Performance of a Benchmark Implementation of the Van Slyke and Wets Algorithm for Stochastic Programs on the Alliant FX/8

K. A. Ariyawansa

CRPC-TR91146
May, 1991

Center for Research on Parallel Computa
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Revised July, 1991.

ABSTRACT

Ariyawansa and Hudson recently presented a benchmark parallel implementation of the Van Slyke and Wets algorithm for stochastic linear programs, as well as the results of a carefully designed numerical experiment using the implementation on the Sequent Balance. An important use of this implementation is as a benchmark to assess the performance of certain approximation algorithms for stochastic linear programs. These approximation algorithms are best suited for implementation on parallel vector processors like the Alliant FX/8. Therefore, the performance of the benchmark implementation on the Alliant FX/8 is of interest. In this paper, we present results observed when a portion of Ariyawansa and Hudson's numerical experiment is performed on the Alliant FX/8. These results indicate that the implementation makes satisfactory use of the concurrency capabilities of the Alliant FX/8. The results also indicate that the vectorization capabilities of the Alliant FX/8 are not satisfactorily utilized by the implementation.

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

PERFORMANCE OF A BENCHMARK IMPLEMENTATION OF THE VAN SLYKE AND WETS ALGORITHM FOR STOCHASTIC PROGRAMS ON THE ALLIANT FX/8*†

K. A. Ariyawansa‡

Mathematics and Computer Science Division

Preprint MCS-P242-0591

May 1991

*This research was supported in part by DOE Grant DE-FG-06-87ER25045, NSF Grant DMS-8918785, the NSF Science and Technology Center for Research in Parallel Computation, and the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†A summary of this paper was presented at the Fifth SIAM Conference on Parallel Processing for Scientific Computing held in Houston, Texas, during March 25–27, 1991. An earlier version of this paper appeared as Technical Report TR 91-3, Department Pure and Applied Mathematics, Washington State University, Pullman, WA 99164.

‡Permanent Address: Department of Pure and Applied Mathematics, Washington State University, Pullman, WA 99164

1. Introduction. The two-stage stochastic program with recourse, and with a discretely distributed random variable with a finite number of realizations, is the following:

$$\begin{aligned}
&\text{Find } x^* \in \Re^{n_1} \text{ such that when } x := x^* \\
&z(x) := c^T x + \mathcal{Q}(x) \text{ is minimized, and} \\
&Ax = b,\ x \geq 0, \text{ where} \\
&\mathcal{Q}(x) := E[Q(x,h,T)] = \sum_{k=1}^{K} p^k Q(x,h^k,T^k), \\
&Q(x,h,T) := \inf_{y \in \Re^{n_2}} \{q^T y : My = h - Tx, y \geq 0\}, \\
&A \in \Re^{m_1 \times n_1},\ b \in \Re^{m_1},\ c \in \Re^{n_1},\ q \in \Re^{n_2},\ M \in \Re^{m_2 \times n_2} \text{ are deterministic and given} \\
&\text{and } h \in \Re^{m_2},\ T \in \Re^{m_2 \times n_1} \text{ are random with } (h,T) \text{ having the given probability} \\
&\text{distribution } F := \{((h^k,T^k),p^k), k = 1,2,\ldots,K\}. \qquad (1)
\end{aligned}$$

Problem (1) arises in operations research problem areas including industrial management, scheduling, and transportation; in control theory; and in economics. The monograph [5] for example, contains details of specific applications.

A popular algorithm for (1) is due to Van Slyke and Wets [8]. In [3], a parallel benchmark implementation of the Van Slyke and Wets algorithm for (1) was presented, as well as the results of a carefully designed numerical experiment on the Sequent Balance. An important use of this implementation is as a benchmark to assess the performance of the approximate algorithms for (1) described in [4, 2]. These approximate algorithms are best suited for implementation on parallel vector processors like the Alliant FX/8. Therefore, the performance of the benchmark implementation on the Alliant FX/8 is of interest. In Section 2 of this paper, we present a slightly modified version of the implementation given in [3] that is better suited for the Alliant FX/8. We have performed a portion of the numerical experiment described in [3] using this implementation on the Alliant FX/8. We indicate our results in Section 3.

2. A Parallel Benchmark Implementation of the Van Slyke and Wets Algorithm on the Alliant FX/8. As in [3], throughout the paper we shall make the following assumptions regarding (1).

(A1) The set $\{x : Ax = b, x \geq 0\}$ is nonempty and bounded.

(A2) The set $\{w : My = w, y \geq 0\} = \Re^{m_2}$.

(A3) The set $\{v : M^T v \leq q\}$ is nonempty.

It can be verified that when (A1), (A2), and (A3) are satisfied, (1) has a finite minimum. Therefore, issues of unboundedness and infeasibility of (1) do not arise.

We now present the pseudo-code on which our Fortran implementation on the Alliant FX/8 is based.

Algorithm 1:

input:

m_1; n_1; m_2; n_2; K; A; c; b; M; q; $F := \{((h^k, T^k), p^k), k = 1, 2, \ldots, K\}$;
nprocs; *maxcut*; *tol*.
/* we assume that $K >$ *nprocs* */

output:

x; z.

begin
/* begin initializations */
$t := 0$;
for $l := 1$ **to** *maxcut* **do**
$E^l := 0$; $e^l := 0$;
end do;
{call DPLO to solve the lp
find $x \geq 0, \theta \in \Re$ such that
$c^T x + \theta$ is minimized, and
$Ax = b$,
$(E^l)^T x + \theta \geq e^l$, $l = 1, 2, \ldots,$ *maxcut*
from scratch};
call *optimality_cut* (m_2, n_2, K, M, q, F, x, *nprocs*, E, e, $\mathcal{Q}$);
$t := t + 1$;
$E^t := E$; $e^t := e$;
call *lowerbound* (m_1, n_1, t, A, b, $(E^l, e^l, l = 1, 2, \ldots, t)$, *maxcut*, x, θ, z);
call *optimality_cut* (m_2, n_2, K, M, q, F, x, *nprocs*, E, e, $\mathcal{Q}$);
/* end initializations */
while $|\mathcal{Q} - \theta| / \max\{1, |\mathcal{Q}|\} <$ *tol* **do**
$t := t + 1$;
if $t >$ *maxcut* **then**
{report that more cuts than *maxcut* need to be added and stop};
else
$E^t := E$; $e^t := e$;
call *lowerbound* (m_1, n_1, t, A, b, $(E^l, e^l, l = 1, 2, \ldots, t)$, *maxcut*, x, θ, z);
call *optimality_cut* (m_2, n_2, K, M, q, F, x, *nprocs*, E, e, $\mathcal{Q}$);
end if;
end while;
end.

optimality_cut:

input:

m_2; n_2; K; M; q; F; x; $nprocs$.

output:

E; e; Q.

begin

$Etemp := 0$; $etemp := 0$; $E := 0$; $e := 0$;

/* dimensions of $Etemp$ and $etemp$ are $n_1 \times nprocs$ and $nprocs$ respectively */

$bnchsz := \lfloor K/nprocs \rfloor$;

for $bunch := 1$ **to** $nprocs$ **in parallel do**

$firstk := (bunch - 1) * bnchsz + 1$;

if $bunch \neq nprocs$ **then**

$lastk := bunch * bnchsz$

else

$lastk := K$

end if;

call $dobnch(firstk, lastk, m_2, n_2, M, q, F, x, Etemp(1 : n_1, bunch), etemp(bunch))$;

end do;

for $bunch := 1$ **to** $nprocs$ **do**

$E := E + Etemp(1 : n_1, bunch)$;

$e := e + etemp(bunch)$;

end do;

$Q := e - E^T x$;

return;

end.

dobnch:

input:

$firstk$; $lastk$; m_2; n_2; M; q; F; x.

output:

E; e.

begin

for $k := firstk$ **to** $lastk$ **do**

$w := h^k - T^k x$;
if $k = firstk$ then
{call DPLO to solve the lp
find $y \geq 0$ such that
$q^T y$ is minimized, and
$My = w$
from scratch to obtain dual maximizer v};
else
{call DPLO with restart procedure and with new options to prevent
disk reads, and to prevent forming LU factorization of initial basis
to solve the lp
find $y \geq 0$ such that
$q^T y$ is minimized, and
$My = w$
to obtain dual maximizer v};
end if;
$E := E + p^k (T^k)^T v$;
$e := e + p^k (h^k)^T v$;
end do;
return;
end.

lowerbound:

input:

m_1; n_1; t; A; b; E^l, e^l, $l = 1, 2, \ldots, t$; $maxcut$.

output:

x; θ; z.
begin
{call DPLO with restart procedure, and with new option to prevent
disk reads, and with a specification that $(m_1 + t)$-th row of constraint
matrix changes to $[(E^t)^T, 1]$, to solve the lp
find $x \geq 0$, $\theta \in \Re$ such that
$c^T x + \theta$ is minimized, and
$Ax = b$,
$(E^l)^T x + \theta \geq e^l$, $l = 1, 2 \ldots, maxcut$};
$z := c^T x + \theta$;
return;
end.

In Algorithm 1, *nprocs* is the number of processors. The input parameter *maxcut* essentially places an upper bound on the number of calls to routine *optimality_cut*, while the input parameter *tol* is a tolerance used in the stopping criterion. Justification for using these parameters is given in [3].

Algorithm 1 and Algorithm 2 of [3] are equivalent (in exact arithmetic). Algorithm 2 of [3] does not contain a routine *dobnch*, and the steps of routine *dobnch* of Algorithm 1 are explicitly included in the routine *optimality_cut* of Algorithm 2 of [3]. This configuration however, prevented the Alliant Fortran compiler from choosing the "parallel do loop" in routine *optimality_cut* of Algorithm 2 of [3] for concurrent execution. The reorganization of Algorithm 2 of [3] into Algorithm 1 above by introducing the routine *dobnch* makes the Alliant Fortran compiler choose the "parallel do loop" in Algorithm 1 for concurrent execution as desired.

The routine DPLO referred to in Algorithm 1 is an lp solver of Hanson and Hiebert [6]. The restart procedure of and modifications to DPLO mentioned in Algorithm 1 are all described in [3]. In addition to these modifications, concurrent execution of the "parallel do loop" in routine *optimality_cut* on the Alliant FX/8 requires modifications to eliminate certain COMMON blocks. (See [1, Section 5.7.11].) When we refer to DPLO in the statement of Algorithm 1 we mean a version in which we have made these additional modifications.

3. Performance Results. We have developed a Fortran implementation of Algorithm 1 on the Alliant FX/8. Using this implementation, we have performed a portion of the numerical experiment described in [3]. The numerical experiment described in [3] is quite comprehensive, consisting of four parts. Since the numerical results in [3] indicate that each part could be used to assess the performance of the implementation adequately, we selected part (b) as our experiment here. Part (b) of the experiment in [3] involves four problem sizes denoted by (i), (ii), (iii) and (iv). Problem sizes (i), (ii), (iii) and (iv) respectively correspond to dimension values $m_1 := 40$, $n_1 := 60$, $m_2 := 10$, $n_2 := 15$; $m_1 := 60$, $n_1 := 88$, $m_2 := 15$, $n_2 := 22$; $m_1 := 80$, $n_1 := 120$, $m_2 := 20$, $n_2 := 30$; and $m_1 := 100$, $n_1 := 148$, $m_2 := 25$, $n_2 := 37$. For each problem size we use a binomial-related multivariate distribution (see [3] for details) F with $K := 100$, $K := 1000$, and $K := 10000$. For each case characterized by problem size and value of K, we generate three random instances of problem (1) and run our implementation on these three instances. Any time value we give is the average of the three time values observed for the corresponding three random instances of (1). We observe two time values for each case characterized by problem size and value of K: the user time for the overall algorithm, and the user time for the routine *optimality_cut*. The latter time is important because when K is "large" the computation in Algorithm 1 is dominated by the computations in the routine *optimality_cut* (see [3] for a discussion of this fact). In fact, as described in [3], the parallel implementation of the Van Slyke and Wets algorithm given in [3] and in Algorithm 1 is motivated by this observation.

Before presenting timing results we mention four options that may be used to compile and link a code on the Alliant FX/8. More details about these options can be found in [1, Chapters 4,5]. These options are global optimizations (also referred to as general or scalar optimizations which generally optimize the code for better performance), associativity (which recognizes certain code forms that perform computations equivalent to intrinsic functions that have been implemented for good performance), vectorization, and concurrency.

In Table 1 below we report the time values observed with global optimizations, associativity, vectorization, and concurrency used to compile and link the code and all eight processors used to run the resulting executable. These and all other timing results reported in this paper were obtained on the Alliant FX/8 at the Advanced Computing Research Facility of Argonne National Laboratory, Argonne, Illinois. To measure the effect of concurrency, we ran the same executable on a single processor. (This and all other single-processor runs mentioned in this paper were made using the Alliant command *execute -c1*.) The observed time values are reported in Table 2, while the resulting speedup (referred to as speedup 1) values are indicated in Table 3.

Table 1. Time (sec) observed on eight processors with global optimizations, associativity, vectorization, and concurrency

Problem size	Overall time when K is			*optimality_cut* time when K is		
	100	1000	10000	100	1000	10000
(i)	10.8	42.2	388.4	7.5	39.7	384.8
(ii)	21.1	152.7	1543.5	15.1	143.1	1529.7
(iii)	62.3	345.7	1039.4	43.1	321.2	1029.4
(iv)	80.8	254.6	1981.3	50.9	230.9	1955.4

Table 2. Time (sec) observed on a single processor with global optimizations, associativity, vectorization, and concurrency

Problem size	Overall time when K is			*optimality_cut* time when K is		
	100	1000	10000	100	1000	10000
(i)	51.1	264.7	2451.3	47.4	261.8	2447.1
(ii)	101.4	931.2	9809.2	94.3	919.7	9792.8
(iii)	277.8	2162.4	7070.7	254.6	2132.3	7058.5
(iv)	325.9	1407.9	12486.9	289.1	1378.9	12454.9

Table 3. Values of speedup 1

Problem size	Overall speedup when K is			*optimality_cut* speedup when K is		
	100	1000	10000	100	1000	10000
(i)	4.7	6.3	6.3	6.3	6.6	6.4
(ii)	4.8	6.1	6.4	6.2	6.4	6.4
(iii)	4.5	6.3	6.8	5.9	6.6	6.9
(iv)	4.0	5.5	6.3	5.7	6.0	6.4

The speedup values in Table 3 are satisfactory for an eight-processor configuration and indicate that the concurrency capabilities of the Alliant FX/8 are utilized satisfactorily. They also compare well with similarly computed speedup values quoted in [7] for an algorithm for bound-constrained optimization problems.

As mentioned earlier, apart from concurrency, the Alliant FX/8 can enhance performance through global optimizations, associativity, and vectorization. To see how our implementation utilizes these features, we obtained timing values by repeating part (b) of the experiment in [3] on a single processor with the code compiled and linked as follows: with no global optimizations, no associativity, no vectorization, and no concurrency; with global optimizations, but with no associativity, no vectorization, and no concurrency; and with global optimizations, associativity, and vectorization but with no concurrency. The time values observed are indicated in Tables 4, 5, and 6, respectively.

Table 4. Time (sec) observed on a single processor with no global optimizations, no associativity, no vectorization, and no concurrency

Problem size	Overall time when K is			*optimality_cut* time when K is		
	100	1000	10000	100	1000	10000
(i)	79.1	454.3	4755.0	72.9	449.6	4748.3
(ii)	164.5	1663.5	18733.5	153.1	1643.1	18703.7
(iii)	467.5	3917.6	13743.2	423.8	3860.2	13723.7
(iv)	542.0	2552.6	24023.4	481.0	2507.1	23973.3

Table 5. Time (sec) observed on a single processor with global optimizations, but with no associativity, no vectorization, and no concurrency

Problem size	Overall time when K is			*optimality_cut* time when K is		
	100	1000	10000	100	1000	10000
(i)	37.8	213.2	2198.1	34.8	210.9	2194.9
(ii)	76.6	760.8	8573.1	71.1	751.3	8559.5
(iii)	215.2	1776.8	6215.0	194.9	1750.9	6205.5
(iv)	245.0	1144.8	10729.1	217.1	1123.7	10705.9

Table 6. Time (sec) observed on a single processor with global optimizations, associativity, and vectorization, but with no concurrency

Problem size	Overall time when K is			*optimality_cut* time when K is		
	100	1000	10000	100	1000	10000
(i)	35.4	182.1	1722.0	32.5	179.8	1718.9
(ii)	68.5	622.6	6558.9	63.3	613.8	6546.5
(iii)	185.4	1397.2	4598.6	166.4	1373.7	4589.2
(iv)	212.8	893.5	7965.2	185.0	871.9	7941.8

Before examining the time values in Tables 4, 5, and 6, let us observe that a speedup (referred to as speedup 2) alternative to speedup 1 above may be defined as follows to measure the utilization of concurrency by the implementation: speedup 2 is the time observed on a single processor with the code compiled and linked with global optimizations, associativity, and vectorization but with no concurrency, divided by the time observed on eight processors with the code compiled and linked with global optimizations, associativity, vectorization, and concurrency. Therefore, speedup 2 values are computed using time values in Tables 6 and 1 instead of the time values in Tables 2 and 1 used in computing speedup 1 values. The values of speedup 2 that result are reported in Table 7 below.

Table 7. Values of speedup 2

Problem size	Overall speedup when K is			*optimality_cut* speedup when K is		
	100	1000	10000	100	1000	10000
(i)	3.3	4.3	4.4	4.3	4.5	4.5
(ii)	3.2	4.1	4.2	4.2	4.3	4.3
(iii)	3.0	4.0	4.4	3.9	4.3	4.6
(iv)	2.6	3.5	4.0	3.6	3.8	4.1

The speedup 2 values in Table 7 compare well with similarly computed values reported in [9] for a quadratic programming algorithm.

Let us now consider the utilization of the vectorization capabilities of the Alliant FX/8 by our implementation. To measure these. we computed a speedup (referred to as speedup 3) by dividing the time observed with the code compiled and linked with global optimizations only and run on a single processor, by the time for the code compiled and linked with global optimizations, associativity, and vectorization and run on a single processor. Speedup 3 values are thus computed by using corresponding values in Tables 5 and 6. We report the values that result in Table 8 below.

Table 8. Values of speedup 3

Problem size	Overall speedup when K is			*optimality_cut* speedup when K is		
	100	1000	10000	100	1000	10000
(i)	1.1	1.2	1.3	1.1	1.2	1.3
(ii)	1.1	1.2	1.3	1.1	1.2	1.3
(iii)	1.2	1.3	1.4	1.2	1.3	1.4
(iv)	1.2	1.3	1.4	1.2	1.3	1.4

The speedup values in Table 8 are not satisfactory. (The possibility of vectorization speeding up computations by a factor of two to four is mentioned in [1, Section 1.2.2].) Since the computation in Algorithm 1 is dominated by work in the routine *optimality_cut* (especially for large K), and the routine *optimality_cut* essentially involves calls to DPLO, the poor vectorization indicated by the speedup 3 values in Table 8 is due to poor vectorization of DPLO. DPLO is a large package of subroutines, and in the runs mentioned in this paper we have not attempted to manually introduce Alliant compiler directives into DPLO for better performance; that is, only the automatic optimizations (including vectorization) introduced by the Alliant compiler were in effect. The speedup 3 values in Table 8 suggest that we may be able to improve the performance by manually tuning routines in DPLO for better vectorization.

Let us now examine the effect of global optimizations on the performance of our implementation. A convenient approach would be by examining the values of a speedup (referred to as speedup 4) obtained by dividing the time on a single processor for the code compiled and linked with no global optimizations, no associativity, no vectorization, and no concurrency, by the corresponding time on a single processor for the code compiled and linked with global optimizations, but with no associativity, no vectorization, and no concurrency. The values of speedup 4 are indicated in Table 9 below.

Table 9. Values of speedup 4

Problem size	Overall speedup when K is			*optimality_cut* speedup when K is		
	100	1000	10000	100	1000	10000
(i)	2.1	2.1	2.2	2.1	2.1	2.2
(ii)	2.1	2.2	2.2	2.2	2.2	2.2
(iii)	2.2	2.2	2.2	2.2	2.2	2.2
(iv)	2.2	2.2	2.2	2.2	2.2	2.2

Speedup 4 values in Table 9 indicate that the global optimizations can speed up the performance of the unoptimized code by a factor of about two. Again, since most of the computation takes place within subroutines of DPLO (especially when K is large), this performance improvement occurs in those subroutines.

The above performance results indicate that the implementation satisfactorily utilizes all the major performance improvement features of the Alliant FX/8 except its vectorization capabilities, and that it may be possible to manually tune DPLO to make the implementation utilize vectorization better.

Acknowledgment. I would like to thank Gail Pieper for her comments on an earlier version of this paper.

REFERENCES

1. Alliant Computer Systems Corporation, *FX/Fortran Programmer's Handbook*, 1988.
2. K. A. Ariyawansa, *Parallel processing in the solution of two-stage stochastic linear programs with complete recourse*, in Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing, J. J. Dongarra, P. Messina, D. C. Sorensen and R. G. Voigt, eds., Society for Industrial and Applied Mathematics, Philadelphia, 1990, pp. 143–148.
3. K. A. Ariyawansa and D. D. Hudson, *Performance of a benchmark parallel implementation of the Van Slyke and Wets algorithm for two-stage stochastic programs on the*

Sequent/Balance, Concurrency—Practice and Experience, 1991 (in press).

4. K. A. Ariyawansa, D. C. Sorensen and R. J.-B. Wets, *Parallel schemes to approximate the values and subgradients of the recourse function in certain stochastic programs*, Preprint, 1987.
5. M. A. H. Dempster, *Stochastic Programming*, Academic Press, New York, 1980.
6. R. J. Hanson and K. L. Hiebert, *A sparse linear programming subprogram*, Report SAND81-0297, Sandia National Laboratories, Alburquerque, New Mexico, and Livermore, California, 1981.
7. J. J. Moré, *On the performance of algorithms for large-scale bound constrained problems*, Preprint MCS-P140-0290, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, 1990.
8. R. Van Slyke and R. J.-B. Wets, *L-shaped linear programs with applications to optimal control and stochastic programming*, SIAM J. Appl. Math., 17 (1969), pp. 638–663.
9. S. Wright, *A fast algorithm for equality-constrained quadratic programing on the Alliant FX/8*, Annals of Operations Research, 14 (1988), pp. 225-243.