Performance of a Benchmark Parallel Implementation of the Van Slyke and Wets Algorithm for Two-Stage Stochastic Programs on the Sequent/Balance* †

DOE/ER/25045--3
DE90 005983

K. A. Ariyawansa and D. D. Hudson
Department of Pure and Applied Mathematics
Washington State University
Pullman, WA 99164-2930

Abstract: We describe a benchmark parallel version of the Van Slyke and Wets (1969) algorithm for two-stage stochastic programs and an implementation of that algorithm on the Sequent/Balance. We also report results of a numerical experiment using random test problems and our implementation. These performance results, to the best of our knowledge, are the first available for the Van Slyke and Wets (1969) algorithm on a parallel processor. They indicate that the benchmark implementation parallelizes well, and that even with the use of parallel processing, problems with random variables having large numbers of realizations can take prohibitively large amounts of computation for solution. Thus, they demonstrate the need for exploiting *both* parallelization *and* approximation for the solution of stochastic programs.

Key words: two-stage stochastic programs, decomposition algorithms, Van Slyke and Wets algorithm, parallel processing

1. Introduction

Two-stage stochastic programs with recourse[1] are a class of optimization problems with applications in a wide variety of areas in operations research. See for example Dempster (1980), Ermoliev and Wets (1988). The functions that describe these problems involve expectations with respect to multivariate random variables. The random variable in a problem instance that is actually solved usually has a discrete distribution with a finite number of realizations. This is either because the random variable has such a distribution naturally, or if not, prior to solution, the actual distribution is usually replaced by a sequence of such

*Research supported in part by DOE Grant DE-FG-06-87ER25045

†A summary of this paper was presented at the Fourth SIAM Conference on Parallel Processing for Scientific Computing, December 11–13, 1989, Chicago, Illinois.

[1]A precise description will be provided later in the paper.

---

distributions converging to it in some sense. The number of realizations of these random variables in meaningful problems is 'large' (say, more than 5000).

Many algorithms for two-stage stochastic programs have been proposed. The algorithms described by Kall (1979) are good representatives based on large scale linear programming (lp) techniques. The algorithm proposed by Van Slyke and Wets (1969) and extended by Wets (1982) is a good representative of those based on decomposition—the other major approach used to derive algorithms for stochastic programs.

It is well known (see Wets (1982,85)) that when the number of realizations of the random variable describing the problem is large, even for moderate values for dimensions that determine the size of the problem, these algorithms take prohibitively large amounts of computation. Due to this, computational experience with algorithms for two-stage programs is limited to problem instances with small numbers of realizations.

Dantzig (1985) and Wets (1985) were the first to make the observation that algorithms for stochastic programs based on decomposition can take advantage of computers with parallel processing capabilities. Wets (1985) made the further point that even with the use of parallel processors, algorithms that attempt to solve stochastic programs *exactly* may take prohibitively large amounts of computation, when the number of realizations of the random variable associated with the problem is large. (See also the timing results presented in this paper.) Wets (1985) then sketched out the rudiments of an approximating scheme, in which the given two-stage stochastic program is replaced by an approximate one constructed to have the following properties: decomposition algorithms would solve the approximate problem with considerably smaller amount of computation than the given problem; and the possibility of parallelization of decomposition algorithms for the approximate problem is retained or even enhanced relative to the possible parallelization of decomposition algorithms for the given problem. The approach, then, is to sacrifice hopefully a little in accuracy to gain speed through approximation and parallelization.

Following the work of Wets (1985), Ariyawansa, Sorensen and Wets (1987) have described implementable algorithms that generate the approximate problem in the above sense for a given two-stage stochastic program. The first author of this paper is involved in a research project to design decomposition algorithms for the approximate problem generated by the algorithms of Ariyawansa, Sorensen and Wets (1987) and to implement such algorithms as well as the algorithms of Ariyawansa, Sorensen and Wets (1987) on parallel processors. One of the aims of that project is to carefully assess the performance of these approximation procedures. Since these procedures achieve speed through approximation and parallelization at the expense of accuracy, the natural basis for evaluation would be relative to the perfor-

mance of a benchmark implementation of a parallelizable algorithm for the exact problem. The natural criteria for comparison would be measures of parallelization for the approximate and exact procedures, speed of the approximate procedure relative to the speed of the exact procedure, and measures of accuracy of solution returned by the approximate procedure relative to the solution returned by the exact procedure.

As mentioned above, published computational results for algorithms for stochastic programs are limited. Moreover, the little published computational results available do not deal with problems with random variables having large numbers of realizations and do not deal with implementations on parallel processors. Therefore, in order to meet the need mentioned in the previous paragraph we have developed a parallel version of the Van Slyke and Wets (1969) algorithm (in the form given in Wets (1982)[2]). We have also developed an implementation of this parallel version on the Sequent/Balance.

After we developed this implementation for use as a benchmark to evaluate the performance of the approximation procedures mentioned above, it occurred to us that results of a carefully designed, repeatable computational experiment using our implementation would be of interest in its own right. The purpose of this paper, then, is to describe our benchmark parallel version of the Van Slyke and Wets (1969) algorithm, its implementation on the Sequent/Balance, and the results of a computational experiment using this implementation. Our numerical results, to the best of our knowledge, are the first available on the performance of the Van Slyke and Wets (1969) algorithm on a parallel processor.

[2]Henceforth in this paper we only quote the paper by Van Slyke and Wets (1969) in connection with this algorithm.

2. The Van Slyke and Wets (1969) Algorithm for the Two-Stage Stochastic Program with Recourse

The two-stage stochastic program with recourse and with a discretely distributed random variable with a finite number of realizations, is the following[3]:

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

We refer the reader to survey papers by Wets (1982, 88) for general discussions on theory, applications and algorithms pertinent to (1) and its extensions. We mention however, that (1) arises in decision making contexts that fit into the following general format. Suppose that *at present* a decision x has to be chosen from the set $\{x : Ax = b, x \geq 0\}$ and that *at a future point in time* a realization (h^k, T^k) of $(\boldsymbol{h}, \boldsymbol{T})$ becomes available. At that point in time, the decision maker is allowed to take a corrective action y, if necessary, to allow for the difference $h^k - T^k x$. The recourse function Q in (1) models this correction process, and problem (1) is a well-posed version of this two-stage decision making problem. The solution of (1) yields a decision x^* that the decision maker can take *at present* and it hedges against the *presently* unknown realizations of $(\boldsymbol{h}, \boldsymbol{T})$.

In the rest of the paper we shall make the following assumptions regarding problem (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 is easy to verify that when (A1), (A2) and (A3) hold, problem (1) has a minimizer x^* with a *finite* minimum $z(x^*) = c^T x^* + \mathcal{Q}(x^*)$. If (A1), (A2) and (A3) are not satisfied, then it is possible for problem (1) to be infeasible or to be unbounded below. Therefore, we have

[3] We use boldface letters to denote random variables and the corresponding normal-face letters to denote their realizations.

assumed that (A1), (A2) and (A3) hold in developing our benchmark parallel implementation of the Van Slyke and Wets (1969) algorithm for (1). We hasten to add that it is possible to add necessary steps to our implementation so that problems which do not satisfy (A1), (A2) or (A3) can be handled, and infeasibility or unboundedness can be detected.

We now state the Van Slyke and Wets (1969) algorithm for problem (1), when assumptions (A1), (A2) and (A3) are satisfied.

Algorithm 2.1: (Van Slyke and Wets (1969))

input:

m_1; n_1; m_2; n_2; K; A; c; b; M; q; $F := ((h^k, T^k), p^k), k = 1, 2, \ldots, K)$.

output:

x; z.

begin
/* begin initializations */
$t := 0$;
{solve the lp
find $x \geq 0$ such that
$c^T x$ is minimized, and
$Ax = b$};
call *optimality_cut* (m_2, n_2, K, M, q, F, x, E, e, $\mathcal{Q}$);
$t := t + 1$;
$E^t := E$; $e^t := e$;
call *lowerbound* (m_1, n_1, t, A, b, $(E^l, e^l, l = 1, 2, \ldots, t)$, x, θ, z);
call *optimality_cut* (m_2, n_2, K, M, q, F, x, E, e, $\mathcal{Q}$);
/* end initializations */
while $\theta < \mathcal{Q}$ **do**
$t := t + 1$;
$E^t := E$; $e^t := e$;
call *lowerbound* (m_1, n_1, t, A, b, $(E^l, e^l, l = 1, 2, \ldots, t)$, x, θ, z);
call *optimality_cut* (m_2, n_2, K, M, q, F, x, E, e, $\mathcal{Q}$);
end while;
end.

optimality_cut:

input:

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

output:

E; e; $\mathcal{Q}$.

begin
  $E := 0$; $e := 0$;
  for $k := 1$ **to** K **do**
    $w := h^k - T^k x$;
    {solve the lp
      find $y \geq 0$ such that
      $q^T y$ is minimized, and
      $My = w$
    to obtain dual maximizer v};
    $E := E + p^K (T^k)^T v$;
    $e := e + p^k (h^k)^T v$;
  end do;
  $\mathcal{Q} := e - E^T x$;
  return;
end.

lowerbound:

input:

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

output:

x; θ; z.

begin
  {solve the lp
    find $x \geq 0$, $\theta \in \Re$ such that

$c^T x + \theta$ is minimized, and
$Ax = b$,
$(E^l)^T x + \theta \geq e^l$, $l = 1, 2 \ldots, t\}$;
$z := c^T x + \theta$;
return;
end.

In exact arithmetic, Algorithm 2.1 terminates after a finite number of iterations of the '**while** $\theta < \mathcal{Q}$ **do**' loop with $\theta = \mathcal{Q}$. We refer the reader to Van Slyke and Wets (1969) for details of Algorithm 2.1 including this finite termination property. We mention however, that Algorithm 2.1 utilizes the fact $\mathcal{Q}$ has the representation

$$\mathcal{Q}(x) = \max\{e^l - (E^l)^T x : l = 1, 2, \ldots, T\}, \quad x \in \{x : Ax = b, x \geq 0\}$$

so that problem (1) could be written as

find $x \in \Re^{n_1}$, $\theta \in \Re$ such that
$c^T x + \theta$ is minimized, and
$Ax = b$,
$(E^l)^T x + \theta \geq e^l$, $l = 1, 2, \ldots, T$,
$x \geq 0$

for appropriate $E^l \in \Re^{n_1}$, $e^l \in \Re$, $l = 1, 2, \ldots, T$ where T is *finite.* The routine *optimality_cut* generates the 'cuts' $(E^l)^T x + \theta \geq e^l$ one at a time. The routine *lowerbound* therefore solves increasingly less relaxed (i.e. using not all the constraints, $(E^l)^T x + \theta \geq e^l$, $l = 1, 2, \ldots, T$) versions of (1). Consequently, x, θ returned by routine *lowerbound* are such that $\theta \leq \mathcal{Q}(x)$ and therefore, if $\theta = \mathcal{Q}(x)$ then x must be a minimizer of problem (1). Of course, the hope is that we shall satisfy this condition well before generating all T cuts, since T is usually large.

3. A Parallel Benchmark Version of Algorithm 2.1

We begin by noting that almost all of the computational work in Algorithm 2.1 is done in routines *lowerbound* and *optimality_cut.* The work in routine *optimality_cut* essentially involves solution of the K lp's in its '**do** loop'. The work in routine *lowerbound* essentially involves the solution of a single lp. In general therefore, when the number of realizations

K is large, the computational work in Algorithm 2.1 is dominated by the computational work in routine *optimality_cut*. Routine *optimality_cut* of course is obviously parallelizable, and consequently, when K is large, we may expect to observe good parallel performance of Algorithm 2.1, simply by parallelizing routine *optimality_cut*. The purpose of this section is to state a parallel version of Algorithm 2.1, keeping in mind the need of a benchmark mentioned in Section 1.

Algorithm 2.1 needs an lp solver and certain features of the parallel version of Algorithm 2.1 that we describe here are motivated by the lp solver DPLO that we have decided to use. Before we state the parallel version of Algorithm 2.1, we therefore briefly mention certain pertinent features of DPLO. DPLO is a double precision Fortran subroutine for solving linear programs using the Revised Simplex Method (see Dantzig (1963)). It is the current version of the subroutine DPLP (single precision versions of DPLO and DPLP are named SPLO and SPLP respectively), written by R. J. Hanson and K. L. Hiebert. The reader is referred to the paper by Hanson and Hiebert (1981) for details on DPLP and SPLP.

The ability to exploit sparsity in the constraint matrix of the lp, the availability of an optional restart procedure (for solving an lp starting with the information at optimality of a related lp solved previously), robustness, portability and availability of software in public domain were the main reasons for our choice of DPLO. Ability to exploit sparsity is important for being able to solve problems with large dimensions. DPLO uses the routines of Reid (1976) for handling sparse linear programming bases through sparse LU factors. The restart procedure is useful for handling the lp's solved by routines *lowerbound* and *optimality_cut*.

DPLO maintains the constraint matrix of a given lp, and the LU factors of the current lp basis using certain sparse data structures. Suppose that a user wishes to solve an lp, and after its solution wishes to solve another lp related to the previous one (of same dimensions). He can then use DPLO as follows. He can solve the first lp from scratch, and then use an option of DPLO that allows him to write the sparse data structure representing the constraint matrix and the column indices of the optimal basis *on to a disk file*. He can then call DPLO to solve the second lp, using another option of DPLO that would allow him to read the information saved on the disk file and to start the computations with the basis specified by the column indices saved.

We wish to emphasize three points regarding the above solution process. First, DPLO allows the possibility of the two lp's having different constraint matrices, and therefore, as part of the solution of the second lp, the sparse LU factors of the starting basis is *formed*. Of course, if the constraint matrices of the two lp's were the same, (as in the context of the lp's in routine *optimality_cut*) then this computation is unnecessary, as the LU factors of the

optimal basis of the first lp is available at the end of its solution. Secondly, the basis specified by the column indices saved may not be feasible for the second lp. DPLO uses a penalty method (see pp. 12, 13 of Hanson and Hiebert (1981)) to handle this situation. Hanson and Hiebert (1981) (see p. 13) state that this procedure "almost always will reach an optimal solution with fewer iterations", relative to a solution from scratch. Thirdly, DPLO performs disk writes and reads in connection with this restart procedure, partly to save high speed memory, and partly to allow the user to use information at optimality of the first lp, when the second lp is solved in a *different* program. If one solves the two lp's in the same program, and high speed memory limitations are not a problem, then the expensive disk writes and reads are unnecessary.

We now describe our benchmark parallel version of Algorithm 2.1. Consider the routine *optimality_cut* first. Suppose that we have *nprocs* processors available. We parallelize the routine *optimality_cut* by dividing the K lp's equally among the *nprocs* processors for solution in parallel. We say that a processor would handle a 'bunch' of lp's. The lp corresponding to the smallest k (loop index) value in a bunch is solved *from scratch* using DPLO. The remaining lp's in the bunch are solved using the *restart procedure of* DPLO, *modified as follows.* Since disk reads and writes are time consuming we introduced a new option to DPLO which when turned on will use data in high speed memory to retrieve information about the previously solved lp when the restart procedure is invoked, rather than reading that information from disk. Since the lp's in routine *optimality_cut* have the same constraint matrix, when using the restart procedure to solve an lp in the routine, we do not need to form the LU factorization of the starting basis. We therefore introduced a second additional option to DPLO, which when turned on will make the restart procedure retrieve the LU factorization of the starting basis from high speed memory, rather than forming it. Thus, the lp's in the bunch other than the one corresponding to the least k value are solved using the restart procedure of DPLO with both these new options turned on.

Consider now the implementation of routine *lowerbound.* Prior to the first call to *lowerbound* in Algorithm 2.1, the lp

$$\begin{aligned} &\text{find } x \in \Re^{n_1} \text{ such that} \\ &c^T x \text{ is minimized, and} \\ &Ax = b \\ &x \geq 0 \end{aligned} \tag{2}$$

needs to be solved. Suppose that we decide to place a limit *maxcut* on the number of cuts

to be added by Algorithm 2.1. Then (2) may be written as

$$\begin{aligned} &\text{find } x \in \Re^{n_1}, \theta \in \Re \text{ such that} \\ &c^T x + \theta \text{ is minimized, and} \\ &Ax + \theta = b \\ &(E^l)^T x + \theta \geq e^l, \quad l = 1, 2, \ldots, maxcut \\ &x \geq 0 \end{aligned} \tag{3}$$

where $E^l = 0$, $e^l = 0$ for $l = 1, 2, \ldots, maxcut$. When a call is made to routine *lowerbound* with $1 \leq t \leq maxcut$, an lp of the form (3) needs to be solved, where E^l, e^l, $l = 1, 2, \ldots, t$ will have values generated by calls to *optimality_cut* and $E^l = 0$, $e^l = 0$, $l = t+1, t+2, \ldots, maxcut$. In our implementation we therefore solve the lp in routine *lowerbound*, using the restart procedure of DPLO as follows. We first solve lp (2), treating it in the form (3), *from scratch*. Routine *lowerbound* solves its lp using the restart procedure of DPLO with the new option that prevents disk reads turned on. The fact that the $(m_1 + t)$-th row of the constraint matrix changes to $[(E^t)^T, 1]$ from a row of zeros can be specified to DPLO as part of this restart procedure. We however, cannot use the LU factorization of the optimal basis of the lp of the form (3) solved prior to this call to *lowerbound*. We therefore *do not* turn on our new option that prevents DPLO from forming the LU factorization of the starting basis.

For the stopping criterion in Algorithm 2.1 we use the following in our parallel version. If

$$\frac{|\mathcal{Q} - \theta|}{\max\{1, |\mathcal{Q}|\}} \leq tol$$

where *tol* is a prescribed tolerance, we terminate.

We can now state our benchmark parallel version of Algorithm 2.1.

Algorithm 3.1: (Benchmark Parallel Version of Algorithm 2.1)

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*.

output:

x; z.

begin

/* begin initializations */
$t := 0$;
$l := 1$ **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; $\mathcal{Q}$.

begin
  $E := 0$; $e := 0$;
  $bnchsz := \lfloor K/nprocs \rfloor$;
  for $bunch := 1$ **to** $nprocs$ **in parallel do**
    for $k := (bunch - 1) \cdot bnchsz + 1$ **to** $\max\{bunch \cdot bnchsz, K\}$ **do**
      $w := h^k - T^k x$;
      if $k = (bunch - 1) \cdot bnchsz + 1$ **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;
  end do;
  $\mathcal{Q} := e - E^T x$;
  return;
end.

lowerbound:

input:

m_1; n_1; t; b; E^t; 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.

4. The Experiment

We have developed a FORTRAN implementation of Algorithm 3.1 on the Sequent/Balance. The only portion of the code that needs parallelization, as indicated in the statement of Algorithm 3.1, is that portion representing the outer '**do** loop' in routine *optimality_cut*. That parallelization was done using the 'doacross' compiler directive on the Sequent, as described in Chapter 4 of Sequent Guide to Parallel Programming (1987).

In the rest of this section we describe a computational experiment that we have performed with the implementation on problems of the form (1) satisfying assumptions (A1), (A2) and (A3). All computations were done in double precision arithmetic.

The test problems for the experiments were generated using the test problem generator GENSLP of Kall and Keller (1985). Once the user specifies the problem dimensions m_1, n_1, m_2 and n_2, ranges for elements of vectors and matrices, and densities for the matrices, GENSLP generates the deterministic data A, b, c, q and M with elements of vectors and matrices uniformly distributed in the ranges specified. GENSLP guarantees that M it generates satisfies (A2), and that the set $\{x : Ax = b, x \geq 0\}$ is nonempty. Note that we can ensure that $\{x : Ax = b, x \geq 0\}$ is bounded by selecting the range for the elements of A so that they are positive. Assumption (A3) can be satisfied by simply selecting the range for the elements of q so that these elements are nonnegative. If GENSLP cannot generate A and M so that they have the densities specified, it reports the densities it has been able to achieve.

GENSLP does not generate the distribution F for $(\boldsymbol{h}, \boldsymbol{T})$. However, it has the following

mechanism so that the user can generate a distribution F for $(\boldsymbol{h}, \boldsymbol{T})$. GENSLP treats that $\boldsymbol{h}$ and $\boldsymbol{T}$ satisfy

$$\boldsymbol{h} = a^0 + \sum_{j=1}^{\kappa} \boldsymbol{S}^j a^j \tag{4.1}$$

$$\boldsymbol{T} = A^0 + \sum_{j=1}^{\kappa} \boldsymbol{S}^j A^j. \tag{4.2}$$

In (4.1), (4.2), $a^j \in \Re^{m_2}$, $A^j \in \Re^{m_2 \times n_1}$ for $j = 0, 1, \ldots, \kappa$ are generated randomly by GENSLP once the user specifies the nonnegative integer κ, the ranges for the elements of a^j and A^j, and the density for the matrices A^j, $j = 0, 1, \ldots, \kappa$. (If GENSLP cannot generate A^j, $j = 0, 1, \ldots, \kappa$ so that they have the density specified, it reports the density it has been able to achieve.) $\boldsymbol{S}^j$, $j = 1, 2, \ldots, \kappa$ are univariate random variables for which the user can specify *univariate* distributions, thereby inducing a multivariate distributions on $(\boldsymbol{h}, \boldsymbol{T})$. In this experiment two discrete multivariate distributions on $(\boldsymbol{h}, \boldsymbol{T})$ are generated using the univariate binomial and Poisson distributions as follows. (These distributions were first described in Lessor (1988) in connection with an experiment on the performance of the algorithms of Ariyawansa, Sorensen and Wets (1987).) First a positive integer vector $r \in \aleph^{\kappa}$ is selected, where r_j is the number of realizations desired for $\boldsymbol{S}^j$, $j = 1, 2, \ldots, \kappa$. Let s_{ij} be the i-th realization of $\boldsymbol{S}^j$ and let $P_{ij} := \Pr(\{\boldsymbol{S}^j = s_{ij}\})$; $i = 1, 2, \ldots, r_j$; $j = 1, 2, \ldots, \kappa$. Note that if s_{ij} and P_{ij}; $i = 1, 2, \ldots, r_j$; $j = 1, 2, \ldots, \kappa$ are available then the following multivariate distribution on $(\boldsymbol{h}, \boldsymbol{T})$ can be described. Suppose that i_j is any integer such that $1 \leq i_j \leq r_j$; $j = 1, 2, \ldots, \kappa$, and let $i := [i_1, i_2, \ldots, i_\kappa]^T \in \aleph^{\kappa}$. Then the integer κ-tuple i defines the unique realization, say (h^k, T^k) of $(\boldsymbol{h}, \boldsymbol{T})$ via (4.1) and (4.2), where $\boldsymbol{S}^j := s_{i_j j}$; $j = 1, 2, \ldots, \kappa$, with probability $p^k = \Pi_{j=1}^{\kappa} P_{i_j j}$. There are $K := \Pi_{j=1}^{\kappa} r_j$ such different integer κ-tuples and corresponding realizations (h^k, T^k) of $(\boldsymbol{h}, \boldsymbol{T})$. It is not too difficult to verify that $\sum_{k=1}^{K} p^k = 1$, so that $F := ((h^k, T^k), p^k), k = 1, 2, \ldots, K$ indeed is a distribution on $(\boldsymbol{h}, \boldsymbol{T})$. Note also that by appropriately selecting the value of r, the value of K can be controlled.

It now remains to specify the way s_{ij} and P_{ij}; $i = 1, 2, \ldots, r_j$; $j = 1, 2, \ldots, \kappa$ are generated. Suppose that the discrete distributions for $\boldsymbol{S}^j$, $j = 1, 2, \ldots, \kappa$ are on a common range $[\alpha, \beta]$, where $\alpha, \beta \in \Re$, $\alpha < \beta$ are given. Then two distributions are specified by selecting s_{ij} and P_{ij}, $i = 1, 2, \ldots, r_j$; $j = 1, 2, \ldots, \kappa$ as follows.

(i) Binomial-related distribution on $(\boldsymbol{h}, \boldsymbol{T})$:

Suppose that $p \in (0, 1)$ is given and that $r_j > 1$. Then let

$$s_{ij} := \frac{(\beta - \alpha)}{(r_j - 1)}(i - 1) + \alpha \quad \textit{and}$$

$$P_{ij} := \binom{r_j - 1}{i - 1}(1-p)^{r_j - i}p^{i-1},$$

for $i = 1, 2, \ldots, r_j$. This is repeated for $j = 1, 2, \ldots, \kappa$.

(ii) Poisson-related distribution on $(\boldsymbol{h}, \boldsymbol{T})$:
Suppose that $\lambda \in \Re$, $\lambda > 0$ is given and that $r_j > 1$. Then let

$$s_{ij} := \frac{(\beta - \alpha)}{(r_j - 1)}(i-1) + \alpha \quad for\ i = 1, 2, \ldots, (r_j - 1);$$

$$P_{ij} := \frac{e^{-\lambda}\lambda^{i-1}}{(i-1)!} \quad for\ i = 1, 2, \ldots, (r_j - 1);$$

$$s_{r_j j} := \beta; \quad and$$

$$P_{r_j j} := 1 - \sum_{i=1}^{r_j - 1} P_{ij}.$$

This is repeated for $j = 1, 2, \ldots, \kappa$.

The random number generator used in conjunction with GENSLP is that of Schrage (1979). The description of the test problems of the form (1) used in the experiment is now complete.

As mentioned in Section 3, the lp solver used is the subroutine DPLO of Hanson and Hiebert (1981). The numerical tolerances pertinent to DPLO that were important in obtaining the numerical results reported here are 'TUNE', 'FACTOR' and 'TOLABS'. 'TUNE' and 'FACTOR' are used in determining if an lp is feasible with respect to a relative tolerance, while 'TOLABS' is used in determining if an lp is infeasible with respect an absolute tolerance if the relative tolerance criteria cannot be met.

The experiment that we have performed consists of four parts: (a), (b), (c) and (d). Part (a) may be treated as a base case. Parts (b), (c) and (d) differ from part (a) in a single key factor that may affect the performance of Algorithm 3.1. The aim therefore is to be able to see whether conclusions on the performance of Algorithm 3.1 can be made with due consideration of these factors.

We first describe part (a) of the experiment. It involves four problem sizes: (i), (ii), (iii) and (iv). The corresponding values of problem dimensions are indicated in Table 4.1.

problem size	problem dimensions				random number seed
	m_1	n_1	m_2	n_2	
(i)	40	60	10	15	7756915
(ii)	60	88	15	22	5489721
(iii)	80	120	20	30	552119
(iv)	100	148	25	37	82353

Table 4.1 Problem dimensions and random number seeds

For each problem size, we used a binomial-related distribution with $K := 100$, 1000 and 10000. Values $\kappa := 3$ and $p := 0.4$ were used for all three K values. Values of α, β, r_1, r_2 and r_3 used are indicated in Table 4.2.

K	α	β	r_1	r_2	r_3
100	-112.0	128.0	5	5	4
1000	-52.0	68.0	10	10	10
10000	-10.0	26.0	25	25	16

Table 4.2 Distribution parameters that depend on K

For each problem size (i), (ii), (iii) and (iv), and for each value of $K := 100$, 1000, 10000 we generated 3 problems of the form (1) using GENSLP. The ranges for elements of A, c, q, M, and a^j, A^j, $j = 0, 1, 2, 3$ were specified as in Table 4.3 below.

data variable	range for elements
A	$[0.5, 5]$
c	$[0.5, 1000/m_1]$
q	$[0.1, 10]$
M	$[1, 50]$
a^0	$[60, 180]$
a^j, $j \neq 0$	$[50/\kappa, 150/\kappa]$
A^0	$[120/n_1, 200/n_1]$
A^j, $j \neq 0$	$[120/(n_1\kappa), 200/(n_1\kappa)]$

Table 4.3 Ranges for problem data elements specified to GENSLP

The densities specified to GENSLP for the matrices A, M and A^j, $j = 0, 1, 2, 3$ were 20%, 40% and 10% respectively. Our aim is to obtain cpu time that Algorithm 3.1 and routine *optimality_cut* take, when Algorithm 3.1 is run parallely and sequentially with the same

value of *nprocs*, on the same problem. It is to allow for the variation of timing results due to randomness of data produced by GENSLP that we generated three problems for each case specified by problem size and value of K. Cpu time values that we report are average values computed using three values observed for the three problems. In Table 4.1 we also indicate the random number seed that we specified for each problem size, so that our experiment is repeatable. (We did not change this seed for different K values for the same problem size.)

In order to complete the description of part (a) of the experiment, we now specify the values of the tolerances TUNE, FACTOR and TOLABS used for DPLO, and the values of *tol* and *maxcut* used for Algorithm 3.1. The values TUNE := 100ε, FACTOR := 1, TOLABS := $\sqrt{\varepsilon}$, *tol* := $\sqrt{\varepsilon}$ and *maxcut* := 15 were used. Here ε is the largest relative spacing for double precision floating point numbers and is approximately 0.222×10^{-15} for the Sequent.

In Tables 4.4 and 4.5 we report the parallel and sequential cpu time values observed when *nprocs* := 7. In Table 4.6 we report the resulting speed-up values. Note that in Algorithm 3.1, we parallelize only routine *optimality_cut*, and therefore, more calls to *optimality_cut* would help improve the speed-up values. In order that the reader can put the speed-up values reported into proper perspective, in Table 4.6 we also record the number of cuts added by Algorithm 3.1 (= number of calls to *optimality_cut* − 1). As one would expect both overall and *optimality_cut* speed-up values improve as K increases for a given problem size. The overall speed-up values around 5.5 achieved for K := 10000 on a 7 processor machine are encouraging.

Throughout part (a) of the experiment we kept the densities of the matrices A, M and A^j, $j = 0, 1, \ldots, \kappa$ at the values stated above. Note that decreasing the density of A relative to the density of M is likely to improve the speed-up values. In part (b) of the experiment we investigated this hypothesis by repeating part (a) with density of A specified as 2.5% with all other details unchanged. Tables 4.7, 4.8 and 4.9 respectively indicate the information corresponding to Tables 4.4, 4.5 and 4.6, when this change is made. As expected, overall speed-up values improve.

Part (c) of the experiment is designed to see the effect of the distribution form on the performance. In part (c), we therefore repeated part (a) with the binomial-related distribution replaced by a Poisson-related distribution (keeping all other details unchanged) as follows. The parameter λ of the underlying Poisson distribution was set at 3.0. The performance results are as indicated in Tables 4.10, 4.11 and 4.12. Comparing these three tables with the corresponding tables for part (a), we observe that the speed-up values for parts (a) and (c) are similar.

Finally, we investigated the effect of the number processors on the performance by repeating part (a) with $nprocs := 14$ (but with all other details unchanged). At this point it should be mentioned that parts (a), (b) and (c) of the experiment were performed on the Sequent/Balance at the Department of Computer Science of Washington State University. This computer is configured with 10 processors. Therefore, part (d) was performed on the Sequent/Balance at the Advanced Computer Research Facility, Argonne National Laboratory, Argonne, Illinois. The latter computer has 24 processors. The results of part (d) are tabulated in Tables 4.13, 4.14 and 4.15.

As mentioned earlier, if GENSLP cannot generate a problem with the densities specified for A, M and A^j, $j = 0, 1, \ldots, \kappa$, then it reports this fact and the densities it has been able to achieve for these matrices in the problem it generated. In all parts of the experiment GENSLP was able to achieve the densities we specified for these matrices.

problem size	overall parallel cpu time (sec) when K is			*optimality_cut* parallel cpu time (sec) when K is		
	100	1000	10000	100	1000	10000
(i)	1.3069×10^2	3.4976×10^2	2.2542×10^3	3.8683×10^1	2.6030×10^2	2.1640×10^3
(ii)	6.5846×10^2	1.3522×10^3	6.5616×10^3	1.3253×10^2	8.5357×10^2	6.1249×10^3
(iii)	1.1959×10^3	2.1727×10^3	9.5141×10^3	1.8606×10^2	1.1614×10^3	8.5694×10^3
(iv)	3.2918×10^3	7.1409×10^3	2.1286×10^4	4.3730×10^2	3.5840×10^3	1.8812×10^4

Table 4.4 Part (a) of experiment: parallel cpu time

problem size	overall sequential cpu time (sec) when K is			*optimality_cut* sequential cpu time (sec) when K is		
	100	1000	10000	100	1000	10000
(i)	3.1102×10^2	1.7040×10^3	1.3315×10^4	2.1906×10^2	1.6146×10^3	1.3225×10^4
(ii)	1.2751×10^3	5.8787×10^3	3.8976×10^4	7.4731×10^2	5.3800×10^3	3.8539×10^4
(iii)	1.9569×10^3	7.9039×10^3	5.1742×10^4	9.8651×10^2	6.9294×10^3	5.0839×10^4
(iv)	5.5268×10^3	2.6114×10^4	1.1802×10^5	2.4847×10^3	2.2263×10^4	1.1539×10^5

Table 4.5 Part (a) of experiment: sequential cpu time

problem size	overall speed-up when K is			*optimality_cut* speed-up when K is			no. of cuts added when K is		
	100	1000	10000	100	1000	10000	100	1000	10000
(i)	2.37	4.87	5.91	5.66	6.20	6.11	2, 2, 2	2, 2, 2	2, 2, 2
(ii)	1.94	4.35	5.94	5.64	6.30	6.29	6, 4, 1	4, 5, 2	5, 2, 2
(iii)	1.64	3.64	5.44	5.30	5.97	5.93	2, 2, 3	2, 2, 3	2, 2, 2
(iv)	1.68	3.66	5.54	5.68	6.21	6.13	2, 6, 2	2, 10, 2	2, 4, 2

Table 4.6 Part (a) of experiment: speed-up values and no. of cuts added

problem size	overall parallel cpu time (sec) when K is			*optimality_cut* parallel cpu time (sec) when K is		
	100	1000	10000	100	1000	10000
(i)	7.6072×10^1	3.0042×10^2	3.0532×10^3	5.2394×10^1	2.8200×10^2	3.0274×10^3
(ii)	1.5943×10^2	1.1306×10^3	1.2295×10^4	1.1391×10^2	1.0553×10^3	1.2186×10^4
(iii)	4.7303×10^2	2.6342×10^3	8.6025×10^3	3.1238×10^2	2.4233×10^3	8.5256×10^3
(iv)	5.9763×10^2	1.8929×10^3	1.6342×10^4	3.7702×10^2	1.7213×10^3	1.6156×10^4

Table 4.7 Part (b) of experiment: parallel cpu time

problem size	overall sequential cpu time (sec) when K is			*optimality_cut* sequential cpu time (sec) when K is		
	100	1000	10000	100	1000	10000
(i)	3.1583×10^2	1.8145×10^3	1.8953×10^4	2.9228×10^2	1.7962×10^3	1.8927×10^4
(ii)	6.7912×10^2	6.7202×10^3	7.6310×10^4	6.3366×10^2	6.6447×10^3	7.6201×10^4
(iii)	1.9418×10^3	1.6022×10^4	5.6620×10^4	1.7820×10^3	1.5811×10^4	5.6544×10^4
(iv)	2.1976×10^3	1.0621×10^4	1.0084×10^5	1.9743×10^3	1.0448×10^4	1.0065×10^5

Table 4.8 Part (b) of experiment: sequential cpu time

problem	overall speed-up when K is			*optimality_cut* speed-up when K is			no. of cuts added when K is		
size	100	1000	10000	100	1000	10000	100	1000	10000
(i)	4.15	6.04	6.21	5.58	6.37	6.25	5, 2, 2	2, 3, 2	2, 2, 6
(ii)	4.26	5.94	6.21	5.56	6.30	6.25	3, 2, 3	2, 9, 2	2, 7, 10
(iii)	4.11	6.08	6.58	5.70	6.52	6.63	2, 10, 2	2, 13, 3	2, 2, 2
(iv)	3.68	5.61	6.17	5.24	6.07	6.23	2, 2, 5	2, 2, 2	3, 2, 2

Table 4.9 Part (b) of experiment: speed-up values and no. of cuts added

problem size	overall parallel cpu time (sec) when K is			*optimality_cut* parallel cpu time (sec) when K is		
	100	1000	10000	100	1000	10000
(i)	1.3341×10^2	3.5300×10^2	2.2567×10^3	3.8733×10^1	2.6030×10^2	2.1675×10^3
(ii)	4.4245×10^2	1.5515×10^3	6.0375×10^3	8.4394×10^1	9.7851×10^2	5.6459×10^3
(iii)	1.1315×10^3	2.0047×10^3	1.1561×10^4	1.6858×10^2	1.0478×10^3	1.0483×10^4
(iv)	3.5647×10^3	5.6141×10^3	1.7436×10^4	4.9032×10^2	2.7281×10^3	1.5378×10^4

Table 4.10 Part (c) of experiment: parallel cpu time

problem size	overall sequential cpu time (sec) when K is			*optimality_cut* sequential cpu time (sec) when K is		
	100	1000	10000	100	1000	10000
(i)	3.1352×10^2	1.7076×10^3	1.3347×10^4	2.1893×10^2	1.6149×10^3	1.3257×10^4
(ii)	8.3443×10^2	6.7179×10^3	3.5868×10^4	4.7706×10^2	6.1442×10^3	3.5473×10^4
(iii)	1.8030×10^3	7.1793×10^3	6.3529×10^4	8.9432×10^2	6.2723×10^3	6.2489×10^4
(iv)	6.0442×10^3	1.9933×10^4	9.5863×10^4	2.7361×10^3	1.6789×10^4	9.3696×10^4

Table 4.11 Part (c) of experiment: sequential cpu time

problem size	overall speed-up when K is			*optimality_cut* speed-up when K is			no. of cuts added when K is		
	100	1000	10000	100	1000	10000	100	1000	10000
(i)	2.35	4.84	5.91	5.65	6.20	6.12	2, 2, 2	2, 2, 2	2, 2, 2
(ii)	1.89	4.33	5.94	5.65	6.28	6.28	3, 2, 1	4, 7, 2	3, 3, 2
(iii)	1.59	3.58	5.50	5.31	5.99	5.96	2, 2, 2	2, 2, 2	3, 2, 3
(iv)	1.70	3.55	5.50	5.58	6.15	6.09	2, 7, 2	2, 6, 2	2, 2, 2

Table 4.12 Part (c) of experiment: speed-up values and no. of cuts added

problem size	overall parallel cpu time (sec) when K is			*optimality_cut* parallel cpu time (sec) when K is		
	100	1000	10000	100	1000	10000
(i)	1.1472×10^2	2.2629×10^2	1.2062×10^3	2.2361×10^1	1.3662×10^2	1.1159×10^3
(ii)	6.0517×10^2	9.4863×10^3	3.5677×10^3	7.5894×10^1	4.4951×10^1	3.1299×10^3
(iii)	1.1223×10^3	1.6211×10^3	5.3568×10^3	1.0463×10^2	6.1158×10^2	4.4061×10^3
(iv)	3.1156×10^3	5.4538×10^3	1.2063×10^4	2.5897×10^2	1.9179×10^3	9.5813×10^3

Table 4.13 Part (d) of experiment: parallel cpu time

problem size	overall sequential cpu time (sec) when K is			*optimality_cut* sequential cpu time (sec) when K is		
	100	1000	10000	100	1000	10000
(i)	3.1620×10^2	1.7282×10^3	1.3417×10^4	2.2372×10^2	1.6384×10^3	1.3326×10^4
(ii)	1.3222×10^3	5.9778×10^3	3.9135×10^4	7.9166×10^2	5.4763×10^3	3.8696×10^4
(iii)	2.0152×10^3	8.0491×10^3	5.2092×10^4	1.0423×10^3	7.0697×10^3	5.1194×10^4
(iv)	5.6231×10^3	2.6609×10^4	1.1882×10^5	2.5423×10^3	2.2678×10^4	1.1617×10^5

Table 4.14 Part (d) of experiment: sequential cpu time

problem size	overall speed-up when K is			*optimality_cut* speed-up when K is			no. of cuts added when K is		
	100	1000	10000	100	1000	10000	100	1000	10000
(i)	2.76	7.64	11.12	10.00	11.99	11.94	2, 2, 2	2, 2, 2	2, 2, 2
(ii)	2.18	6.30	10.96	10.43	12.18	12.36	6, 4, 1	4, 5, 2	5, 2, 2
(iii)	1.80	4.97	9.72	9.96	11.56	11.61	2, 2, 3	2, 2, 3	2, 2, 2
(iv)	1.80	4.88	9.85	9.82	11.82	12.12	2, 6, 2	2, 10, 2	2, 4, 2

Table 4.15 Part (d) of experiment: speed-up values and no. of cuts added

5. Conclusion

We have described a simple, parallel version of the Van Slyke and Wets (1969) algorithm for two-stage stochastic programs with recourse. We have also described an implementation of that algorithm on the Sequent/Balance, and the results of a carefully designed numerical experiment with the implementation on random test problems generated by the test problem generator of Kall and Keller (1985).

The parallel implementation described in the paper was motivated by the need of a benchmark to assess the performance of parallel, *approximate* algorithms that are being developed by the first author. The numerical results presented indicate that the benchmark implementation parallelizes well. They also indicate that even with the use of parallel processing, meaningful stochastic programs can take prohibitively large amounts of computation for solution. Thus, they demonstrate the need for exploiting *both* parallelization *and* approximation for the solution of stochastic programs.

We conclude this paper with the following two comments. First is that an alternative way to solve the lp's in routine *optimality_cut* of Algorithm 3.1, is to use the Dual Simplex Method (see Dantzig (1963)). An implementation of the Dual Simplex Method is not available as part of DPLO and therefore we used the restart procedure described above. The second comment we wish to make is that techniques termed 'bunching' are described by Wets (1982) for the solution of the lp's in routine *optimality_cut* of Algorithm 3.1. The performance of algorithms based on these techniques have not been studied carefully, especially on parallel processors. We plan to perform a careful study of the performance of such algorithms relative to the benchmark implementation described in this paper.

Acknowledgements

We appreciate the helpful comments of Dr. R. J. Hanson on the use of DPLO. We thank the Advanced Computer Research Facility, Argonne National Laboratory, Argonne, Illinois, and the Department of Computer Science, Washington State University, for giving us permission to use their Sequent/Balance computers.

References

1. Ariyawansa, K. A., D. C. Sorensen and R. J-B. Wets (1987) "Parallel schemes to approximate values and subgradients of the recourse function in certain stochastic programs", Manuscript, Department of Pure and Applied Mathematics, Washington State Univer-

sity, Pullman, WA 99164-2930.

2. Dantzig, G. B. (1963) *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey.
3. Dantzig, G. B. (1985) "Parallel processors for planning", Paper presented at the 12th International Symposium on Mathematical Programming, Boston, MA, August 5–9, 1985.
4. Dempster, M. A. H. (1980), *Stochastic Programming*, Academic Press, New York.
5. Ermoliev, Y. and R. J-B. Wets (1988) "Stochastic programming, an introduction", in *Numerical Techniques for Optimization*, eds., Yu. Ermoliev and R. J-B. Wets, Springer-Verlag, New York, pp. 1–32.
6. Hanson, R. J. and K. L. Hiebert (1981), "A sparse linear programming subprogram", Report SAND81-0297, Sandia National Laboratories, Alburquerque, New Mexico and Livermore, California.
7. Kall, P. (1979), "Computational methods for solving two-stage stochastic linear programming problem", *Z. Agnew. Math. Phy.*, Vol. 30, pp. 261–271.
8. Kall, P. and E. Keller (1985), "GENSLP: A program for generating input for stochastic linear programs with complete fixed recourse", Manuscript, Institüt fur Operations Research der Universität Zürich, Zürich CH-8006, Switzerland.
9. Lessor, K. S. (1988), "A preliminary numerical evaluation of a parallel algorithm for approximating the values and subgradients of the recourse function in a stochastic program with complete recourse", MS Project Report, Department of Computer Science, Washington State University, Pullman, WA 99164-1210.
10. Reid, J. (1976), "Fortran subroutines for handling sparse linear programming bases", Technical Report AERE-R 8269, Computer Science and Systems Division, AERE Harwell, Didcot, Oxfordshire, United Kingdom.
11. Schrage, L. (1979), "A more portable random number generator", *ACM Transactions of Mathematical Software*, Vol. 5, No. 2, pp. 132–138.
12. Van Slyke, R. and R. J-B. Wets (1969), "L-shaped linear programs with applications to optimal control and stochastic programming", *SIAM J. Appl. Math.*, Vol. 17, pp. 638–663.
13. Wets, R. J-B. (1982), "Stochastic programming: Solution techniques and approximation schemes", in *Mathematical Programming—The State of the Art*, eds., A. Bachem, M. Grötschel and B. Korte, pp. 566–603.
14. Wets, R. J-B. (1985), "On parallel processors design for stochastic programs", Working Paper WP 85-67, International Institute of Applied Systems Analysis, A-2361, Laxen-

burg, Austria.

15. Wets, R. J-B. (1988), "Stochastic Programming", to appear in *Handbook on Operations Research*.