NASA Contractor Report 181812

ICASE REPORT NO. 89-17

ICASE

# PARALLEL SOLUTION OF SPARSE ONE-DIMENSIONAL DYNAMIC PROGRAMMING PROBLEMS

David M. Nicol

(NASA-CR-181812) PARALLEL SOLUTION OF SPARSE ONE-DIMENSIONAL DYNAMIC PROGRAMMING PROBLEMS Final Report (Institute for Computer Applications in Science and Engineering) 23 p N89-21542

CSCL 09B G3/61 Unclas 0200232

Contract No. NAS1-18605
March 1989

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

# Parallel Solution of Sparse One-Dimensional Dynamic Programming Problems

*David M. Nicol*[*]
*Department of Computer Science*
*The College of William and Mary*
*Williamsburg, VA 23185*

**Abstract**

Parallel computation offers the potential for quickly solving large computational problems. However, it is often a non-trivial task to effectively use parallel computers. Solution methods must sometimes be reformulated to exploit parallelism; the reformulations are often more complex than their slower serial counterparts. We illustrate these points by studying the parallelization of *sparse* one-dimensional dynamic programming problems, those which do not obviously admit substantial parallelization. We propose a new method for parallelizing such problems, develop analytic models which help us to identify problems which parallelize well, and compare the performance of our algorithm with existing algorithms on a multiprocessor.

[*]Research was supported under the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

The availability of commercial parallel computers offers the potential for quickly solving large computational problems. However, to exploit parallelism it may be necessary to reformulate the solution algorithm; the construction of effective parallel algorithms is still an art in its infancy. Parallel solutions may be more complex than their serial counterparts, and may rely on insights not generally called for in serial algorithms.

Consider the following general one-dimensional dynamic programming problem: find the solutions to the functional equation

$$\begin{aligned} V(1) &= 0 \\ V(j) &= \min_{i \in N(j)} \{C(i,j) + V(i)\} \qquad \text{for } j = 2, \ldots, n, \end{aligned} \tag{1}$$

where for each $j$, $N(j)$ is a non-empty subset of $\{1, 2, \ldots, j-1\}$. It can be useful to view this problem as that of finding the length of the shortest path from node 1 to every other node in an acyclic graph, where $i \in N(j)$ implies the existence of a directed edge from $i$ to $j$, $i < j$, weighted by $C(i,j)$. This formulation is extremely general, since the nodes of any directed acyclic graph may be so ordered. If instead the nodes are numbered so that the edges go from higher to lower nodes, this formulation arises in classic problems such as optimal production scheduling (with or without backlogging) when holding costs are concave[3].

We are interested in solving this problem on a medium-scale multiprocessor, where the size of the graph is significantly larger than the number of processors. An obvious parallel solution method, the *vertical method*, computes the $V(j)$ values serially, but employs $P$ multiple processors to compute the edge sums specified on the right-hand-side of (1). The set $N(j)$ is partitioned among the processors, who each find the index optimizing $C(i,j) + V(i)$ among their assigned indices. Finally, the processors cooperatively compute $V(j)$ in $O(\log P)$ steps. This scheme can effectively exploit parallelism, provided there is a large amount of necessary parallelizable computation. This condition is violated if the average cardinality of the $N(j)$ sets is small, or equivalently, the average indegree of a node is small. In this situation, relatively little parallel work is performed between each synchronization, and its attendant overhead. The vertical approach exploits *intra*-node parallelism; *sparse* graphs with relatively few edges tend to have little intra-node parallelism. We propose and analyze a method which exploits *inter*-node parallelism, where a multiprocessor works concurrently on the solution of multiple $V(j)$ values. We call this type of approach *horizontal*.

This work is an outgrowth of earlier work[7] where we considered the parallel solution of equations

$$\begin{aligned} V(1) &= 1 \\ V(j) &= C(j) + \min_{i(j) \le i < j} \{V(i)\} \qquad \text{for } j = 2, \ldots, n, \end{aligned}$$

where $i(j)$ is nondecreasing in $j$. This type of equation arises in the context of mapping computations onto parallel computers[6]. Even though the indegree of a node may be large, the vertical approach is inefficient relative to the optimal serial solution. An efficient serial solution maintains a priority heap of previously computed $V(j)$ values, allowing it to find the min term for a given $j$ in $O(\log n)$ time. By contrast, the vertical method requires $O(n/P + \log P)$ time to find the same quantity. We proposed a horizontal method which computes $P$ min terms *optimistically* each step, meaning that if $V(i)$ and $V(j)$ $(i < j)$ are computed concurrently in a step, then $V(j)$ assumes that its min term is not defined by $V(i)$ (the value of which is not yet known). A simple check determines whether the optimism is warranted—if $V(j)$'s min term *is* defined by $V(i)$, then the function values for that step are recomputed serially. The probability of serialization is $O(1/P)$ if $j - i(j)$ is $O(P^2)$ and the $C(j)$ values are independent and identically distributed random variables. Our algorithm is based upon this approach, and we will derive similar results for it.

Bertsekas and Tsitsiklis (1989) explain other parallel dynamic programming algorithms[2]. These algorithms are iterative, requiring multiple passes over the node set. Our algorithms solve the problem in a single pass, but use more global synchronizations. We present empirical results showing our methods to be superior on the problems we study.

This study relies on random directed graphs in order to characterize general properties of problems, and algorithm behavior. These graphs are described by two parameters, a (constant) expected indegree, and the size of an interval from which a node's incoming edges may be taken. All of our experiments and analyses assume this model.

The paper is organized as follows. §2 discusses our methods for generating random graphs. §3 describes the block window algorithm, while §4 derives a performance model, and identifies classes of graphs which support the algorithm's approach. §5 describes our computational experience on a sixteen processor shared memory architecture. We find that our algorithm outperforms both the vertical and iterative methods, achieving good speedups for a variety of problem types. §6 presents our conclusions.

## 1 Random Graph Generation

Before discussing the algorithm, we first describe the methods used to generate random graphs. Our experiments and analysis concern graphs whose edges and weights are created randomly. We view a graph as a linear sequence of nodes $1, \ldots, n$, where directed edges feed forward from lower to higher numbered nodes. We study random graphs that are connected, where an edge $(i, j)$ is constrained by $j - i \leq w$ for some $w$, and where, whenever $i, j > w$, nodes $i$ and $j$ have the same expected number of incoming edges. To construct such graphs we employ a node interval length $w$ and an indegree parameter $D \leq w$, both user defined. For each node $j$ we first create an *initial edge* from some node $i$ in the interval $[\max\{1, j - w\}, j - 1]$, chosen uniformly at random. This ensures that every node is reachable from node 1. We then compute an edge probability

$$p_j = \min\left\{\frac{(D-1)}{\min\{w, j-1\} - 1}, 1\right\}.$$

$p_j$ takes value 1 when $j \leq D$. $p_j$ is constant for $j > w$, we denote that constant

$$p_e = (D-1)/(w-1).$$

For each potential edge from nodes in $j$'s interval (other than the initial edge) we perform a Bernoulli trial, accepting the edge with probability $p_j$. The expected indegree of $j$ is $D$ when $j \geq D$, and is $j - 1$ otherwise. Random edge weights are constructed using a variety of methods described in §4.

## 2 Block Window Algorithm

The block window algorithm partitions the graph into blocks $P$ nodes wide. The algorithm slides a window from block to block, from left to right. When the window is positioned over a block, the $V$ values for nodes within the block are computed in parallel. When all the block's $V$ values have been correctly computed the window is repositioned over the next block. Figure 1 depicts a partitioned graph, and the assignment of processors within a block. We label the nodes within the window by $w_1$ through $w_P$. Processor $j$ is responsible for computing $V(w_j)$.

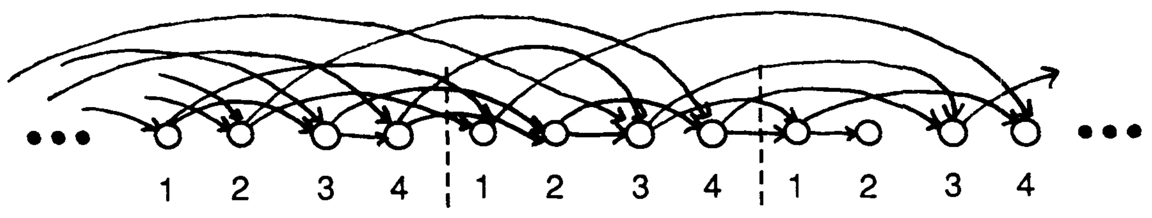

Figure 1: Partitioning a graph into blocks

Consider the computation associated with a node $w_j$ in the window: compute

$$V(w_j) = \min_{(i,w_j) \in N(w_j)} \{C(i,w_j) + V(i)\}.$$

For each $w_j$ we take $N(w_j) = N_I(w_j) \cup N_O(w_j)$, the union of edges rooted inside and outside the window, respectively. Block processing has two phases. In a first phase, each processor $j$ computes

$$u_j = \min_{(i,w_j) \in N_O(w_j)} \{C(i,w_j) + V(i)\},$$

in parallel with the others, assigns $V(w_j) = u_j$, and then synchronizes globally. With this assignment processor $j$ is optimistically assuming that $V(w_j)$ is not defined by a $V$ value within the current block. If all processors are correct in this assumption, then the $V$ values computed for this block are correct. The second phase determines whether any such $V$ value is incorrect. For every edge $(w_i, w_j) \in N_I(w_j)$ processor $j$ compares $C(w_i, w_j) + u_i$ with $u_j$. If any such sum is strictly less than $u_j$ we do not have $u_j = V(w_j)$. Following this test, the processors again synchronize globally, each one passing a flag indicating whether $u_j \neq V(w_j)$. Let $j(\min)$ be the smallest index of a processor indicating this inequality; define $j(\min) = n + 1$ in the complete absence of the inequality. If $j(\min) \leq P$, then values $V(w_{j(\min)}), \ldots V(w_P)$ are recomputed serially. The algorithm's correctness rests on the following proposition.

**Proposition 1** *Let $k \in [1, P]$. If*

$$u_j \leq \min_{(w_i, w_j) \in N_I(w_j)} \{C(w_i, w_j) + u_i\},$$

*for all $j = 1, \ldots, k-1$, then $V(w_j) = u_j$ for $j = 1, \ldots, k$.*

**Proof:** $N_I(w_1)$ is empty, so that $V(w_1) = u_1$. Suppose then that for some $j < k$, $V(w_i) = u_i$ for $i = 1, \ldots, j-1$. Now

$$\begin{aligned} u_j &= \min_{(i, w_j) \in N_O(w_j)} \{C(i, w_j) + V(i)\} \\ &= \min \left\{ \min_{(i, w_j) \in N_O(w_j)} \{C(i, w_j) + V(i)\}, \min_{(w_i, w_j) \in N_I(w_j)} \{C(w_i, w_j) + u_i\} \right\} \\ &= \min \left\{ \min_{(i, w_j) \in N_O(w_j)} \{C(i, w_j) + V(i)\}, \min_{(w_i, w_j) \in N_I(w_j)} \{C(w_i, w_j) + V(i)\} \right\} \\ &= \min_{(i, w_j) \in N(w_j)} \{C(i, w_j) + V(i)\} \\ &= V(j+1). \end{aligned}$$

The conclusion follows by repeated application of this argument to $j = 2, \ldots, k$.

□

The algorithm is described in pseudo-code in figure 2. The parallel synchronization routine CheckSerial identifies $j(\text{min})$, and computes $V(w_{j(\text{min})}), \ldots, V(w_n)$ serially.

It is worthwhile pointing out that the the block window method in no way requires the problem of interest to be a minimization problem. We could equally well use it to solve equations having the general form of (1), but with a max operator. This observation increases the applicability of the algorithm to classic problems such as the one-resource integer knapsack problem[3].

The section to follow constructs a performance model for this algorithm, and identifies a class of graphs on which the second phase serializes with low probability.

## 3 Analysis

To achieve good performance the parallel algorithm must balance the workload well, avoid undue overhead, and avoid serialization in the second phase. In this section we construct a performance

```
BlockWindow(pid)
/*———— pid is processor id, from 1 to $P$ ————*/
    int pid;
{ int $u_j$, $w_0, w_j$, correct;
/*———— $w_0$ is node "before" window ————*/
    $w_0 = 0$;
    while ($w_0 < n$)
        /*———— compute node index ————*/
    {   $w_j = w_0 + $ pid;
        /*———— first phase ———*/
        $u_j$ = minimum value of $C(i, w_j) + V(i)$ over all $i \in N_O(w_j)$;
        /*———— store optimistic solution, synchronize—*/
        $V(w_j) = u_j$;
        /*———— synchronize globally ————*/
        Barrier(pid);
        /*———— begin second phase —*/
        If ($(C(i, w_j) + V(i) < u_j$ for any $i \in N_I(w_j)$ ) then correct = 0;
        else correct = 1;
        /*———— synchronize and correct values —*/
Synch2: CheckSerial(pid,$w_j$,correct);
        w0 = w0 + $P$;
    }
}
```

Figure 2: Pseudo-code for block window algorithm

model for the algorithm, and show that if serialization is infrequent or if the indegree is moderately large we should achieve good speedups. We then show that if edge weights are independent and identically distributed (*i.i.d.*) and $w$ is $O(P^3)$, then the probability of serialization is only $O(1/P)$.

## 3.1 Performance Model

The time required to process a block is determined by the processor with the heaviest workload. Our first task is to estimate the expected load on the most heavily loaded processor in the first phase, and then in the second phase. We can compare the sum of those loads to "perfectly balanced" loads to determine the extent of load imbalance. For simplicity, our analysis ignores the startup anomalies, and only considers blocks sufficiently far into the graph, where the average indegree of each node is $D$. Let $O_j$ be the cardinality of $N_O(w_j)$. Due to the graph construction method the random variables $O_1, \ldots, O_P$ are independent. However, they are not identically distributed; consider the construction of $N(w_j)$. The initial edge is in $N_I(w_j)$ with probability $(j-1)/w$. In this case, each of the possible $w-j+1$ edges for $N_O(w_j)$ has probability $p_e$ of being chosen, and $O_j$ has a binomial distribution, $B(w-j+1, p_e)$. If the initial edge is in $N_O(w_j)$ then we know that $O_j$ has the distribution of one plus a $B(w-j, p_e)$ random variable. Consequently, $O_j$ has the mixed distribution

$$(2) \qquad \text{Distribution of } O_j \equiv \left(\tfrac{j-1}{w}\right) B(w-j+1, p_e) + \left(\tfrac{w-j+1}{w}\right)(1 + B(w-j, p_e)).$$

Now let $I_j$ be the cardinality of $N_I(w_j)$. The distribution of $I_j$ is also found by conditioning on where the initial edge lies, and so is found to be the mixture

$$(3) \qquad \text{Distribution of } I_j \equiv \left(\tfrac{w-j+1}{w}\right) B(j-1, p_e) + \left(\tfrac{j-1}{w}\right)(1 + B(j-2, p_e)).$$

It is straightforward to estimate the expected values of $X_O = \max_j\{O_j\}$, and $X_I = \max_j\{I_j\}$. For example, let

$$\begin{aligned} F_j(s) &= Pr\{O_j \le s\} \\ &= \left(\frac{j-1}{w}\right) Pr\{B(w-j+1, p_e) \le s\} + \left(\frac{w-j+1}{w}\right) Pr\{B(w-j, p_e) \le s-1\}. \end{aligned}$$

Now, $X_O \le s$ if and only if $O_j \le s$ for all $j$. Consider the identity: $E[Y] = \sum_{s=0}^{\infty} Pr\{Y > s\}$ for any discrete nonnegative random variable $Y$. Using it we find that

$$E[X_O] = \sum_{s=0}^{\infty} Pr\{X_O > s\} = \sum_{s=0}^{\infty} \left(1 - \prod_{j=1}^{P} F_j(s)\right).$$

An identical approach allows us to compute $E[X_I]$. For computational reasons we will approximate the binomial distributions involved with Poisson distributions (this is a reasonable approximation[8]). A $B(x, r)$ random variable is approximately equal to the number of Poisson arrivals in time $[0, x]$, where $r$ is the arrival rate. We can view $E[X_0]$ as the expected time to complete the first phase, measured in units where the time required to process one edge takes one time unit. $E[X_I]$ is an estimate of the time needed to complete the second phase, if no serialization occurs. $E[X_I]$ is not exact because the knowledge that no serialization occurs alters these distributions. If serialization does occur, we estimate the number of edges processed serially by $S = E[\sum_{j=1}^{P} I_j]$. Again, this is not exact because knowledge that serialization occurs alters the distributions. It is also not exact because our implementation does not recompute all inter-block edges once the need for serialization is detected. With careful programming we can reduce the number of edge sums computed serially by ignoring those already known to be larger than their corresponding $u_j$ values. If $f$ is the fraction of blocks requiring serialization, we estimate the time required to complete the second phase (ignoring the synchronization cost) by

$$E[T_I(f)] = (1-f)E[X_I] + fS.$$

The sum $E[X_O] + E[T_I(f)]$ is a measure of the time required to process a block (again, ignoring synchronization delays).

The expected value of $N(j)$ is just $D$; the expected number of edges processed in a block is consequently $PD$. If we could assign the load to processors perfectly, every processor would process, on average, $D$ edges. A natural measure of load imbalance given the probability of serialization is $b(f)$, the *balance efficiency*:

$$b(f) = \frac{D}{E[X_O] + E[T_I(f)]}.$$

Figure 3 plots the maximal and minimal balance efficiencies (take $f = 0$ and $f = 1$) as a function of $w$, for fixed $D = 4$, $D = 16$, and $D = 256$, and sixteen processors. We see that good efficiencies can be achieved even when $f = 0$, especially when $D$ or $w$ are large. In these cases the number of edges from outside the block is so large relative to those in the block that serialization scarcely affects performance. Poor load balance occurs if $f$ is high and both $w$ and $D$ are small. §3.2 will show the dependence of $f$ on the behavior of the edge weights.

Load imbalance is only once source of performance degradation. Parallel algorithms frequently have a higher overhead per unit of computation than do their serial counterparts. We suppose that a serial solution requires $\alpha$ units of time per edge processing, while the parallel solution requires $\beta \geq \alpha$ units of time per edge. We call $\alpha/\beta$ the *overhead efficiency*. Both $\alpha$ and $\beta$ are problem sensitive; for example, they depend on the amortization of loop bounds checking over the number of loop iterations. The overhead efficiency can be estimated by dividing the execution time of the parallel algorithm running on one processor by the execution time of the specifically serial version. For the problems we studied the overhead efficiencies range from 0.5 to 0.83.

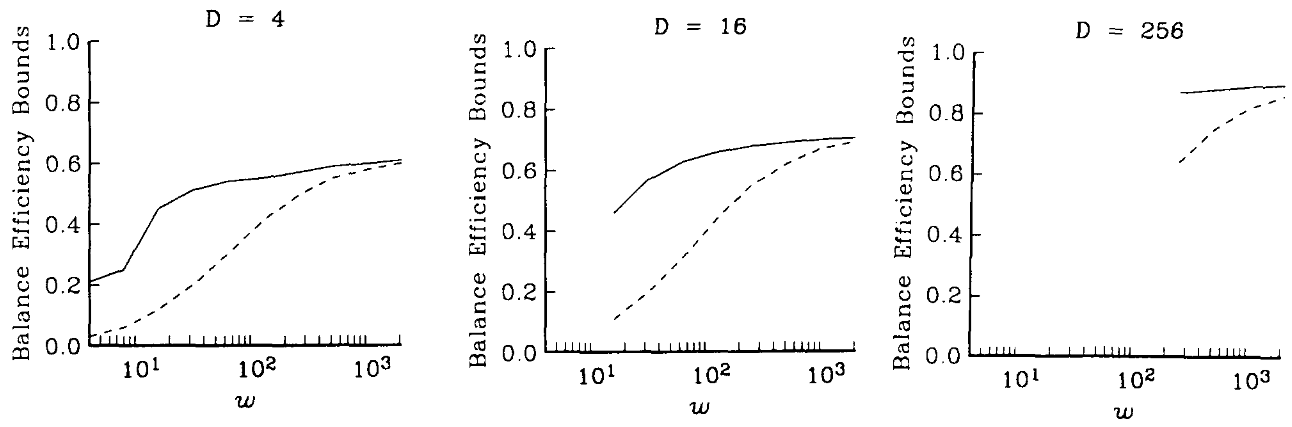

Figure 3: Bounds on sixteen processor balance efficiencies, as function of $w$, for $D = 4, D = 16$ and $D = 256$

There is also a delay cost associated with executing the global synchronization; let $G(P)$ be the time required for $P$ processors to perform the global synchronization between windows. Our algorithm uses two synchronizations per block. The overall efficiency of the parallel solution is the speedup divided by the number of processors used, and so is given by

$$\begin{aligned} \text{Efficiency} &= \frac{\text{Optimized Serial Time}}{\text{Processors} \times \text{Parallel Time}} = \frac{\alpha P D}{P\,(\beta(E[X_O] + E[T_I(f)]) + 2G(P))} \\ (4) \qquad &= \frac{(\alpha/\beta)b(f)}{1 + 2G(P)/(\beta(E[X_O] + E[T_I(f)]))}. \end{aligned}$$

It is important to note that true speedup measurements use an optimal, specifically serial program.

It is well-known that a barrier synchronization can be performed in $O(\log P)$ time. However, on the architecture we used the fastest barrier algorithm has a *linear* cost[1]. We consequently use the model $G(P) = \gamma P$, for some $\gamma$. Measured in units where $\alpha = 1$, we have found $\gamma \approx 0.5$. Our test suite of problems consists of four replicates each of problems where $D = 8, 16, 32, 64, 96, 128$, and $w = 64$ and $w = 256$. On this suite the average magnitude of the relative error between measured speedup for up to eight processors, and that predicted by (4) is approximately five percent. These figures used measured frequencies of serialization. The performance model sometimes overestimates speedup for sixteen processors by as much as 33%, although the absolute error never exceeded 1.2.

This error seems to be due primarily to an under-estimation of $G(16)$; although it is also possible that $\beta$ increases in $P$, a phenomenon unaccounted for by the model.

Computer manufacturers are now planning architectures with fast hardware support for global synchronization. It is reasonable to ask how good the performance would be if synchronization costs were negligible. Equation (4) helps answer this question. If $G(P)$ vanishes the overall efficiency is just the product of the load and overhead efficiencies. We have already seen that load efficiencies are often good, at least under the assumed analytic model. The overhead efficiencies are between 50% and 83%. The product of load and overhead efficiences (assuming sixteen processors) can exceed 50%, yielding a speedup of at least 8. Of course, it can be much lower. §4 will show that speedups depend primarily on $D$; with $D = 128$ and $w = 256$ we do achieve 50% efficiencies on sixteen processors.

## 3.2 Probability of Serialization

We now turn to a more qualitative analysis of the algorithm, where we focus on the behavior of $V$ as a random function. To highlight the difference between the length of shortest path to $j$ on a *given* problem, and the random length of the shortest path to $j$ under our model, we will use $\mathcal{V}(j)$ to denote the random variable. Likewise, we will use $\mathcal{C}(i,j)$ to describe the random weight on extant edge $(i,j)$.

We first show that if the edge weight random variables increase stochastically in their length, then $\mathcal{V}(j)$ increases stochastically in $j$. We then show that if the edge weights are *i.i.d* or if the edge weights are identically distributed and all edge weights into a node are identical, and if $w$ is $O(P^3)$, then the probability of serialization is $O(1/P)$. Surprisingly, the data in figure 3 show that this is a secondary concern, at least when $D$ is not small. Nevertheless, it is an important question when $D$ is small and the time spent in the second phase contributes significantly to the overall execution time.

Our understanding of $V(i)$ as the length of the optimal path from node 1 to $i$ suggests that $\mathcal{V}(i)$ should tend to increase in $i$, at least if edge weights tend to increase in their length. Our first result affirms this intuition by using order relations between random variables. Principally, random variable $X$ is said to be stochastically larger than random variable $Y$ if for all $a$, $Pr\{X > a\} \geq Pr\{Y > a\}$. This is denoted $X \geq_{st} Y$, or $Y \leq_{st} X$. An equivalent definition says that $Y \leq_{st} X$ if there exists a random variable $Z$ distributed identically as $Y$, with the property that $Z \leq X$. A common way to show that $Y \leq_{st} X$ is to construct a variable $Z$ "based on" $X$, show that $Y$ and $Z$ are identically distributed, and show that $Z \leq X$. Called *coupling*, this will be our approach to showing that when $\mathcal{V}(i-1) \leq_{st} \mathcal{V}(i)$ when $i > w$ and edge weights increase stochastically in length.

**Proposition 2** *Suppose that $\mathcal{C}(i,j-1) \leq_{st} \mathcal{C}(i,j)$ whenever $i < j-1$, and that for all $j$, $\mathcal{C}(i,j)$ is independent of $\mathcal{V}(i)$. Then $\mathcal{V}(i-1) \leq_{st} \mathcal{V}(i)$ for $i = w+1, \ldots, n$.*

**Proof:** We use a coupling argument as described above. Define $N(j-1)$, given $N(j)$, as follows. Whenever $(i,j) \in N(j)$ and $i \neq j-1$, then place $(i,j-1) \in N(j-1)$. If $(j-1,j) \in N(j)$, then define $(j-1-w, j-1) \in N(j-1)$. Because the edge weights increase stochastically in length, whenever $i < j-1$ and $(i,j) \in N(j)$ has a weight $c$, we may chose a weight $\hat{c} \leq c$ for $(i,j)$'s corresponding edge in $N(j-1)$. If $(j-1-w, j-1) \in N(j-1)$ we may weight it with an arbitrary sample from $\mathcal{C}(j-1-w, j-1)$'s distribution. Let $Z$ be the length of the shortest path from node 1 to node $j-1$. $Z$ has $\mathcal{V}(j-1)$'s distribution, because the choice of arcs into $j-1$ is driven by the appropriate distributions, and the choice of edge weights are also from the appropriate distributions. We must now show that $Z \leq \mathcal{V}(j)$.

Let $H$ be the sum of all edge weights in the graph. For every $i$ and $j$ with $i < j$ define

$$B_i^j = \begin{cases} \mathcal{C}(i,j) + \mathcal{V}(i) & \text{if } (i,j) \in N(j) \\ H & \text{otherwise} \end{cases}.$$

By construction $B_i^j \geq B_i^{j-1}$ for all $i \in [j-w, j-2]$. Now define

$$M = \min_{j-w \leq i \leq j-2} \{B_i^j\}, \quad \text{and} \quad \hat{M} = \min_{j-w \leq i \leq j-2} \{B_i^{j-1}\}$$

It follows that $M \geq \hat{M}$. Now

$$\begin{aligned} \mathcal{V}(j) = \min\{M, B_{j-1}^j\} &\geq \min\{\hat{M}, Z\} \\ &= \min\left\{\hat{M}, \min\{B_{j-1-w}^{j-1}, \hat{M}\}\right\} \\ &= \min\{B_{j-1-w}^{j-1}, \hat{M}\} \\ &= Z. \end{aligned}$$

Consequently, $\mathcal{V}(j-1) \leq_{st} \mathcal{V}(j)$.

□

The proposition above is also true in the case of a maximization problem; the proof above works by replacing all min's with max's.

Two special cases are of particular interest. One occurs when the edge weights are independent and identically distributed, a degenerate case of $\mathcal{C}(i,j-1) \leq_{st} \mathcal{C}(i,j)$. A second case occurs when edges are identically distributed, and all weights on edges entering a node are identical. Both cases are consistent with the hypothesis of proposition 1, and lead to the situation where we can stochastically order the edges entering a node. That is, if $N(j)$ is ordered increasing as $N(j) = \{i_1, \ldots, i_k\}$, then for $h = 2, \ldots, k$, $\mathcal{C}(i_{h-1}, j) + \mathcal{V}(i_{h-1}) \leq_{st} \mathcal{C}(i_h, j) + \mathcal{V}(i_h)$. To see this, observe that $\mathcal{V}(i_{h-1}) \leq_{st} \mathcal{V}(i_h)$ implies the existence of a random variable $Z$ having the distribution of $\mathcal{V}(i_{h-1})$, and $Z \leq \mathcal{V}(i_h)$. Under the identified conditions we can choose a weight $\hat{c}$ having the distribution of $\mathcal{C}(i_{h-1}, j)$, where $\hat{c} \leq \mathcal{C}(i_h, j)$. Then $\hat{c} + Z$ has the distribution of $\mathcal{C}(i_{h-1}, j) + \mathcal{V}(i_{h-1})$ and $\hat{c} + Z \leq \mathcal{C}(i_h, j) + \mathcal{V}(i_h)$, establishing proposition 3.

**Proposition 3** *Suppose that all edge weights are i.i.d., or that for each fixed $j$, $\mathcal{C}(i,j)$ is independent of $\mathcal{V}(i)$ and $\mathcal{C}(i,j) = \mathcal{C}(k,j)$ for all $i, k \in N(j)$. If $N(j)$ is ordered increasing as $N(j) = \{i_1, \ldots, i_k\}$, then for $h = 2, \ldots, k$, we have*

$$\mathcal{C}(i_{h-1}, j) + \mathcal{V}(i_{h-1}) \leq_{st} \mathcal{C}(i_h, j) + \mathcal{V}(i_h).$$

□

Let

$$q_h = Pr\left\{ \min_{i \in N(j)} \{\mathcal{C}(i,j) + \mathcal{V}(i)\} = \mathcal{C}(i_h, j) + \mathcal{V}(i_h) \right\}.$$

In the case of a tie between two or more edges, we define the edge from the least indexed nodes as the minimum. If the conditions of proposition 3 are satisfied, an immediate consequence of its conclusion is that $q_1 \geq q_2 \geq \ldots \geq q_k$. Now for any $j$ such that $1 \leq j \leq k$,

$$\frac{1}{j} \sum_{h=1}^{j} q_h \geq \frac{1}{k} \sum_{h=1}^{k} q_h.$$

Since $\sum_{h=1}^{k} q_h = 1$, we have $\sum_{h=1}^{j} q_h \geq j/k$. It follows that if $e_j$ is the edge defining $\mathcal{V}(w_j)$, then

$$\begin{aligned} Pr\{e_j \in N_O(w_j)\} &\geq E\left[\frac{O_j}{O_j + I_j}\right] \\ &\geq \frac{E[O_j]}{E[O_j + I_j]} = \frac{E[O_j]}{D}. \end{aligned}$$

The second step follows from Jensen's inequality[8]. When $w$ is large relative to $P$, then $E[O_j]$ is close to $D$. Such a situation supports our optimistic hope that serialization not be required.

To avoid serialization we must have $e_j \in N_O(w_j)$ for *all* $j$. The events that $e_j \in N_O(w_j)$ and $e_i \in N_O(w_i)$ are not independent. Intuitively though, we expect these events to be positively correlated—one way for $e_i \in N_O(w_i)$ is to have it rooted in a node with an unusually low $\mathcal{V}$ value. If that node has an edge to $w_j$ as well, the probability that $e_i$ is also rooted in that node increases. A formal proof of positive correlation appears to be formidable. Assuming positive correlation, the probability of serializing the second phase is bounded from above by the probability of serializing if the events were independent. This gives

$$Pr\{e_j \in N_O(w_j) \text{ for } j = 1,\ldots,P\} \geq \prod_{j=1}^{P} \frac{E[O_j]}{D}. \tag{5}$$

We can compute $E[O_j]$ from (2). By subtracting the product from 1 we obtain an upper bound on the probability of serialization. Figure 4 plots this bound for $D = 4$ when $P = 4, 8$ and 16, as a function of $w$ (the same problem set represented in figure (3). Plots for $D = 16$ and $D = 256$ are indistinguishable from that of $D = 4$. We see that low serialization probabilities are achieved when $w$ is large compared with $P$. In fact, we next demonstrate that the probability of serialization is $O(1/P)$ when $w$ is $O(P^3)$. This is significant, because then the *expected* complexity of the second phase is the same as the complexity if no serialization occurs.

**Proposition 4** *Let the conditions of proposition 3 be met, and suppose that $w = O(P^3)$. Then the probability of serialization is $O(1/P)$.*

**Proof:** Beginning with (5) we have

$$\begin{aligned} Pr\{e_j \in N_O(w_j) \text{ for } j = 1,\ldots,P\} &\geq \prod_{j=1}^{P} \frac{E[O_j]}{D} \\ &= \prod_{j=1}^{P} \left(1 - \frac{E[I_j]}{D}\right) \\ &\geq 1 - \frac{1}{D}\sum_{j=1}^{P} E[I_j]. \end{aligned} \tag{6}$$

The last step follows from the inequality[4]: $\prod_{j=1}^{P}(1 - a_j) \geq 1 - \sum_{j=1}^{P} a_j$ whenever $0 \leq a_j < 1$ for

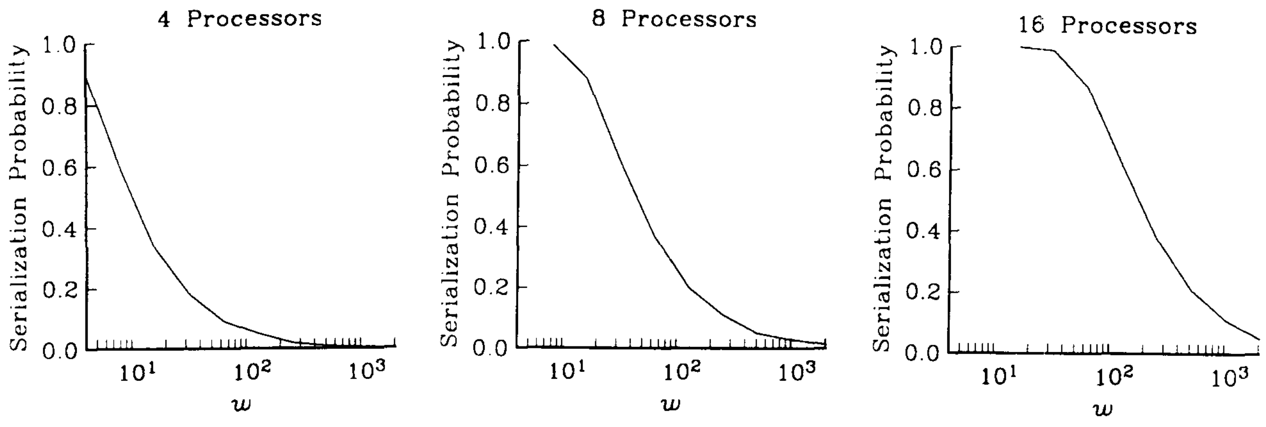

Figure 4: Probability of serialization as a function of $w$, when $D = 4$ and $P = 4, 8, 16$

all $j$. Note that $E[I_j] \leq E[I_P]$ for all $j$; consulting (3) we see that

$$\begin{aligned} E[I_P] &= \left(\frac{P-1}{w}\right)(1 + p_e(P-2)) + \left(\frac{w-P+1}{w}\right) p_e(P-1) \\ &= O(1/P^2) + O(D/P^2). \end{aligned}$$

The last step depends on the fact that $p_e = (D-1)/(w-1)$, and that $w = O(n^3)$. Picking up from (6) we have

$$\begin{aligned} Pr\{e_j \in N_O(w_j) \text{ for } j = 1, \ldots, P\} &\geq 1 - \frac{1}{D}\sum_{j=1}^{P} E[I_j] \\ &\geq 1 - \left(\frac{P}{D}\right) E[I_P] \\ &= 1 - O(1/P) \qquad \text{because } E[I_P] = O(D/P^2). \end{aligned}$$

It follows that the probability of serialization is $O(1/P)$.

□

It is interesting to note that we place no conditions on $D$ in order to achieve a $O(1/P)$ probability of serialization.

We strongly suspect that the probability of serialization is low more generally than just under the conditions of proposition 3. For example, one intuitively suspects that if edge weights are concave in the edge length, then the edge defining a given $V(j)$ value will tend to be long, and hence rooted outside of $V(j)$'s block. An algorithmically uninteresting, but extreme case of this occurs if edge weights are exactly proportional to edge distance. In this case $V(j) = \tau j$ for all $j$, where $\tau$ is the constant of proportionality, and $C(i,j) + V(i) = \tau j$ for all $i \in N(j)$. Assuming that ties are broken by choosing the longest edge, $V(j)$ is always defined by the edge rooted furthest from $j$.

## 4 Computational Experience

We tested the block window algorithm against two competitors, an iterative, asynchronous "contraction" algorithm, and the straightforward vertical method. We programmed all of these methods on the Flex/32[5], a shared memory multiprocessor. We find that under our implementations on this architecture, the block window algorithm substantially outperforms both of the other methods. Furthermore, efficiencies in excess of 50% are achieved when $D$ is sufficiently large relative to the number of processors.

We have already mentioned the vertical method: the edges into each node are partitioned among processors, who then cooperatively compute the minimum edge sum. The pseudo-code in figure 5 describes this algorithm. For each $j$ we denote $N(j) = \{j_1, \dots, j_k\}$. The synchronization routine ComputeMinimum determines the minimum value passed to it, and writes that into $V(j)$.

Load balancing under the vertical method is good when $D$ is large—at every point no processor computes more than one more edge sum than any other. However, the method suffers when $D$ is small, and it will always suffer a synchronization cost at each $V$ point.

An alternate method is based on iterative methods[2]. We annotate each $V$ value with a superscript describing an "iteration number". We initialize by setting $V^0(1) = 0$, and $V^0(i) = H$ for all $i > 1$, where $H$ is the sum of all edge weights, and consequently bounds the true value of $V(i)$ for each $i$. The iterative computation described by

$$V^k(j) = \min_{i \in N(j)} \{C(i,j) + V^{k-1}(i)\} \qquad \text{for } k > 0 \tag{7}$$

will converge to the correct solution. This computation uses two arrays for $V$ values, one for the "old" (iteration $k-1$) values, and one for the new (iteration $k$). Processor $j$ is responsible for computing values $V^k(j + iP)$ for $i = 0, \dots,$ does so in increasing sequence, and synchronizes

```
VerticalMethod(pid)
/*———————— pid is processor id ————————*/
    int pid;
{ int $j$, MinEdgeCost ;

    for $j = 2$ to $n$
    {
        MinEdgeCost = minimum value of $C(j_h, j) + V(j_h)$ over all $j_h \in N(j)$,
                $h = \text{pid} + tP$ for some $t$;
        ComputeMinimum(pid, $j$, MinEdgeCost);
    }
}
```

Figure 5: Pseudo-code for vertical method

globally after such a pass. This algorithm has the disadvantage of needing multiple passes over the $V$ array; it has the advantage of synchronizing only between passes. If $L$ is the minimal number of edges in any shortest path from node 1 to node $P$, then convergence is detected in $L + 1$ iterations. We can improve upon this substantially by allowing asynchrony. We use only one $V$ array, so that when computing $V(j)$, the values $V(i), i \in N(j)$ may actually be "new". For example, if we use only one processor, the true $V$ values are correct after the first pass, and convergence is detected on the second. One processor processing of (7) will require $L + 1$ passes to detect convergence.

The pseudo-code in figure 6 describes an asynchronous iterative algorithm. The global synchronization routine CheckConvergence returns a zero if any processor passed a "Change" value of one to it; it otherwise returns a one.

All of the results we report use $n = 1024$. We did test larger problems, but found that the performance figures were largely unaffected. Four different methods of creating edge weights were used. Method 1 chooses each node weight independently, and uniformly from $[0, n]$. Method 2 makes the weight sensitive to the edge length, adding $j - i$ to a uniform $[0, 2n]$ random variable. Methods 3 and 4 both use uniform $[0, n]$ weights; Method 3 ensures that edge weights rooted in a node increase as a function of their destination. Method 4 forces all weights on a node's incoming

```
IterativeMethod(pid)
/*————————— pid is processor id ——————*/
    int pid;
{ int $j$, MinEdgeCost, Converged, Change;

    Converged = 0;
    /*————————— loop through another pass —————*/
    while( Converged = 0)
    {  /*————————— pid does every $Pth$ point ——*/
        $j$ = pid;
        Change = 0;
        while ($j \leq n$ )
        {   MinEdgeCost = minimum value of $C(i,j) + V(i)$ over all $i \in N(j)$;
            /*————————— look for improvement in solution ————*/
            If (MinEdgeCost < $V(j)$)
                {   $V(j)$ = MinEdgeCost;
                    Change = 1;
                }
            $j = j + P$;
        }
    /*————————— synchronize with others, check convergence ——*/
        Converged = CheckConvergence(pid,Change);
    }
}
```

Figure 6: Pseudo-code for iterative algorithm

edges to be identical. All of these methods satisfy the hypothesis of proposition 2; only Methods 1 and 4 satisfy the hypothesis of proposition 3. We take $w = 64$ and $w = 256$; we also vary the average indegree, taking $D = 8, 16, 32, 48, 64, 96, 128$, subject to $D \leq w$.

Early evaluation of the algorithms revealed that the execution times of both the block window and vertical methods on problems this large are highly insensitive to the particular edges and edge weights, and are insensitive to the different methods we used of generating edge weights. Some law of large numbers seems to keep the measured coefficient of variation (standard deviation divided by mean) well under 0.01 on ten replications. By comparison, execution time variation for the iterative method was large. The asynchrony allows different runs (even on the same problem) to have different convergence rates.

For every set of graph characteristics ($D$,$w$,weight generation method) we generate four independent graphs, and solve the shortest path problem with each algorithm, using $2, 4, 8$, and 16 processors. Each of our performance graphs plots the average efficiency as a function of $D$ and the algorithm used—measurements from different problem types but with common $w$,$D$, and algorithm characteristics are averaged together. Figure 7 shows four plots of measured mean efficiencies, when $w = 256$. Each graph reports measurements from a fixed set of processors. Similar tests were run using $w = 64$, but surprisingly very little difference was observed, at least for the vertical and window block methods. Measurements from $w = 256$ tend to be better, but only marginally so. The iterative method does somewhat better on $w = 256$, because the number of edges in a shortest path tends to be smaller.

Three trends in this data are interesting. First and foremost, performance of the block window method increases as the graph density ($D$) increases, and is often quite respectable. On the given implementations, architecture, and problem set, the other algorithms are clearly inferior. Second, better efficiencies are achieved using a smaller number of processors. For a fixed problem size it is almost always the case that increasing the number of processors decreases the efficiency, because more processors usually imply more overhead. The third trend is that the vertical method's effectiveness decreases as the number of processors increases. This is due to the fact that the cost of synchronization increases as the number of processors increases, as does relative load imbalance.

## 5 Summary

This paper proposes the block window algorithm for solving sparse dynamic programming problems on a parallel computer. Dynamic programming problems are characterized by their use of subproblem solutions to construct "larger" problem solutions. Sparse problems tend to force a completely serial execution if, when constructing a larger problem solution, we always wait for the solutions to all possible subproblems that *might* be needed. The key idea behind our algorithm is to optimistically *assume* in a first phase that the particular subproblem solution that will be needed is one that has already been computed. A second phase checks this assumption's veracity, and corrects any erroneous calculations.

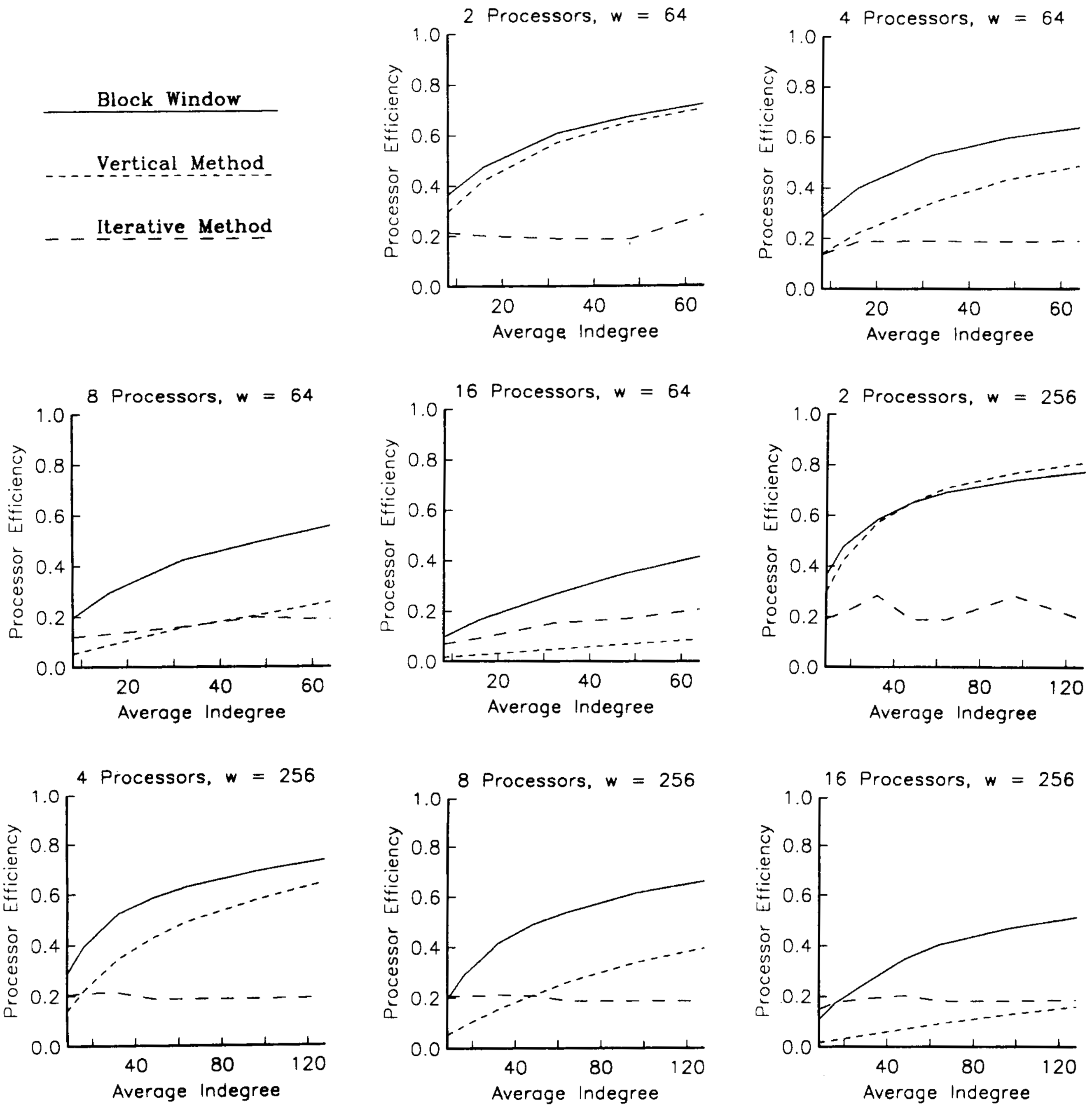

Figure 7: Measured efficiencies for different algorithms

The bulk of this paper analyzes the method's performance quantitatively and qualitatively. The analysis gives us insight into the type of performance we can expect, depending on problem and architectural characteristics. We then compare its performance against two other algorithms on a shared-memory multiprocessor, observe that on the given problem set it performs markedly better than the others, and note that efficiencies in excess of 50% can be achieved using sixteen processors.

## References

[1] N.S. Arenstorf and H.F. Jordan. *Comparing Barrier Algorithms.* Technical Report 87-65, ICASE, September 1987.

[2] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation:Numerical Methods.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[3] E.V. Denardo. *Dynamic Programming: Models and Applications.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[4] D.E. Knuth. *The Art of Computer Programming, vol. 1.* Addison-Wesley, New York, 1968.

[5] N. Matelan. The Flex/32 multicomputer. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 209–213, Computer Society Press, June 1985.

[6] D.M. Nicol. *Parallel Algorithms for Mapping Pipelined and Parallel Computations.* Technical Report 88-2, ICASE, January 1988.

[7] D.M. Nicol. Parallel solution of a dynamic programming equation using optimistic evaluation. In *Publications in Operations Research Series*, Elsevier-North Holland, 1989. To appear.

[8] H.S. Ross. *Stochastic Processes.* Wiley, New York, 1983.

NASA
National Aeronautics and Space Administration

# Report Documentation Page

| | | |
|---|---|---|
| 1. Report No. NASA CR-181812 ICASE Report No. 89-17 | 2. Government Accession No. | 3. Recipient's Catalog No. |
| 4. Title and Subtitle PARALLEL SOLUTION OF SPARSE ONE-DIMENSIONAL DYNAMIC PROGRAMMING PROBLEMS | | 5. Report Date March 1989 |
| | | 6. Performing Organization Code |
| 7. Author(s) David M. Nicol | | 8. Performing Organization Report No. 89-17 |
| | | 10. Work Unit No. 505-90-21-01 |
| 9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225 | | 11. Contract or Grant No. NAS1-18605 |
| 12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225 | | 13. Type of Report and Period Covered Contractor Report |
| | | 14. Sponsoring Agency Code |

15. Supplementary Notes

Langley Technical Monitor: Richard W. Barnwell

ORSA Journal on Computing

Final Report

16. Abstract

Parallel computation offers the potential for quickly solving large computational problems. However, it is often a non-trivial task to effectively use parallel computers. Solution methods must sometimes be reformulated to exploit parallelism; the reformulations are often more complex than their slower serial counterparts. We illustrate these points by studying the parallelization of sparse one-dimensional dynamic programming problems, those which do not obviously admit substantial parallelization. We propose a new method for parallelizing such problems, develop analytic models which help us to identify problems which parallelize well, and compare the performance of our algorithm with existing alogirithms on a multiprocessor.

| | |
|---|---|
| 17. Key Words (Suggested by Author(s)) dynamic programming, shortest path algorithms, parallel algorithms | 18. Distribution Statement 61 - Comp. Prog. & Software 66 - Systems Analysis Unclassified - Unlimited |

| 19. Security Classif. (of this report) | 20. Security Classif. (of this page) | 21. No. of pages | 22. Price |
|---|---|---|---|
| Unclassified | Unclassified | 22 | A03 |

NASA FORM 1626 OCT 86