stage of the decision process; (c) a transition function f_i relating a state and decision u_i at a stage to a state at a future stage, usually the succeeding one, i.e., $x_{i+1} = f_i(x_i, u_i)$; (d) a criterion function giving the cost of making a particular decision at a given state and stage, $g_i(x_i, u_i)$, along with a rule for computing overall cost, e.g., $\sum_{i=1}^{N} g_i(x_i, u_i)$; and (e) possible constraints upon states and/or decisions at different stages.

The optimization problem that we wish to solve can be stated as follows: Find a policy $(u_1, ..., u_N)$ and corresponding trajectory $(x_1, ..., x_N)$ which minimize

$$\sum_{i=1}^{N} g_i(x_i, u_i), \tag{1-1}$$

subject to the constraints

$$x_{i+1} = f_i(x_i, u_i), \qquad i = 1, ..., N-1, \tag{1-2}$$

$$x_i \in X_i, \qquad u_i \in U_i, \qquad f_i: X_i \times U_i \to X_{i+1}, \qquad g_i: X_i \times U_i \to R, \tag{1-3}$$

where at stage i, X_i is the allowable state set, U_i is the set of admissible decisions, and R is the real line.

Large classes of deterministic and stochastic problems, in such areas as nonlinear programming, optimal control, network flow, and combinatorial theory can be formulated in the above framework.

The dynamic programming algorithm is a simple computational process which, in general, involves the following two computational steps.

(a) Evaluation of an optimal value function V at a suitable number of state values at each stage. The function V_i is defined to be

$$V_i(x_i) = \min_{u_i, ..., u_{N-1}} \left[\sum_{j=1}^{N} g_j(x_j, u_j) \right], \qquad x_i \in X_i, \qquad i = 1, ... N, \tag{2-1}$$

$$V_{N+1}(x_{N+1}) = 0, \qquad i = N, N-1, ..., 1. \tag{2-2}$$

The *principle of optimality* (Ref. 5) allows us to evaluate the optimal value function recursively by the relation

$$V_i(x_i) = \min_{u_i \in U_i} [g_i(x_i, u_i) + V_{i+1}(f_i(x_i, u_i))], \qquad i = 1, 2, ..., N. \tag{3}$$

The optimal feedback decision $\hat{u}_i(x_i)$ is the argument which minimizes the r.h.s. of the above formula.

(b) Trace back recovery of an optimal solution starting at optimal initial state $\hat{x}_1$, where $\hat{x}_1$ is determined as

$$\min_{x_1 \in X_1}[V_1(x_1)].$$

Starting with the optimal decision $\hat{u}_1(\hat{x}_1)$, the optimal policy and trajectory are then recursively determined, using the equation $\hat{x}_{i+1} = f_i(\hat{x}_i, \hat{u}_i(\hat{x}_i))$.

Oftentimes, especially in control problems, a *feedback* solution is desired, which simply means that, given any state of the process, an optimum decision is desired which minimizes the objective of the problem at that point. This is obtained via dynamic programming by performing computational step (a) alone.

For computational purposes, the sets X_i, U_i are assumed to be finite; in cases where they are infinite, the state and decision variables are quantized to a finite number of values. Consequently, if the state is a vector and each component has the same number of quantized values (QS), then the total number of quantized states at a given stage is $(QS)^n$, where n is the dimension of the vector. Likewise for decisions, assuming equal quantization of all components, the total number of quantized decisions at a state is $(QD)^m$, where QD is the number of quantized values of each component and m is the dimension of the decision vector.

The solution of Eq. (3), referred to as Bellman's equation or the dynamic programming formula, is by far the most time-consuming part of the dynamic programming computations. The approximate computation time τ is

$$\tau = \sum_{i=1}^{N} (\Delta t)(QD_i)^{m_i}(QS_i)^{n_i},$$

where Δt is the time to solve Eq. (3) once (at one state using one decision choice), m_i is the number of components in decision vector at stage i, n_i is the number of components in state vector at stage i, QD_i is the number of quantized decision choices per component at stage i, and QS_i is the number of quantized state values per component at stage i.

Note the exponential growth of the computing time with both the number of states and number of decisions. Growth due to either or both of these factors can be reduced or eliminated by the parallel computation schemes to be discussed in the next section.

2. Characteristics of a Parallel Computer

Before turning to the algorithms, let us briefly describe what we mean by a parallel computer. For our purposes, a machine with the following characteristics is needed.

It is capable of processing many data streams simultaneously, using the same instruction stream. We assume that is is made up of a set of processing elements (PEs), each having the capability of executing logical and arithmetic instructions. The PEs are controlled by the central processing unit (CPU) which broadcasts the same sequence of instructions to all PEs.

Each PE is connected to its own rapid access memory and also to at least two of its *nearest neighbors* with which it is able to communicate. Figure 1 illustrates this overall desired architecture.

The ILLIAC IV machine at the NASA Ames Research Center (Ref. 1) has the above characteristics, with 64 PEs. However, since it is being designed as a general purpose machine, it is a great deal more complex than is necessary for execution of the algorithms to be described here.

3. Parallel States, Decisions, and Stages Algorithms

Evaluation of the optimal value function at all states and stages generally involves three nested interative loops, as follows: (a) iterate over all stages; (b) iterate over all stages at a stage; (c) iterate over all

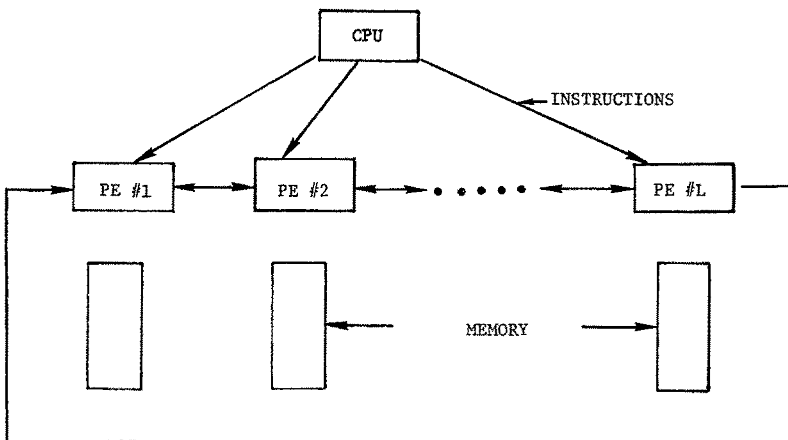

Fig. 1. Parallel computer architecture.

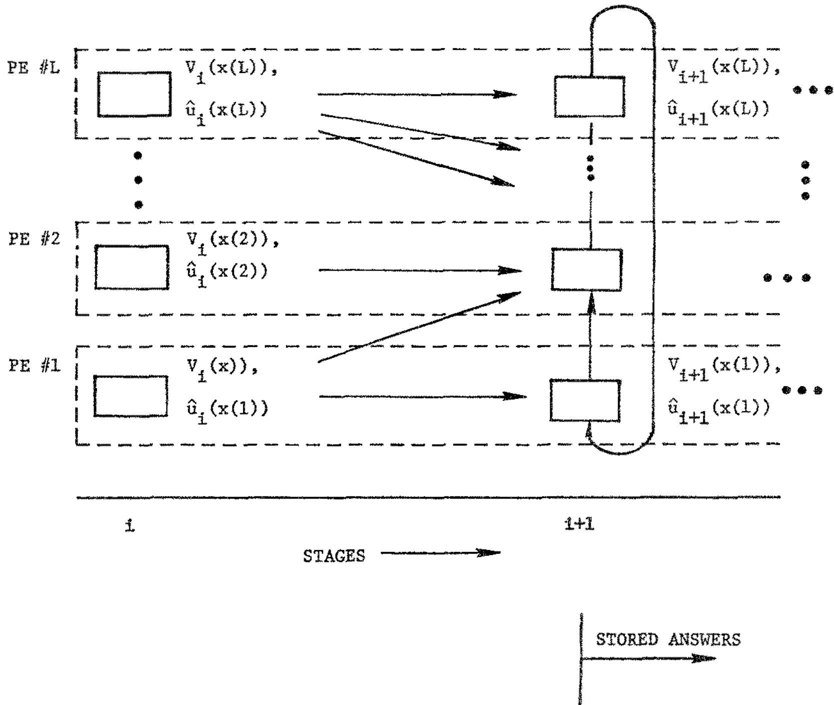

Fig. 2. Schematic diagram of parallel states algorithm.

decisions; evaluate r.h.s. of iterative formula (3) using one decision,[5] compare answer with previous value, and save minimum.

The following algorithms each eliminate one of the above loops.

3.1. Parallel States. If the number of processing elements in a parallel machine is equal to or greater than $(QS_i)^{n_i}$ for each i, then each PE can do the computations for one quantized state value. Hence, all the computations at one stage are done in parallel by the following procedure.

(i) Compute r.h.s. of Eq. (3) using V_{i+1} stored in PE, and save.

(ii) Route V_{i+1} to neighboring PE as shown in Fig. 2.

(iii) Compute second value of r.h.s. of Eq. (3) using new V_{i+1} and compare answer with result of Step (i), saving the minimum.

(iv) Return to Step (ii) and repeat procedure for all admissible quantized values of the decision variable.

[5] r.h.s. refers to the quantity in backets on the right-hand side of the recursive formula.

Note that $V_i(x_i)$ is evaluated in parallel at all states x_i during L evaluations, comparisons, and storage of minimum r.h.s. of the recursive formula, with L routes of stored values $V_{i+1}(x_{i+1})$ in between.

Depending on the form of the state transition function [Eq. (1)], if a given quantized state x_i and decision u_i does not yield a quantized next state x_{i+1}, then one of the following schemes can be used, depending on the problem being solved: (a) interpolation between two surrounding values of $V_{i+1}(x_{i+1})$ to obtain the proper $V_{i+1}(x_{i+1})$; (b) interpolation between successive decision values to obtain proper quantized next state x_{i+1}; and (c) *round off* to the next higher or lower quantized state value.

3.2. Parallel Decisions. If the number of processing elements in a parallel machine is equal to or greater than $(QD_i)^{m_i}$ for each i, then the optimum decision at each quantized state can be found efficiently by the following parallel scheme.

(i) Each PE computes r.h.s. of Eq. (3) for one value of the decision variable, all at the same state value, as shown in Fig. 3.

(ii) The optimum decision for that state is then found by the binary route and comparison routine depicted in Fig. 4. Each PE performs routes of +1, +2, +4,... PEs with a comparison and storage of the minimum performance after each routing. When m routes and comparisons have been made, each PE has the optimum performance $V_i(x_i)$ and corresponding decision stored in it.

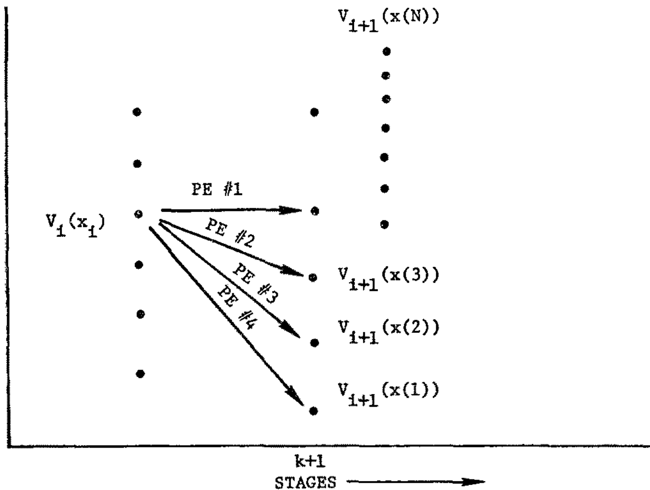

Fig. 3. Parallel decisions algorithm.

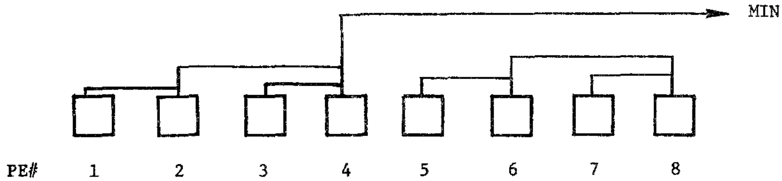

Fig. 4. Binary minimization procedure.

(iii) The computation iterates over all states and stages using Steps (i) and (ii).

3.3. Comparison with the Serial Algorithm. It is very easy to make comparisons between the previous two algorithms and straightforward serial computation, since in one case the state loop was eliminated, and in the other the decision loop was eliminated. Figures 5–6

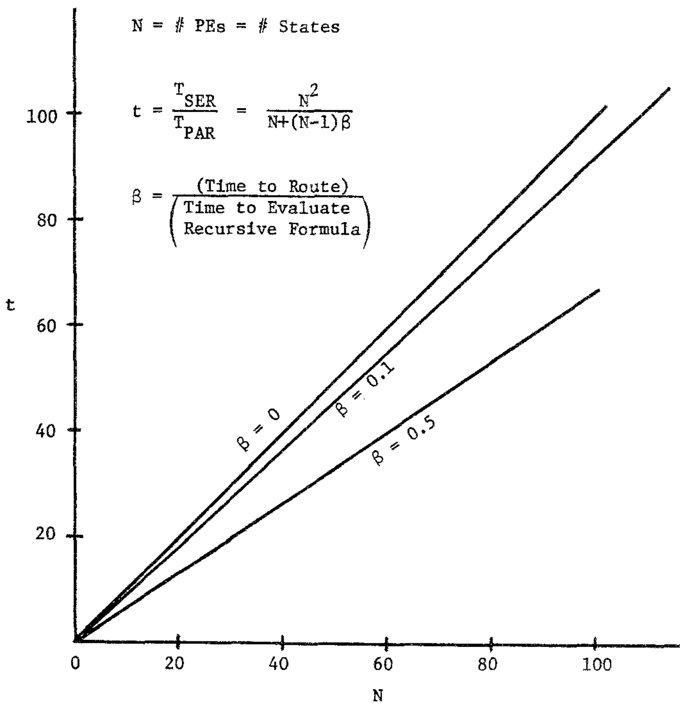

Fig. 5. Parallel states time savings.

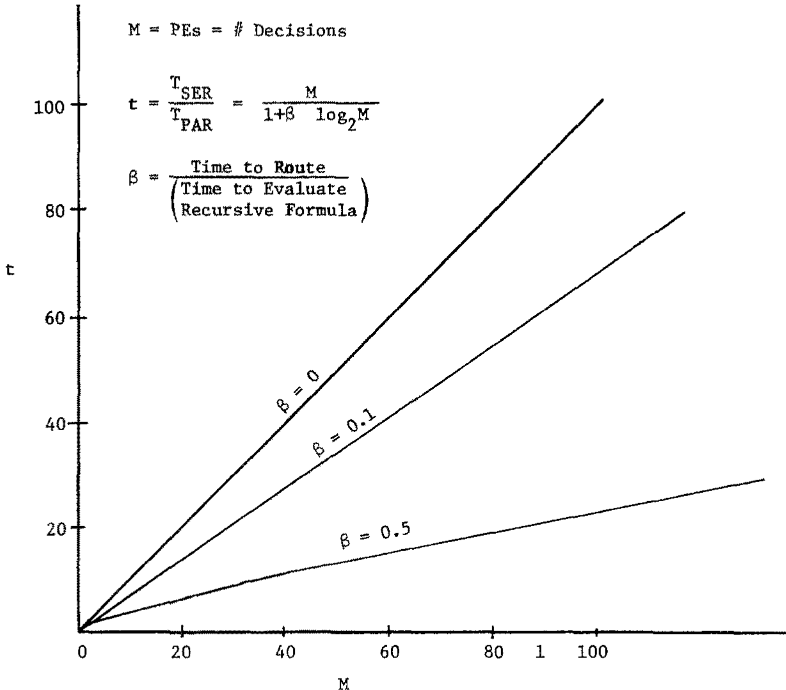

Fig. 6. Parallel decisions time savings.

illustrate the time savings of the parallel schemes over serial computation. In both cases, the actual position of the curves is governed by the amount of time necessary to route information between PEs. In the case of parallel states, this amount of overhead simply grows linearly with the number of states. In the parallel decisions scheme, the overhead computation is a function of $\log_2$ (number of decisions) and, hence, diminishes the efficiency of the parallel algorithm more for a small number of decisions than for a large number.

4. Parallel Stages and States

Replacement of the stage loop in the evaluation of the recursive formula is not as straightforward as the state and decisions loops. However, in certain special cases, the following parallel stages and states algorithm may quickly converge to an optimal solution. The required number of PEs is equal to the product of the number of states at a stage times the number of stages. Figure 7 shows the layout of PEs, each PE

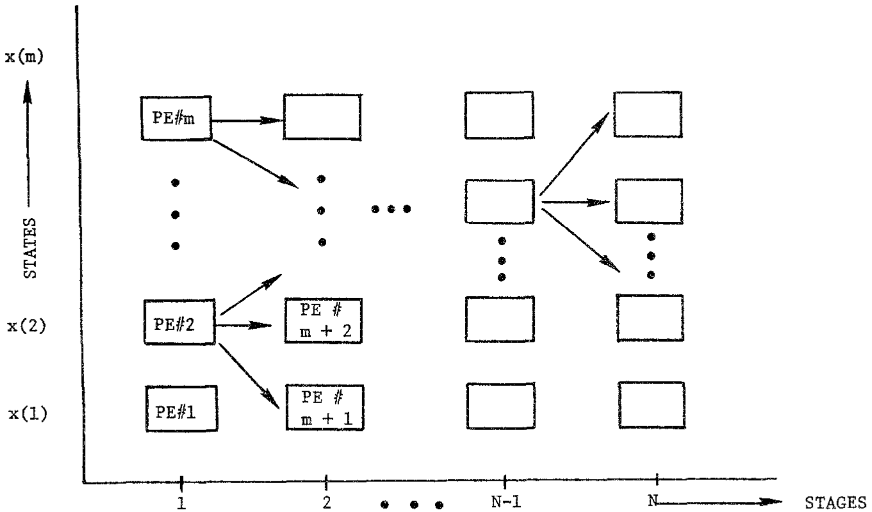

Fig. 7. Parallel stages and states.

corresponding to one quantized state at a stage. Each PE is connected to its neighbors to which possible state transitions might take place. The algorithm executes the following steps.

(i) Each PE is given a nominal optimal value function $V_i(x_i)$ to start computations.

(ii) Each PE computes the index of one of its neighbors using a quantized decision choice.

(iii) Each PE retrieves the nominal $V_i(x_i)$ from the proper neighbor chosen in Step (ii).

(iv) Each PE computes an updated value of its own nominal $V_i(x_i)$ using the r.h.s. of Eq. (3), compares the answer with the previously stored value, and retains the minimum value.

(v) Return to Step (ii), at least until all decisions are used once, and until the difference between successive stored values of $V_i(x_i)$ is small.

Convergence of the above algorithm to an optimum at all states and stages is guaranteed in at most N cycles through the decision choices. In that time, the optimum answers will have passed from stage N down to stage 1. With well-chosen starting values, convergence may be expected to occur much sooner (Ref. 2).

An example of another approach to the parallel stages algorithm is given in Ref. 6, where a multistage allocation problem is treated.

5. Parallelism with Shift Vectors

Any process with scalar input that can be represented by the following nth-order difference equation (discretized nth-order differential equation):

$$x_{i+n} = f_i(x_{i+n-1}, \ldots, x_{i+1}, x_i, u_i),$$

may also be written in the equivalent shift vector form (Ref. 3)

$$\begin{aligned} x_{i+1}^1 &= x_i^2, \\ x_{i+1}^2 &= x_i^3, \\ &\vdots \\ x_{i+1}^n &= f_i(x_i^1, \ldots, x_i^n, u_i), \end{aligned}$$

where the entire shift vector is $x_i = \text{col}(x_i^1, \ldots, x_i^n)$.

The advantage of using shift vectors with dynamic programming is that the usual n-dimensional problem is reduced to a number of parallel one-dimensional problems. Figure 8 illustrates this for a two-dimensional state. Th shift vector equations are

$$\begin{aligned} x_{i+1}^1 &= x_i^2, \\ x_{i+1}^2 &= f_i(x_i^1, x_i^2, u_i), \qquad i = 1, 2, \ldots, n. \end{aligned}$$

Thus, for any fixed value of x_i^2, the value of x_{i+1}^1 is fixed and consequently all transitions from the line $x_i^2 = C$ lie along the line $x_{i+1}^1 = C$, C a constant. Therefore, to solve Eq. (3) along any line at stage i, only the stored data along its corresponding line at stage $i + 1$ is needed in high speed memory. Furthermore, all parallel lines at stage i, corresponding

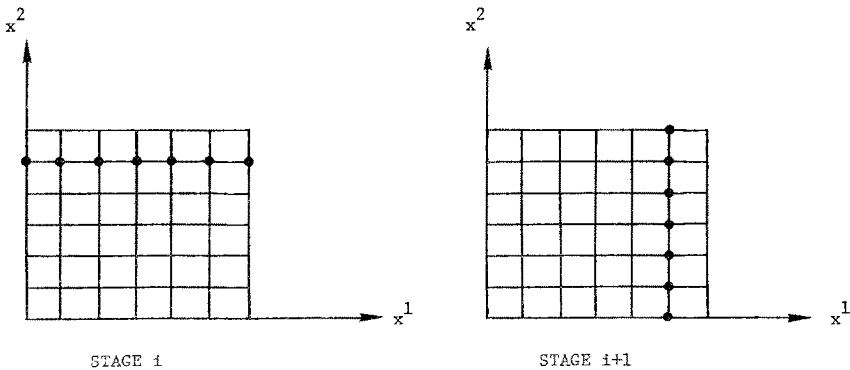

Fig. 8. Admissible state transitions, two-dimensional case.

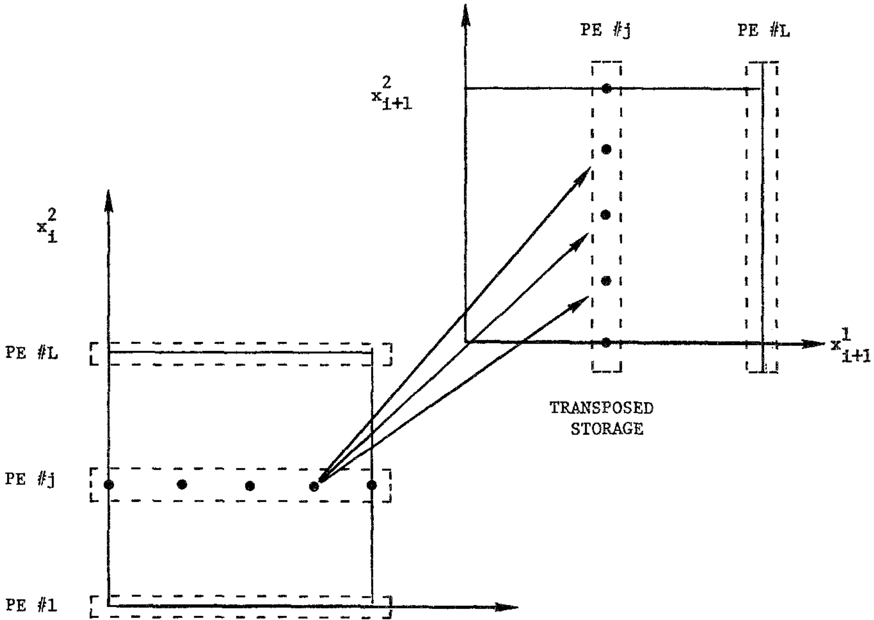

Fig. 9. Parallel two-dimensional case.

to all quantized values of x_i^2, can be processed simultaneously. Figure 9 depicts this scheme. The same is true for the n-dimensional case except that a PE is needed for each value of the quantized vector $(x_i^2, \ldots, x_i^n)$, as shown in Fig. 10.

Hence, given a parallel processor with $(QS)^{n-1}$ PEs, the computation time necessary to solve an n-dimensional problem using shift vectors is

$$\text{FIXED} \begin{cases} x^1_{i+1} = x^2_i \\ x^2_{i+1} = x^3_i \\ x^{n-1}_{i+1} = x^n_i \end{cases}$$

FIXED FOR EACH PE

$$\text{VARIABLE} \longrightarrow \quad x^n_{i+1} = f_i(x^1_i, x^2_i, \ldots, x^n_i, u_i)$$

VARIABLE IN EVERY PE

Fig. 10. Required PEs $= (QS)^{n-1}$ for parallel solution in n-dimensional case.

the same as the time for solving a scalar state problem with the same quantization, plus the slight additional time needed to transpose the stored answers between stages.

6. State Increment Dynamic Programming in Parallel

The state increment dynamic programming method developed by Larson (Ref. 2) applies to continuous time systems which are discretized in time. The system dynamics are assumed to be described by the difference equation

$$x(t + \delta t) = x(t) + f(x, u, t)\,\delta t,$$

where δt is the time interval over which the decision variable $u(t)$ is held constant. We wish to minimize a performance integral over a fixed time interval (t_0, t_f), that is,[6]

$$\text{minimize} \int_{t_0}^{t_f} g(x, u, t)\,dt.$$

The dynamic programming recursive formula for obtaining the feedback solution is (Ref. 2)

$$V(x, t) = \min_u [g(x, u, t)\,\delta t + V(x + f(x, u, t)\,\delta t, t + \delta t)], \tag{4}$$

with starting condition $V(x, t_f) = 0$, where the optimal value function V is defined as

$$V(x, t) = \min_u \int_t^{t_f} g(x, u, t)\,dt.$$

The state increment procedure is similar to the previously described finite-dimensional methods in that Eq. (4) is solved at a finite number of quantized state values and also at specific stages, determined by choosing a fixed time increment Δt and quantizing the time axis. Using the state increment method, however, the optimal value function $V(x, t)$ at a point is evaluated by considering just its *nearest neighbors*, as shown in Fig. 11. The neighbors are defined as those which are one state increment away (plus or minus) from the state value in question.

In order to solve Eq. (4), the time increment δt is allowed to vary until either one increment of change takes place in one of the components

[6] With the discretization in time, this integral is equivalent to a finite sum of constant performance values over all time intervals.

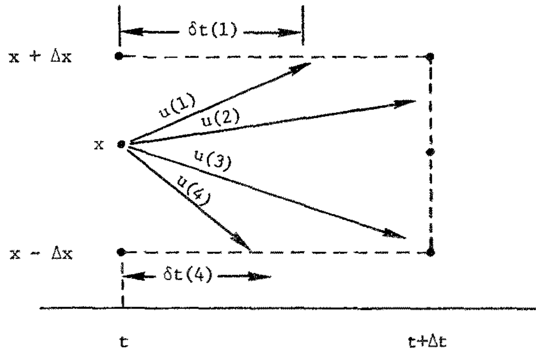

Fig. 11. Computation at a state using its nearest neighbors.

of the state, using a particular quantized decision value, or until $\delta t = \Delta t$. This is signified by the formula

$$\delta t(j) = \min_{i=1,\ldots,n} [| \Delta x^i \mid f^i(x, u(j), t)|, \Delta t]. \tag{5}$$

Hence, all candidate r.h.s.'s of Eq. (4) can be evaluated by interpolating between values of $V(x, t)$ stored at the nearest neighbors.

The state increment method can be used within the parallel stages and states algorithm, which was previously described. The advantage of using the state increment method is that far fewer connections and data transfers between PEs are necessary. The necessary communication is illustrated in Fig. 12.

7. Successive Approximation

Larson and Korsak have developed an iterative technique, originally described in Ref. 7, which greatly alleviates storage problems caused by

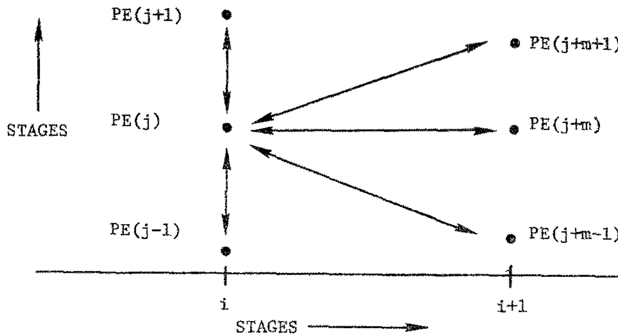

Fig. 12. Communication links for state increment dynamic programming in parallel.

$(QS_i)^{n_i}$, that is, the dimensionality and quantization of the state vector. This method works best when the state equations are decoupled to some degree, but this requirement is not necessary. The algorithm successively solves lower-dimensional problems and, thus, improves the value of the performance by optimizing with respect to one or a small number of decision variables at a time. This procedure can be carried out in parallel by allowing each PE to optimize with respect to one decision variable or group of decision variables $(U^1,..., U^m)$. Figure 13 shows the overall computational process.

The lower-dimensional subproblem which is solved by PE_j is defined as

$$\min_{\{u_i^j\}} \sum_{i=1}^{N} h_i(x_i^j, u_i^j),$$

where

$$x_{i+1}^j = f_i^j(x_i^1, x_i^2,..., x_i^j,..., x_i^n, u_i^j),$$

$$x_1 = C, \qquad x_i^j \in X^j, \qquad u_i^j \in U^j.$$

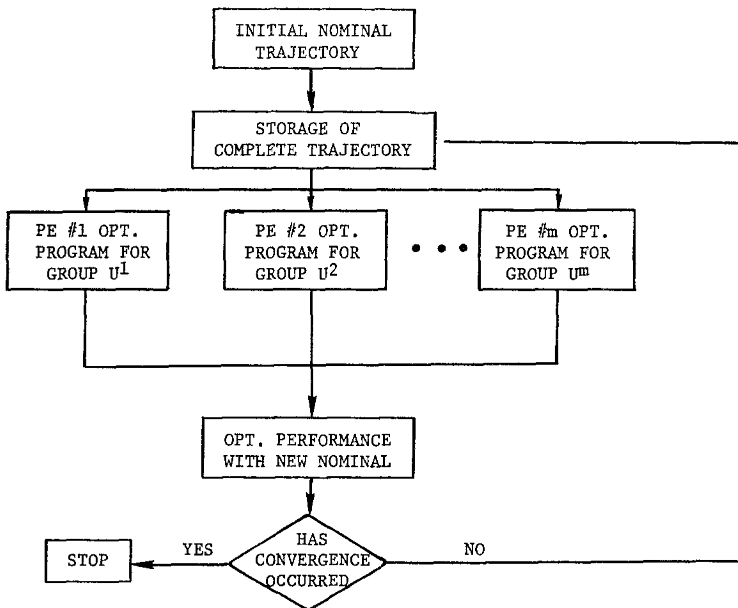

Fig. 13. Flowchart for successive approximations in parallel.

Here, the sequences $\{x_i^k, i = 1, 2,..., N, k \neq j\}$ are fixed, which imposes $n - 1$ constraints on the sequence $\{u_i^k, k \neq j\}$, that is,

$$u_i^k = \alpha_i(\{x_i^k\}),$$
$$x_{i+1}^k = f_i^k(x_i^1,..., x_i^n, u_i^k), \qquad k \neq j.$$

Also, the functions $\{h_i\}$ are determined from the original functions $\{g_i\}$ by fixing the sequences $\{x_i^k, k \neq j\}$.

8. Discussion

A number of alternatives for the use of parallel processors to solve dynamic programming problems have been presented. It may seem that some of the algorithms proposed require a number of PEs well in excess of any machine currently being developed or even being considered for development. However, this objection is resolved when we consider that the types of computational processes under consideration do not require the flexibility of a general purpose computer. A machine such as ILLIAC IV has very powerful PEs from a computational viewpoint, but it would be of limited value in this application because it has only 64 PEs. However, it is certainly feasible, and well within the realm of current electronic technology, to consider constructing special-purpose computers with thousands or even millions of primitive PEs whose only purpose would be to deal with important subclasses of the problems presented above.

In closing, we remark that, when faced with a specific problem, the proper choice of algorithm is generally not at all clear. Whether to use parallel states, decisions, stages, or a combination of these methods is highly dependent upon the problem and the computational resources available. It is unreasonable to expect a uniform rule for algorithm selection which would apply to all cases, although experience will undoubtedly supply several rules-of-thumb. The determination of these rules, as well as the investigation of related areas such as quasilinearization, invariant imbedding, etc., will be reported in subsequent papers.

References

1. Barnes, G. H., *et al.*, *The ILLIAC IV Computer*, IEEE Transactions on Computers, Vol. C-17, No. 8, 1968.
2. Larson, R. E., *State Increment Dynamic Programming*, American Elsevier Publishing Company, New York, New York, 1968.

3. WONG, P. J., *Dynamic Programming Using Shift Vectors*, Stanford University, Center for Systems Research, Report No. 6453-1, 1967.
4. LARSON, R. E., and KORSAK, A. J., *A Dynamic Programming Successive Approximations Technique with Convergence Proofs*, Automatica, Vol. 6, 1970.
5. BELLMAN, R. E., *Dynamic Programming*, Princeton University Press, Princeton, New Jersey, 1957.
6. GILMORE, P. A., *Structuring of Parallel Algorithms*, Journal of the Association for Computing Machinery, Vol. 15, No. 2, 1968.
7. BELLMAN, R. E., and DREYFUS, S. E., *Applied Dynamic Programming*, Princeton University Press, Princeton, New Jersey, 1962.

Additional Bibliography

KARP, R., and MILLER, R., *Parallel Program Schemata*, Journal of Computer and System Science, Vol. 3, 1969.

TABAK, D., *Computational Improvement of Dynamic Programming by Multiprocessing Computers*, IEEE Transactions on Automatic Control, Vol. AC-13, No. 5, 1968.

Printed in Belgium