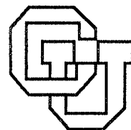
PARALLEL FACTORIZATION ON THE iPSC/860 OF STRUCTURED MATRICES ARISING IN STOCHASTIC PROGRAMMING

E.R. Jessup
Department of Computer Science
University of Colorado, Boulder 80309-0430

Dafeng Yang and Stavros A. Zenios
Operations & Information Management Department
The Wharton School, University of Pennsylvania

CU-CS-701-94 January 1994



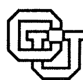
University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Parallel Factorization on the iPSC/860 of Structured Matrices Arising in Stochastic Programming

E.R. Jessup
Department of Computer Science
University of Colorado, Boulder 80309–0430

Dafeng Yang and Stavros A. Zenios
Operations & Information Management Department
The Wharton School, University of Pennsylvania

CU-CS-701-94 January 1994



University of Colorado at Boulder

Technical Report CU-CS-701-94
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Parallel Factorization on the iPSC/860 of Structured Matrices Arising in Stochastic Programming

E.R. Jessup
Department of Computer Science
University of Colorado, Boulder 80309–0430

Dafeng Yang and Stavros A. Zenios
Operations & Information Management Department
The Wharton School, University of Pennsylvania
Philadelphia, PA 19104.

January 1994

Abstract

Solving the deterministic equivalent formulation of two-stage stochastic programs using interior point algorithms requires the solution of linear systems of the form

$$(AD^2A^\top)dy = b.$$

The constraint matrix A has a dual, block-angular structure. This system of equations is dense and difficult to solve. We develop a parallel matrix factorization procedure using the Sherman-Morrison-Woodbury formula, based on the work of Birge and Qi [4]. This procedure requires the solution of smaller, independent systems of equations. With the use of optimal communication algorithms and careful attention to data layout we obtain a parallel implementation that achieves near perfect speedup Intel iPSC/860 hypercube for a variety of problems on a wide range of machine dimensions. An analysis of the computational and communication steps of the algorithm along with the timing data reveals the problem structures for which almost linear speedups can be achieved. Results are reported with the solution of linear systems arising when solving stochastic programs with up to 2048 scenarios. The largest deterministic equivalent linear program solved has 89,601 constraints and 407,130 variables.

1 Introduction

Stochastic programming is used to model a wide range of practical applications with uncertain input data. When the input data are discretely distributed — represented, for example, by a set of scenarios — the stochastic program can be formulated as a deterministic equivalent linear program with a dual, block-angular constraint matrix. For large number of scenarios these linear programs could be extremely large and computationally intractable.

The literature on stochastic programming is extensive, dating back to the 1950's with the works of G.B. Dantzig and M. Beale, and we do not stop to review it here. Significant progress has been made over the last decade. Developments in interior point algorithms hold great

promise for the solution of large-scale programs of this form. Substantial improvements are also expected with the use of parallel computers. For an introduction to stochastic programming and the deterministic equivalent formulation see Wets [21]. Applications are found in recent papers by Birge and Holmes [5], Mulvey and Vladimirou [15], Zenios [22] and their references. Algorithmic approaches for solving stochastic programs are found in Rockafellar and Wets [18], Birge and Holmes [5], Mulvey and Ruszczynski [14], Nielsen and Zenios [17] and their references. Parallel algorithms are also discussed in [5, 14, 17].

One strand of research [14, 17, 18] develops decomposition algorithms that solve the stochastic program by solving a sequence of smaller subproblems — one for each scenario — and combining the results through a master program or an aggregation step. A second strand of research [4, 5, 13] — which is followed by this paper as well — attacks the linear programming formulation of a stochastic program using interior point algorithms, and seeks matrix factorization techniques that exploit the special structure of the constraint matrix. Efficient techniques are sought for the computation of the dual step dy by solving the symmetric, positive definite system

$$(AD^2A^\top)dy = b,$$

where the constraint matrix A has a dual, block-angular structure. Birge and Holmes [5] review different methods that have been proposed to solve this system. In particular they find that a factorization technique proposed in Birge and Qi [4] (abbreviated: BQ) that uses the Sherman-Morrison-Woodbury updating formula is more efficient, stable and accurate than alternative methods based on *problem reformulations* suggested by Lustig et al. [13] or on methods using Schur complements. This conclusion is supported by numerical experiments: BQ is up to 20 times faster than methods based on problem reformulation. Worst case complexity analysis agrees with the empirical findings.

Birge and Holmes also suggest that BQ is suitable for parallel computation. However, their experiments — on a distributed network of DECstations — show only modest speedups. The best speedup was approximately 3.0 on 4 workstations. Adding more workstations worsened performance. The authors therefore conclude that "speedups are not linear with the number of processors since the communication requirements quickly overtake the benefits provided by the distribution of computational work" [5]. We demonstrate that this need not necessarily be the case. It was true for the distributed, networked environment they used because the communication was too expensive for the small-sized problems they could solve. Using a "true" parallel machine (an iPSC/860 hypercube multiprocessor with 128 processors), we show that the BQ method can be implemented in a way that achieves almost linear speedups for a wide range of problem orders and machine dimensions. We develop a model of the performance of the algorithm that, together with the experimental data, relates the execution time — both in computation and communication — to the *aspect ratio* of the blocks of the constraint matrix. In the process, we solve test problems with 2048 scenarios. The deterministic equivalent linear programs have 89,601 constraints and 407,130 variables.

Factorization procedures similar to BQ were proposed by Choi and Goldfarb [6] for multicommodity network flow problems. Their worst case analysis shows substantial savings in computations, but they ignore the cost of communication and report no empirical results. The techniques developed here are applicable to multicommodity network flow problems as well.

Section 2 establishes notation and reviews the BQ matrix factorization procedure. Section 3 identifies the basic parallelism of the factorization procedure and its communication requirements. Section 4 develops appropriate communication schemes on the targe machine, the Intel

iPSC/860 hypercube. Section 5 discusses the implementation and develops a model for the performance of the computation and communication steps of the algorithm. Section 6 interprets the model and presents the results of the computational experiments, and Section 7 concludes the paper.

2 The matrix factorization procedure

2.1 Problem formulation

The two-stage stochastic program determines an optimal first-stage decision vector $x_0 \in \Re^{n_0}$ with cost vector $c_0 \in \Re^{n_0}$ before some random coefficients are observed, and then it makes an optimal corrective (or *recourse*) action after the random coefficients become known. We assume $l = 1, 2, \ldots, N$ scenarios of the random coefficients: $c_l \in \Re^{n_l}$, $T_l \in \Re^{m_l \times n_0}$, $W_l \in \Re^{m_l \times n_l}$, $b_l \in \Re^{m_l}$. We also use $(A)_{i.}$ and $(A)_{.i}$ to denote the i-th row and column of the matrix A respectively. With this notation the program is written as

$$
\begin{array}{lll}
\text{Minimize} & c_0^\top x_0 + \sum_{l=1}^N c_l^\top y_l & \\
\text{Subject to:} & & \\
& A_0 x_0 & = b_0 \\
& T_l x_0 + W_l y_l & = b_l, \text{ for } l = 1, 2, \ldots, N \\
& x_0, \quad y_l & \geq 0.
\end{array}
$$

This problem has $n = n_0 + \sum_{l=1}^N n_l$ variables and $m = m_0 + \sum_{l=1}^N m_l$ constraints. A_0 and W_l are assumed to have full row rank, with $m_l \leq n_l$ for all $l = 0, 1, 2, \ldots, N$, and $n_0 \leq \sum_{l=1}^N n_l$.

The application of a primal-dual, path-following interior point algorithm to this linear program requires the repeated solution of symmetric, positive definite linear systems of the form

$$
(AD^2A^\top)dy = b. \tag{1}
$$

A is the constraint matrix of the two stage stochastic program

$$
A = \begin{pmatrix} A_0 & & & \\ T_1 & W_1 & & \\ \vdots & & \ddots & \\ T_N & & & W_N \end{pmatrix}.
$$

The matrix $D^2 \in \Re^{n \times n}$ is positive definite and diagonal.

This system of equations calculates the dual step dy. Its solution accounts for more than 90% of the computation in practical applications of the algorithm. The derivation of this system can be found in most references on interior-point algorithms, see, e.g., Birge and Holmes [5].

2.2 The Birge-Qi matrix factorization procedure

The procedure for solving (1) is based on the following result:

Theorem 1 *Let* $M = AD^2A^\top$, $S = diag\{S_0, S_1, \ldots, S_N\}$ *where* $S_l = W_l D_l^2 W_l^\top \in R^{m_l \times m_l}, l = 1, \ldots N$, $S_0 = I_2 \in R^{m_0 \times m_0}$ *and* $D_l \in R^{n_l \times n_l}$ *is the (diagonal) submatrix of* D *corresponding to*

the l-th block. Also, let

$$G_1 = (D_0)^{-2} + A_0^\top A_0 + \sum_{l=1}^{N} T_l^\top S_l^{-1} T_l,$$

$$G = \begin{bmatrix} G_1 & A_0^\top \\ -A_0 & 0 \end{bmatrix}, \quad U = \begin{pmatrix} A_0 & I_2 \\ T_1 & 0 \\ \vdots & \vdots \\ T_N & 0 \end{pmatrix}, \quad V = \begin{pmatrix} A_0 & -I_2 \\ T_1 & 0 \\ \vdots & \vdots \\ T_N & 0 \end{pmatrix}$$

If A_0 and W_l, $l = 1, \ldots N$, have full row rank then M and $G_2 \equiv -A_0 G_1^{-1} A_0^\top$ are invertible and

$$M^{-1} = S^{-1} - S^{-1} U G^{-1} V^\top S^{-1} \tag{2}$$

Proof : See Birge and Holmes [5, pp. 257–258]. □

It is easy to verify that, using equation (2), the solution of $M dy = b$ is given by $dy = p - r$, where p is given by $Sp = b$, and r is given by solving

$$Gq = V^\top p, \text{ and } Sr = Uq. \tag{3}$$

The vector p can be computed component-wise by solving $S_l p_l = b_l$, $l = 1, \ldots N$.

In order to solve for q we exploit the block structure of G :

$$Gq = \begin{bmatrix} G_1 & A_0^\top \\ -A_0 & 0 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} \hat{p}_1 \\ \hat{p}_2 \end{bmatrix}, \text{where} \begin{bmatrix} \hat{p}_1 \\ \hat{p}_2 \end{bmatrix} \equiv V^\top p. \tag{4}$$

Hence, we get

$$q_2 = -G_2^{-1}(\hat{p}_2 + A_0 G_1^{-1} \hat{p}_1) \tag{5}$$

$$q_1 = G_1^{-1}(\hat{p}_1 - A_0^\top q_2) \tag{6}$$

The system of equations (1) is therefore solved according to the folowing procedure from [5]:

Procedure 2.1 (Finddy($S, A_0, T_1, \ldots, T_N, b, dy$))

1. *(Solve $Sp = b$). Solve $S_l p_l = b_l$ for p_l, $l = 0, \ldots, N$.*
2. *(Solve $Gq = V^\top p$).*
   - *(a) Form G_1 by solving $S_l(u_l)_i = (T_l)_i$ for $(u_l)_i$, $l = 1, \ldots, N$, $i = 1, \ldots, n_0$ and setting $(G_1)_i = (D_0)_{ii}^{-2} + \sum_{l=1}^{N} T_l^\top (u_l)_i + (A_0^\top A_0)_i$.*
   - *(b) Form $\hat{p}_1$ and $\hat{p}_2$ using equation (4).*
   - *(c) Solve $G_1 u = \hat{p}_1$ for u and set $v = \hat{p}_2 + A_0 u$.*
   - *(d) Form G_2 by solving $(G_1) w_i = (A_0^\top)_i$ for w_i, for $i = 1, \ldots, m_0$ and setting $G_2 = -A_0[w_1, \ldots, w_{m_0}]$.*
   - *(e) Solve $G_2 q_2 = -v$ for q_2, and solve $G_1 q_1 = \hat{p}_1 - A_0^\top q_2$ for q_1.*
3. *(Solve $Sr = Uq$). Set $r_0 = A_0 q_1 + q_2$, and solve $S_l r_l = T_l q_l$ for $r_l \in \mathcal{R}^{m_l}$, $l = 1, \ldots, N$.*
4. *(Form dy.) Set $(dy)_l = p_l - r_l$ for $l = 0, \ldots, N$. Return $dy = (dy_0, \ldots, dy_N)$.*

3 The Parallel Matrix Factorization Procedure

Procedure 2.1 (Finddy) is well-suited for parallel implementation because the operations involving separate submatrices of the matrix A can be carried out independently of one another. If the computation begins with the submatrices T_{l+1}, W_{l+1} and D_{l+1} and the vector segment b_{l+1} located on node l, for $l = 0, \ldots, N-1$, node l can compute S_{l+1} and proceed independently with all computations involving only these data. Interprocessor data communication is necessary at only three points in the algorithm. In particular, in steps 2a and 2b, the nodes must communicate to form the matrix G_1 and the vectors $\hat{p}_1$ and $\hat{p}_2$. If steps 2c, 2d, and 2e proceed serially in node 0, nodes must then communicate to broadcast the computed vector q_1. Steps 3 and 4 require only the distributed data S_{l+1}, T_{l+1}, q_{l+1}, and p_{l+1} on node l and so may be carried out with full parallelism. A final communication step accumulates all the partial vectors $(dy)_{l+1}$ in node 0 for use in subsequent iterations. Thus, all of the communication steps require either broadcasting of data from one node to all others (*one-to-all* communication) or gathering of data distributed among processors. The data gathers can be either *all-to-all* or *all-to-one* depending on intended use of the data and speed of the gather routines.

In this paper we demonstrate the parallel efficiency of the Finddy procedure on a distributed-memory MIMD multiprocessor. In section 4, we introduce the iPSC/860 hypercube multiprocessor used for our implementation and experiments. We also describe the optimal one-to-all and all-to-all communication routines that are the basis for the communication routines used in our parallel implementation. In section 5, we present the details of our parallel implementation. We explain both how the computations are divided between the processors and how the generic communication routines presented in Section 4 are adapted for use in our implementation.

4 The Communication Schemes on the Target Machine

We performed our experiments on an Intel iPSC/860 hypercube multiprocessor. For more information on portability of our code, see Section 5.4.

A hypercube is a distributed-memory MIMD message-passing parallel computer in which processors are connected according to a hypercube graph. A single processor together with its associated local memory is referred to as a *node*. The size of a hypercube is defined by its dimension d, and a hypercube graph of dimension d and the multiprocessor based on it are called *d-cubes*. A d-cube graph has $p = 2^d$ nodes, and the nodes of the hypercube computer are located at the nodes of the graph.

Nodes in a d-cube are assigned d-bit binary identifiers (from 0 through $p-1$) such that the d nodes connected to node j all have identifiers differing from j in exactly one bit. A d-cube can be constructed by connecting corresponding nodes of two $(d-1)$-cubes in any of d ways. For example, the familiar 3-cube can be made by linking corresponding processors of the two squares or 2-cubes forming its top and bottom, left and right, or front and back faces. In a d-cube, the d neighbors of node j define the d nodes corresponding to node j in the d different $(d-1)$-cubes.

The special properties of the hypercube graph are the basis for the efficient interprocessor data communication routines on the hypercube multiprocessor. While modern hypercube multiprocessors support *wormhole routing* by which messages may be sent efficiently between any pair of nodes, the most efficient codes are still those in which messages do not cross on communication wires. One way to ensure contention free routing of messages is to ensure that

only neighboring nodes in the cube communicate with one another. We have used that rule in our code design.

The first communication scheme broadcasts data from node 0 to all others via the links of a spanning tree of the hypercube graph rooted at node 0. All communication takes place between nearest neighbors. This procedure is termed a *spanning tree broadcast (STB)*. The spanning tree of height d is embedded into a d-cube by simple bit manipulation of the node identifiers. Node 0 is at the root of the tree. The broadcast proceeds for a total of d communication steps where at step l, for $l = 1, \ldots, d$, each node with identifier $j < 2^l$ pairs with the node with identifier different from j in bit l only. For each pair, the node with the smaller identifier is the parent node and sends its data to its child. Thus, at step l of the algorithm, 2^{l-1} nodes receive the data. When a k-byte message is broadcast from node 0 to all others, the communication cost is

$$d(\beta + k\tau),$$

where β is the startup time for a message passed between two processors, and τ is the time for one byte of data to pass between two neighboring processors. We use an STB to broadcast the vector q_1 in the parallel implementation of step 2e of Procedure 2.1 (Finddy). (See section 5 for more details on all steps.)

The STB moves data from one node to all others. Suppose, in contrast, that each of the p nodes begins with k bytes of data but that the full pk bytes of data need to be accumulated in node 0. The distributed data can be gathered by traversing the spanning tree from its leaves to its roots. Again, the gather proceeds for d steps, but every node receiving data from its child appends that data to its own data and forwards the accumulated data to its parent in the next communication step. Upon completion, node 0 holds all pk bytes of data. The cost of this *spanning tree gather (STG)* is

$$\begin{aligned}(\beta + k\tau) + (\beta + 2k\tau) + \cdots + (\beta + 2^{d-1}k\tau)\\ = d\beta + (p-1)k\tau.\end{aligned}$$

We use a global sum routine based on an STG in the parallel implementation of steps 2a and 2b of Finddy.

The second communication routine is based on an *alternate direction exchange (ADE)* as described in [19]. An ADE can be used, for example, to accumulate in all $p = 2^d$ nodes pk bytes of data initially distributed with k bytes per node. In each of d communication steps, the d-cube splits into a different pair of $(d-1)$-cubes. The 2^{d-1} pairs of corresponding nodes from the two cubes exchange and accumulate their data sets. Thus, as in the STG, the size of the data set received by any processor at communication step l is $2^{l-1}k$ bytes, $l = 1, \ldots, d$.

All communication takes place between neighboring nodes. Thus, the time to perform this ADE on a d-cube is

$$\begin{aligned}\alpha[(\beta + k\tau) + (\beta + 2k\tau) + \cdots + (\beta + 2^{d-1}k\tau)]\\ = \alpha[d\beta + (p-1)k\tau],\end{aligned}$$

where the parameter α lies between 1 and 2, inclusive. It is 2 when the exchange between $(d-1)$-cubes is accomplished by a send-receive-send-receive pattern of communication commands. By synchronizing all nodes, α can be made closer to 1 on the iPSC/860 [20] when k is large. We do not synchronize the nodes and so use the approximation $\alpha = 2$ in the models we develop in Section 5.

STB, STG, and ADE are optimal broadcast and gather routines in the sense that they take only the minimum number of communication steps — d steps on a d-cube [19]. Minimizing the number of communication steps is important on present-day distributed-memory computers as the cost of communication is typically high in comparison to the cost of computation. In particular, the cost of communicating an m-byte message from one hypercube node to a neighboring one is $\beta + m\ \tau$, where the communication startup latency β is generally large in comparison to the transmission time per byte τ. (On the iPSC/860, $\beta/\tau \approx 340$ for messages of more than 100 byte [10].)

5 The Parallel Implementation

In this section, we step through the parallel implementation of Procedure 2.1 (Finddy) presenting first the communication requirements and then the computation requirements. In the process, we develop a model of the communication and computation costs of the parallel algorithm. We summarize the parallel algorithm as Procedure 5.1 (Parallel Finddy) in Section 5.

5.1 The communication steps

Each of the three communication steps in Parallel Finddy is based either on a spanning tree broadcast/gather (STB or STG) or on an alternate direction exchange (ADE). In what follows, we detail how each communication step proceeds and estimate its cost.

Steps 2a and 2b. In step 2a, node l, $l = 0, \ldots, N-1$ solves the system $S_{l+1}(u_{l+1})_i = (T_{l+1})_i$ for $(u_{l+1})_i$, $i = 1, \ldots, n_0$, then determines the matrix $\hat{T}_{l+1} = (T_{l+1}^\top (u_{l+1})_1, \ldots, T_{l+1}^\top (u_{l+1})_{n_0})$. (Note that S_{l+1} need be factored only once to solve the n_0 systems.) Then, using a communication routine based on an STG rooted at node 0, nodes communicate to accumulate the matrix sum G_1. In this routine, each node at a leaf of the spanning tree initializes a partial sum with its own matrix $\hat{T}_{l+1}$ and sends that partial sum to its parent node. As the gather proceeds, a node receives partial sums from all of its children, adds them to its own partial sum, then sends the result to its parent. We call this variation of an STG a *spanning tree global sum*. After the root node 0 receives the partial sums of its children, it sums them with $(D_0)^{-2} + A_0^\top A_0$ to form the matrix G_1.

Step 2b has similar communication requirements. For $l = 0, \ldots, N-1$, node l forms $\hat{t}_{l+1} = T_{l+1}^\top p_{l+1}$. All nodes then determine a global sum of all terms $\hat{t}_{l+1}$ by means of a spanning tree global sum rooted at node 0. The final sums $\hat{p}_1 = A_0^\top p_0 + \sum_{l=1}^{N} \hat{t}_l$ and $\hat{p}_2 = -p_0$ are found in node 0.

The Intel communication routine GDSUM performs a global sum of distributed scalar variables only and so is not appropriate for steps 2a and 2b. Thus, we have used our own spanning tree global sum routine GpGATHER for Steps 2a and 2b. This routine packs the matrix partial sums (from step 2a) and the vector partial sums (from step 2b) into one common block of size $n_0 \times (n_0 + 1)$ and sends them together to save on message startups. Communication proceeds according to the spanning tree gather just described with both matrix and vector sums performed together at each step.

The time for a GpGATHER of double precision (8-byte) quantities is

$$t_{Gp} \quad = \quad d[(n_0^2 + n_0)]\omega + d[\beta + 8(n_0^2 + n_0)\tau]$$

$$= T_{Gp}^{comp} + T_{Gp}^{comm},$$

where ω is the time for a floating point operation.

Step 2e. Step 2e produces the vector q_1 in node 0. For step 3 to proceed, node 0 broadcasts this vector to all others. For this broadcast, we use our own spanning tree broadcast routine STB. For double precision vector elements, the time for the STB is

$$t_{STB} = d\beta + 8d\frac{n_0}{p}\tau.$$

Our timing tests have shown that our own routine STB takes time identical to that needed by the Intel communication routine csend to effect a one-to-all broadcast. The STB is also faster than our ADE routine for broadcasting q_1.

Step 4. The final step in FINDDY is to gather the pieces of the computed vector dy together in order in node 0. While our STB and Intel's GCOLX provide equivalent ways to do this gather, our timings have shown that an ADE is in fact the fastest way of gathering the ordered vector. Our ADE routine that orders the vector properly in node 0 is called dyGATHER.

The time to perform a dyGATHER is

$$\begin{aligned} t_{dy} &= 2[\beta + \frac{8Nm_1}{p}\tau + \beta + \frac{16Nm_1}{p}\tau + \ldots + \beta + 8 * 2^{d-1}\frac{Nm_1}{p}\tau] \\ &= 2[d\beta + 8(2^d - 1)\frac{Nm_1}{p}\tau]. \end{aligned}$$

The total time for communication in the parallel Finddy procedure is thus

$$T_{comm} = 4d\beta + [8d(n_0^2 + n_0) + 8d\frac{n_0}{p} + 16(2^d - 1)\frac{Nm_1}{p}]\tau \qquad (7)$$

5.2 The computation steps

In this section, we describe the distribution of computation in the parallel implementation and derive a model of the computational costs. To begin, we assume that for $l = 0, \ldots, N-1$, node l starts with the submatrices S_{l+1} and T_{l+1} and the vector segment b_{l+1}. (That is, the number of processors p equals the number of submatrices N.) Node 0 also holds A_0, S_0, D_0, and b_0. We conclude this section by generalizing our model to the case of N/p submatrices and vector segments per node. Recall that $A_0 \in \Re^{m_0 \times n_0}$, $T_{l+1} \in \Re^{m_{l+1} \times n_0}$, $S_{l+1} \in \Re^{m_{l+1} \times m_{l+1}}$, $D_{l+1} \in \Re^{n_{l+1} \times n_{l+1}}$, and $b_{l+1} \in \Re^{m_{l+1}}$. In this section, we count all of the floating point operations performed by the parallel Finddy algorithm. We then present an estimate of the total time required by the algorithm by charging a time of ω to each operation. As we will show in Section 6.2, however, the proper value of omega is strongly dependent not only on the basic operation cost but also on the related memory access costs and the language used for the code (e.g., assembler or Fortran.) For more information on general matrix operation costs, see [11].

The one problem per node distribution means that the N sparse system solutions carried out in steps 1 and 2a can be carried out in parallel. Each S_{l+1} is a sparse, symmetric positive definite $m_1 \times m_1$ matrix. We compute the factors of each matrix by the (serial) supernodal Cholesky factorization algorithm of Ng and Peyton [16]. The cost of this algorithm is very

highly problem dependent and is impossible to write as a general analytic expression. However, we know that the total cost is bounded above by

$$T_{factor} \le \delta \frac{m_1^3}{3} \omega,$$

where δ is the fraction of nonzero elements in the Cholesky factor of sparse matrix S_{l+1}. (In determining a cost over all l, we take δ to be the largest value for any Cholesky factor.) Solving the system $S_{l+1}p_{l+1} = b_{l+1}$ in step 1 and each of the n_0 systems $S_{l+1}(u_{l+1})_i = (T_{l+1})_i, i = 1, \ldots n_0$, in step 2a is also done by the algorithm of [16] so that the time for one solve is very roughly bounded above by

$$T_{solve} \le \delta 2 m_1^2 \omega.$$

The operations performed in step 2b are also fully parallel. Forming the matrix $\hat{T}_{l+1} = (T_{l+1}^{\top}(u_{l+1})_1, \ldots, T_{l+1}^{\top}(u_{l+1})_{n_0})$ costs

$$2n_0^2 m_1 \omega,$$

and constructing the vector $\hat{t}_{l+1} = T_{l+1}^{\top} p_{l+1}$ costs

$$2n_0 m_1 \omega.$$

The computation cost of the GpGATHER introduced in section 5.1 is T_{Gp}^{comm}. After GpGATHER completes, node 0 takes another $(2n_0^2 + 2n_0 m_0 + n_0 + m_0)\omega$ to form G_1 and $\hat{p}_2$. Thus, the total computation costs of steps 1, 2a, and 2b are (retaining quadratic and higher order terms)

$$T_{1,2a,2b} = [T_{factor} + (n_0 + 1)T_{solve} + (2n_0^2 m_1 + 2n_0^2 + 2n_0 m_0 + 2n_0 m_1)]\omega + T_{Gp}^{comp}.$$

While steps 1, 2a, and 2b are fully parallel, the same does not hold for the rest of step 2. In particular, once node 0 has formed the $n_0 \times n_0$ matrix G_1 via the call to GpGATHER, it is more efficient for node 0 to carry out the computations involving G_1 (e.g., steps 2c, 2d, and 2e) serially than to distribute them among all nodes. The following deliberations illustrate that step 2d should not be implemented in parallel.

The time needed to run step $2d$ serially is

$$T_{2d} = 2(n_0^2 m_0 + m_0^2 n_0)\omega, \tag{8}$$

where the first term is the cost of solving $G_1 w_i = (A_0^{\top})_i, i = 1, \ldots m_0$, with G_1 factored in step 2c, and the second term is the cost of forming $G_2 = -A_0[w_1, \ldots w_{m_0}]$.

A parallel implementation of step $2d$ would proceed as follows (with m_0 divisible by p):

- Processor 0 sends a copy of matrix G_1 to every node. Doing this by a spanning tree broadcast takes time $d[\beta + 8n_0^2\tau]$.
- Processor l solves $\frac{m_0}{p}$ of the systems needed to form G_2 and multiplies the solution vectors by $-A_0$ in parallel on processors $l = 0, \ldots, p-1$. The cost of this step is $2(n_0^2 m_0 + n_0 m_0^2)\frac{\omega}{p}$.
- Processor 0 gathers back all the columns $(-A_0 w)_i, i = 1, \ldots m_0$, to form G_2. As we did for dyGATHER, we use an ADE, and the total time is $2[d\beta + 8(2^d - 1)\frac{n_0 m_0}{p}\tau]$.

The total time needed to run the parallel implementation of step 2d is then

$$T_{2d}^{parallel} = 3d\beta + 8[dn_0^2 + 2(2^d - 1)\frac{n_0 m_0}{p}]\tau + 2(n_0^2 m_0 + n_0 m_0^2)\frac{\omega}{p}. \tag{9}$$

For the problems of interest, n_0 and m_0 are small – usually less than 100. Small matrix orders mean that the communication time can quickly overwhelm the small computation time in the parallel algorithm. For example, suppose that $m_0 = 50$ and $n_0 = 100$. On the iPSC/860, $\beta \approx 75$ μsec for small messages, $\tau \approx 0.4$ μsec, and $\omega \approx 0.1$ μsec (ignoring memory access costs) [10]. Therefore, the serial algorithm would take at least 150 msec, while the parallel algorithm would take at least 143 msec when $d = 3$, 168 msec when $d = 4$, 196 msec when $d = 5$, and even greater times for larger machine sizes as the cost of the ADE increases with increasing machine dimension. When n_0 or m_0 is smaller, the efficiency of the parallel algorithm is even poorer. Thus, the parallel implementation of step 2d is never faster than the serial implementation for practical problems on reasonably sized machines. For instance, the test problems we examine in Section 6.4 have $m_0 = 10$ and $n_0 = 70$ or $m_0 = 1$ and $n_0 = 90$.

Similar arguments show that steps 2c and 2e, which involve only small order matrix computations, cannot be distributed efficiently. The total cost for step 2c is $\frac{1}{3}n_0^3\omega$ to factor the dense, symmetric, positive definite matrix G_1, $2n_0^2\omega$ to solve the system $G_1 u = \hat{p}_1$, and $(2m_0 n_0 + m_0)\omega$ to form v. The computation cost for step 2e is $(\frac{2}{3}m_0^3 + 2m_0^2)\omega$ to factor and solve the system $G_2 q_2$, and $2n_0(n_0 + m_0 + \frac{1}{2})\omega$ to form the righthand side of and solve the second system. The computation time for steps 2c through 2e is then

$$T_{2c,2d,2e} = (\frac{1}{3}n_0^3 + \frac{2}{3}m_0^3 + 2m_0^2 n_0 + 2n_0^2 m_0 + 4m_0^2 + 3n_0^2 + 4m_0 n_0)\omega.$$

Steps 3 and 4 are again completely parallel. Node 0 determines r_0 in time $m_0(2n_0 + 1)\omega$. All nodes find the matrix vector product $T_{l+1}q_{l+1}$ in time $m_1 n_0 \omega$, and all nodes solve their sparse systems in time bounded by $2\delta m_1^2\omega$. (Remember that the factors of S_{l+1} were computed in step 1.) In the final step, all nodes take time $m_1\omega$ to determine $(dy)_l$. Steps 3 and 4 thus run in time

$$T_{3,4} = 2[m_0 n_0 + m_1(n_0 + \delta m_1)]\omega.$$

Totalling the contributions of all steps gives a total computation time of

$$\begin{aligned} T_{comp}^{N=p} &= T_{1,2a,2b} + T_{2c,2d,2e} + T_{3,4} \\ &= T_{factor} + (n_0 + 1)T_{solve} + n_0^2 m_1 + \frac{1}{3}n_0^3 + \frac{2}{3}m_0^3 \\ &+ m_0^2 n_0 + n_0^2 m_0 + 2m_0^2 + 4n_0^2 + 3m_0 n_0 + 2m_1 n_0]\omega + T_{Gp}^{comp}. \end{aligned}$$

When each processor holds not just one submatrix but rather $\frac{N}{p}$ of them, the costs of the steps increase to

$$\begin{aligned} T_{1,2a,2b} &= [T_{factor} + (n_0 + 1)T_{solve} + 2n_0^2 m_1]\frac{N}{p}\omega + T_{Gp}^{comp} \\ T_{2c,2d,2e} &= (\frac{1}{3}n_0^3 + \frac{2}{3}m_0^3 + 2m_0^2 n_0 + 2n_0^2 m_0 + 4m_0^2 + 3n_0^2 + 4m_0 n_0)\omega \\ T_{3,4} &= 2[m_0 n_0 + m_1(n_0 + \delta m_1)]\frac{N}{p}\omega. \end{aligned}$$

This means that the total computational cost for the parallel algorithm is (to cubic order)

$$T_{comp} = \{[T_{factor} + T_{solve} + 2n_0m_1^2 + 2n_0^2m_1]\frac{N}{p} + (\frac{1}{3}n_0^3 + \frac{2}{3}m_0^3 + 2m_0^2n_0 + 2n_0^2m_0)\}\omega + T_{Gp}^{comp} \quad (10)$$

5.3 The parallel algorithm

For simplicity of presentation, we assume that the number of processors p equals the number of subproblems N so that the parallel algorithm proceeds as follows. (Our code does handle arbitrary problem sizes by spreading leftover tasks evenly among the processors.)

Procedure 5.1 (Parallel Finddy($S, A_0, T_1, \ldots, T_N, b, dy$))
Begin with the following data distribution:

Node l holds S_{l+1}, T_{l+1}, and b_{l+1}, $l = 0, \ldots, N-1$.

Node 0 also holds A_0, S_0, D_0, and b_0.

1. *(In parallel, solve $Sp = b$). On node zero, solve $S_0p_0 = b_0$. In parallel (on nodes $l = 0, \ldots, N-1$), solve $S_{l+1}p_{l+1} = b_{l+1}$ for p_{l+1}.*
2. *(Solve $Gq = V^\top p$).*
   (a) *In parallel on nodes $l = 0, \ldots, N-1$, solve $S_{l+1}(u_{l+1})_i = (T_{l+1})_i$ for $(u_{l+1})_i$, $i = 1, \ldots, n_0$.*
   (b) *In parallel on nodes $l = 0, \ldots, N-1$), multiply $T_{l+1}^\top p_{l+1}$.*
   Serially (on node 0), compute $\hat{p}_2$.
   Call GpGATHER to form G_1 and $\hat{p}_1$ on node 0. Node 0 (serially) combines its data with that accumulated by GpGATHER to form $G_1 = (D_0)_{ii}^{-2} + (A_0^T A_0) + \sum_{l=1}^N T_l^T (u_l)_i$ and $\hat{p}_1 = A_0^T p_0 + \sum_{l=1}^N T_l^T p_l$.
   (c) *Serially (on node 0), solve $G_1 u = \hat{p}_1$ for u and set $v = \hat{p}_2 + A_0 u$.*
   (d) *Serially (on node 0), form G_2 by solving $(G_1)w_i = (A_0^\top)_i$ for w_i, for $i = 1, \ldots, m_0$ and setting $G_2 = -A_0[w_1, \ldots, w_{m_0}]$.*
   (e) *Serially (on node 0), solve $G_2q_2 = -v$ for q_2, and solve $G_1q_1 = \hat{p}_1 - A_0^\top q_2$ for q_1. Call STB to distribute q_1 to all nodes.*
3. *(In parallel, solve $Sr = Uq$). On node 0, set $r_0 = A_0q_1 + q_2$. On nodes $l = 0, \ldots, N-1$, solve $S_{l+1}r_{l+1} = T_{l+1}q_{l+1}$ for $r_{l+1} \in \mathcal{R}^{m_{l+1}}$.*
4. *(In parallel, form dy.) On node zero, set $dy_0 = p_0 - r_0$. On nodes $l = 0, \ldots, N-1$, set $(dy)_{l+1} = p_{l+1} - r_{l+1}$. Call dyGATHER to gather the vector $dy = (dy_0, \ldots, dy_N)$ on node 0.*

In steps 1 and 2a of our Fortran code, we use the memory-efficient sparse system solver routines SUPFCT and SUPSLV developed by Ng and Peyton [16]. We use the LAPACK routines DPOTRF and DPOTRS to factor and solve the dense linear systems involving the symmetric positive definite matrix G_1, and we use the LAPACK routines DGETRF and DGETRS to factor and solve the dense linear systems involving the matrix G_2 [1]. Throughout the code, we use assembler BLAS routines wherever appropriate [9].

For all tested problems, the solution computed in double precision had a residual error of no more than 10^{-13}.

5.4 Portability of the code

In this paper, we have examined only a version of this code geared expressly toward the hypercube multiprocessor. However, the modular structure of the code makes it straightforward to port to any other machine suitable for medium- to large-grained parallel tasks. For example, the nearest neighbor communication schemes used in this code translate easily into contention free broadcast or gather routines for a two-dimensional mesh. (See, for example, [2, 7].)

6 Timing and Speedup Analysis

In this section, we examine the efficiency of our parallel algorithm and its implementation. For this purpose, we employ the expressions for the theoretical communication and computation costs summarized in equations (7) and (11) as well as data from experiments on random matrices. We begin with a desciption of the test problems in Section 6.1. In Section 6.2, we examine the validity of the communication and computation models. While the communication is easy to model nearly exactly, it is very difficult to devise an accurate computation model of an optimized code. Thus, we show how the analytic model and the experimental results can be combined to produce a combination analytical and empirical model that accurately reflects the behavior of our algorithm for our test problems on the target machine. In Section 6.3, we present and interpret our experimental results. In Section 6.4, we apply our implementation to solve very large real-world problems.

6.1 The random test problems

The test matrices used in this section were generated with the stochastic programming test problem generator GENSLP of Kall and Keller [12]. The precise dimensions of each test problem are reported below together with the computational results. The entries of the matrices A_0 and W_l are uniformly distributed in the range [-112.0, 128.0]. Two percent of the entries of these matrices are non-zero while the matrices T_l are dense. Details of the procedure used to generate the test problems, once the user parameters are specified, are given in [12]. Since the test problems are randomly generated we expected some variation in solution time when solving two problems with identical parameters. To eliminate any noise from our observations we generated 5 test problems for each experiment and report the average solution time. The variation in solution time over each set of five problems was never greater than 1%.

We used a total of 44 random test matrices. The matrix orders are provided in Section 6.3.

6.2 Validation of the models

The total communication cost of the parallel algorithm was modelled in Section 5.1. To validate this communication model, we computed the relative error in the model values using the given matrix dimensions and the machine parameters $\beta = 136$ (long message startup time) and $\tau = 0.4$ (byte transfer time) from [10]. This error is defined by

$$E = (T^{obs}_{comm} - T_{comm})/T^{obs}_{comm},$$

where T_{comm} is defined in equation (7) and T^{obs}_{comm} is the measured communication time on the hypercube. For all matrices tested, the magnitude of the relative error E was no greater than 4.5%.

Validating the computation model is more difficult. The time to perform a single floating point operation is $\omega_{i860} = 0.1$ μsec in a test program written in a high-level programming language and incurring no cache misses [10]. This value is reduced when operations can be pipelined. The minimum pipelined operation time is about 3 cycles or 0.06 μsec on the 50 MHz i860 processor [8]. Our code, however, includes assembler routines (the BLAS), and our test problems are large enough that they do not always fit into cache. Furthermore, the times for the sparse matrix routines SUPFCT and SUPSLV are strongly problem dependent [16]. Thus, the proper value of ω to use for the floating point operation time must be adjusted throughout the algorithm to account for language used, problem size, and the context in which the operation is performed. We now adapt the total computation cost of equation (11) to more accurately reflect the true cost of the algorithm.

The major cost of the algorithm is incurred in steps 1, 2a, and 2b which contribute the first term of equation (11). The first part of this term, $(T_{factor} + n_0 T_{solve})\frac{N}{p}\omega$, represents the factoring and solving of the $\frac{N}{p}$ sparse submatrices S_l on each node and then solving $n_0 + 1$ systems using those factors. In our test problems, all of the $\frac{N}{p}$ sparse submatrices are identical, meaning that this term reduces to about $T_{factor} + (n_0 T_{solve})\frac{N}{p}\omega$, and this term, in turn, is bounded by

$$T_{factor} + (n_0 T_{solve})\frac{N}{p}\omega \leq \delta(\frac{1}{3}m_1^3 + 2n_0 m_1^2 \frac{N}{p})\omega. \qquad (11)$$

The tightness of this bound is strongly dependent on both the matrix order and the number of nonzero matrix elements. For all of our test matrices, the sparse Cholesky factors of the sparse submatrices S_l have about half of their elements as zeros. In particular, $\delta \approx 0.46$ for all test problems. Thus, we concentrate only on the matrix size to determine a reasonable value for the time ω_{sparse} for floating point operations performed during the sparse matrix operations. With increased matrix order comes increased memory access time, and increased memory access time translates into an increased value of ω_{sparse}.

The sizes of our test problems are such that they are contained fully within the 8 Mbytes of main memory on each node (i.e., we do not use the CFS node I/O disk). Furthermore, a fill parameter $\delta = 0.46$ indicates that sparse systems of order up to about 66 can be solved wholly within the cache. Our timings of Parallel Finddy roughly corroborate this prediction: the best value of ω_{sparse} for tested values of $m1 \leq 50$ is 0.1 μsec while a value of 0.5 μsec is better for $m1 \geq 100$. That is, the cost per floating point operation is about 5 times greater for systems requiring main memory access than for those incurring no cache misses.

Steps 1, 2a, and 2b actually require that $(n_0 + 1)$ $m_1 \times m_1$ systems be solved for each of the $\frac{N}{p}$ problems on each node. When all systems can fit in cache simultaneously (for example, when $n_0 = m_0 = 10$, ω_{sparse} drops further to around .05 μsec.) The values of ω_{sparse} less than the basic operation time ω_{i860} can be accounted for by some combination of the roughness of the bound in equation (11) and by the pipelining of operations. To summarize, for our tests,

$$\omega_{sparse} = \begin{cases} 0.10 & m_1 < 66; \\ 0.50 & m_1 \geq 66; \\ 0.05 & \text{if } n_0 m_1^3 < 1000 \text{ regardless of } m_1. \end{cases} \tag{12}$$

The version of the triangular solver SUPSLV we used for these experiments allows us to solve only one system at a time. We are presently working on a variant of the code to handle multiple righthand sides. Such code would allow more system solves for each cache load and so would lower the overall cost of the algorithm (and with it the value of ω_{sparse}.)

The remaining terms in equation (11) result either from calls to BLAS routines or calls to LAPACK routines. The latter also do most of their computation via calls to BLAS routines. For all of our experiments, we linked our codes not with the Fortran BLAS but rather with the optimized assembler BLAS provided by the manufacturer. As intended, the assembler BLAS are more efficient than the compiled Fortran BLAS, and the speed advantage of the assembler BLAS increases with increasing problem size even though the larger problems may incur larger memory access costs. In particular, we find by studying our timings that when the BLAS are called with matrix or vector arguments of order m_0 or n_0, the floating point operation time is well-approximated by

$$\omega_{assem} = \begin{cases} 0.67 & m_0, n_0 \leq 50; \\ 0.50 & 50 < m_0, n_0 \leq 100; \\ 0.25 & 100 < m_0, n_0 \leq 200. \end{cases} \tag{13}$$

The total computation time then becomes

$$\begin{aligned} T^{\omega}_{comp} &= \delta(\frac{m_1^3}{3} + 2n_0 m_1^3 \frac{N}{p})\omega_{sparse} \\ &+ (2n_0 m_1^2 + 2n_0^2 m_1 \frac{N}{p} + \frac{1}{3}n_0^3 + \frac{2}{3}m_0^3 + 2m_0^2 n_0 + 2n_0^2 m_0)\omega_{assem} \\ &+ T^{comp}_{Gp}(\omega_{assem}/\omega_{i860}), \end{aligned} \tag{14}$$

and the maximum relative error (obtained by using the approximations for ω_{sparse} and ω_{assem} from equations 12 and 13, respectively) is

$$E_{comp} = (T^{obs}_{comp} - T^{\omega}_{comp})/T^{obs}_{comp}.$$

The largest value of E_{comp} for our test problems was about 13%.

Notice, however, that the granularity of the approximations ω_{sparse} and ω_{assem} reflect the granularity of the test matrix orders. In particular, they were derived only for the test matrix orders $m_1 = 10, 20, 50, 100, 500$ and $n0, m0 = 20, 50, 100, 200$. Furthermore, all tested matrices had fill factor $\delta \approx 0.46$. We therefore cannot expect our compuational model to be accurate for every test problem. We instead employ it and the accurate communication model as a general framework for interpretting our experimental results.

6.3 Experimental results

In this section we study how the *aspect ratio* of the constraint matrix (i.e., ratio of the row and column dimensions of the submatrices) affects the performance of Procedure 5.1 (Parallel Finddy). We do this via three sets of experiments using randomly generated test problems with user specified dimensions. (We assume throughout that submatrices corresponding to distinct scenarios, i.e., T_l, W_l, have identical sizes and that all submatrices S_l are not only identical in size but also in element values.) For all experiments, the dimension $n_1 = 1000$.

In the first experiment we fix the submatrix orders (i.e., fix n_0, m_0, and m_1) and vary the number of scenarios N. In the second experiment we fix the number of scenarios N and the submatrix dimensions m_0 and n_0 but change the number of second-stage constraints m_1. For the third experiment we fix the number of scenarios N and m_1, and vary the number of first-stage variables $m_0 = n_0$.

For each experiment we report speedup results for hypercubes with 2 to 16 nodes. These results are evaluated with reference to the theoretical communication and computation costs of equations (7) and (15). The empirical results are used to draw conclusions about the problem sizes and aspect ratios for which best efficiencies can be achieved.

6.3.1 Experiment I: Varying the number of scenarios N

For the first group of experiments, we use the fixed parameters $m_0 = 100$, $n_0 = 100$, and $m_1 = 100$ and vary the parameter N, the number of scenarios in the second stage.

The total time to run the parallel algorithm on p nodes can be expressed as the equation

$$T_p = T_{N/p} + T_{serial} + T_{comm},$$

where the total time to run the algorithm on one node is

$$T_1 = pT_{N/p} + T_{serial}.$$

That is, $pT_{N/P}$ is the time for the fully parallelizable part of the one processor algorithm, and T_{serial} is the time for the part that cannot be implemented in parallel.

With this expression, the speedup can be written $S_p = T_1/T_p$ and the efficiency as $E_p = S_p/p = T_1/(pT_p)$. Perfect speedup is then p while perfect efficiency is 1.0 or 100%. The time T_{comm} is accurately represented by equation (7). The total computation time on one node $T_{N/p} + T_{serial}$ is roughly approximated by equation (15) for some choices of ω_{sparse} and ω_{assem}. These equations suggest and our expermental results confirm that T_{serial} is negligible in comparison to $T_{N/p}$ for the parameters used in this experiment. Thus, the attainable speedup and efficiency is most influenced by the relative values of $T_{N/p}$ and T_{comm}.

We first examine the effects of communication. The large matrix orders used for these experiments mean that data messages are always long. In particular, data messages always exceed both the 100 byte cutoff for fast message transfer on the iPSC/860 and the 336 byte (42 double precision number) point at which the message transfer time (336τ) becomes greater than the message startup time β on that machine. The expression for T_{comm} shows that the transfer time is linearly dependent on the number of scenarios N. In particular, it is linearly dependent on the number of scenarios per processor N/p.

Table 1 shows how the efficiency attained is related to the percentage of total time spent in communication (i.e., T_{comm}/T_p) for all matrices in this test set. Note that a 4% or less

N	p	T_{comm}	T_{comm}/T_p	E_p
16	2	39	0.8%	0.97
	4	74	2.9%	0.90
	8	110	7.5%	0.80
	16	151	16.0%	0.61
64	2	54	0.3%	0.99
	4	98	1.0%	0.97
	8	136	2.8%	0.93
	16	176	6.6%	0.85
128	2	74	0.2%	1.00
	4	128	0.7%	0.98
	8	171	1.8%	0.96
	16	214	4.3%	0.91
256	2	114	0.2%	1.00
	4	189	0.5%	0.99
	8	240	1.3%	0.98
	16	288	3.0%	0.95

Table 1: Communication times for problems of varying N with $n0 = m0 = m1 = 100$ (in msecs).

communication time corresponds to a 90% or better efficiency. The percentage of time spent in communication grows as more processors are applied to a given matrix. This suggests the obvious: with all else equal, it is best to maximize N/p, the number of scenarios per processor. The increase in computation time outweighs the increase in communication time.

6.3.2 Experiment II: Varying the number of second-stage constraints

For the second set of experiments, we fix the parameters $N = 128$, $m_0 = 100$, and $n_0 = 100$ and vary $m_1 = 100$, the number of second-stage constraints.

The communication time T_{comm} varies linearly with m_1. That this linear variation mimics the linear variation with N is evident in a comparison of Tables 1 and 2. (Compare, for example, the percentages of time spent in communication for $N = 128$ and $m_1 = 100$.) The two tables also show a similar interdependence of the percentage of time spent in communication and the efficiency. Compare, for example, the two entries showing $T_{comm}/T_p = 4.3\%$ and $E_p = 91\%$.

As m_1 decreases, however, the effect of communication time grows. Consider the two table entries showing ratios of 2.8%. At the Table 1 entry, $m_0 = n_0 = 100$, eight scenarios are supplied to each processor, and the efficiency is 91%. At the Table 2 entry, $m_0 = n_0 = 100$, 64 scenarios are supplied to each processor, but the efficiency is only 88%. Because the fraction of time spent in communication is the same for both entries, the efficiency drop is attributed to a relative increase in the time for serial computation T_{serial}. Indeed, examination of equation (11) shows that when m_1 shrinks in relation to to n_0, the relative contribution of T_{serial} (largely the dense $n_0 \times n_0$ system solutions) is greater.

Thus, all else being equal, the second set of experiments shows that the ratio n_0/m_1 should be kept as small as possible in order to attain best efficiency.

m_1	p	T_{comm}	T_{comm}/T_p	E_p
10	2	38	2.8%	0.88
	4	73	8.8%	0.70
	8	108	17.8%	0.49
	16	146	28.6%	0.29
20	2	44	2.0%	0.93
	4	83	6.5%	0.80
	8	119	14.3%	0.61
	16	158	25.1%	0.41
50	2	54	1.1%	0.97
	4	98	3.7%	0.90
	8	138	8.9%	0.78
	16	176	17.8%	0.60
100	2	74	0.2%	1.00
	4	127	0.7%	0.98
	8	174	1.8%	0.96
	16	217	4.3%	0.91
500	2	237	0.03%	1.00
	4	377	0.1%	1.00
	8	453	0.2%	1.00
	16	516	0.5%	0.88

Table 2: Communication times for problems of varying m_1 with $n0 = m0 = 100$ and $N = 128$ (in msecs).

n_0	p	T_{comm}	T_{comm}/T_p	E_p
20	2	81	0.6%	0.99
	4	124	1.7%	0.98
	8	144	3.9%	0.95
	16	156	8.0%	0.91
50	2	90	0.3%	1.00
	4	140	0.8%	0.99
	8	168	1.9%	0.98
	16	188	4.1%	0.95
100	2	114	0.2%	1.00
	4	189	0.5%	0.99
	8	240	1.2%	0.98
	16	288	3.0%	0.95
200	2	209	0.1%	0.99
	4	379	0.5%	0.98
	8	526	1.3%	0.96
	16	667	3.1%	0.91

Table 3: Communication times for problems varying $m_0 = n_0$ with $m_1 = 100$ and $N = 256$ (in msecs).

6.3.3 Experiment III: Varying the number of first-stage variables

In this group of experiments, we fix the parameters $N = 256$ and $m_1 = 100$ and vary $m_0 = n_0 = 100$, the number of first-stage constraints.

Table 3 shows the efficiencies attained. These results largely reiterate the results of Experiment II: the smaller n_0/m_1 is, the easier it is to get good efficiency. This is most pronounced for $n_0 = m_0 = 20$ and $p = 16$ so that $n_0/m_1 = 0.2$. In this case, a communication percentage of 8 still allows an efficiency of 91%. When n_0 is as small as 10, the serial time T_{serial} becomes nearly negligible in comparison to the other time requirements.

While T_{comm} is linearly dependent on N and m_1 (in fact, on the product Nm_1), it is quadratically dependent on n_0. The difference is most evident in the distributions of times among the different communication tasks. Tables 4 and 5 show how the times for the three communication routines change with changing $n0$ and N, respectively. (A table for changing m_1 closely resembles Table 5.) The most pronounced change is the increased fraction of time spent in GpGATHER with increasing n_0. This corresponds to the building of a larger serial problem and, hence, to an increased T_{serial}. Again, the data points to the value of a small dimension n_0.

6.4 Solving large scale problems

As a final step, we applied the Parallel Finddy procedure to solve two sets of test problems arising in real-world applications. One is the SCSD8 set, a stochastic version of a model to find the minimal design of a multistage truss [3]. The second is SEN, a telecommunication network design problem communicated to us by Suvrajeet Sen from University of Arizona. SCSD8 has dimensions $m_0 = 10, n_0 = 70, m_1 = 20, n_1 = 140$. SEN has dimensions $m_0 = 1, n_0 = 90, m_1 = 175, n_1 = 795$.

$n0$	p	GpGATHER	STB	dyGATHER	Total Commun. Time
20	2	1	0	80	81
	4	3	0	121	124
	8	5	1	138	144
	16	7	1	148	156
50	2	10	0	80	90
	4	19	0	121	140
	8	29	1	138	168
	16	38	2	148	188
100	2	34	0	80	114
	4	67	1	121	189
	8	100	2	138	240
	16	137	3	148	288
200	2	129	0	80	209
	4	257	1	121	379
	8	386	2	138	526
	16	515	4	148	667

Table 4: Communication time breakdown for problems of varying n0 (in msecs.)

$n0$	p	GpGATHER	STB	dyGATHER	Total Commun. Time
16	2	34	0	5	39
	4	66	1	7	74
	8	100	2	8	110
	16	137	3	9	151
64	2	34	0	20	54
	4	67	1	30	98
	8	100	2	34	136
	16	137	3	36	176
128	2	34	0	40	74
	4	67	1	60	128
	8	100	2	69	171
	16	137	3	74	214
256	2	34	0	80	114
	4	67	1	121	189
	8	100	2	138	240
	16	137	3	148	288

Table 5: Communication time breakdown for problems of varying N (in msecs.)

Problem	Scenarios	Rows of A	Columns of A
SCSD8.16	16	330	2,310
SCSD8.64	64	1,290	9,030
SCSD8.256	256	5,130	35,910
SCSD8.512	512	10,250	71,750
SCSD8.2048	2048	40,970	286,790
SEN.16	16	2,801	12,810
SEN.64	64	11,201	50,970
SEN.256	256	44,801	203,610
SEN.512	512	89,601	407,130

Table 6: Characteristics of the large-scale, real-world test problems.

Problem	iPSC/860 nodes	Solution time (sec)
SCSD8.16	16	0.144
SCSD8.64	16	0.228
SCSD8.256	16	0.572
SCSD8.512	16	1.031
SCSD8.2048	16	3.820
SEN.16	16	0.970
SEN.64	64	1.043
SEN.256	128	2.077
SEN.512	128	3.929

Table 7: Solution time for the large-scale, real-world test problems.

The problems in each set have a user-specified number of scenarios. Thus it is possible to generate extremely large test problems by increasing the number of scenarios. Table 6 summarizes the characteristics of the test problems. To put the size of these problems in perspective we mention that Birge and Holmes [5] solved SCSD8 with only 32 scenarios. To the best of our knowledge the largest SEN problem solved previous to this paper has 100 scenarios.

Figure 1 illustrates the speedups achieved when solving the SCSD8 set on an iPSC/860 with up to 128 nodes. Figure 2 illustrates the speedup achieved when solving the SEN set on an iPSC/860 with up to 128 nodes. Table 7 summarizes the solution times. For cube sizes up to 16 nodes, the speedup curves quickly level off at $S_p \approx p$. That is, near perfect speedup is achieved for most matrix orders. For the larger cube sizes, the number of scenarios per processor N/p in our tests is insufficient to provide better than aobut 70% efficiency for the largest SCSD8 problem and 79% efficiency for the largest SEN problem on 128 nodes. (Note that the better efficiency does occur for the test problem with the smaller ratio of n_0 to m_1.) However, the 128-node plots do indicate an increasing trend of speedup versus matrix order even for the largest matrix order tested. Moreover, these largest matrix orders exceed the largest order solved by other means to date.

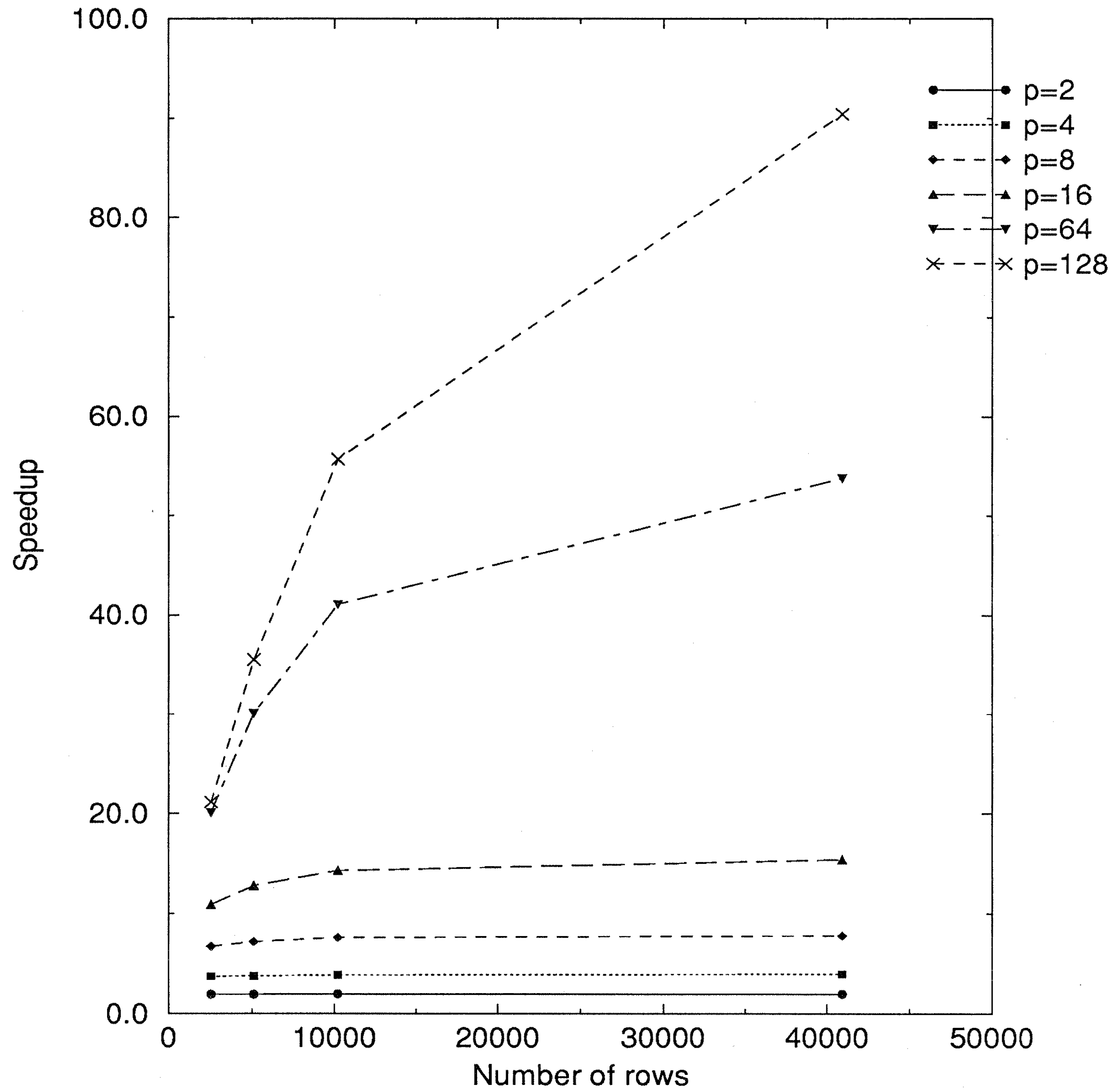


Figure 1: Speedups with the solution of the SCSD8 test problems on an iPSC/860 with up to 128 nodes.

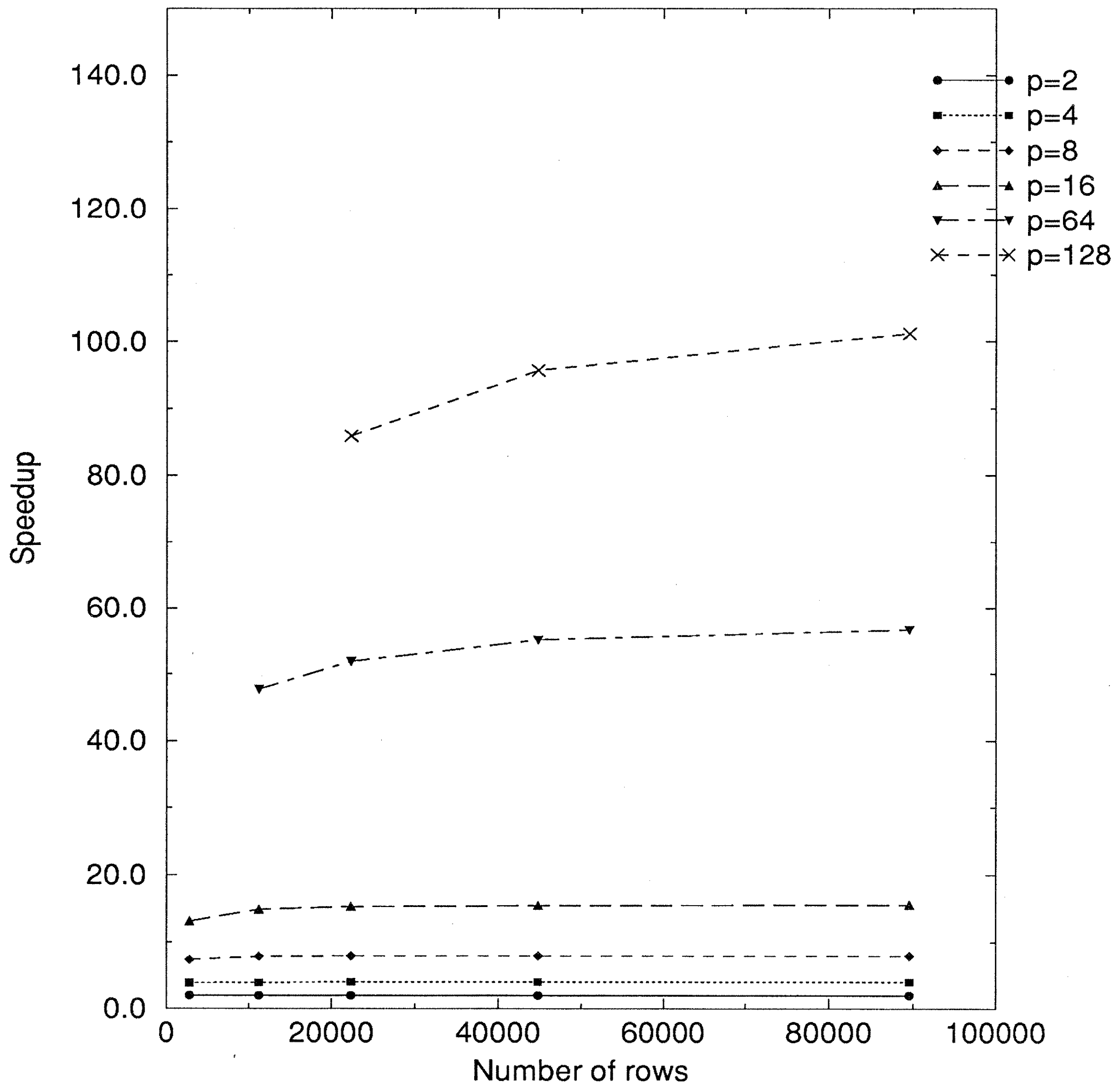


Figure 2: Speedups with the solution of the SEN test problems on an iPSC/860 with up to 128 nodes.

7 Conclusions

Birge and Holmes have shown that the BQ matrix factorization procedure is a fast and accurate method for solving large-scale stochastic programming problems. We have shown in this paper that this method can achieve excellent speedups on a distributed-memory multiprocessor (like the Intel iPSC/860). These speedups are achieved with the development of efficient communication schemes that have been adapted to the needs of the BQ algorithm.

With a careful implementation of the algorithm — using memory efficient and fast matrix factorization routines — we have been able to obtain an implementation that is robust and modular. We have also shown why the features of this implementation that make it efficient also stand in the way of a straightforward analytical model of the computation time. However, by using the approximate model together with the experimental data, we have shown that very good speedups can be expected for a wide range of problem structures. In particular, good speedups result when the the number of first-stage constraints (n_0) is small in comparison to the number of second-stage constraints (m_1) for a sufficiently large number of scenarios (N/p).

The test problems solved in this paper are, to the best of our knowledge, some of the largest problems reported in the literature. An interesting question that deserves further work is to investigate the efficiency of a parallel implementation of this algorithm on even larger scale machines or other architectures, like the mesh of the Paragon machine or the fat-tree of the Connection Machine CM–5.

Acknowledgements:

The authors wish to thank E. Ng and B. Peyton for providing their codes along with much helpful information about their use. The experiments described in this paper were performed on the iPSC/860 machine at Oak Ridge National Laboratory.

The research of Elizabeth R. Jessup was funded by DOE contract DE-FG02-92ER25122 and by an NSF National Young Investigator Award. The research of Dafeng Yang and Stavros A. Zenios was supported in part by NSF grant CCR–91–04042. This work was completed while Zenios was visiting the Universities of Urbino and Bergamo, Italy, under a fellowship from the GNAFA and GNIM groups of Consiglio Nazionale delle Richerche (CNR). He is currently a visiting Professor of Management Science, Department of Public and Business Administration, University of Cyprus, Nicosia, Cyprus.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, and S. Hammarling. *LAPACK User's Guide.* SIAM, 1992.

[2] M. Barnett, D.G. Payne, and R. van de Geijn. Optimal broadcasting in mesh-connected architectures. Technical Report TR-91-38, Dept. of Computer Science, University of Texas at Austin, 1991.

[3] J.R. Birge. Decomposition and partitioning methods for multistage stochastic linear programs. *Operations Research*, 33:989–1007, 1985.

[4] J.R. Birge and L. Qi. Computing block-angular Karmarkar projections with applications to stochastic programming. *Management Science*, 34(12):1472–1479, Dec. 1988.

[5] J.R. Birge and D.F. Holmes. Efficient solution of two-stage stochastic linear programs using interior point methods. *Computational Optimization and Applications*, 1:245– 276, 1992.

[6] I. Choi and D. Goldfarb. Solving multicommodity network flow problems by an interior point method. In T. Coleman and Y. Li, editors, *Large Scale Numerical Optimization*, pages 58–69. SIAM, 1990.

[7] S. Crivelli and E.R. Jessup. Optimal eigenvalue computation on distributed-memory mimd multiprocessors. To appear in Parallel Computing (pending revision).

[8] R. Dewar and M. Smosna. *Microprocessors: A Programmer's Point of View.* McGraw-Hill, 1990.

[9] J.J. Dongarra, J. Du Croz, Iain Duff, and Sven Hammarling. A set of level 3 basic linear algebra subroutines. Preprint no. 2, Argonne National Laboratory, 1988.

[10] T.H. Dunigan. Performance of the Intel iPSC/860 hypercube. Technical Report ORNL/TM-11491, Oak Ridge National Laboratory, 1990.

[11] G.H. Golub and C.F. Van Loan. *Matrix Computations.* The Johns Hopkins Press, Baltimore, MD, 2nd edition, 1989.

[12] P. Kall, and E. Keller. GENSLP: A program for generating input for stochastic linear programs with complete fixed recourse. Manuscript, Institute for Operations Research der University Zurich, Zurich CH-8006, Switzerland, 1985.

[13] I.J. Lustig, J.M. Mulvey, and T.J. Carpenter. Formulating two-stage stochastic programs for interior point methods. *Operations Research*, 39:757–770, 1991.

[14] J.M. Mulvey and A. Ruszczynski. A diagonal quadratic approximation method for large scale linear programs. *Operations Research Letters*, 12:205–215, 1992.

[15] J.M. Mulvey and H. Vladimirou. Stochastic network programming for financial planning problems. *Management Science*, 38:1643–1664, 1992.

[16] E. Ng and B. Peyton. A supernodal Cholesky factorization algortihm for shared-memory multiprocessors. *SIAM J. Sci. Comput.*, 14:761–769, 1993.

[17] S. Nielsen and S.A. Zenios. A massively parallel algorithm for nonlinear stochastic network problems. *Operations Research*, 41(2):319–337, 1993.

[18] R.T. Rockafellar and R.J.-B. Wets. Scenarios and policy aggregation in optimization under uncertainty. *Mathematics of Operations Research*, 16:119–147, 1991.

[19] Y. Saad and M.H. Schultz. Data communication in hypercubes. Research Report 428, Dept Computer Science, Yale University, 1985.

[20] S. Seidel, M.-H. Lee, and S. Fotedar. Concurrent bidirectional communication on the Intel iPSC/860 and iPSC/2. Technical Report CS-TR 90-06, Dept. Computer Science, Michigan Technological University, 1990.

[21] R. J. B. Wets. Stochastic programs with fixed resources: the equivalent deterministic problem. *SIAM Review*, 16:309–339, 1974.

[22] S.A. Zenios. A model for portfolio management with mortgage-backed securities. *Annals of Operations Research*, 43, 1993.