

AD-A240 443



2

SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF OPERATIONS RESEARCH
STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305-4022



Solving Stochastic Linear Programs
on a Hypercube Multicomputer

by

George B. Dantzig, James K. Ho and Gerd Infanger

TECHNICAL REPORT SOL 91-10

August 1991

91-10788



Accession For	
DTIC Tab	
Unannounced	
Justification	
By	
Distribution	
Availability Codes	
Dist	Avail and/or
A-1	Special

Research and reproduction of this report were partially supported by the Office of Naval Research Grant N00014-89-J-1659; the National Science Foundation Grants ECS-8906260, DMS-8913089; and the Electric Power Research Institute Contracts RP 8010-09 and CSA-4005335. The Office of Naval Research Grant N00014-90-J-1796 at the University of Tennessee where the participation of the second author was initiated. Additional research was supported by the Austrian Science Foundation, "Fonds zur Förderung der wissenschaftlichen Forschung," Grant J0323-PHY.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do NOT necessarily reflect the views of the above sponsors.

Reproduction in whole or in part is permitted for any purposes of the United States Government. This document has been approved for public release and sale; its distribution is unlimited.

9 1 9 16 108

Solving Stochastic Linear Programs on a Hypercube Multicomputer *

George B. Dantzig[†], James K. Ho[‡], and Gerd Infanger[†]

[†] Department of Operations Research
Stanford University, Stanford, CA 94305-4022, U.S.A.

[‡] Department of Information and Decision Sciences
University of Illinois at Chicago, Chicago, IL 60680, U.S.A.

Abstract

Large-scale stochastic linear programs can be efficiently solved by using a blending of classical Benders decomposition and a relatively new technique called importance sampling. The paper demonstrates how such an approach can be effectively implemented on a parallel (Hypercube) multicomputer. Numerical results are presented.

* Research and reproduction of this report were partially supported by the Office of Naval Research Grant N00014-89-J-1659, the National Science Foundation Grants ECS-8906260, DMS-8913089, the Electric Power Research Institute Contract RP 8010-09, CSA-4O05335, and the Austrian Science Foundation, "Fonds zur Förderung der wissenschaftlichen Forschung," Grant J0323-Phy at Stanford University, and the Office of Naval Research Grant N00014-90-J-1796 at the University of Tennessee where the participation of the second author was initiated. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do NOT necessarily reflect the views of the above sponsors.

Kingsley Gnanendran and R.P. Sundarraj provided crucial programming support in early phases of this project. Access to the iPSC/2 hypercube computer was by courtesy of Oakridge National Laboratory.

1. Hypercube Multicomputers

Advances in VLSI (very large-scale integration) for digital circuit design are leading to much less expensive and much smaller computers. They have also made it possible to build a variety of "supercomputers" consisting of many small computers combined into an array of concurrent processors. We shall refer to such an architecture as multicomputers. Each individual processor is called a node. At this writing, multicomputers with up to 128 nodes are commercially available from at least half a dozen manufacturers. Typically, the nodes are the same kind as those used in high-end microcomputers and are relatively inexpensive. Significant computational power can be obtained by making many of them work in parallel at costs that are much lower than an equivalent single processor. Obviously, the effectiveness of the approach depends on whether an application can be reduced to a well-balanced distribution of asynchronous tasks on the nodes. Linear programming and especially stochastic linear programs solved by decomposition naturally fit into this framework.

A Hypercube multicomputer is essentially a network of 2^n processors interconnected in a binary n -cube (or hypercube) topology. The connections for $n \leq 4$ are illustrated in Figure 1. Each processor (or node) has its local memory and runs asynchronously of the others. Communication is done by means of messages. A node can communicate directly with its n neighbors. Messages to more distant nodes are routed through intermediate nodes. The hypercube topology provides an efficient balance between the costs of connection and the benefits of direct linkages. Usually, a host computer serves as an administrative console and as a gateway to the hypercube for users.

For the work reported in this paper, we used an Intel iPSC/2 d6 with 64 nodes at the Oak Ridge National Laboratory. Each node consists of Intel's 32-bit 80386 CPU (4 MIPS) coupled with a 80387 (300 Kflops) numeric coprocessor for floating

point acceleration. It has 4 MBytes of local memory. The hypercube (or Cube) is accessed via a host (or System Resource Manager) which is also a 80386-based system with 8 MByte memory and a 140 MByte hard disk. The operating system on the host is the UNIX System V/386 (Release 3.0). The data transfer rate between the System Resource Manager and the Cube has a peak value of 2800 KBytes/sec.

Although the nodes are physically connected as the edges of a hypercube, a trade-marked routing network called DIRECT-CONNECT provides essentially uniform communication linkages between all the nodes. The earlier "store and forward" method used in first-generation hypercubes is replaced by a hardware switching system, the Direct-Connect Module (DCR) on each node. Each DCR provides seven full-duplex channels for internodal communication and one for connection to the System Resource Manager or I/O devices. The network uses a special algorithm for messages longer than 100 bytes. It first sends ahead a header message to the destination node. This header sets gates in each DCR on the intermediate nodes to clear a data path for the message. Once communication with the destination node is established with acknowledgment of receipt of the header, the message is sent through at essentially hardware data transfer rates. The implication of this improved technology is that computational efficiency is essentially independent of the problem domain to machine topology mapping. The hypercube can be programmed as an ensemble of processors with an arbitrary communications network in which each node can communicate more or less uniformly with all other nodes. The host machine allows the user to perform the following tasks.

- To edit, compile and link host/node programs.
- To access and release the cube (or a partition thereof).
- To execute the host program.
- To start or kill processes on the cube.

Operations peculiar to the hypercube are controlled either by UNIX-type commands

(iPSC/2 commands) or by extensions to standard programming languages such as Fortran and C (iPSC/2 routines). The iPSC/2 commands are used to gain access to the cube, to load, start or kill cube processes and to relinquish access to the cube. These commands may be input from a terminal or they may be invoked using a shell script. The iPSC/2 routines, on the other hand, are mainly used to manage internodal messages. Nevertheless, it should be noted that almost all of the tasks that can be performed with iPSC/2 commands can also be accomplished from within the user programs by iPSC/2 routines with similar names. The iPSC/2 commands and routines for the Fortran programming environment are documented in Intel (1988a).

To execute a typical parallel program, the following steps are used.

- I – Compile and link the host and node programs to create executable modules.
- II – Obtain a partition of the cube (a subcube) of suitable size by invoking the GETCUBE command. The user has the option of providing a name to identify this partition. For example, the command


```
“getcube -c sugar -t d3”
```

 allocates to the user an exclusive subcube named sugar with dimension 3 (i.e. 8 processors) identified by the node numbers 0, 1, 2, . . . , 7.
- III – Run the host program by invoking the name of the executable host module. Node programs are loaded on to the appropriate nodes at runtime in response to calls to the LOAD subroutine in the host program.
- IV – On termination, kill all node processes and flush messages by invoking the KILLCUBE command.
- V – Relinquish access to the subcube by the RELCUBE command.

Internodal and host-to-node communication is done by subroutine calls in the corresponding programs. The subroutine to send messages is called CSEND. Its argu-

ments are:

- message type (ID)
- message location (address)
- message length in bytes
- destination node ID
- destination process ID.

The subroutine to receive messages is called CRECV. Its arguments are:

- message type (ID)
- address of buffer for storing message
- length of buffer in bytes.

Both CSEND and CRECV are blocking commands in the sense that the calling process halts until the message has been transmitted and received, respectively. Non-blocking versions of these commands are also provided as ISEND and IRECV respectively. Other features necessary for our purpose are the following functions:

- IPROBE (): indicating whether a message of a particular type has been received;
- MYHOST (): indicating the node ID of the host;
- MCLOCK (): returning elapsed times on the nodes and CPU times on the host; and
- MSGWAIT (): blocking the calling process until the outgoing message has been copied to the operating system buffer.

2. Two-Stage Stochastic Linear Programs

An important class of stochastic models are two-stage stochastic linear programs with recourse. These models are the analog extensions of deterministic dynamic systems which have a staircase structure: x denotes the first, y the second stage decision variables, A , b represent the coefficients and right hand sides of the

first stage constraints and D , d represent the second stage constraints, which together with the transition matrix B , couples the two periods. In the literature D is often referred to as the technology/recourse matrix. The first stage parameters are known with certainty. The second stage parameters are random variables ω that assume certain outcomes with certain probabilities $p(\omega)$. They are known only by their probability distribution of possible outcomes at time $t = 1$, where actual outcomes will be known later at time $t = 2$. Uncertainty occurs in the transition matrix B and in the right hand side vector d . The second stage costs f and the elements of the technology/recourse matrix D are assumed to be known with certainty. We denote an outcome of the stochastic parameters with $\omega, \omega \in \Omega$, with Ω being the set of all possible outcomes. The two-stage stochastic linear program can be written as follows:

$$\begin{array}{ll} \min Z = cx + E_{\omega}(fy^{\omega}) \\ s/t & Ax = b \\ & - B^{\omega}x + Dy^{\omega} = d^{\omega} \\ & x, y^{\omega} \geq 0, \quad \omega \in \Omega, \quad p(\omega) \text{ known.} \end{array}$$

The problem is to find a first stage decision x which is feasible for all scenarios $\omega \in \Omega$ and has the minimum expected costs. Note the adaptive nature of the problem: While the decision x is made only with the knowledge of the distribution $p(\omega)$ of the random parameters, the second stage decision y^{ω} is made later after an outcome ω is observed. The second stage decision compensates for and adaptes to different scenarios ω .

Two-stage stochastic linear programs have been studied extensively in the literature since Dantzig (1955), for example Birge (1985), Ermoliev (1983), Frauendorfer (1988), Higle and Sen (1989), Kall (1979), Pereira et al. (1989), Rockafellar and Wets (1989), Ruszczyński (1986), Wets (1984) and others contributed in this area (Ermoliev and Wets (1988)). Parallel decomposition for deterministic linear programs are reported e.g. in Entriken (1989), Ho and Gnanendran (1989) and Ho, Lee and Sundarraaj (1988). Examples of using parallel processing for solving

stochastic programs are Ariyawansa and Hudson (1990), Hiller and Eckstein (1990), Vladimirou and Mulvey (1990), Wets (1985), and Zenios (1990).

The difficulty of solving large-scale stochastic problems arises from the need to compute multiple integrals or multiple sums. The expected value of the second stage costs, e.g. for given first stage decision variables $\hat{x}$, $z = E(fy^\omega) = E(C)$ is an expectation of functions $C(v^\omega)$, $\omega \in \Omega$, where $C(v^\omega)$ is obtained by solving a linear programming problem. V is a h -dimensional random vector parameter, e.g. $V = (V_1, \dots, V_h)$, with outcomes $v^\omega = (v_1, \dots, v_h)^\omega$. Clearly, V is composed of the random elements of the transition matrix B and the random elements of right hand side d . For example V_i represents the percent of generators of type i down for repair or transmission lines not operating and v_i^ω the observed random percent outcome, or V_i represents an uncertain electricity demand in demand region i and v_i the observed demand realization. We also will denote the vector v^ω by v . The corresponding probability is denoted by $p(v^\omega)$ sometimes $p(v)$ or p^ω . We assume independence of the stochastic parameters Ω . The set of all possible random events, is constructed by crossing the sets of outcomes $\Omega_i, i = 1, \dots, h$ as $\Omega = \Omega_1 \times \Omega_2 \times \dots \times \Omega_h$. The expectation $E C(V)$ takes on the form of a multiple integral $E C(V) = \int \dots \int C(v)p(v)dv_1 \dots dv_h$, or, in case of discrete distributions, the form of a multiple sum $E C(V) = \sum_{v_1} \dots \sum_{v_h} C(v)p(v)$, where $p(v) = p_1(v_1) \dots p_h(v_h)$.

In the following discussion we concentrate on discrete distributions. In this case Ω takes on K values. For relevant practical problems, K can get very large and easily out of hand. Consider, for instance, the number of stochastic parameters h being as small as 20 and Ω_i , the set of possible outcomes of parameter i containing $K_i = 5$ possible outcomes each. Each term requires a function evaluation which can be computationally expensive since its value is obtained as the optimal solution of a linear program. The number of terms in the multiple sum computation, $K = 5^{20} \approx 10^{14}$. It is clear that the problem is no longer practical to be solved by direct

summation.

Using discrete distributions, one can express a stochastic problem as a deterministically equivalent linear program by writing down the second stage constraints for each scenario $\omega \in \Omega$ one below the other. The objective function carries out the expected value computation by direct summation. Clearly, this formulation leads to linear programs of enormous sizes.

$$\begin{array}{llll}
 \min Z = & cx + p^1 f y^1 + & p^2 f y^2 + \cdots + p^K f y^K & \\
 s/t & Ax & & = b \\
 & -B^1 x + Dy^1 & & = d^1 \\
 & -B^2 x & + Dy^2 & = d^2 \\
 & \vdots & \ddots & \vdots \\
 & -B^K x & & + Dy^K = d^K \\
 & x, y^1, y^2, \dots, y^K & & \geq 0.
 \end{array}$$

The method which we apply to solve large-scale stochastic linear programs uses Benders decomposition and importance sampling. The method and the underlying theory of our approach is developed in Dantzig and Glynn (1990) and Infanger (1990, 1991). Dantzig and Infanger (1991) report on the solution of large-scale problems. Entriken and Infanger (1990) discuss how reliability constraints can be handled by additionally using Dantzig-Wolfe decomposition. In the following we give a brief review of the concept. Using decomposition techniques we split the problem into a series of tractable smaller problems. Using sampling techniques we compute an estimate of the expected costs and variances. Importance sampling is the key to obtaining accurate estimates, i.e. unbiased estimates with low variances, with low sample size.

3. Benders Decomposition

We decompose the two stage stochastic linear program using Benders (1962) dual decomposition. According to Van Slyke and Wets (1961) we express the sum of second stage costs by a scalar θ and replace the second stage conditions sequentially by "cuts", which are necessary conditions expressed only in terms of the first stage

decision variables x and θ . The problem then decomposes into a **master** problem and into independent **subproblems**, one for each $\omega \in \Omega$. The latter are used to generate the cuts, unbiased estimates, and variances.

The master problem:

$$\begin{array}{ll} \min z_M & = cx + \theta \\ \text{s/t} & Ax = b \\ \text{cuts :} & -G^l x + \alpha^l \theta \geq g^l, \quad l = 1, \dots, L \\ & x, \theta \geq 0, \end{array}$$

where "cuts", are initially absent and are added one each iteration. On iteration l the master problem is optimized to obtain an approximate optimal feasible solution $x = \hat{x}^l$ (using only the l cuts generated so far) which is passed as input to the subproblems. The value of the scalar θ gives an approximation to the expected subproblems costs and $z_M = c\hat{x}^l + \hat{\theta}$ gives a lower bound estimate of $\min Z$.

The sub problems:

The solution $\hat{x}^l$ of the master problem of iteration l is sent as input to each subproblem ω which is then solved to obtain the optimal costs of the second stage problem: namely, for each scenario $\omega \in \Omega$ for given $\hat{x}^l$, the following subproblem is solved:

$$\begin{array}{ll} \min z^\omega & = fy^\omega \\ \text{s/t } \pi^\omega : & Dy^\omega = d^\omega + B^\omega x \\ & y^\omega \geq 0, \quad \omega \in \Omega, \text{ e.g. } \Omega = \{1, 2, \dots, K\}. \end{array}$$

$z^\omega = z^\omega(\hat{x}^l)$ is the optimal objective function value as a function of $\hat{x}^l$. The dual multipliers $\pi^\omega = \pi^\omega(\hat{x}^l)$ corresponding to the constraints in scenario ω , are then used to generate the next cut l to augment the set of $l - 1$ cuts found so far for the master problem.

The cuts (definition of G, g, π^ω for cut l):

$$G = E(\pi^\omega B^\omega), \quad g = E(\pi^\omega d^\omega), \quad \pi^\omega = \pi^\omega(\hat{x}^l)$$

Note that if a subproblem is infeasible a different definition of the cut is used. $\alpha^l = 0$ corresponds to feasibility cuts and $\alpha^l = 1$ corresponds to optimality cuts.

The expected value of the second stage costs:

$$z(\hat{x}^l) = E(z^\omega(\hat{x}^l))$$

Lower LB^L and upper UB^L bounds to the problem: $UB^0 = \infty$,

$$LB^L = z_M^L, \quad UB^L = \min\{UB^{L-1}, c\hat{x}^l + z(\hat{x}^l)\},$$

The optimum objective function value z_M^l of the master problem in iteration l , provides a lower bound of the objective function value of the optimum solution of the problem which monotonically increases with l . The expected costs $c\hat{x}^l + z(\hat{x}^l)$ associated with a trial solution $\hat{x}^l$, provide an upper bound to the optimal costs of the problem. These upper bounds, however, do not monotonically decrease with l , hence we recursively redefine the best solution as the one associated with the lowest upper bound obtained so far.

Computing these expectations exactly can be in practice an impossible task. Solving all subproblems $\omega \in \Omega$ once they are seeded with a trial solution $\hat{x}^l$ from the master problem means total evaluation for all ω and these can be too many. Instead we use a specialized Monte Carlo sampling technique to select a sample of subproblems ω , $\omega \in S$, to compute **estimates** of the second stage costs z^l and estimates of the gradients G^l and right hand sides g^l of the cuts. Using estimates for the gradient and the right hand side of the cuts, and estimates of the second stage costs we obtain estimates of the lower and upper bounds to the problem. These lower and upper bound estimates are viewed as sample means drawn from a population of $i \sim d$ random terms. If the sample size is forty or more the sample means for all practical purposes can be assumed to be normally distributed. The estimation process also provides estimates of the variances of these sample means.

A 95% confidence interval for the objective of the obtained optimal solution is computed. A Student-t test is used to test whether the lower and upper bounds of the objective are sufficiently close. If yes, the problem is considered solved and the iterative process terminated.

4. Importance Sampling

Monte Carlo Methods is the way recommended by numerical analysts to compute multiple integrals or multiple sums of functions of v where v is a point in a higher dimensional space h , say $h \geq 4$, (Davis and Rabinowitz (1984)). Suppose $C^\omega = C(v^\omega)$ are independent random variates of $v^\omega, \omega = 1, \dots, n$ with expectation z , where n is the sample size. An unbiased estimator of z is

$$\bar{z} = (1/n) \sum_{\omega=1}^n C^\omega.$$

with variance $\sigma_z^2 = \sigma^2/n$, $\sigma^2 = \text{var}(C(V))$. Note that the standard error decreases with $n^{-0.5}$ and the convergence rate of $\bar{z}$ to z is independent of the dimension of the sample space h . Note also the inherent parallelism of the approach. C^ω are random variates obtained by solving a linear program. The computation of C^ω can be carried out in parallel. Different sample problems are assigned to different processors and solved concurrently. As the computation of C^ω is the computationally most expensive part in the Monte Carlo scheme, the parallel implementation can be anticipated to be highly efficient, an anticipation which is borne out by the experimental results to be presented later.

Importance sampling is a variance reduction technique often applied in simulation models (Glynn and Iglehart (1989)). We rewrite $z = \sum_{\omega \in \Omega} C(v^\omega)p(v^\omega)$ as

$$\sum_{\omega \in \Omega} \frac{C(v^\omega)p(v^\omega)q(v^\omega)}{q(v^\omega)}$$

by introducing a new probability mass function $q(v^\omega)$ and we obtain a new estimator for z ,

$$\bar{z} = \frac{1}{n} \sum_{\omega=1}^n \frac{C(v^\omega)p(v^\omega)}{q(v^\omega)}$$

by sampling from the distribution $q(v^\omega)$. The variance of $\bar{z}$ is given by

$$\text{var}(\bar{z}) = \frac{1}{n} \sum_{\omega \in \Omega} \left(\frac{C(v^\omega)p(v^\omega)}{q(v^\omega)} - z \right)^2 q(v^\omega).$$

Choosing $q^*(v^\omega) = \frac{C(v^\omega)p(v^\omega)}{\sum_{\omega \in \Omega} C(v^\omega)p(v^\omega)}$ would lead to $\text{var}(\bar{z}) = 0$, which means one could get a perfect estimate of the multiple sum from a sample size $n = 1$. However, this is a useless result since to compute $q(v^\omega)$ we would need to know $z = \sum_{\omega \in \Omega} C(v^\omega)p(v^\omega)$, which is what we wanted to compute in the first place. Nevertheless, this suggests the following heuristic for choosing q . It should be proportional to the product $C(v^\omega)p(v^\omega)$ and should be of such a form that it can be integrated easily. Thus a function $\Gamma(v^\omega) \approx C(v^\omega)$ is sought, which can be integrated with less effort than $C(v^\omega)$. Additive and multiplicative (in the components of the stochastic vector v) approximation functions and combinations of these are candidate approximation functions. In particular, we have been getting good results using as our approximation to $C(v)$ a function of the form $\sum_{i=1}^h C_i(v_i)$ where $C_i(v_i)$ is (in general) a non-linear function of a scalar variable v_i . We compute q as

$$q(v^\omega) \approx \frac{C(v^\omega)p(v^\omega)}{\sum_{i=1}^h \sum_{\omega \in \Omega_i} C_i(v^\omega)}.$$

To understand the motivation for the importance sampling scheme, assume for convenience $C_i(v_i^{\omega_i}) > 0$ and let $\Gamma(v^\omega) = \sum_{i=1}^h C_i(v_i^{\omega_i})$. If $\sum C(v^\omega)p(v^\omega)$ were used as an approximation of $\bar{z}$ it can be written

$$\sum_{\omega=1}^n \Gamma(v^\omega)p(v^\omega) = \sum_{i=1}^h \alpha_i \sum_{\omega=1}^n \left[\frac{C_i(v_i^{\omega_i})}{\alpha_i} \right] p_1(v_1^{\omega_1}) p_2(v_2^{\omega_2}) \dots p_h(v_h^{\omega_h})$$

where $\omega = (\omega_1, \omega_2, \dots, \omega_h)$ and where we define

$$\alpha_i = \sum_{\omega_i \in \Omega_i} C_i(v_i^{\omega_i}) p_i(v_i^{\omega_i}),$$

which is relatively easy to compute since it can be evaluated by summing only one of the dimensions of ω . Note that

$$\bar{p}_i(v_i^{\omega_i}) = \frac{C_i(v_i^{\omega_i})p_i(v_i^{\omega_i})}{\alpha_i} \geq 0, \quad \omega_i \in \Omega_i$$

may be viewed as a modified probability distribution of v_i associated with the i term. It is, of course, a trivial matter to directly sum each term i since each of its factors, being independent probability distributions, sum to one. Suppose, however, one does not notice this fact and decides to estimate the sum by estimating each of the h terms by Monte Carlo sampling. The i -th term would then be evaluated by randomly sampling v_i from the distribution $\bar{p}_i(v_i^{\omega_i})$ and all the rest of the components v_j of v from the distributions $p_i(v_i^{\omega_i})$.

In an analogous manner, we let

$$\rho(\omega) = \frac{C(\omega)}{\Gamma(\omega)}$$

and write

$$\begin{aligned} \bar{z} &= \sum C(\omega)p(\omega) = \sum \rho(\omega)\Gamma(\omega)p(\omega) \\ &= \sum_{i=1}^h \alpha_i \sum_{\omega=1}^n \rho(\omega) \left[\frac{C_i(v_i^{\omega_i})}{\alpha_i} \right] p_1(v_1^{\omega_1}) p_2(v_2^{\omega_2}) \dots p_h(v_h^{\omega_h}). \end{aligned}$$

If our approximation $\Gamma(\omega)$ to $C(\omega)$ is any good, $\rho(\omega)$ will be roughly 1 for almost all ω . This suggests the heuristic that the sampling be carried out differently for each term i . The importance sampling scheme then is to sample v_i of the i -th term according to the distribution $\bar{p}_i(v_i^{\omega_i})$ and to sample all other components $v_j^{\omega_j}$ of the i -th term according to the distribution $p_j(v_j^{\omega_j})$.

If the additive function turns out to be a bad approximation of the cost function, as indicated by the observed variance being too high, it is easily corrected by increasing the size of the sample.

Actually we use a variant of the additive approximation function. By introducing $C(\tau)$, the costs of a base case, we make the model more sensitive to the impact of the stochastic parameters v . Our approximation function is computed as follows:

$$\Gamma(V) = C(\tau) + \sum_{i=1}^h \Gamma_i(V_i), \quad \Gamma_i(V_i) = C(\tau_1, \dots, \tau_{i-1}, V_i, \tau_{i+1}, \dots, \tau_h) - C(\tau)$$

We refer to this as a **marginal cost** approximation. We explore the cost function at the margins, e.g. we vary the random elements v_i to compute the costs for all outcomes v_i while we fix the other random elements at the level of the base case. τ can be any arbitrary chosen point of the set of k_i discrete values of v_i , $i = 1, \dots, h$. For example we choose τ_i as that outcome of V_i which leads to the lowest costs, *ceteris paribus*.

Summarizing, the importance sampling scheme has two phases: the preparation phase and the sample phase. In the preparation phase we explore the cost function $C(V)$ at the margins to compute the additive approximation function $\Gamma(V)$. For this process $n_{prep} = 1 + \sum_{i=1}^h (k_i - 1)$ subproblems have to be solved. Using $\Gamma(V)$ we compute the approximate importance density

$$q(v^\omega) = \frac{\Gamma(v^\omega)p(v^\omega)}{C(\tau) + \sum_{i=1}^h \sum_{\omega \in \Omega_i} \Gamma_i(v^\omega)p(v^\omega)}.$$

Next we sample n scenarios from the importance density and, in the sample phase, solve n linear programs to compute the estimation of $\bar{z}$ using the Monte Carlo estimator. We compute the gradient G and the right hand side g of the cut using the same sample points at hand from the expected cost calculation. See Infanger (1990, 1991) for the computation of the cuts and details of the estimation process. The function evaluations in the preparation phase and the sample phase are “made to order” for parallel processing.

5. The parallel algorithm

The Hypercube computer has the architecture of loosely coupled multiprocessors. The nodes of the cube are independent processors, where each processor has

its own operating system and its own memory. The nodes are connected via a communication network. Information is exchanged between nodes only by sending messages. The hypercube architecture defines which nodes are directly connected and which nodes are only indirectly connected via third nodes. Message routing systems of modern Hypercube computers, like the Intel iPSC/2 computer that we are using, ensure that communication between indirectly connected nodes is very fast. Thus the difference in the communication time between directly and indirectly connected nodes is neglectable. However, the time spent for communication can be significant, if much information is exchanged between nodes. Therefore the design of a parallel algorithm for loosely connected multiprocessors should be laid out in such a way that only minimum amounts of information have to be exchanged between nodes.

The main work is in the repeated solving of the master problem, and the subproblems in the preparatory phase and in the sample phase. All other tasks are comparably unimportant with respect to computing time. We assign processor 0 to be the master processor. Besides its main task of solving the master problem, the master processor also controls the computation and synchronizes the algorithm. The other processors (1 - 63) were assigned to be subprocessors, with the main task of solving subproblems. This design requires communication between the master processor and the sub processors. No information needs to be exchanged between different sub-processors.

In addition there is a host processor which has access to data storage devices and manages data input and output. The execution of the parallel program follows the following general steps: The host processor loads the host module (the executable file for the host processor) into its memory and starts the execution. Next the executable files for the master processor and the sub processors are loaded into the host and then sent to the master processor and the sub processors respectively.

The master processor and the sub processors after they receive their modules start execution. After processing the input data and sending it to master and subs, the host remains inactive and waits until it receives the optimal solution from the master processor. During this time the algorithm is performed entirely in the cube and the master processor controls the execution of the program. After receiving the optimal solution, the host processor outputs the solution to the disk, stops the execution of the programs of the master and sub processors, and releases the cube, terminating the parallel program.

The problem data includes the problem specification of the master and the sub, the stochastic information and control parameters for the execution of the program. The input data for specifying the master problem and the sub problem are given in the form of an MPS file. Internally the problems are stored in the form of the data structures used by the linear programming solver, which we use as a subroutine. We adapted LPM1 (Tomlin 1973), a linear programming optimizer, for our purposes. Clearly, the master processor only receives the data for the master problem and the sub processors only get the data for the subproblem. Thus no switching between different problems is necessary, as it would be in a serial implementation. Both master and subprocessor receive the complete stochastic information. The stochastic data include the identification of the stochastic parameters within the problem and their distributions.

An index vector $\nu^\omega = (\nu_1, \dots, \nu_h)^\omega$ completely defines a scenario ω . We define $\nu_i \in \Omega_i$ or $\nu_i = 1, \dots, k_i$, $i = 1, \dots, h$. For example $\nu^\omega = (1, 3, 2)$ would denote a scenario given by the first outcome of random parameter 1 the third outcome of random parameter 2 and the second outcome of random parameter 3. Thus only the index vector ν^ω is transmitted from the master processor to a sub processor to identify the scenario subproblem to be to be solved. For example for $h = 20$ and a 4 byte integer representation 80 bytes have to be sent. Besides the scenario

information ν^ω the current solution of the master problem, $\hat{x}^l$, is needed to set-up the scenario problem ω . We only pass $\hat{x}^l$ to each subprocessor j once per iteration at the beginning of the preparation phase. The flag $I_x \in \{0, 1\}$ tells the subprocessor if an $\hat{x}$ has to be received (1) or not (0).

Now subprocessor j looks up the outcomes of the stochastic parameters corresponding to ν to set up the the vector $b(\nu)$ and the matrix $B(\nu)$. Using $\hat{x}$ the right hand side $b(\nu) + B(\nu)\hat{x}$ is computed and the sub-problem is solved.

In any case the optimal objective function value $z(\nu)$ has to be sent to the master processor. Dual information for the coefficients G and the right hand side g of the cut is all that is needed from the base case scenario and all sample scenarios. In this case we compute the products $G(\nu) = B(\nu)\pi(\nu)$ and $g(\nu) = b(\nu)\pi(\nu)$ and send the result to the master processor. The flag I_c tells the subprocessor if the computation and the sending of $G(\nu)$ and $g(\nu)$ is requested if (1), or not if (0).

In our design the subprocessors do not have any information of the status of the algorithm. The subprocessors set-up and solve the subproblems and post-process the solution. The computation is controlled by the master processor through the flags I_x and I_c .

The master processor runs the entire algorithm except obtaining solutions of subproblems. An important task concerns the controlling of the assignment of subproblems ω to subprocessors j in the case where more sample subproblems have to be solved per iteration than there are subprocessors available. Assigning subproblems in equal proportions to subprocessors is not always possible for all sample sizes nor is it most efficient. Different subproblems need different amounts of time for getting solved. The solution time mainly depends on how many columns of the starting basis (from which the solving procedure is started) differs from the optimal basis of the subproblem. Clearly, it makes sense and is convenient to use as starting basis the optimal basis of the subproblem which was last solved on the same

processor.

We implemented an algorithm to adaptively balance the work load of the sub-processors. In our scheme the master processor keeps track if a subprocessor j is busy or idle. At the beginning of each solving phase (preparatory and sample phase) all subprocessors are idle. The master processor initiates a subprocessor working by sending the first message (I_x, I_c) to it. At this time the subprocessor is set to busy. It is set to idle again when it's solution has arrived at the master processor. Given a queue of subproblems to be solved, the first subproblem in the queue is assigned to the next idle subprocessor. The master processor keeps switching between sending out problems and receiving solutions until all subproblems are solved. Of course the mapping $\omega \rightarrow j$ is not unique because different subproblems ω are solved by one subprocessor j . However, because we only send a new problem after the solution of the previous problem has been received, the solution of a subproblem ω can be identified as uniquely coming from subprocessor j .

We can now summarize and state the algorithm. Step 2 is computationally the most expensive part and is the part computed by using parallel processors.

The Algorithm

Host processor

Step H: 0.0 Load host executable modul from disk.

Step H: 0.1 Load master modul from disk.

Send master module to processor 0.

Load sub module from disk

Send sub module to processors $j, j = 1, \dots, J$.

Step H: 0.2 Read data and from disk.

Send control data and stochastic data to processors $j, j = 0, \dots, J$.

Send master problem data to processor 0.

Send sub problem data to processors $j, j = 1, \dots, J$.

Step H: 6 Receive optimal solution.

Write solution report.

Kill cube. Stop.

Master Processor

- Step M: 0** Receive master module from host processor.
Receive control and stochastic data from host processor.
Receive master problem data from host processor.
Initialize: $l = 0, UB^0 = \infty$.
- Step M: 1** Solve the relaxed master problem.
Obtain a trial solution $\hat{x}^l$ and a lower bound LB^l .
- Step M: 2.0** $l = l + 1$.
- Step M: 2.1** Determine preparatory scenarios $\nu^\omega = (\nu_1, \dots, \nu_h)^\omega, \omega = 1, \dots, n_{prep}$.
- Step M: 2.2** $\omega = 1, \dots, n_{prep}$:
Determine $\omega \rightarrow j$.
Send I_x^j, I_c^ω to subprocessor j .
Send $\hat{x}^l$ to subprocessor j .
Send ν^ω to subprocessor j .
 $\omega = 1, \dots, n_{prep}$:
Receive z^ω from subprocessor j .
If $I_c^\omega = 1$: Receive G^ω, g^ω from subprocessor j .
- Step M: 2.3** Compute the importance distribution.
- Step M: 2.4** Sample scenarios $\nu^\omega = (\nu_1, \dots, \nu_h)^\omega, \omega = 1, \dots, n$ from the importance distribution.
- Step M: 2.5** $\omega = 1, \dots, n$:
Determine $\omega \rightarrow j$.
Send I_x^j, I_c^ω to subprocessor j .
Send ν^ω to subprocessor j .
 $\omega = 1, \dots, n$:
Receive z^ω from subprocessor j .
Receive G^ω, g^ω from subprocessor j .
- Step M: 2.6** Obtain estimates of the expected second stage cost, the coefficients and the right hand side of the cut. Add the cut to the master problem. Obtain an upper bound UB^l .
- Step M: 3** Solve the master problem.
Obtain a trial solution $\hat{x}^l$ and a lower bound LB^l .
- Step M: 4** $s = UB^l - LB^l + TOL$
If $s \geq 0$ (Student-t test) go to Step 2.
- Step M: 5** Obtain a solution and compute confidence interval.
- Step M: 6** Send optimal solution to host processor.

Sub Processor j :

- Step S: 0** Receive sub module from host processor.
Receive control and stochastic data from host processor.
Receive sub problem data from host processor.
- Step S: 2.1** Receive I_x, I_c from the master processor.
If $I_x = 1$: Receive $\hat{x}$ from the master processor.
Receive ν from the master processor.

Step S: 2.2 Compute $B(\nu)$, $b(\nu)$ and the right hand side $b(\nu) + B(\nu)\hat{x}$.

Step S: 2.3 Solve scenario subproblem ν .

Step S: 2.4 Send $z(\nu)$ to the master processor.

Step S: 2.5 If $I_c = 1$:

 Compute $G(\nu) = \pi(\nu)B(\nu)$, $g(\nu) = \pi(\nu)b(\nu)$.

 Send $G(\nu)$, $g(\nu)$ to the master processor.

Step S: 2.6 Go to Step S: 2.1

6. Performance Measures

Parallel processing main purpose is to speed up computing time relative to conventional sequential computation. In the case when large sample sizes are necessary in order to obtain good approximate solutions to stochastic linear programs, parallel processing is an important part of the solution technique, because the solution times on sequential computers may exceed time limits for practically solving the problem.

Assuming that a number p of processors are available and allocated to solve the problem at hand, we compare the parallel time utilizing p processors to the sequential time using only 1 processor. We define the parallel time t_p the duration from start to finish of the solution process in the parallel implementation. In terms of CPU times t_p covers the disjoint union (nonoverlapping total) of CPU times of all processors. We define the sequential time t_s the sum of all CPU times of all processors. The sequential time t_s differs from a sequential time obtained by actually solving the problem on one processor. This would require a different implementation and would not be directly comparable. In a serial version no messages are sent. On the other hand computing resources are needed for alternately switching between solving the master problem and the subproblems.

The speedup S in using p processors instead of one is given by $S = \frac{t_s}{t_p}$. The efficiency is defined by $E = \frac{S}{p} \times 100\%$.

A simple set of algebraic formulae can be used to predict the sequential time t_s and the parallel time t_p . We denote t_{MA} the mean duration to compute the tasks

assigned to the master processor per iteration. We define t_{SUB} the mean duration to compute the tasks assigned to a subprocessor (mainly solving one subproblem) when starting from the optimal solution of the previously solved subproblem and t_{SUB}^0 the mean duration if solving a subproblem from scratch. Thus with L being the number of iterations,

$$\frac{t_s}{L} = t_{MA} + t_{SUB}^0 + (n_{prep} + n) t_{SUB}$$

and

$$\frac{t_p}{L} = \begin{cases} t_{MA} + t_{SUB}^0 + \frac{(n_{prep} + n)}{p-1} t_{SUB}, & \text{if } n, n_{prep} \geq p-1; \\ t_{MA} + t_{SUB}^0 + \frac{(1+n)}{p-1} t_{SUB}, & \text{if } n \geq p-1, n_{prep} < p-1. \end{cases}$$

If the sample size n is smaller than the number of sub-processors the parallel algorithm is not efficient because not all computer resources are utilized. Using the above formulae, we can compute the efficiency e.g. for the case of $n, n_{prep} \geq p-1$ as

$$E = \frac{t_{MA} + t_{SUB}^0 + (n_{prep} + n) t_{SUB}}{p t_{MA} + p t_{SUB}^0 + \frac{p}{p-1} (n_{prep} + n) t_{SUB}}.$$

One can see for a fixed number of processors the efficiency approaches 100% as sample size increases. This is obvious because increasing the sample size means adding computational work which can be conducted in parallel. Thus the parallel implementation is most efficient when solving problems which require large sample sizes. On the other hand one can also see that for a given sample size the efficiency decreases with increasing number of processors. The maximum number of processors which can be utilized meaningfully is $1 + \max \{n_{prep}, n\}$.

7. Numerical Results

Experiments were conducted to validate the parallel implementation and to obtain measures of computing time, speedup and efficiency. Test problems taken from the literature are usually small with a small number of stochastic parameters.

In order to test our methodology on truly large-scale problems we build two classes of models based on practical planning models in electric power and financial investment. Numerical results using the serial implementation of the algorithm are reported in Infanger (1991) and Dantzig and Infanger (1991). Here we report on the performance of the parallel algorithm on one of these large-scale test problems.

All experiments were performed on the large-scale test problem BIGNEW which is a modified version of the capacity expansion planning model WRPM, a description of which can be found in Dantzig et al. (1989). It is a multi-area capacity expansion planning problem for western USA from Canada to Mexican border. The model is quite detailed and covers 6 regions, 3 demand blocks, 2 seasons, and several kinds of generation and transmission technologies. The objective is to determine optimum discounted least cost levels of generation and transmission facilities for each region of the system. The model minimizes the total discounted costs of supplying electricity (investment and operating costs) to meet the exogenously given demand subject to expansion and operating constraints.

In the stochastic version of the model the availabilities of generators, transmission lines, and demands are subject to uncertainty. There are 11 stochastic parameters (8 stochastic availabilities of generators and transmission lines and 3 uncertain demands) with discrete distributions with 3 or 4 outcomes. While other implementations of WRPM cover up to 3 future time periods, BIGNEW covers a planning horizon of only one future time period and is formulated as a two-stage stochastic linear program with recourse. The problem is large-scale but is by far not the largest we have solved serially. The number of universe scenarios is about 10^6 ; the equivalent deterministic formulation of the problem (if it were possible to state it explicitly) would have more than 0.3 billion constraints.

This test problem has been solved repeatedly using different numbers of processors and different sample sizes. The parallel implementation has been improved

as we learned more about its characteristics. For example in our first implementation, tables of indices $\nu^\omega, \omega = 1, \dots, n_{prep}$ and $\nu^\omega, \omega = 1, \dots, n$ were sent to each sub processor and the sub processors extracted the ν^ω from the table when solving subproblem ω . In this case the table lookups are done in parallel. However, it requires considerable communication. When sending tables of indices to all processors the message length in byte is n_{prep} or n times larger than an alternative procedure which sends only ν^ω to the corresponding sub processor. Table 1 gives a comparison of computing time of sending tables versus only index arrays. E.g. a table has 3.8 kBytes versus an index array has only 60 Bytes. At this stage the mapping of subproblems to processors $\omega \rightarrow j$ was hardwired. Thus the number of subproblems to be solved in each iteration (number of preparatory subproblems and sample size) was limited by the number of processors at hand for the computation. The comparison of the two implementation shows differences in the CPU time which increase approximately linearly with the sample size. The differences are small, however, compared to the total CPU time. The comparison shows that communications to the extent required by this algorithm do not influence the performance significantly. However, it is clear that extensive communication on some problems could increase the computing time significantly.

Next we varied the sample size between the range of 20 and 63, where we always have at least as many processors at hand as subproblems have to be solved in one parallel phase. Table 2 represents the results. The computing time (measured in CPU minutes per iteration) is approximately constant at a level of 0.12 minutes per iteration from sample size 20 up to 29. Then it jumps to a level of approximately 0.17 min per iteration where it again remains approximately constant.

In the test example the number of preparatory subproblems to compute the importance distribution is 29. Figure 2 shows how the algorithm parallelizes to indicate the efficiency of the parallel algorithm. The figure shows schematically

busy and idle times for different processors in case of sample size 63 during the first two iterations. Note the two phases of solving subproblems the preparatory phase and the sample phase. While in the preparatory phase only 29 subproblems have to be solved compared with having to solve 63 subproblems in the sample phase. Each optimization is started using the basis of the optimal solution of the problem previously solved on the same processor. At the beginning, all problems are started from scratch as no basis is available. In the first iteration processors 1 to 29 start from scratch in the preparatory phase but use the optimal bases from the preparatory subproblems in the sample phase. Processors 30 to 63 do not solve subproblems in the preparatory phase, thus the sample sub-problems assigned to these processors are started from scratch.

Solving a subproblem from scratch takes considerably more time than solving it with a good starting basis (warm start). The master processor starts operation when all necessary subproblems are solved completely, both in the preparatory phase and the sampling phase. The computing time in each phase is determined by the maximum duration spent for solving a subproblem. In the first iteration processors 30 to 63 are idle during the preparation phase and solve subproblems from scratch in the sample phase; the maximum time spent in the sample phase by these processors is much larger than the maximum time spent by processors 1 to 29. The duration of the sample phase in the first iteration is therefore much larger in the case of sample sizes larger than 29, the number of preparatory subproblems. The jump in the computing time at sample size 30 is due to this effect.

Besides the impact of the starting basis in the first iteration, there is also an impact in all other iterations. A basis of the optimal solution of a subproblem of the current iteration is expected to be a better starting basis than a basis of the optimal solution of a subproblem of the previous iteration. Note that the effect only occurs if $n_{prep} \leq p - 1$ and $n > n_{prep}$. We overcome this effect by supplying a proper

basis to subprocessors 30 to 63. In general one could copy the optimal basis of the subproblem which has finished first in the preparatory phase to processors 30 to 63 to warm start all subs in the sample phase. As idle processors are not used for any other tasks and cannot be used in a timesharing mode by other users it is more efficient (as no communication is necessary) to assign a preparatory subproblem (e.g. subproblem 1) also to processors 30 to 63 and solve it to have the optimum starting basis ready for the sampling phase. Table 2 also shows the results for warm starting all subs. The computing time remains approximately constant over the whole range of sample sizes. The results show that the effect is completely compensated. When using the warm start feature no time differences resulting from $n_{prep} < n$ can be observed. Thus the model for determining the parallel time t_p is valid for all numbers of preparatory problems n_{prep} .

The analysis so far has concerned previous implementations where the assignment of subproblems to subprocessors was hardwired. In our current implementation sub problems are sent to the next idle node. This implementation allows for any size of subproblems n_{prep} and n per iteration and divides up the number of subproblems efficiently to the number of processors available. If necessary the warm start procedure is used. In the following we are interested in the efficiency of the method both with respect to the sample size and with respect to the number of processors.

For determining the efficiency we use the formulae developed in the previous section 6. Varying the sample size over a sufficiently large range, we estimate the parameters for determining the computing time. Table 3 gives the results for sample sizes from 100 to 600 using 64 processors representing the parallel computation time versus the sample size for both the actual time measurements and the estimates from the formulae. One can see that the algebraic formulae give an excellent estimate of the actual parallel computing time. We estimate the parameters $t_{MA} + t_{SUB}^0$

to be 0.0962 and t_{SUB} to be 0.0149. Using these parameters we compute the corresponding serial time t_s , the speedup S and the efficiency E , which are also reported in Table 3. While the efficiency is low for small sample sizes it rapidly improves with increasing sample size. In the case of sample size 600, we obtain a speedup of about 37.5 which means using 64 processors we reduce the computation time by a factor of 37.5. The total parallel time is 17.3 minutes while in a serial implementation the time to solve the problem would be 652 minutes. Figure 3 shows the dependency of the efficiency upon the sample size when 64 processors are used.

Using estimates based on the formulae for the parallel time, we compute the efficiency as a function of the number of parallel processors used. Figure 4 gives a graphical representation. For small numbers of processors the effect of only $p - 1$ processors operating in parallel when using p processors dominates the result. For example when using 2 processors we switch between the master processor and only one sub processor. There is no parallel overlapping in the computation. In this case we perform a serial computation distributed to 2 processors. The efficiency hence is 50%. The efficiency increases until the above mentioned effect is not dominating anymore. E.g. for sample size 600 and using 12 processors, the efficiency is about 82%. The efficiency decreases with increasing numbers of processors beyond 12. Using 64 processors, we obtain an efficiency of 58.54% when sample size is 600.

Corresponding to the runs documented in Table 3, Table 4 reports on the optimum objective function value and the 95% confidence interval. The lower bound distributions have less variance than the upper bound distributions, hence the confidence interval is asymmetric. Using a sample size of 100 (out of about 1 million universe scenarios) we obtain an optimal solution of 188348.7 with a 95% confidence interval of 0.08% on the lower side and 0.018% on the upper side. Even with only small sample sizes we obtain amazingly accurate results. The parallel time to run

the problem was 8.3 minutes on the Hypercube multicomputer.

The optimal objective function value remains stable when increasing the sample size. That again shows that we obtained good estimates. The confidence interval decreases with increasing sample size and the rate of $n^{-0.5}$ is verified by the computational results.

Using a sample size of 600 we obtain an optimal objective function value of 188351.8 with a 95% confidence interval of 0.04% on the left side and of 0.06% on the right side. Thus the optimal solution lies with 95% confidence within $188276.7 \leq z^* \leq 188473.0$. All solutions reported in Table 4 fall within this range. The computation time on the Hypercube multicomputer was 17.3 minutes. It is interesting to note that during the process of solving the problem about 43400 subproblems of the size of 289 rows and 302 columns each and 69 master problems were solved.

Table 1: Communication time

nodes reserved	p	n	iter	CPU (min) sending tables	CPU (min) sending indices	diff	obj
32	32	20	66	7.916	7.898	0.018	188482
32	32	30	64	11.236	11.173	0.063	188271
64	51	50	63	11.118	11.034	0.084	188378
64	51	60	70	12.167	12.035	0.132	188549

Table 2: Warm start all subs

				with no warm start		with a warm start		
nodes reserved	p	n	iter	CPU (min)	time/it	CPU (min)	time/it	obj
32	32	20	66	7.898	0.120	7.898	0.120	188382
32	32	24	63	7.502	0.119	7.502	0.119	188025
32	32	26	61	7.866	0.129	7.866	0.129	188236
32	32	27	56	6.219	0.111	6.319	0.111	188232
32	32	28	52	6.434	0.124	6.434	0.124	188195
32	32	29	60	7.303	0.122	7.303	0.122	188492
32	32	30	64	11.173	0.175	7.770	0.121	188271
32	32	31	60	10.767	0.179	7.331	0.122	188301
64	33	32	64	12.409	0.194	N/A	N/A	188347
64	36	35	59	10.334	0.175	N/A	N/A	188295
64	41	40	63	10.898	0.173	7.516	0.119	188261
64	51	50	63	11.034	0.175	7.528	0.119	188378
64	61	60	70	12.035	0.172	8.374	0.120	188549
64	64	63	75	12.645	0.169	8.821	0.118	188492

Table 3 : Speedup and Efficiency

n	iter	t_p	t_p	t_s	S	E
		actual	est. by formula		speedup	efficiency
100	63	0.132	0.135	2.024	14.99	23.456
200	72	0.159	0.159	3.519	22.13	34.674
300	76	0.182	0.182	5.014	27.55	42.973
400	84	0.213	0.206	6.508	31.59	49.360
500	69	0.229	0.230	8.003	34.80	54.428
600	69	0.250	0.253	9.497	37.54	58.547

Table 4: Optimal Solution

			95% confidence interval				
n	iter	obj	lower	upper	total	%	CPU
					lower +upper	of obj	(min)
100	63	188348.7	153.0	344.4	497.4	0.26	8.3
200	72	188390.9	144.8	161.8	306.6	0.16	11.4
300	76	188344.9	100.5	180.2	280.7	0.15	13.8
400	84	188328.4	79.9	153.7	233.5	0.12	17.9
500	69	188304.0	78.0	131.1	209.0	0.11	15.8
600	69	188351.8	75.1	121.2	196.3	0.10	17.3

Figure 1. Hypercubes of dimension $n \leq 4$

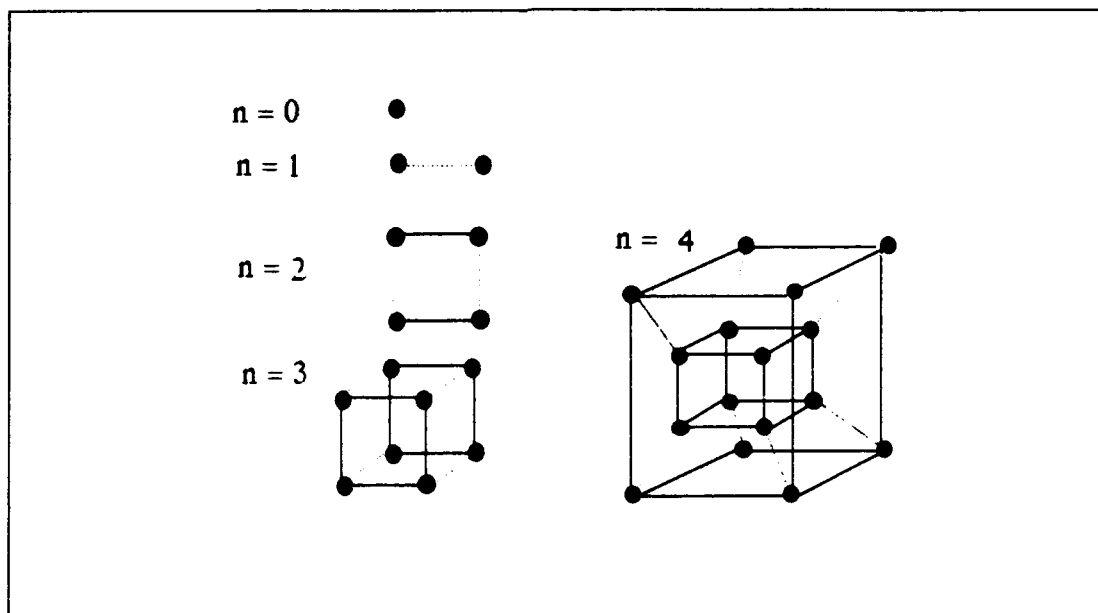


Figure 2. Efficiency of the parallel implementation

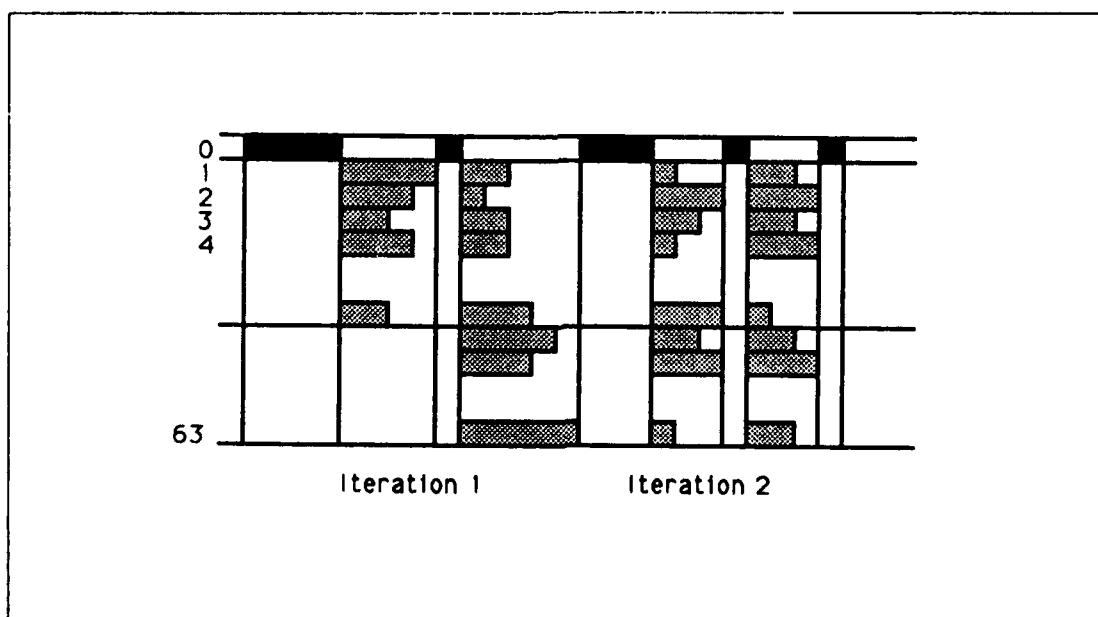


Figure 3. Efficiency versus sample size

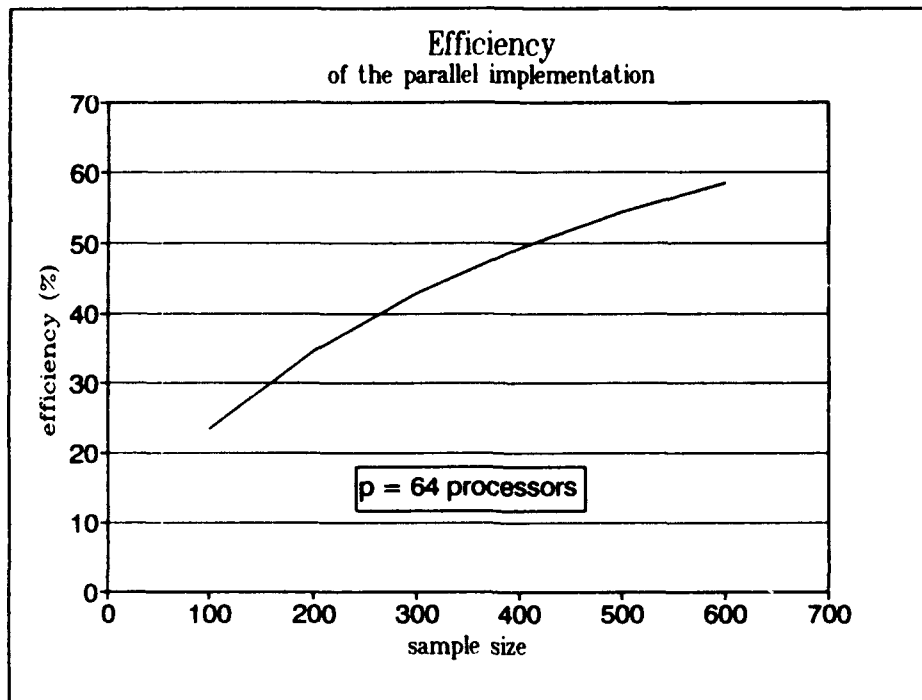
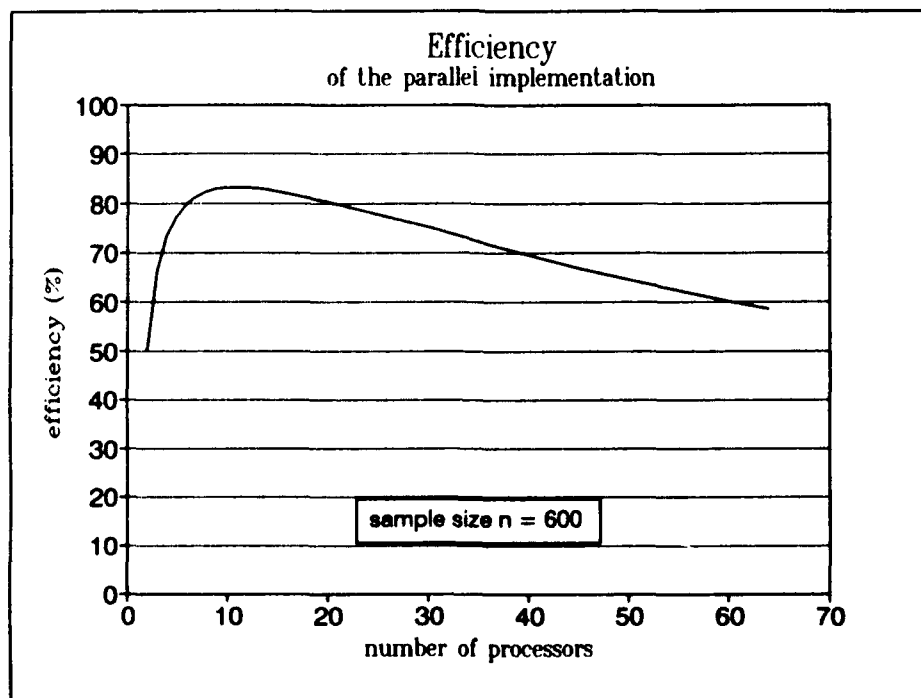


Figure 4. Efficiency versus number of processors



Literature

- Ariyawansa, K.A. and Hudson, D.D. (1990): Performance of a Benchmark Parallel Implementation of the Van Slyke and Wets Algorithm for Two-Stage Stochastic Programs on the Sequent/Balance, Working Paper, Dept. of Pure and Applied Mathematics, Pullman, WA.
- Benders, J.F. (1962): Partitioning Procedures for Solving Mixed-Variable Programming Problems, *Numerische Mathematic* 4, 238- 252.
- Birge, J.R. (1985): Decomposition and Partitioning Methods for Multi-Stage Stochastic Linear Programming, *Operations Research* 33, 989-1007.
- Dantzig, G.B. (1955): Linear Programming under Uncertainty, *Management Science* 1, 197-206.
- Dantzig, G.B. and Glynn, P.W. (1990): Parallel Processors for Planning Under Uncertainty, *Annals of Operations Research* 22, 1-21.
- Dantzig, G.B., Glynn, P.W., Avriel, M., Stone, J., Entriken, R., Nakayama, M. (1989): Decomposition Techniques for Multiarea Generation and Transmission Planning under Uncertainty, EPRI report 2940-1.
- Dantzig, G.B. and Infanger, G. (1991): Large-Scale Stochastic Linear Programs — Benders Decomposition and Importance Sampling, Technical Report SOL 91-04, Department of Operations Research, Stanford University.
- Dantzig, G.B. and Madansky M. (1961): On the Solution of Two-Staged Linear Programs under Uncertainty, *Proc 4th Berkely Symp. on Mathematical Statistics and Probability I*, ed. J. Neyman, 165-176.
- Davis, P.J., and Rabinowitz, P. (1984): Methods of Numerical Integration, Academic Press, London.
- Entriken, R. (1989): The Parallel Decomposition of Linear Programs, Technical Report SOL 89-17, Department of Operations Research, Stanford University.
- Entriken, R. and Infanger, G. (1990): Decomposition and Importance Sampling for Stochastic Linear Models, *Energy, The International Journal*, Vol. 15, No 7/8, 645-659.
- Ermoliev, Y. (1983): Stochastic Quasi-gradient Methods and Their Applications to Systems Optimization, *Stochastics* 9, 1- 36.
- Ermoliev, Y. and R.J. Wets Eds. (1988): Numerical Techniques for Stochastic Optimization, Springer Verlag.
- Frauendorfer, K. (1988): Solving SLP Recourse Problems with Arbitrary Multivariate Distributions - The Dependent Case, *Mathematics of Operations Research*,

Vol. 13, No. 3, 377-394.

- Glynn, P.W. and Iglehart, D.L. (1989): Importance Sampling for Stochastics Simulation, *Management Science*, Vol 35, 1367-1392.
- Higle, J.L. and Sen, S. (1989): Stochastic Decomposition: An Algorithm for Two Stage Linear Programs with Recourse, to appear in *Mathematics of Operations Research*.
- Hiller, R S. and Eckstein J. (1990): Stochastic Dedication: Designing Fixed Income Portfolios Using Massively Parallel Benders Decomposition, Working Paper 91-025, Harvard Business School.
- Ho, J.K. and Gnanendran, S.K. (1989): Distributed Decomposition of Block-Angular Linear Programs on a Hypercube Computer, College of Business Administration, University of Tennessee, Knoxville, TN 37996-0562.
- Ho, J.K. Lee, T.C. and Sundarraaj, R.P. (1988): Decomposition of Linear Programs using Parallel Computation, *Mathematical Programming* 42, 391-405.
- Infanger, G. (1990): Monte Carlo (Importance) Sampling within a Benders Decomposition Algorithm for Stochastic Linear Programs, Technical Report SOL 89-13R, Department of Operations Research, Stanford University.
- Infanger, G. (1991): Monte Carlo (Importance) Sampling within a Benders Decomposition Algorithm for Stochastic Linear Programs, Extended Version: Including Large-Scale Results, Technical Report SOL 91-06, Department of Operations Research, Stanford University, to appear in *Annals of Operations Research*.
- Intel Corporation (1988a): iPSC/2 Fortran Programmer's Reference Manual, Order No. 311019-003.
- Intel Corporation, (1988b)iPSC/2 Green Hills Fortran Language Reference Manual (Preliminary), Order No. 311020-003.
- Kall, P. (1979): Computational Methods for Two Stage Stochastic Linear Programming Problems, *Z. angew. Math. Phys.* 30, 261-271.
- Pereira, M.V., Pinto, L.M.V.G., Oliveira, G.C. and Cunha, S.H.F. (1989): A Technique for Solving LP-Problems with Stochastic Right- Hand Sides, CEPEL, Centro del Pesquisas de Energia Electria, Rio de Janeiro, Brazil.
- Rockafellar, R.T. and Wets, R.J. (1989): Scenario and Policy Aggregation in Optimization under Uncertainty, *Mathematics of Operations Research* (to appear).
- Ruszczynski, A. (1986): A Regularized Decomposition method for Minimizing a Sum of Polyhedral Functions, *Mathematical Programming* 35, 309-333.
- Van Slyke and Wets, R.J. (1969): L-Shaped Linear Programs with Applications

- to Optimal Control and Stochastic Programming, *SIAM Journal of Applied Mathematics*, Vol 17, 638-663.
- Tomlin, J. (1973): "LPM1 User's Guide", Unpublished Manuscript, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford CA 94305.
- Vladimirou, H. and Mulvey, J.M. (1990): Parallel and Distributed Computing for Stochastic Network Programming, Statistics and Operations Research Series Report, SOR-90-11, Princeton University, Princeton, New Jersey 08544.
- Wets R.J. (1984): Programming under Uncertainty: The Equivalent Convex Program, *SIAM J. on Appl. Math.* 14, 89-105.
- Wets R.J. (1985): On Parallel Processors Design for Solving Stochastic Programs, Proceedings of the 6th Mathematical Programming Symposium, Japan
- Zenios S.A. (1990): Massively Parallel Algorithms for Financial Modelling under Uncertainty, Decision Sciences Department, The Wharton School, University of Pennsylvania, Philadelphia, PA 19104.

REPORT DOCUMENTATION PAGEForm Approved
OMB No 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE August 1991	3. REPORT TYPE AND DATES COVERED Technical Report	
4. TITLE AND SUBTITLE Solving Stochastic Linear Programs on a Hypercube Multicomputer			5. FUNDING NUMBERS N00014-89-J-1659	
6. AUTHOR(S) George B. Dantzig, James K. Ho and Gerd Infanger			8. PERFORMING ORGANIZATION REPORT NUMBER 1111MA	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Department of Operations Research - SOL Stanford University Stanford, CA 94305-4022				
9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research - Department of the Navy 800 N. Quincy Street Arlington, VA 22217			10. SPONSORING/MONITORING AGENCY REPORT NUMBER SOL 91-10	
12a. DISTRIBUTION AVAILABILITY STATEMENT UNLIMITED			12b. DISTRIBUTION CODE UL	
13. ABSTRACT (Maximum 200 words) Large-scale stochastic linear programs can be efficiently solved by using a blending of classical Benders decomposition and a relatively new technique called importance sampling. The paper demonstrates how such an approach can be effectively implemented on a parallel (Hypercube) multicomputer. Numerical results are presented.				
14. SUBJECT TERMS Linear Programming; Large-Scale; Stochastic; Importance Sampling; Parallel Processors; Hypercube Multicomputer			15. NUMBER OF PAGES 34 pages	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT SAR	