SUPERCOMPUTER OPTIMIZATIONS FOR STOCHASTIC OPTIMAL CONTROL APPLICATIONS*

Siu-Leung Chung, Floyd B. Hanson and Huihuang Xu

Laboratory for Advanced Computing
Department of Mathematics, Statistics, and Computer Science
University of Illinois at Chicago
P. O. Box 4348; M/C 249
Chicago, IL 60680

SUMMARY

Supercomputer optimizations for a computational method of solving stochastic, multibody, dynamic programming problems are presented. The computational method is valid for a general class of optimal control problems that are nonlinear, multibody dynamical systems, perturbed by general Markov noise in continuous time, i.e., nonsmooth Gaussian as well as jump Poisson random white noise. Optimization techniques for vector multiprocessors or vectorizing supercomputers include advanced data structures, loop restructuring, loop collapsing, blocking, and compiler directives. These advanced computing techniques and supercomputing hardware help alleviate Bellman's *curse of dimensionality* in dynamic programming computations, by permitting the solution of larger multibody problems. Possible applications include lumped flight dynamics models for uncertain environments, such as large scale and background random aerospace fluctuations.

INTRODUCTION

The objective of this research is to provide a general, highly optimized, computational treatment of stochastic optimal control applications in continuous time. Advanced computing techniques have been implemented so that stochastic dynamic programming algorithms can be used to solve larger optimal control problems than possible by ordinary computing methods. Optimization techniques will help alleviate *Bellman's curse of dimensionality*, in that the computational and memory requirements grow exponentially as the dimension of the state space increases, limiting the size of the control problem that can be computed. Computer optimization techniques can help alleviate Bellman's curse by permitting larger, but still hardware limited problems to be computed. Optimization consists of parallelization and vectorization methods to enhance performance on advanced computers, such as parallel processors and vectorizing supercomputers. Preliminary results for massively parallel processors are also presented.

General Markov random noise in continuous time consists of two kinds, Gaussian and Poisson. Gaussian white noise, being continuous but nonsmooth, is useful for modeling background random fluctuations, such as turbulence and moderate environmental variations. Poisson white noise (its frequency spectrum is also flat like Gaussian noise), being discontinuous, is useful for modeling large random fluctuations, such as shocks, collisions, unexpected external events and large environmental changes. Our general feedback control approach combines the treatment of both linear and nonlinear (i.e., singular and

*This work was supported by the National Science Foundation Computational Mathematics Program under grant DMS-88-06099 at the University of Illinois at Chicago, by the Argonne National Laboratory Advanced Computing Research Facility, by the University of Illinois at Urbana National Center for Supercomputing Applications, and by the UIC Workshop Program on Scientific Supercomputing.

nonsingular) control through the use of small to moderate quadratic costs. The methods also handle deterministic and stochastic control in the same code, making it convenient for checking the effects of stochasticity on the application. Some actual applications are models of resources in an uncertain environments [16], [13], [8].

The Markov, multibody dynamical system is illustrated in Figure 1 and is governed by the stochastic differential equation (SDE):

$$d\mathbf{y}(s) = \mathbf{F}(\mathbf{y}, s, \mathbf{u})ds + G(\mathbf{y}, s)d\mathbf{W}(s) + H(\mathbf{y}, s)d\mathbf{P}(s) \ , \tag{1}$$

with initial value $\mathbf{y}(t) = x$, $0 < t < s < t_f$, $\mathbf{y}(s) \in \mathcal{D}_y$, $\mathbf{u} \in \mathcal{D}_u$, where $\mathbf{y}(s)$ is the $m \times 1$ multibody state vector at time s starting at time t, $\mathbf{u} = \mathbf{u}(\mathbf{y}, s)$ is the $n \times 1$ feedback control vector, $\mathbf{F}$ is the $m \times 1$ deterministic nonlinearity vector, $\mathbf{W}$ is the r-dimensional normalized Gaussian white noise vector, $\mathbf{P}$ is the independent q-dimensional Poisson white noise vector with jump rate vector $[\lambda_i]_{q\times 1}$, G is an $m \times r$ diffusion coefficient array, and H is an $m \times q$ Poisson amplitude coefficient array. In a more general treatment, the Poisson jump amplitude can also be random.

The control criterion is the optimal expected cost performance,

$$V^*(\mathbf{x}, t) \ = \ \min_{\mathbf{u}} \left[\underset{\{\mathbf{P},\mathbf{W}\}}{\text{MEAN}} \left[V[\mathbf{y}, s, \mathbf{u}, \mathbf{P}, \mathbf{W}] \mid \mathbf{y}(t) \ = \ \mathbf{x} \right] \right] , \tag{2}$$

where the random total cost is

$$V[\mathbf{y}, t, \mathbf{u}, \mathbf{P}, \mathbf{W}] \ = \ \int_t^{t_f} ds \ C(\mathbf{y}(s), s, \mathbf{u}(\mathbf{y}(s), s)) \ , \tag{3}$$

on the time horizon $(t, \ t_f)$. The instantaneous cost function $C \ = \ C(\mathbf{x}, t, \mathbf{u})$ is assumed to be at least a quadratic function of the control,

$$C(\mathbf{x}, t, \mathbf{u}) \ = \ C_0(\mathbf{x}, t) \ + \ \mathbf{C}_1^T(\mathbf{x}, t)\mathbf{u} \ + \ \tfrac{1}{2}\mathbf{u}^T C_2(\mathbf{x}, t)\mathbf{u} \ . \tag{4}$$

C_2 is assumed to be positive definite, so that large controls are much more costly on a per unit basis. In addition, the dynamics in (1) are assumed to be linear in the controls,

$$\mathbf{F}(\mathbf{x}, t, \mathbf{u}) \ = \ \mathbf{F}_0(\mathbf{x}, t) \ + \ F_1(\mathbf{x}, t)\mathbf{u} \ , \tag{5}$$

remaining nonlinear in the state variable $\mathbf{x}$.

The Bellman functional PDE of dynamic programming (or Hamilton-Jacobi-Bellman Equation),

$$\begin{aligned} 0 \ &= \ \frac{\partial V^*}{\partial t} \ + \ \mathcal{H}[V^*] \\ &\equiv \ \frac{\partial V^*}{\partial t} \ + \ \mathbf{F}_0^T \nabla V^* \ + \ \tfrac{1}{2} G G^T(\mathbf{x}, t) : \nabla \nabla^T V^* \\ &+ \ \sum_{l=1}^{q} \lambda_l \cdot [\ V^*(\mathbf{x} + \mathbf{H}_l(\mathbf{x}, t), t) - V^*(\mathbf{x}, t) \ ] \\ &+ \ C_0 \ + \ (\tfrac{1}{2}\mathbf{U}^* - \mathbf{U}_R)^T C_2 \mathbf{U}^* \ , \end{aligned} \tag{6}$$

follows from the generalized *Itô* chain rule for Markov SDEs as in [7] and [16]. Here, $\mathbf{U}^*$ is the optimal feedback control computed by constraining the unconstrained or regular control,

$$\mathbf{U}_R(\mathbf{x},t) \;=\; -C_2^{-1}(\mathbf{C}_1 \;+\; F_1\nabla V^*)\;, \tag{7}$$

to the control set D_u, under the assumption of positive definite quadratic costs. In general, the Bellman equation (6) is nonlinear with discontinuous coefficients due to the quadratic last term, $(\frac{1}{2}\mathbf{U}^* - \mathbf{U}_R)^T C_2 \mathbf{U}^*$, in (6) and due to the compact relationship between the constrained, optimal control and the unconstrained, regular control,

$$U_i^*(\mathbf{x},t) = \min[U_{\max,i}, \max[U_{\min,i}, U_{R,i}(\mathbf{x},t)]], \tag{8}$$

for $i = 1$ to n controls. Here, $\mathbf{U}_{\min}$ is the minimum control constraint vector and $\mathbf{U}_{\max}$ is the maximum. As the constraint components are attained, the optimal control component U_i^*, changes from the regular control component, $U_{R,i}$, to components of the constraints, $U_{min,i}$ or $U_{max,i}$, which in general are functions of state and time. In (6), the symbol (:) denotes the scalar matrix product $A : B = \sum_{i=1}^{m}\sum_{j=1}^{m} A_{ij}B_{ij}$, assuming B is symmetric. It is important to note that the principal equation, the Bellman equation (6), is an exact equation for the optimal expected value V^* and does not involve any sampling approximations such as the use of random number generators in simulations.

As the number of state variables, m, increases, the spatial dimension rises, and computational difficulties are present that can compare to those of three-dimensional fluid dynamics computations. Thus there is a great need to make use of advanced-architecture computers, to use parallelization as well as vectorization, in order to solve larger state space systems. The Panel on Future Directions in Control Theory [6] stresses the importance of making gains in such areas as nonlinear control, stochastic control, optimal feedback control and computational methods for control. This paper is a report on our efforts to treat all of the above mentioned areas combined from the point of view of computational control.

SYMBOLS

$\mathbf{C}, C_0, C_1, C_2$	cost coefficients (eq. (4))
$\mathbf{DX}, DT$	state mesh increment, time increment (eq. (9))
$\mathbf{F}, \mathbf{F}_0, F_1, FV$	nonlinearity function coefficients (eq. (5))
G	Gaussian noise amplitude matrix (eq. (1))
$H, \mathbf{H}_l$	Poisson noise jump amplitude (eq. (1), (6))
$\mathcal{H}$	Hamiltonian for Bellman Equation (eq. (6))
$\mathbf{j}, js, jv$	indices for state mesh points (eq. (9), (15), (18))
m, M	state dimension, number of mesh points for each state (eq. (1), (9))
n	control space dimension (eq. (1))
$\mathbf{P}$	Poisson noise vector (eq. (1))
q	Poisson noise dimension (eq. (1))
r	Gaussian noise dimension (eq. (1))
s	forward time variable (eq. (1))
t, t_f, T_k	backward time variable, final time, discrete time (eq. (3), (9))
$\mathbf{u}$	control vector (eq. (1))
$\mathbf{U}_R, \mathbf{U}*$	regular control, optimal control (eq. (7), (8))
$\mathbf{U}_{min}, \mathbf{U}_{max}$	control constraint vectors (eq. (8))
$V, V*$	total cost, optimal expected total cost (eq. (3), (2))
$\mathbf{W}$	Gaussian noise vector (eq. (1))
$\mathbf{x}, X_{j_i}$	initial state vector, discrete state (eq. (2), (9))

$\mathbf{y}$	forward state variable (eq. (1))
λ_l	component of Poisson jump rate vector (eq. (6))

THE BASIC COMPUTATIONAL PROCEDURE

The integration of the PDE in (6) is backward in time, because V^* is specified finally at the final time $t = t_f$, rather than at the initial time. A summary of the discretization in state and backward time is given below:

$$\begin{aligned} \mathbf{x} &\longrightarrow \mathbf{X_j} = [X_{ij_i}]_{m\times 1} = [X_{i1} + (j_i - 1)\cdot DX_i]_{m\times 1}\ , \\ \mathbf{j} &= [j_i]_{m\times 1}\ ,\ \text{where } j_i = 1 \text{ to } M_i\ ,\ \text{for } i = 1 \text{ to } m\ ; \\ s &\longrightarrow T_k = t_f - (k-1)\cdot DT\ ,\ \text{for } k = 1 \text{ to } K\ ; \\ V^*(\mathbf{X_j}, T_k) &\longrightarrow V_{\mathbf{j},k}\ ;\quad \mathcal{H}[V^*](\mathbf{X_j}, T_{k+\frac{1}{2}}) \longrightarrow \mathcal{H}_{\mathbf{j},k+\frac{1}{2}}\ ; \end{aligned} \tag{9}$$

where DX_i is the mesh size for state i and DT is the step size in backward time.

The numerical algorithm is a modification of the predictor corrector, Crank Nicolson methods for nonlinear parabolic PDEs in [5]. Modifications are made for control feedback, switch term optimization and delay term calculations. Derivatives and Poisson induced differences are approximated with an accuracy that is second order in the local truncation error $\mathcal{O}^2(DX_i)$, at all interior and boundary points. Even though the Bellman equation (6) is a single PDE, the process of solving it not only produces the optimal expected cost V^*, but also the optimal expected feedback control law $\mathbf{U}^*$. This is because the Bellman equation is a functional PDE, in which the computed regular control feeds back into the optimal cost and the optimal cost feeds back into regular control through its gradient. The nonstandard part of the algorithm is to decompose this tightly coupled analytical feedback system so that both the cost and the control can be calculated by successive iterations, such that each successive approximation of one quantity improves the next approximation of the other quantity. While our procedure may look superficially like a standard application of finite differences, it is not due to the nonstandard control features mentioned above. For these reasons, we are not aware of any other successful stochastic dynamic programming code that treats anywhere near the generality of applications that we treat and with the advanced computing techniques that we use, especially with regard to Poisson noise. Variations of this algorithm have been successfully utilized in [16] and [8]. Quadrat and his co-workers [1] discuss several algorithms for stochastic dynamic programming problems that admit stationary solutions, and describe an expert system for their solution.

Prior to calculating the values, $V_{\mathbf{j},k+1}$, at the new $(k+1)^{st}$ time step for $k = 1$ to $K-1$, the old values, $V_{\mathbf{j},k}$ and $V_{\mathbf{j},k-1}$, are assumed to be known, with $V_{\mathbf{j}0} \equiv V_{\mathbf{j}1}$. The algorithm begins with an convergence accelerating *extrapolator (x) start*:

$$V^{(\mathbf{x})}_{\mathbf{j},k+\frac{1}{2}} = \tfrac{1}{2}(3\cdot V_{\mathbf{j},k} - V_{\mathbf{j},k-1})\ . \tag{10}$$

The extrapolated values are use to calculate updated values of the gradient DV, the second order derivatives DDV, the Poisson functional terms (V^* evaluated at $(\mathbf{x}+\mathbf{H})$), the regular control UR, the optimal feedback control U^*, and the spatial functional $\mathcal{H}_{\mathbf{j},k+0.5}$ of the Bellman equation. These evaluations are used in the *extrapolated predictor (xp) step*:

$$V^{(\mathbf{xp})}_{\mathbf{j},k+1} = V_{\mathbf{j},k} + DT\cdot\tfrac{1}{2}\mathcal{H}^{(\mathbf{x})}_{\mathbf{j},k+\frac{1}{2}}\ . \tag{11}$$

which are then used in the *predictor evaluation (xpe) step*:

$$V^{(\mathrm{xpe})}_{\mathbf{j},k+\frac{1}{2}} = \tfrac{1}{2}(V^{(\mathrm{xp})}_{\mathbf{j},k+1} + V_{\mathbf{j},k}) , \tag{12}$$

and continuing with other terms of the spatial functional $\mathcal{H}$. The evaluated predictions are used in the *corrector (xpec) step*:

$$V^{(\mathrm{xpec},\gamma+1)}_{\mathbf{j},k+1} = V_{\mathbf{j},k} + DT \cdot \mathcal{H}^{(\mathrm{xpe},\gamma)}_{\mathbf{j},k+\frac{1}{2}} \tag{13}$$

for $\gamma = 0$ to γ_{max} until the stopping criterion is met, with *corrector evaluation (xpece) step*:

$$V^{(\mathrm{xpece},\gamma+1)}_{\mathbf{j},k+\frac{1}{2}} = \tfrac{1}{2}(V^{(\mathrm{xpec},\gamma+1)}_{\mathbf{j},k+1} + V_{\mathbf{j},k}) . \tag{14}$$

The stopping criterion for the corrections is formally derived from a comparison to a predictor corrector convergence criterion for a linearized, constant coefficient PDE. A robust mesh selection method is used to determine the stopping criterion, so that only a couple of corrections are needed, except at the first time step. The proper selection of the time to state mesh ratio guarantees that the corrections for the comparison equation converge, whether the Bellman equation is parabolic-like when the Gaussian noise is present or hyperbolic-like when there is no Gaussian noise.

Current efforts are concentrated on implementing the code on the Alliant FX/8, Cray X-MP/48, Cray 2S/4-128, and the Connection CM-2 for more general multi-state and multi-control applications. In order to implement the code for arbitrary state space dimension, a more flexible data structure is needed for the problem arrays, $\mathbf{F}$, G and H, as well as for the solution arrays, V along with its derivatives and $\mathbf{U}$.

The advantages of the algorithm is that it 1) permits the treatment of general continuous time Markov noise or deterministic problems without noise in the same code, 2) maintains feedback control, 3) permits the cheap control limit to linear singular control to be found from the same quadratic cost code, and 4) produces very vectorizable and parallelizable code whose performance is described in the next section.

ADVANCED SUPERCOMPUTER OPTIMIZATION

The code for the algorithm has been developed and tested on three advanced architecture machines, the ACRF Alliant FX/8 vector multiprocessor at Argonne National Laboratory; the NCSA Cray X-MP/48 and the NCSA Cray 2S/4-128 at the University of Illinois in Urbana; the massively parallel Connection Machine CM-2 at both the ACRF and NCSA. The Alliant FX/8, with its superb concurrent outer, vector inner (COVI) parallelizing compiler, is mainly used to test for the parallelization of the code. The Cray X-MP/48, noted for its very fast pipelined processing unit, is used for the testing of small and moderate size code (less than 1 MW, where MW denotes a megaword or one million words). As the number of states grows, the problem size grows exponentially, we have to make use of the huge internal memory (up to 128 MW) of the Cray 2S/4-128 or large numbers of parallel processors on the Connection Machine CM-2.

The present code under testing has been obtained from the three-state, three-control modification of Hanson's two-state, two-control resource model [8]. Modifications have been made to the present code so that it can apply to arbitrary number of state variables and mesh points by just changing a few parameters. numbers of state variables m and mesh points M.

Initial parallelization and vectorization of the algorithm were done by what might be called the

"Machine Computational Model Method," i.e., tuning the code to optimizable constructs that are automatically recognized by the compiler, with the Alliant FX/8 vector multiprocessor [2] in mind. All inner double loops were reordered to fit the Alliant *concurrent-outer, vector-inner* (*COVI*) model. All non-short single loops were made *vector-concurrent.* Short loops became *scalar-concurrent* only. Multiple nested loops were reordered with the two largest loops innermost. A total of 37 out of 39 loops was optimized for the two-state code, two-control model with Poisson noise. Detailed results are reported in [8], [9] and [10].

The relative performance of column oriented versus row oriented code is discussed in [11]. Dongarra, Gustavson, and Karp [4] have demonstrated that loop reordering gives vector or supervector performance for standard linear algebra loops on a Cray 1 type column oriented FORTRAN environment with vector registers. However, for the stochastic dynamic programming application, the dominant loops are non-standard linear algebra loops, so that the preference for column oriented loops is not a rule, as demonstrated on the Alliant vector multiprocessor [11]. The present code under testing has up to four states and controls, with Gaussian as well as Poisson noise. This code is a general modification of the two-state, two-control model.

Vector Data Structure

In the original code, the data structure for the problem arrays, $\mathbf{F}$ and G, the solution arrays V, the derivative arrays, and the control arrays $\mathbf{U}$, depend on all the numerical node indices, $js(is)$, for all state variables. The resulting data structure takes the form:

$$F(is, js(1), js(2), \cdots, js(m)) \tag{15}$$

for each state equation, $is = 1$ to m, with the nonlinearity function used as an example. If it is assumed that there are a common number $M = M_1 = M_2 = \cdots = M_m$ of nodes per state, then $js(is) = 1$ to M points for $is = 1$ to m states. As a consequence, the typically dominant loops for the computation of the nonlinearity function F, the solution gradient DV, and similar arrays, are nested to a depth of at least $m + 1$. A typical loop will take the form:

```
      do 1 i = 1, m
         do 1 j1 = 1, M
                 :
            do 1 jm = 1, M
                    :
1                F(i,j1,j2,···,jm) = ·····
```

This state size dependent loop nest depth level of $m + 1$ inhibits the development of general multibody algorithms, especially when the state size m is incremented and the number of loops in each nest has to be changed. Also, vectorization is inhibited for compilers that vectorize only the most inner loop. As the number of states grows, the computational load will grow asymptotically like some multiple of

$$m \cdot M^m = m \cdot e^{m \ln(M)}, \tag{16}$$

i.e., the load grows exponentially in the number of states m. The exponential growth in (16) is merely a quantitative expression of *Bellman's curse of dimensionality.*

One way around this inhibiting structure is to use a *vector data structure* [12]:

$$\mathsf{FV(is, jv)} \tag{17}$$

to replace the original *hypercube type of data structure* in (15), using the nonlinearity vector as an example, such that all the numerical nodes are collected into a single vector indexed by the global state index jv, where $jv = 1$ to M^m over all state nodes.

$$jv = \sum_{i=1}^{m} (js(i) - 1) \cdot M^{i-1} + 1, \tag{18}$$

in the the case that the state mesh size, M_i has a common value of M for all i.

Both the direct mapping from the original data structure to the vector data structure and the inverse mapping are needed to compute the amplitude functions, $\mathbf{F}$, G and H, as well as the derivatives of V^*, because these quantities depend on the original formulation. The pseudo-inverse of the vector index in (18) can be shown to permit the recovery of the individual state indices by way of integer arithmetic:

$$js(is; jv) = 1 + [jv - 1 - \sum_{i=is+1}^{m} (js(i; jv) - 1) \cdot M^{i-1}]/M^{is-1}, \tag{19}$$

recursively, for $is = m$ to 1, by back substitution, with $\sum_{i=m+1}^{m} a_i \equiv 0$, as long as each state has the same number of discrete nodes M. The vector data structure of (17) to (19) results in major do loop nests of the order of 1 to 2, rather than order of $m + 1$.

A typical vector data structure loop has the form

```
      do  2  i = 1, m     !  parallel loop.
         do  2  jv = 1, M**m     !  vector loop.
                  ⋮
2                FV(i,jv) = ·····
```

resulting in collapsing the loop nest depth from $m + 1$ to a depth 2, independent of the number of states m. This is analogous to the automatic compiler technique of *loop collapsing* on the Alliant for simple loops.

Table I shows the performance of the code for $m = 3$ states and $M = 16$ nodes per state on the Alliant FX/8 at Argonne National Laboratory's ACRF. implemented and run on vector multiprocessors which will be discussed in the following two subsections.

Parallelization in Alliant FX/8

When loop 2 above is executed on multiprocessors such as the Alliant FX/8, due to the *COVI* (concurrent-outer, vector-inner) compiler optimization scheme, the i-loop will run in parallel while the jv-loop is vectorized. For machines with such architecture, the gain in speed is achieved through the full exploitation of all its processors. If the number states m is less than the maximum number of processors (the maximum number of processors is eight on the Alliant), performance ceases to improve beyond m processors as demonstrated in Table I when the number of processors p is greater than three. The speedup $S_{p,2} = T_{1,2}/T_{p,2}$ for loop 2 also levels off at roughly 2.9 in this table starting at $p = 3$. This means degradation in efficiency because a large proportion of processors available are sitting idle.

One simple modification of the loop structure will solve the problem by parallelizing and vectorizing the entire loop nest, further enhancing the performance. This is illustrated by the restructuring of loop 2 by a compiler directive in loop 3 below.

```
      do  3  jv = 1, M**m     !  vector-concurrent loop.
CVD$L NOVECTOR
```

```
              do  3  i = 1, m    !  scalar loop.
3                 FV(i,jv) = ·····
```

Due to the flexibility of the optimization scheme of the FX Fortran compiler, we can choose whichever loop we want to parallelize and vectorize by inserting suitable compiler directives, such as *CVD$L NOVECTOR* in loop 3. The i-loop is moved innermost and is forced to run in *Scalar Mode* by inserting a *CVD$L NOVECTOR* directive. The modification has two effects:

i. the outer jv-loop is forced to run in *Vector-Concurrent Mode*, hence, full parallelization of the entire work load can be achieved through self-scheduling by the compiler;

ii. moving the i-loop inner-most increases the chunk or grain size of each iteration, while overhead for parallelization and vectorization is lessened.

The modification leads to an improvement of 46% of computing time for the code running by 8 processors in the Alliant.

Table II shows the performance of the modified code, for $m = 3$ states and $M = 16$ points per state, on the Alliant FX/8 at Argonne National Laboratory's ACRF. The speedup, also given in the table, reaches a good value over six times executing on all eight Alliant processors using the form of loop 3. The last column compares the results of using loops 2 and 3 for the main stochastic dynamic programming loops and shows that the loop 3 form outperforms the loop 2 form by 1.85 times on all eight processors. Thus, the restructured loop 3 gives better load balancing that the pure vector data structure of loop 2.

Parallelization on the Cray 2

Parallelization in the Cray 2S/4-128 is done through multitasking. Basically, the compiler follows the *COVI* optimization scheme that the outer loop will run in parallel and inner loop is vectorized. In a multi-user environment such as that in NCSA, improvement through multitasking is hard to measure unless the code is run in a dedicated machine. Therefore, performance utilities such as Job Accounting (ja), are used to get an approximate measure of the CPU time and speed-up obtained.

Table III shows the performance of multitasking on the NCSA Cray 2S/4-128. Note that the timings grow drastically as either the state dimension m and the common mesh size M increase.

Performance on the Connection Machine

As the number of states increase, the performance obtained from Cray shows an exponential growth as in Table IV. Thus for a larger size problem, another solution would be to implement the problem on a massively parallel computer system. The Connection Machine CM-2 at the NCSA has $32K$ or $32,768$ bit processors and one floating point processor for every 32 bit processors.

The preliminary results obtained from the Connection Machine are shown in Table IV. We implemented the problem in the Fortran 8X language with array notation extensions and two dimension data structure. Also the CM-2 directives CMF$LAYOUT and CMF$ALIGN overlay different size of arrays, in order to reduce the internal communication time and hence improve the performance. The program is compiled with $-O$ option and run with at most $32K$ data processors in single precision. The preliminary results on the CM-2 indicate that when the problem size increases, the Connection Machine computes the problem with relatively small increase in the execution time. For instance, for the case nx=4 states, when mesh size per state increases from M=8 to M=16, the execution time increases by about 2.1 times for *Real Time* (sum of CM-2 time and the time on the front-end computer) and 5.8 times for *CM*

Time in Table IV, while on the Cray the time increases by about 23.8 times, according to Table III, which is much larger than for the CM-2.

It must be noted that the Cray and the CM-2 have different computational structures and our current Fortran 8X program is translated from our Cray algorithm and data structure. A further goal will be to modify the algorithm and data structure so that the performance on the CM-2 will be competitive with the performance on the Crays in an absolute sense.

Computation for Boundary Points

The computation of the solution gradient DV and the array of second derivatives DDV, which is carried out in the subroutine $GETDV$, requires different algorithms for the interior nodes and the boundary nodes. Due to the complexity and generality of the underlying stochastic dynamical system, the boundary values cannot be specified in general, but must be calculated from the Bellman Equation (6) itself, except for the most trivial boundaries and processes. Use of the Bellman Equation at the boundaries, makes the algorithm segments for updating the boundary values quite different from the interior values in order to maintain the same order of error as at the interior points, i.e., to avoid *numerical pollution* of the order, $\mathcal{O}(DX)^2$, at the interior points. When the vector data structure is used, the boundary nodes ($js(is) = 1$ and M) are scattered throughout the data arrays $FV(is, jv)$. Due to this nonuniform distribution of the boundary nodes, a time-consuming nested if-then-else loop has to be used in the original $GETDV$, which greatly degrades the computation speed. For the current testing code with $m = 3$ and $M = 16$, $GETDV$ takes 34% of the running time in the Alliant runs and 30% in the Cray X-MP runs. One way to alleviate this degradation is by *homogeneous global computation* and then separate *recorrection* for the boundary points.

Since the proportion of boundary nodes is generally small compare with internal nodes ($2/M$ for M mesh points per state) and all of them can be extracted explicitly from the inverse vector index (19). Hence, we can pass the whole data array through a homogeneous computation first, taking all points to be internal nodes, then recorrect the boundary nodes outside the main loop. Artificial or redundant points are added to prevent overwriting valid data, and it will be seen the resulting small addition to the memory by the use of artificial points is worth the benefit in improved performance.

Table V compares the performance for the old and new forms of $GETDV$ for different mesh points M run on the CRAY X-MP. A faster run time for the new version of 1.45 times the old version and a saving of up to 31% of running time is exhibited.

MEMORY REQUIREMENTS

Since the memory requirements grows exponentially with increases in the state variable from Bellman's *curse of dimensionality* (16), a machine with large internal memory is needed for the large state variable case. For the sake of a uniform comparison, all the testings were carried out on the NCSA Cray 2S/4-128, which possesses a huge internal memory (up to 128 MW). Table VI summarizes the memory requirements for different test codes. Also in Table VI, the memory in words is compared to the order of magnitude of the Bellman's *curse of dimensionality* term in (16). The approximate asymptote for large state dimensions m or very fine state meshes M is about 12, which gives the effective number of major loops of nest depth $m + 1$. The CPU time measurements in Table III have similar exponential growth characteristics to that of memory requirements.

CONCLUSIONS

Stochastic dynamic programming can be optimized for the a moderate and perhaps larger number of state variables using a vector multiprocessor. Loop collapsing using a vector data structure, compiler

directives making possible more efficient loop reordering, and homogeneous global computation making boundary value computation more efficient, all help obtain superior optimization of the stochastic dynamic programming code. Parallelization, vectorization, large memories, and other supercomputing features are important in solving larger state space problems In order to handle a large number of state variables, a large number of parallel processors with extremely large memory would be desirable, but Bellman's *curse of dimensionality* appears to very much weakened. Computation with massively parallel processors, like the Connection Machine CM-2, is still preliminary, but shows promise for larger state spaces. These techniques are generally applicable to other vector and parallel computers. Our general code is essentially valid for general Markov noise in continuous time, feedback control, nonlinear control and the cheap control limit.

REFERENCES

1. M. Akian, J. P. Chancelier and J. P. Quadrat, *Dynamic programming complexity and application*, in **Proc. 27th IEEE Conf. on Decision and Control, Vol. 2**, pp. 1551-1558, Dec. 1988.

2. Alliant, **FX/FORTRAN Programmer's Handbook**, Alliant Computer Systems Corporation, Acton, Mass., 1985.

3. R. E. Bellman, **Adaptive Control Processes: A Guided Tour**. Princeton: Princeton University Press, 1961.

4. J. J. Dongarra, F. G. Gustavson, and A. Karp, *Implementation of linear algebra algorithms of dense matrices on a vector pipeline machine*, **SIAM Rev.**, vol. 26, pp. 91-112, 1984.

5. J. Douglas, Jr., and T. DuPont, *Galerkin methods for parabolic equations*, **SIAM J. Num. Anal.**, vol. 7, pp. 575-626, 1970.

6. **Future Directions in Control Theory: A Mathematical Perspective**, W. H. Fleming, Chairman. Philadelphia: Society for Industrial and Applied Mathematics, 1988.

7. I. I. Gihman and A. V. Skorohod, **Controlled Stochastic Processes**. New York: Springer-Verlag, 1979.

8. F. B. Hanson, *Bioeconomic model of the Lake Michigan alewife fishery*, **Can. J. Fish. Aquat. Sci.**, vol. 44, Suppl. II, pp. 298-305, 1987.

9. F. B. Hanson, *Computational dynamic programming for stochastic optimal control on a vector multiprocessor*, **Argonne National Laboratory, Mathematics and Computer Science Division Technical Memorandum ANL/MCS-TM-113**, June 1988, 26 pages.

10. F. B. Hanson, *Computational dynamic programming on a vector multiprocessor*, **IEEE Trans. Automat. Contr.**, 12 pages, to appear, 1990.

11. F. B. Hanson, *Parallel computation for stochastic dynamic programming: Row versus column code orientation*, in **Proceedings 1988 Conference on Parallel Processing, Vol. III Algorithms and Applications**, D. H. Bailey, Editor. University Park: Pennsylvania State University Press, 1988, pp. 117-119.

12. F. B. Hanson, *Stochastic dynamic programming: Advanced computing constructs*, **Proc. 28th IEEE Conf. on Decision and Control, Vol. I**, Dec. 1989, pp. 901-903.

13. F. Hanson and D. Ryan, *Optimal harvesting with density dependent random effects*, **Natural Resource Modeling**, vol. 2, No. 3, pp. 439-455, 1988.

14. D. Ludwig, *Optimal harvesting of a randomly fluctuating resource I: Application of perturbation methods*, **SIAM J. Appl. Math.**, vol. 37, pp. 166-184, 1979.

15. C. D. Polychronopoulos, **Parallel Programming and Compilers**. Boston: Kluwer Academic Publishers, 1988.

16. D. Ryan and F. B. Hanson, *Optimal harvesting of a logistic population in an environment with stochastic jumps*, **J. Math. Biol.**, vol. 24, pp. 259-277, 1986.

Table I: Timings on the Alliant with vector data structure for loop 2 with 3 states and 16 nodes per state.

Number of Processors p	User CPU Time (seconds) $T_{p,2}$	Speedup $S_{p,2}$ $T_{1,2}/T_{p,2}$
1	83.68	1.00
2	56.23	1.49
3	30.07	2.78
4	29.24	2.86
5	29.20	2.87
6	29.24	2.86
7	28.77	2.91
8	28.64	2.92

Table II: Timings on the Alliant with order and directive modified loops for loop 3 with 3 states and 16 nodes per state.

Number of Processors p	User CPU Time (seconds) $T_{p,3}$	Speedup $S_{p,3}$ $T_{1,3}/T_{p,3}$	Improvement Ratio $T_{p,2}/T_{p,3}$
1	95.32	1.00	0.88
2	49.00	1.95	1.15
3	34.61	2.75	0.87
4	25.74	3.70	1.14
5	22.66	4.21	1.29
6	18.82	5.06	1.55
7	17.78	5.36	1.62
8	15.44	6.17	1.85

Table III: CPU time (seconds) for different state dimensions and different mesh sizes on the Cray 2S/4-128 with Multitasking.

State Variables m	Mesh Points M					
	8		16		32	
	T	Sp	T	Sp	T	Sp
2	0.033	1.60	0.130	3.66	0.685	3.54
3	0.104	3.53	1.169	2.86	24.626	1.54
4	2.527	3.14	60.151	1.73	2338.290	2.10

Table IV: CPU time (seconds) for different state dimensions and different mesh sizes on the Connection Machine CM-2 (CM Time) and front-end (Real Time).

State Variables m	Mesh Points M					
	8		16		32	
	Real Time	CM Time	Real Time	CM Time	Real Time	CM Time
3	10.97	2.79	21.51	5.83	52.53	30.14
4	36.02	11.42	76.01	66.41	—	—

Table V: Performance comparison of the old and new forms of GETDV on the Cray X-MP/48 for loop 3 with 3 states.

Number of Mesh Points M	User CPU Time (seconds)		Improvement Ratio T_{old}/T_{new}	Per Cent Savings $100(1 - T_{new}/T_{old})$
	Old GETDV T_{old}	New GETDV T_{new}		
8	0.142	0.098	1.45	31.
16	2.093	1.523	1.37	27.
24	10.378	7.679	1.35	26.
32	32.725	24.234	1.35	26.
40	78.510	58.467	1.34	26.

Table VI: Memory requirements for different state dimensions and different mesh sizes on the Cray 2S/4-128.

State Variables m	Mesh Points M			Mesh Points M		
	8	16	32	8	16	32
	Memory (MW)			Words/$m \cdot M^m$		
2	0.13	0.14	0.16	1000.	270.	78.
3	0.15	0.28	1.30	98.	23.	13.
4	0.34	3.28	50.23	21.	13.	12.

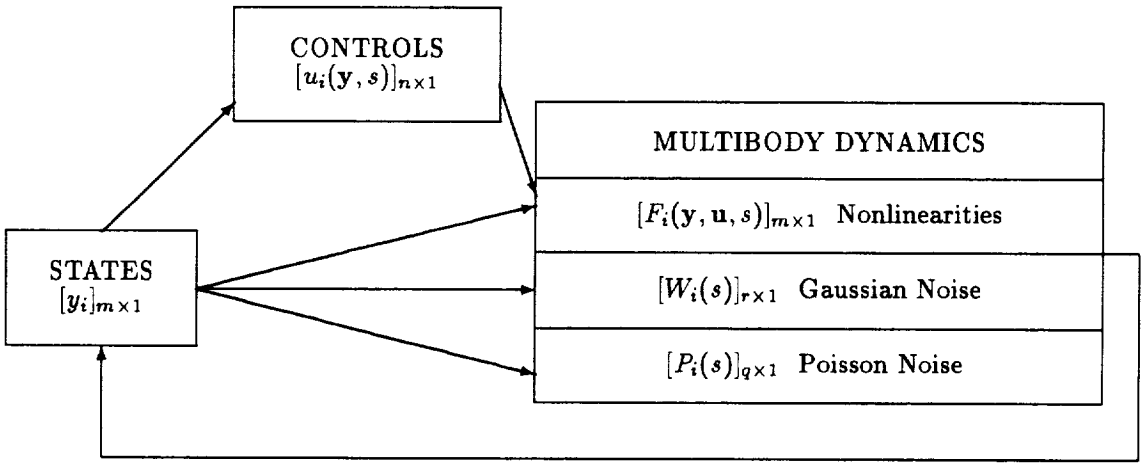

Figure 1: The stochastic multibody system with feedback.