Optimization Methods and Software
Vol. 00, No. 00, Month 200x, 1–20

RESEARCH ARTICLE

On the parallel solution of dense saddle-point linear systems arising in stochastic programming

Miles Lubin, Cosmin G. Petra* and Mihai Anitescu

Mathematics and Computer Science Division, Argonne National Laboratory, IL, USA

(*released October 15, 2010*)

We present a novel approach for solving dense saddle-point linear systems in a distributed-memory environment. This work is motivated by an application in stochastic optimization problems with recourse, but the proposed approach can be used for a large family of dense saddle-point systems, in particular those arising in convex programming. Although stochastic optimization problems have many important applications, they can present serious computational difficulties. In particular, sample average approximation (SAA) problems with a large number of samples are often too big to solve on a single shared-memory system. Recent work has developed interior point methods and specialized linear algebra to solve these problems in parallel, using a scenario-based decomposition that distributes the data and work across computational nodes. Even for sparse SAA problems, the decomposition produces a dense and possibly very large saddle-point linear system that must be solved repeatedly. We developed a specialized parallel factorization procedure for these systems, together with a streamlined method for assembling the distributed dense matrix. Strong scaling tests indicate over 90% efficiency on 1,024 cores on a stochastic unit commitment problem with 57 million variables. Stochastic unit commitment problems with up to 189 million variables are solved efficiently on up to 2,048 cores.

Keywords: stochastic programming, parallel computing, parallel dense linear algebra, saddle-point

AMS Subject Classification: 90C15; 65F05; 68W10

1. Introduction

In this paper we consider two-stage stochastic convex problems with recourse of the form

$$\min \left(\frac{1}{2}x_0^T Q_0 x_0 + c_0^T x_0\right) + \mathbb{E}[G(x_0,\xi)] \;\text{ subject to }\; T_0 x_0 = b_0,\; x_0 \geq 0, \tag{1}$$

where, for a given realization $\tilde{\xi}$ of the random vector ξ, the recourse function $G(x_0,\tilde{\xi})$ is the optimal value of the second-stage problem (2) parameterized by the realization $\tilde{\xi}$. The expectation $\mathbb{E}[\cdot]$ is taken with respect to the density function of ξ. The matrix Q_0 is symmetric positive definite, and the matrix T_0 has full rank. The second-stage problem is a convex quadratic programming problem of the form

$$\min \frac{1}{2}y^T Q y + c^T y \;\text{ subject to }\; Wy = b - Tx_0,\; y \geq 0. \tag{2}$$

*Corresponding author. Email: petra@mcs.anl.gov

ISSN: 1055-6788 print/ISSN 1029-4937 online
© 200x Taylor & Francis
DOI: 10.1080/03081080xxxxxxxxx
http://www.informaworld.com

The problem is parameterized by ξ in the sense that the random entries of the data (Q, c, T, W) form the random vector ξ. We assume that Q is symmetric positive semidefinite, and that the technology matrix T and recourse matrix W have full rank for any realization of ξ.

The convexity of the second-stage quadratic problem implies that the recourse function is convex [3]. Also, the recourse function $G(x_0, \tilde{\xi})$ is nonlinear in general. Therefore, problem (1) is a nonlinear convex optimization problem, although in the literature problem (1) is called a two-stage stochastic convex quadratic problem with recourse (TCQP) [14], and we adopt this terminology. In addition to the fact that the second-stage problem is a QP, the term TCQP is used because any TCQP can be reformulated as an equivalent convex QP when the support of ξ is finite, or it is approximated by a convex QP when the support of ξ is a not finite, as we show below.

Sampling methods such as Monte Carlo, Latin hypercube sampling, and importance sampling, etc. are used to make the computation of the expected value term and its derivative(s) tractable from a computational point of view. Once a finite sample $(\xi_1, \xi_2, \ldots, \xi_N)$ of N realizations of the random vector ξ is obtained, the recourse term $\mathbb{E}[G(x_0, \xi)]$ is approximated by the average of the values $G(x_0, \xi_i)$, $i = 1, 2, \ldots, N$. This is the sample average approximation (SAA) approach, with which one obtains a convex quadratic deterministic approximation to the TCQP (1), which has the following form

$$\begin{array}{lllllll}
\min & \multicolumn{6}{l}{\left(\frac{1}{2}x_0^T Q_0 x_0 + c_0^T x_0\right) + \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{2}x_i^T Q_i x_i + c_i^T x_i\right)} \\
\text{subj to} & T_0 x_0 & & & & & = b_0, \\
 & T_1 x_0 + & W_1 x_1 & & & & = b_1, \\
 & T_2 x_0 + & & W_2 x_2 & & & = b_2, \\
 & \vdots & & & \ddots & & \vdots \\
 & T_N x_0 + & & & & W_N x_N & = b_N, \\
 & \multicolumn{6}{l}{x_0 \geq 0,\ x_1 \geq 0\ , x_2 \geq 0, \ldots x_N \geq 0.}
\end{array} \tag{3}$$

Interior-point methods (IPMs) have been used as early as 1988 to decompose and solve SAA problems [4]. The SAA problems are usually extremely large and even in the sparse case they can be solved only by means of distributed computing. The decomposition of the problem in the context of IPMs is usually achieved at the linear algebra level by taking advantage of the block-separable form of the objective function and the half-arrow shape of the Jacobian. This special structure allows most of the work related to IPM linear solves to be done independently for each sample when a Schur complement mechanism is used. Parallel implementations of IPMs using the Schur complement decomposition have been done in state-of-the-art software packages such as OOPS [11–13] and IPOPT [19].

Recently we implemented PIPS, a parallel IPM solver in C++ based on OOQP [9] that uses the Schur complement decomposition to solve SAA problems. We achieved very good strong scaling from 80 to 1000 cores (77% efficiency) on a stochastic unit commitment problem (described in Section 4.1) with 29 million variables. The main obstacle to solving larger instances of this problem on a larger number of cores was a memory usage bottleneck described in Section 2 that is caused by the number of variables in the first-stage problem. The present work removes this bottleneck by performing the linear algebra related to the first-stage problem in a parallel, distributed-memory MPI-based framework.

In the context of interior-point methods applied to SAA problems of the form

(3), the linear algebra operations associated with the first-stage consist of solving symmetric indefinite systems of the form

$$C = \begin{bmatrix} Q & A^T \\ A & 0 \end{bmatrix}, \tag{4}$$

where Q is a *dense*, symmetric positive definite matrix and A is a full-rank rectangular matrix, see Section 2 for a detailed discussion. Systems with matrices of this form are also known as saddle-point linear systems.

The size of the matrix Q can be very large; for example, it can approach 100,000 by 100,000 in the case of the stochastic unit commitment problem with wind power generation presented in Section 4.1. Such large, dense linear systems can be solved efficiently by using existing libraries for parallel dense linear algebra such as ScaLAPACK, PLAPACK, and Elemental. This is the approach that we follow; however, there are two issues that we address and solve in this paper.

The first issue is the lack of a parallel solver for symmetric indefinite *dense* linear systems. Instead, one must use an LU-based solver for general matrices, which is twice as expensive. We overcome this drawback by implementing a specialized Cholesky-based LDL^T factorization. Such factorization has been previously used in the *sparse* context, see the review article by Benzi et. al. [2], however, to our knowledge, it was not implemented before for dense saddle-point systems in a distributed memory environment.

The second difficulty is specific to stochastic optimization problems and comes from assembling the distributed saddle-point matrix (4). More specifically, C needs to be distributed across processors as required by the particular parallel solver, but all processors contribute to all of the elements of the Q block. Therefore a large amount of inter-process communication (in the form of "reduce" operations) is required in the assembly operation. This can incur a significant cost, possibly greater than the cost of factorization. We describe a technique that yields good large-scale performance. It uses efficient `Reduce_scatter` operations that maximize network bandwidth given the available memory on each node.

The paper is organized as follows. In Section 2 we outline the linear algebra required by interior-point methods for solving the stochastic SAA problems, and we present the Schur complement-based decomposition used to parallelize the computation of Q. In Section 3 we describe the parallel dense linear solvers ScaLAPACK and Elemental and our LDL^T factorization based on Elemental. We also give the implementation details of the specialized reduce operations we use to distribute the saddle-point dense linear system. In Section 4 we investigate and report on the large-scale performance of our code (up to 2,048 cores). The conclusions of this work and related future research directions are given in Section 5.

2. Schur complement decomposition of SAA problems

In this section we present the linear algebra needed to solve convex quadratic SAA problems of the form (3) by interior-point methods. We refer the reader to [11], [12], [13], or [15] for more details on how the linear algebra is derived.

The deterministic SAA problem (3) has a staircase structure that can be exploited to produce highly parallelizable linear algebra. The matrix of the linear system that needs to be solved at each iteration of the interior-point algorithm has an arrow shape of the form

$$K := \begin{bmatrix} K_1 & & & B_1 \\ & \ddots & & \vdots \\ & & K_N & B_N \\ B_1^T & \dots & B_N^T & K_0 \end{bmatrix}. \tag{5}$$

Here we used the following simplifying notation,

$$K_i := \begin{bmatrix} \frac{1}{N}\bar{Q}_i & \frac{1}{N}W_i^T \\ \frac{1}{N}W_i & 0 \end{bmatrix}, \qquad K_0 := \begin{bmatrix} \bar{Q}_0 & W_0^T \\ W_0 & 0 \end{bmatrix},$$
$$B_i := \begin{bmatrix} 0 & 0 \\ \frac{1}{N}T_i & 0 \end{bmatrix}, \qquad i = 1, 2, \dots, N,$$

where $\bar{Q}_i = Q_i + D_i$, $i = 0, 1, \dots, N$, with each D_i being a diagonal matrix with positive diagonal entries occurring from the use of interior-point algorithms.

Solving linear systems of the form $K\Delta z = r$ is the main computational effort at each iteration of the interior-point algorithm. Since K is symmetric, it can be factorized as LDL^T [10], where L is a unit lower triangular matrix and D is a diagonal matrix. One can easily verify that L and D have the following particular structures,

$$L = \begin{bmatrix} L_1 & & & \\ & \ddots & & \\ & & L_N & \\ L_{N1} & \dots & L_{NN} & L_c \end{bmatrix}, \quad D = \begin{bmatrix} D_1 & & & \\ & \ddots & & \\ & & D_N & \\ & & & D_c \end{bmatrix},$$

where

$$L_i D_i L_i^T = K_i, \; i = 1, \dots, N, \tag{6}$$
$$L_{Ni} = B_i^T L_i^{-T} D_i^{-1}, \; i = 1 \dots, N, \tag{7}$$
$$C = K_0 - \sum_{i=1}^{N} B_i^T K_i^{-1} B_i, \tag{8}$$
$$L_c D_c L_c^T = C. \tag{9}$$

We note that C defined by (8) is the Schur complement of the first-stage Hessian block K_0 in the entire Hessian matrix K.

Let $\Delta z_i := \begin{bmatrix} \Delta x_i^T \; \Delta y_i^T \end{bmatrix}^T$, $i = 0, 1, \dots, N$, $\Delta z := \begin{bmatrix} \Delta z_1^T \dots \Delta z_N^T \; \Delta z_0^T \end{bmatrix}^T$, and let r be of the form $\begin{bmatrix} r_1^T \dots r_N^T \; r_0^T \end{bmatrix}^T$. To solve the linear system $K\Delta z = r$ we take the following steps:

$$w_i = L_i^{-1} r_i, \ i = 1, \ldots, N, \tag{10}$$

$$\tilde{r}_0 = r_0 - \sum_{i=1}^{N} L_{Ni} w_i, \tag{11}$$

$$v_i = D_i^{-1} w_i, \ i = 1, \ldots, N, \tag{12}$$

$$w_0 = L_c^{-1} \tilde{r}_0, \tag{13}$$

$$v_0 = D_0^{-1} w_0, \tag{14}$$

$$\Delta z_0 = L_c^{-1} v_0, \tag{15}$$

$$\Delta z_i = L_i^{-T} (v_i - L_{Ni} \Delta z_0), \ i = 1, \ldots, N. \tag{16}$$

Observe that the computations of each of the steps (6)-(8), (10)-(12), and (16) can be done independently for each scenario $i \in \{1, \ldots, N\}$. This observation is the core of the Direct Schur complement (DSC) method which we implemented in PIPS. However, the factorization (9) and steps (13)-(15) need to be performed serially, that is identically on all processors (or only on one processor, while the other processors are waiting). Obviously, the serial steps create a bottleneck in the parallel execution flow, but for problems having a small number of first-stage variables, the bottleneck has little negative impact on the performance of DSC method. Unfortunately, as expected, the performance of the DSC method is considerably affected when problems with a large number of first-stage variables are solved. The Preconditioned Schur Complement (PSC) method we presented in [15] uses a stochastic preconditioner for the Schur complement matrix C and Krylov iterative methods for the solution of linear systems involving C to remove most of the execution bottleneck. Consequently, PSC approach outperforms DSC method on medium-sized first-stage problems (several thousands variables). However, PSC experiences a different bottleneck caused by the insufficient memory in the case of SAA problems with a larger number of first-stage variables (more than ~10,000). The memory usage bottleneck occurs because, for such problems, the Schur complement matrix C does not fit the memory of a single computational node.

As shown in [15], C has the following simplified form,

$$C = \begin{bmatrix} Q & T_0^T \\ T_0 & 0 \end{bmatrix}, \tag{17}$$

where $Q := \bar{Q}_0 + \frac{1}{N} \sum_{i=1}^{N} T_i^T \left(W_i \bar{Q}_i^{-1} W_i^T \right)^{-1} T_i$. Each of the $T_i^T \left(W_i \bar{Q}_i^{-1} W_i^T \right)^{-1} T_i$ terms becomes dense even when all the second-stage matrices are sparse. This adverse behavior is somehow expected since, formally speaking, two matrices are inverted and it is well known that matrix inversion destroys sparsity. Consequently, the $(1, 1)$ block Q of the Schur complement matrix C becomes dense. This is a square block of the size of the number of first-stage variables. PSC, as well as DSC, stores C as a dense matrix on each processor. As we previously mentioned, this approach leads to a memory usage bottleneck because C becomes too large to store completely on a node for some real-life problems with many first-stage variables (more than ~10,000). In this paper we propose an approach to remove the memory bottleneck as well as the execution bottleneck. Our technique parallelizes the first-stage linear algebra(*i.e.*, steps (9) and(13)-(15)) of the DSC method in a distributed-memory computing environment.

3. Factorization and distribution of the dense system

Here, we present our solutions to the issues arising in the parallelization of the dense linear algebra required in the first stage, whose details were just described in Section 2. In Section 3.1 we provide an overview of existing parallel distributed-memory linear algebra libraries, followed by our specialized factorization procedure in Section 3.2. In Section 3.3 we describe the procedure for assembling the distributed matrix.

3.1. *Parallel solvers for dense linear systems*

As described in Section 2, the linear system we must factorize and solve at each iteration is a symmetric indefinite system with the following block form,

$$C = \begin{bmatrix} Q & A^T \\ A & 0 \end{bmatrix}, \tag{18}$$

where Q is fully dense, symmetric positive definite and A $(= T_0)$ is sparse and of full rank. This is known as a standard saddle-point system. In the initial versions of PIPS, we used the symmetric indefinite solver in LAPACK [1] (`DSYSV`), which is based on the Bunch-Kaufman decomposition [6]. For the large-scale problems that PIPS is designed to solve, storing the system entirely in local memory in order to solve it by using LAPACK is infeasible. Our solution is to solve the system in parallel in a distributed memory environment.

A review of the literature yielded a single parallel dense symmetric indefinite solver by Strazdins and Lewis [17]; however, the code has not been maintained in the past 10 years and was not incorporated into any major library. Strazdins confirmed in correspondence that he was unaware of any other efforts. Also, we are not aware of any solver specialized for dense saddle-point systems, either in serial or in parallel.

Historically, the most important and most widely used parallel dense linear algebra packages are ScaLAPACK [5] and PLAPACK [18]. A package currently under development is Elemental [16], which claims significant performance improvements over ScaLAPACK and PLAPACK. We chose to focus on ScaLAPACK and Elemental; PLAPACK did not offer any particular advantages, and one may consider Elemental as its successor. In Sections 3.1.1 and 3.1.2, we describe these packages and their particular methods of distributing the dense matrix across nodes. The matrix distributions are the important, if not defining, characteristic of these packages.

While all of these packages provide routines for LU and Cholesky decompositions, none provides routines for symmetric indefinite systems. Cholesky decomposition is not directly applicable to our linear system, since it is indefinite, and LU decomposition requires double the number of operations necessary. In light of the lack of an existing symmetric indefinite solver, we developed a specialized Cholesky-based LDL^T factorization procedure that exploits saddle-point structure of the matrix; it is described in Section 3.2.

3.1.1. ScaLAPACK

ScaLAPACK (Scalable LAPACK) is a library designed to present an LAPACK-like interface to dense, distributed linear algebra. The procedures follow LAPACK naming and calling conventions; however, not all LAPACK functions have been implemented. In particular, while there is a `PDGESV` for solving general linear systems, there is no "`PDSYSV`" procedure for solving symmetric indefinite systems.

For load balancing, ScaLAPACK uses a *block-cyclic* distribution so that all processors "own" all parts of the matrix. First, a block size b is fixed, and the available processors are arranged into an $n_p \times m_p$ processor grid. The matrix is partitioned into blocks of size $b \times b$. Irregular, small blocks are permitted at the boundaries of the matrix. Using zero-based indexing, the block-cyclic distribution arises by assigning block (i, j) to processor $(i \mod n_p, j \mod m_p)$. Each processor has a single column-oriented local storage buffer, where the blocks are stored in their original shape, *as if* there were no blocks of the matrix between them. See Figure 1 for an example.

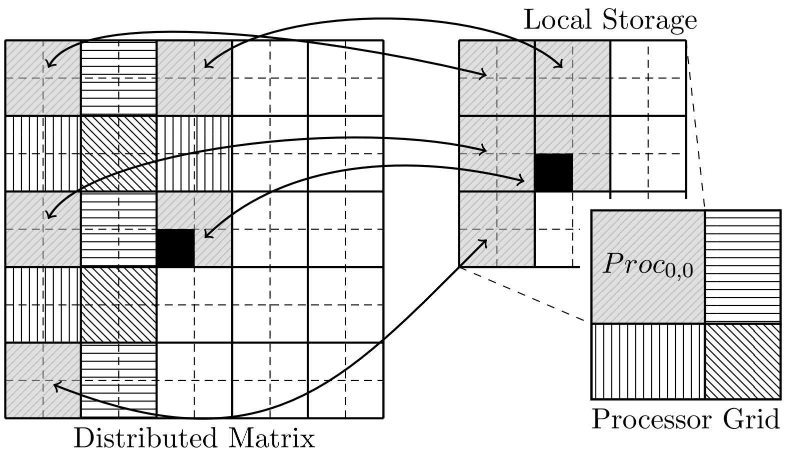

Figure 1. An illustration of the block-cyclic distribution used in ScaLAPACK, with blocksize 2 on a 2×2 processor grid, 10×10 matrix. The mapping is shown between the blocks of the distributed matrix and the local storage on processor $(0, 0)$. The blocks belonging to each processor are marked with a pattern. The highlighted square in black illustrates the element-to-element mapping. The size of blocks in the processor grid indicates the size of the local storage; note that it need not be uniform.

The above is a fairly complete description of the block-cyclic distribution at a high level. Additionally, one may consider non-square blocks, but these are supported only for some operations in ScaLAPACK. The formulas for calculating the exact index mappings become somewhat complex; they are covered in full detail in [5].

A particularity of ScaLAPACK is that these storage blocks are the same blocks used algorithmically. In the ScaLAPACK implementations of direct factorizations (LU and Cholesky), large algorithmic blocks achieve optimal cache performance, while small storage blocks yield the best load balancing. Hence, there is an inevitable compromise when choosing the optimal block size. We also found that the coupling of the storage and algorithmic blocks imposes other, severe restrictions. For example, ScaLAPACK has the ability to perform operations on sub-matrices of the distributed matrix, but the sub-matrices must be completely aligned with the storage blocks, that is, the top left element of a sub-matrix must be the top left element of a storage block. In Section 3.2 we describe how this restriction prevented implementing our specialized LDL^T factorization in ScaLAPACK.

3.1.2. Elemental

Elemental is a new library intended to replace ScaLAPACK and PLAPACK. It is under active development.

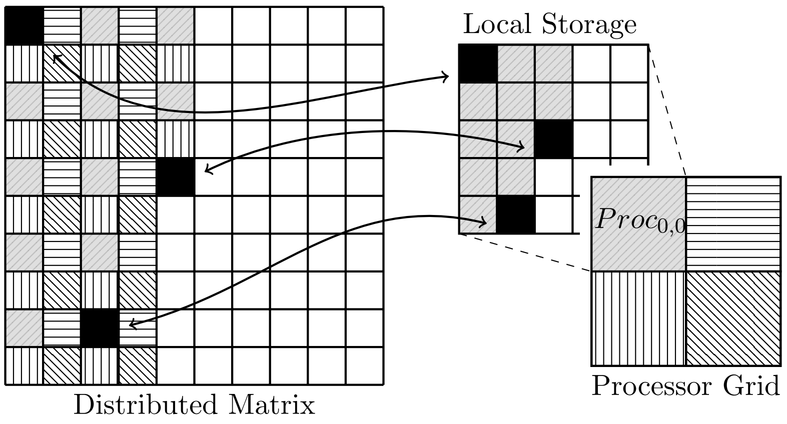

Figure 2. Illustration of the element-cyclic distribution used in Elemental on a 2×2 processor grid, 10×10 matrix. The mapping is shown between elements of the distributed matrix and the local storage on processor $(0, 0)$. The blocks belonging to each processor are marked with a pattern.

The distinguishing characteristics of Elemental are the *element-cyclic* matrix storage distribution, which is precisely the block-cyclic ScaLAPACK distribution with a blocksize of 1 (see Figure 2), and the separation of the storage and algorithmic sizes. In Elemental one can use large algorithmic blocksizes to obtain the best cache performance without compromising the load balancing, which in fact is the best possible since the storage blocks are of size 1. The element-cyclic distribution requires Elemental to use more elaborate and possibly more costly communication patterns than ScaLAPACK. However, the abovementioned benefits of the element-cyclic distribution prevail over the communication overhead, and Elemental outperforms ScaLAPACK in the tests presented in [16]. Another important feature of the "elemental" matrix distribution is the ability to perform operations on arbitrary sub-matrices, because there are no misalignment issues. We fully exploited this feature in our specialized LDL^T factorization.

3.2. *Specialized Cholesky-based LDL^T factorization*

Although using a general LU factorization routine to solve the linear system C given by (18) presents a practicable solution, it is not ideal. We would expect to be able to gain a 2x increase in performance by using an algorithm that at least exploited the symmetric structure. We describe below a specialized LDL^T factorization algorithm that exploits both the symmetric and the saddle-point structure of C, and we analyze the flop count.

3.2.1. Algorithm

Every symmetric indefinite matrix whose diagonal is not all zeros has a decomposition LDL^T where L is lower triangular and D is diagonal [7]. This decomposition is usually avoided in practice because of the numerical instabilities that may arise when the elements of the diagonal of the matrix all approach zero. Instead, a slightly modified decomposition is used, taking D to be a block-diagonal matrix with blocks of size 1 or 2. This is used in the Bunch-Kaufman [6] and Bunch-Parlett [7] methods.

In the case of the saddle-point system C (18), however, because the Q block is positive definite and the matrix A is full-rank, a LDL^T factorization with D strictly diagonal always exists. This can be seen by writing

$$\begin{bmatrix} Q & A^T \\ A & 0 \end{bmatrix} = \begin{bmatrix} M & 0 \\ AM^{-T} & \widetilde{M} \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} M^T & M^{-1}A^T \\ 0 & \widetilde{M}^T \end{bmatrix}, \tag{19}$$

where M and $\widetilde{M}$ are lower triangular Cholesky factors satisfying $MM^T = Q$ and $\widetilde{M}\widetilde{M}^T = AQ^{-1}A^T$. These factors necessarily exist because Q is positive definite, and therefore $AQ^{-1}A^T$ is positive definite as well because A has full rank.

Benzi et al. [2] note that the factorization (19) is more efficient than Bunch-Kaufman because no pivoting is required; in addition, it is sufficiently numerically stable since it couples two Cholesky factorizations. The use of this factorization may be disadvantageous in the sparse case, because a large amount of fill-in may occur in the factors. Obviously, this is not the case in this work since our matrix is dense. To our knowledge, there has been no previous attempt to solve dense saddle-point systems in parallel by using an LDL^T factorization of form (19) or any other specialized approach.

What makes this factorization practical is that it can be performed *in-place* on the distributed matrix. Let us denote the four logical blocks of the distributed matrix as follows:

$$B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} \tag{20}$$

where $B = C$ initially, that is $B_{00} = Q, B_{10} = A, B_{01} = A^T, B_{11} = 0$; in fact, only the lower triangle must be filled. We perform a sequence of standard linear algebra operations on B, after which B contains the lower triangular L factor. See Figure 3 for the procedure.

Specialized LDL^T Procedure

```
In-place factorization
1.  $B_{00} \longleftarrow \text{Cholesky}(B_{00})$      $(= \text{Cholesky}(Q) = M)$
2.  $B_{10} \longleftarrow B_{10}B_{00}^{-T}$             $(= AM^{-T})$                                   (trsm)
3.  $B_{11} \longleftarrow (B_{10})(B_{10})^T$            $(= (AM^{-T})(AM^{-T})^T = AQ^{-1}A^T)$         (syrk)
4.  $B_{11} \longleftarrow \text{Cholesky}(B_{11})$      $(= \text{Cholesky}(AQ^{-1}A^T) = \widetilde{M})$
Solving $Sx = b$
5.  $b \longleftarrow L^{-1}b$                                                                           (trsv)
6.  $b \longleftarrow D^{-1}b$                                                                           (ad-hoc)
7.  $b \longleftarrow L^{-T}b$                                                                           (trsv)
```

Figure 3. Specialized procedure for solving the saddle-point system S. After the factorization, the lower triangle of B contains L. The name of the operations in standard BLAS terms is in parentheses. The operations are described in the text.

The operations used are Cholesky factorization, triangular solve (*trsm, trsv*), and symmetric rank-k update (*syrk*). On line 2, *trsm* is used to solve a matrix triangular system of the form $XY^T = Z$ for X, overwriting Z with the answer. On line 3, a symmetric rank-k update performs the operation $Z \longleftarrow \alpha XX^T + \beta Z$, where we take $\alpha = 1, \beta = 0$. The sequence of operations is similar to a step of blocked right-looking Cholesky factorization [18].

In the solution phase, the *trsv* operation solves a simple triangular system $Zx = b$ or $Z^T x = b$. Note that $D = D^{-1}$. Then, multiplication by D^{-1} can be performed *ad-hoc* by simply negating the lower part of the right-hand side vector.

Cholesky factorization, triangular solve, and symmetric rank-k update are standard operations that, in principle, are provided by both ScaLAPACK and Elemental. In both ScaLAPACK and Elemental, these operations can also operate on sub-matrices of a distributed matrix, a property that is fully exploited above. However, as we previously mentioned, ScaLAPACK has an important caveat: the sub-matrices must align with the storage blocks. Given that Q may be of arbitrary size and that there is a practical limit on the size of a storage block, this restriction is effectively impossible to satisfy in the given context without extensively modifying ScaLAPACK's source code. Therefore, this procedure was only implemented by using Elemental, requiring just five lines of code in C++ to perform the factorization.

3.2.2. *Flop count*

We analyze here the number of operations required by the algorithm above, in order to confirm the desired theoretical 2x decrease in operation count over *LU* factorization. Only the higher-order terms are counted.

Note first that for an $n \times n$ matrix, *LU* decomposition requires $\frac{2}{3}n^3$ floating-point operations, and both Cholesky and Bunch-Kaufman (symmetric indefinite) decompositions require $\frac{1}{3}n^3$. Both *LU* and Bunch-Kaufman require an additional $O(n^2)$ comparisons to perform pivoting [6].

Now let n be the size of Q and m be the number of rows in A ($Q \in \mathbb{R}^{n\times n}, A \in \mathbb{R}^{m\times n}$). Then the entire matrix is $(n+m) \times (n+m)$. We verify that *trsm* requires mn^2 operations and *syrk* requires $m^2 n$ operations.

For *trsm* ($B_{10}B_{00}^{-T}$) one may verify that a triangular solve requires n^2 operations; this is performed m times, resulting in mn^2 operations. For *syrk* ($(B_{10})(B_{10})^T$): this is a normal matrix-matrix multiplication, but only the lower triangle is calculated. Each of the $\frac{1}{2}m^2$ elements requires n multiplications and $n-1$ additions, resulting in $m^2 n$ operations.

The specialized LDL^T factorization therefore requires $\frac{1}{3}n^3 + m^2 n + mn^2 + \frac{1}{3}m^3 = \frac{1}{3}(n+m)^3$ floating-point operations. This is the same number of flops as Bunch-Kaufman and half those of an *LU* decomposition, confirming that we have achieved the goal of a factorization routine that requires half the operations of *LU* and, in theory, should deliver twice the performance. Also note that no comparisons are required in this case, unlike both *LU* and general symmetric indefinite factorization routines.

As a final observation, recall that the A block is in reality sparse, although it has been treated as a block of a dense matrix. We implemented it as such, but one may be able to significantly reduce communication costs and flops in the *trsm* stage by storing A as a sparse matrix on each processor and implementing a specialized triangular solve routine. However, since the number of rows of A is less than the the number of rows of Q, usually much more smaller, this would likely be a minor optimization.

3.2.3. *As a saddle-point solver*

We note that the method proposed applies with only a slight modification to a more general dense saddle-point system of the form

$$C = \begin{bmatrix} Q & A^T \\ A & -S \end{bmatrix}, \tag{21}$$

where S is symmetric positive semidefinite and Q and A are symmetric positive definite and of full rank, respectively, as above. The only modification necessary to the algorithm in Figure 3 is at Step 3 to include S in the Schur complement. This saddle-point system (21) has applications outside of constrained optimization, which are referenced in [2].

3.3. *Assembling the matrix*

We have assumed up to this point that the linear system is already distributed across processors as required by Elemental or ScaLAPACK. However, assembling the matrix and distributing it as required can be a costly operation, possibly more costly than the factorization itself. This operation must be streamlined to obtain acceptable large-scale performance.

We present a simplified version of the summation that was more fully described in Section 2. Let B refer to the distributed matrix, partitioned as in (20). Let $\mathcal{P}$ be the set of processors. The distribution operation we must perform can be described simply as

$$B_{00} = \sum_{p \in \mathcal{P}} M_p, \tag{22}$$

where M_p is calculated locally on processor p and B_{00} is distributed across processors. Here M_p is the local contribution to the sum discussed in Section 2, precisely at Step (8).

In the serial case where LAPACK is used to solve the entire first-stage system on each processor, this operation maps directly to an `Allreduce` in MPI. In the distributed case, we have two important considerations that make the distribution problem significantly more complicated:

- M_p is too large to fit entirely in a node's local memory.
- Every node owns different, non-contiguous elements in B_{00}; however, all nodes contribute to all elements.

To address the first issue, we calculate M_p in blocks of columns that fit in a node's local memory. Then, repeated communication operations are performed to "globally" build B_{00} by blocks of columns.

For the second issue, we observe that the communication pattern required maps closely to a `Reduce_scatter` operation in MPI, in which a large array is "reduced" (summed) across all processors, and then its pieces are partitioned and "scattered" (distributed) to processors.

However, `Reduce_scatter` requires that each processor receive a single contiguous part of the send buffer. Considering the distribution of the matrix across processors, a single contiguous column of the matrix can not be partitioned such that the elements belonging to a given processor are in contiguous memory. Some intermediate steps are therefore necessary.

We first present a method to distribute the entire B_{00} block ("full reduce"), followed by a method to distribute only the lower triangle of B_{00} ("lower triangular reduce"). We describe these methods only for the matrix distribution corresponding to Elemental, but a "full reduce" was also implemented for ScaLAPACK.

3.3.1. Full reduce

In order to apply LU decomposition, the entire distributed matrix must be filled with the corresponding elements, disregarding the symmetry of the matrix. This is not the case for the LDL^T procedure, which requires only the lower triangle and it is twice faster. Nevertheless, we compare our LDL^T factorization to LU decomposition in Section 4, so we present the "full reduce" method that fills the entire B_{00} block. This method is also a starting point for the lower triangular procedure, described in Section 3.3.2.

Figure 4 contains a high-level description of the procedure. While it is generally straightforward, special care is needed at some points to ensure an efficient implementation.

As mentioned above, we build the matrix in blocks of columns. The size of the blocks is governed by the parameter b. This should be as large as possible to maximize the communication bandwidth, given the available memory on each node.

"Full reduce" procedure

```
Initialization
1.   Let $n$ be the size of $B_{00}$.
2.   Fix buffer size $b$.
3.   Allocate $b$ doubles for column buffer and $b$ doubles for send buffer.
4.   Allocate recv buffer (sufficiently large).
5.   step $\longleftarrow b/n$
Main loop
6.   For $i = 0$ to $n-1$, step
7.       endCol $\longleftarrow \min(i + step - 1, n-1)$
8.       Compute columns $i$ to endCol $\longrightarrow$ column buffer
9.       Pack column buffer $\longrightarrow$ send buffer
10.      MPI_Reduce_scatter(send buffer) $\longrightarrow$ recv buffer
11.      Unpack recv buffer $\longrightarrow$ local matrix storage
12.  End For
```

Figure 4. Overall procedure for distributing the full B_{00} block.

The **Pack** step fills the send buffer for `Reduce_scatter`. The send buffer must be arranged such that the elements destined for a processor are in a single, contiguous block, and the blocks must be ordered according to processor number. For fast unpacking, we also require that inside a block, the order of elements match their order in the local matrix storage. We have fully specified a one-to-one map between the location of the elements in the column buffer and their location in the send buffer, and theoretically only a permutation of the column buffer is necessary. An in-place permutation would have poor cache performance, so we allocate a separate array and copy the elements into their positions.

The key to streamlining this copying procedure is to avoid expensive division and modulus operations that one would naively use to calculate the required positions of the elements. Instead, one should loop over continuous memory in the send buffer and copy the corresponding elements of the column buffer, using only addition operations to calculate the addresses. The following loop implicitly copies the elements into the correct order in the send buffer: loop over each processor p in order, and in an inner loop pick out the elements from the column buffer belonging to processor p, copying them to the send buffer, in order. The offsets between elements is known, so this inner loop may be performed just by incrementing the counters correctly. Recall that all buffers store elements in column-major order.

Once the send buffer is filled, `Reduce_scatter` is called. The entries are summed across all processors, and the result is partitioned and distributed to the receive buffers on the desired processors. We have arranged the elements so that they are

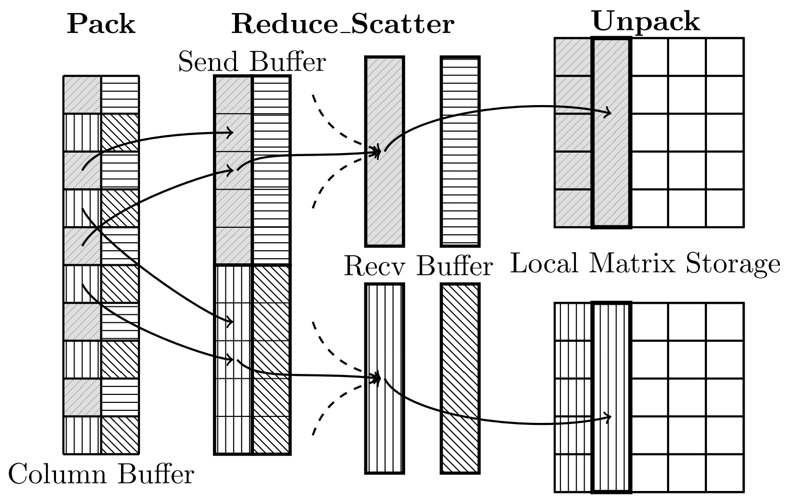

Figure 5. Illustration of a step of the "full reduce" procedure. The 3rd and 4th columns are sent of a 10×10 B_{00} block on a 2×2 processor grid. Note that the local storage contains more rows and columns than displayed; only the elements belonging to B_{00} are shown. Dashed lines indicate communication from other processes. In general, processors will receive more than one column, unlike shown here.

in the correct order for unpacking, so this step is straightforward. Note that we cannot use the local storage directly as the receive buffer, because the local storage has additional rows for the rest of the distributed matrix.

3.3.2. Lower triangular reduce

For the LDL^T factorization procedure, we would be performing unnecessary work by distributing the entire symmetric B_{00} block, when only the lower triangle is required. Also, in initial experiments we noticed that the communication in the reduce step can take a significant amount of time. Therefore, we set out to design a "lower triangular reduce" that should take nearly exactly half the time of the "full reduce" procedure above, excluding computing the columns. We arrived at a procedure that can effectively guarantee requiring only half of the communication time, with little extra overhead.

With this goal in mind, we must fix the size b of the send buffer as above and design a procedure that calls `Reduce_scatter` half the number of times. We need to send only half the number of elements, so this is certainly possible. In a more naive approach, one might be led to loop over fixed-sized blocks of columns as before and send only the lower triangular elements. This approach cannot deliver the performance desired, because it results in the same number of `Reduce_scatter` calls as before and so does not decrease the communication overhead.

The solution for a fixed send buffer size is to vary the number of columns calculated in each iteration, taking exactly as many as whose lower triangular elements fit in the send buffer. This number will increase with each iteration. We may count the lower triangular elements as follows. Let n be the size of the B_{00} block. Then the block of columns starting at column s and ending at column $e-1$ (inclusive, zero indexed) has the following number of elements:

$$\sum_{i=s}^{e-1}(n-i) = -\frac{1}{2}e^2 + \left(n+\frac{1}{2}\right)e + \frac{1}{2}s^2 - \left(n+\frac{1}{2}\right)s =: f(e;n,s). \tag{23}$$

At each iteration, n and s are fixed, and the problem is to find the largest integer

e satisfying $f(e; n, s) \leq b$ and $e \leq n$. This is an easy and inexpensive calculation, given that $f(e; n, s)$ is quadratic.

Besides varying the number of columns at each iteration, the overall procedure is the same as in Figure 4. The **Pack** and **Unpack** operations require a small overhead in addressing, but only in calculating offsets. Instead of describing these in detail, we provide an illustration in Figure 6, which indicates the operations required.

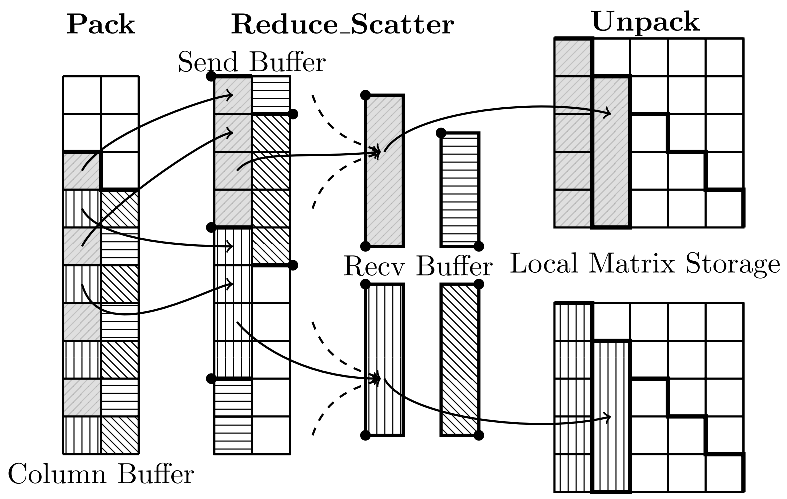

Figure 6. Illustration of a step of the "lower triangular reduce" procedure. The 3rd and 4th columns are sent of a 10×10 B_{00} block on a 2×2 processor grid. Note that the local storage contains more rows and columns than displayed; only the elements belonging to B_{00} are shown, and the lower triangular elements are indicated. Dashed lines indicate communication from other processes. Dots indicate the partitions of the column-major send buffer. In the illustrated case, only two columns fit in the send buffer. Note that in general, not all processors will receive an equal number of elements, because of the properties of the matrix distribution.

4. Numerical experiments

Numerical experiments were performed on the Fusion cluster at Argonne National Laboratory. Each node has 36 GB of RAM and dual quad-core Intel Xeon 2.53 Ghz CPUs, for a total of 8 cores per node. In further discussion, we treat each core itself as a node or processor with its own local memory. The cluster has an Infiniband interconnect. A storage blocksize of 32 is used for ScaLAPACK and an algorithmic blocksize of 96 is used for Elemental, optimized respectively by empirical observation. Additionally, 250 MB is used for buffers during the reduce stage.

We first describe the test problem and then present strong and weak scaling results.

4.1. *The test problem*

We use a formulation of the stochastic unit commitment problem with wind power generation in the tests for PIPS. For brevity, we do not present the full model; instead, we provide an overview of the problem and the terminology used to describe it, and we direct the interested reader to [8] for a complete presentation.

Unit commitment refers to committing power generation *units* to either produce electricity or remain idle. In our problem there are two types of power units: thermal power plants using fossil fuels and wind farms using renewable energy. The thermal power generation units are costly to operate, both economically and environmentally. Hence, they should not be operating in large excess of demand. Each unit has startup, shutdown, and running costs and cannot change state instantaneously.

The stochastic component arises from considering electricity produced by wind farms, which is highly variable. The optimization problem is to minimize operation costs *subject to* satisfying the demand with some safety margin. Solving such problems, we may realize the economic and environmental benefits of wind power while ensuring that it is safely integrated with the power grid.

Each *scenario* is a possible realization of weather patterns, which corresponds to a different amount of electricity produced by the wind farms. These scenarios are generated by simulation using the state-of-the-art Weather Research and Forecast (WRF) model. In the formulation proposed by [8], this is a two-stage stochastic mixed-integer linear program with recourse, and the problem is solved over a 24-hour timeframe with a *recourse* stage to reallocate units at the end of the period. The problem solved by PIPS has one (large) simplification: the mixed-integer problem is relaxed to a continuous problem. This can be considered as the root relaxation problem in a branch-and-bound framework. However, the problems solved are realistically sized, in both the number of variables and the number of second-stage scenarios.

Problems of various sizes that are used in the experiments of this section are obtained by replicating a (small) real-life unit commitment problem (10 thermal units, 12 wind farms) set up for the state of Illinois [8]. We were forced to do this because of the lack of data for a larger area. We mention that our implementation is not tuned to take advantage of any special structure that may be introduced by replications.

4.2. *Solvers*

We compare here the first-stage factorization times for the three solvers tested: LU with ScaLAPACK, LU with Elemental, and LDL^T with Elemental. A fixed problem size of 300 thermal units is used, and we vary the number of processors used by PIPS. The Q block of C is of size 23,436, and the A block has 1,224 rows. This is not an especially large first-stage problem, and so we would expect the solver to be less efficient with a larger number of processors. To verify this, we include cases where only a subset of the processors is used for factoring the matrix. See Table 1.

Table 1. Factorization times. In some cases, a subset of the total CPUs is used for the factorization. (S) indicates ScaLAPACK and (E) indicates Elemental. The LDL^T factorization is performed with Elemental. Values are averages over 5 iterations. There was insufficient memory to run with 32 processors.

# Procs.		Factor (sec)		
	# Factoring	LU (S)	LU (E)	LDL^T
32	32	*	*	*
64	64	55.14	89.18	**29.94**
256	256	15.63	17.68	**9.78**
1024	256	22.34	25.54	**11.48**
	1024	16.59	20.04	**6.71**
2048	256	35.20	42.48	**16.86**
	1024	31.25	41.43	**10.81**
	2048	38.45	56.19	**14.08**

In all cases, the LDL^T factorization is the fastest, and in many cases it takes less than half of the best LU time. We note that ScaLAPACK LU times are better than Elemental LU times. This was a surprising result, given Elemental's claims of improved performance. After discussions with the author, we concluded that the poor performance is most likely related to the characteristics of the Fusion cluster. Elemental makes liberal use of advanced MPI collective operations, while ScaLAPACK primarily uses broadcast operations. This communication pattern may be relatively more efficient on Fusion, whose interconnect is not designed for all-to-all communications. In spite of this issue, the LDL^T factorization, which uses Elemental, does deliver the expected 2x increase in performance over even the LU solver in ScaLAPACK.

We observe that 1024 processors appears to be an optimal number for this problem size; this is clear in the case of 2048 total processors, where factorization time decreases from 256 to 1,024 and increases from 1,024 to 2,048 for all solvers. It is curious that factorization times appear to worsen for a fixed number of factoring processors when the total number of processors is increased. We did not have the opportunity to fully investigate this result.

4.3. *Reduce*

As the reader will later see, the factorization times above are small compared to the total execution time. In fact, the reduce times are generally more significant. The performance of the "full reduce" procedure is generally independent of the ScaLAPACK or Elemental matrix distribution, with most of the time spent in the `Reduce_scatter` step. Therefore, the benefit of ScaLAPACK is minor, and only when we compare between LU factorizations. For simplicity and because of a lack of CPU time allocation, from this point on we present strong scaling results only from Elemental. We will further justify this decision in Section 4.4.

The times for the full and lower triangular reduce operations are compared in Table 2. In all cases, the lower triangular reduce takes about half the time. Note that these times are bigger than the factorization times themselves. Also, reducing onto a subset of processors is slower than reducing onto all processors, because of the load imbalance that arises from the uneven communication costs. This slowdown appears to be greater than the possible improvement in factorization time.

Table 2. Time spent assembling Q block of the distributed matrix, excluding calculating the columns. The operation involves summing contributions from all processors to each of the 549,246,096 elements, and scattering the elements to their required place in the distributed matrix. Values are averages over 5 iterations. There was insufficient memory to run with 32 cores(4 nodes).

# Procs.		Reduce (sec)	
	# Factoring	LU	LDL^T
32	32	*	*
64	64	28.31	**12.96**
256	256	37.55	**17.18**
1024	256	110.45	**45.21**
	1024	54.32	**26.35**
2048	256	167.73	**89.50**
	1024	100.40	**50.80**
	2048	82.41	**43.93**

The reduction step presents a difficulty for strong scaling. With a fixed problem size, the reduce time increases with the number of processors. This result can be explained easily by an increase in communication overhead. Because the lower triangular reduce grows more slowly than the full reduce in absolute terms, we will see that in addition to being faster, the lower triangular reduce also provides the best strong scaling results.

4.4. *Strong scaling*

For the fixed problem size chosen (300 thermal units), the "backsolve" procedure to generate the columns of the terms in the sum of the Q block (17) takes approximately 140 seconds per scenario, independently of the total number of processors. This itself is large compared to the reduction and factorization steps, which are the only significant operations that are not "embarrassingly parallel". With 4,096 scenarios, we would expect very good strong scaling until the point where each processor is assigned a very small number of scenarios. This is the exact behavior we observed. The results are reported in Table 3.

Table 3. Total wall time for 5 interior-point iterations, with a fixed problem size with 4,096 scenarios, divided evenly across processors. All processors are used for factoring. Elemental is used for both LU and LDL^T. The execution time for 64 processors is used as the baseline, and we calculate speedup and efficiency.

Procs.	Tot. Walltime (min)		Speedup (Efficiency)		Peak Mem. (MB per node)
	LU	LDL^T	LU	LDL^T	
64	759.01	**735.37**	64 (100%)	64 (100%)	1818
256	195.62	**193.12**	248.3 (**97.0%**)	243.7 (95.2%)	770
1024	55.76	**50.99**	871.1 (85.1%)	922.9 (**90.1%**)	533
2048	37.9	**30.48**	1282.05 (62.6%)	1534.9 (**75.4%**)	523

The LDL^T solver has the best strong scaling, primarily because of the smaller increases in reduce times. We observe very good scaling (90%) up to 1024 processors, where each processor is assigned four scenarios. Scaling degrades to 75% efficiency with 2048 processors, where each processor is assigned only two scenarios, and the reduction and factorization steps become more significant. The peak memory usage per node decreases with the number of processors because the size of the local matrix storage decreases and fewer scenarios are assigned. Note that memory usage includes the fixed 250 MB used during the reduce step.

Currently, the number of processors is limited by the total number of scenarios. This is not an unreasonable limitation, given that the computational difficulty with SAA problems generally arises from the large number of scenarios. Splitting scenarios across processors is a possibility, and could be accomplished by using parallel sparse libraries to perform the linear algebra in the second-stage.

Earlier we disregarded ScaLAPACK, claiming that the benefit over Elemental's LU solver is insignificant. We now substantiate that claim. For 1,024 processors, the difference in factor times was about 3.5 seconds (20.04 versus 16.59). Recalling that the reduction process is the same, over five iterations, this amounts to about a 17.5 second difference in total execution time. This is, in fact, not very significant compared to the 5 minute difference between LU full reduce and LDL^T lower triangular reduce.

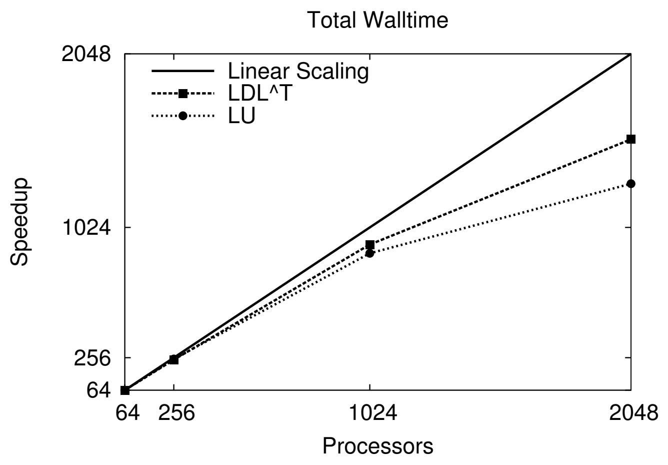

Figure 7. Plot of strong scaling results. See Table 3 for numerical values.

4.5. *Weak scaling*

Strong scaling is more difficult on smaller problems, and so above we used a relatively small first-stage matrix with size 24,660. By itself, this matrix requires about 4.5 GB to store, which does not exceed the capabilities of a modern computer; in the tests above, most of the memory on each node was in fact used to store the data associated with the scenarios. Here, we present weak scaling results, solving larger problems with a fixed number of processors. We solve the unit commitment problem described earlier, now with 640 and 1,000 thermal units on a fixed 1,024 processors with 4,096 scenarios. Table 4 contains the reduce and factorization time, and Table 5 contains the average iteration times. Because of the very large CPU time requirements, we ran three interior-point iterations for 640 and 1,000 thermal units (with 5 iterations for 300); for ScaLAPACK we can present only factorization times.

Table 4. Factorization and reduce times: 1,024 processors with all used for factorization, 4,096 scenarios. (S) indicates ScaLAPACK and (E) indicates Elemental.

Thermal	1st Stage Size	Factor (Sec.)			Reduce (Sec.)	
Units	(Q+A)	LU(S)	LU(E)	LDL^T	LU	LDL^T
300	23436+1224	16.59	20.04	**6.71**	54.32	**26.35**
640	49956+2584	60.67	83.24	**36.77**	256.95	**128.59**
1000	78030+4024	173.67	263.53	**90.82**	565.36	**248.22**

Table 5. Average iteration times and "backsolve" times per second-stage scenario: 1,024 processors with all used for factorization; 4,096 scenarios. Elemental used for LU.

Thermal	Total Variables	Per Scenario		Min./Iter.		Peak Mem.
Units		Vars.	Sec.	LU	LDL^T	(MB per node)
300	57,677,508	14,076	139.55	11.15	**10.19**	533
640	121,764,108	29,716	689.35	53.49	**50.44**	722
1000	189,620,508	46,276	1711.29	132.72	**122.74**	954

Both the factorization and reduce times for LDL^T continue to be about half of the times for LU. These are promising weak scaling results. The reduce times scale quadratically with the size of the Q block, since the operation is a function of the number of elements. The factorization time should scale with the cube of the size of the first-stage matrix; but as the matrix size increases, the factorization routines become more efficient, and so we observe less than cubic scaling at these problem sizes. The matrix of the largest problem has a size of over 82,000, which would take approximately 50 GB to store, and the LDL^T routine factors it in only 90 seconds. This translates to over two teraFLOPS of performance (20% of peak). Large matrices that would be very difficult, if not impossible, to solve in serial present no problem to solve efficiently in parallel.

None of these problems could have been solved previously by PIPS using LAPACK to factor the dense matrices. Problems of this size are real-life problems. For example, 1,000 thermal units and 1,200 wind farms covers the entire Midwest region of the United States. To our knowledge, SAA problems with nearly 80,000 first-stage variables have not been previously solved.

5. Conclusions and future work

We presented a specialized LDL^T factorization procedure for solving dense saddle-point linear systems in parallel. In numerical experiments, this procedure obtains the desired 2x increase in performance over a general LU factorization. Our factorization applies to an entire class of saddle-point systems and requires only five lines of C++ code to implement using an actively maintained parallel dense linear algebra library, Elemental. Currently, it is the only such procedure available. For saddle-point systems, it is likely more efficient than general parallel dense symmetric-indefinite solvers, if any are implemented in the future, because no comparisons or pivoting is required. The procedure scales well to very large systems, and performance will improve with improvements in the Elemental core.

We also presented an efficient method to assemble the matrix in the context of a parallel solver for two-stage stochastic optimization problems with recourse using the SAA approach. These problems are highly parallelizable by distributing the calculation for the second stage scenarios, but one must also solve a large dense linear system in the first stage variables. This work demonstrated how to parallelize solving this system as well. The overhead of parallelization arises in the assembly phase of the matrix, and we were able to reduce this cost by half by assembling only the lower triangle, significantly increasing the strong scaling efficiency. The communication overhead would likely be greatly reduced on a system with tightly coupled nodes and a dedicated collectives network, such as the Blue Gene/P at Argonne National Laboratory. We intend to test PIPS on this system as well as investigate efficient hybrid programming approaches (MPI plus shared-memory parallelization), which have become increasingly important for current and next-generation high performance computing.

By parallelizing the dense factorization, we removed the memory usage bottleneck that prevented PIPS from solving problems with a large number of first-stage variables. Now, PIPS is capable of solving very large real-life problems. This is an important problem for the integration of wind-generated power with the electricity grid, and this work is a necessary step forward in order to be able to solve it and similarly sized large-scale stochastic optimization problems.

Acknowledgments

We are grateful to Jack Poulson, the main developer of Elemental, for his guidance in both implementation and development of the factorization procedure, and to Peter Strazdins for informative discussions. This work was supported by the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[2] M. Benzi, G.H. Golub, and J. Liesen, *Numerical solution of saddle point problems*, ACTA NUMERICA 14 (2005), pp. 1–137.

[3] J.R. Birge and F. Louveaux, *Introduction to stochastic programming*, Springer-Verlag, New York, 1997.

[4] J.R. Birge and L. Qi, *Computing block-angular Karmarkar projections with applications to stochastic programming*, Manage. Sci. 34 (1988), pp. 1472–1479.

[5] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[6] J.R. Bunch and L. Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, Mathematics of Computation 31 (1977), pp. 163–179.

[7] J.R. Bunch and B.N. Parlett, *Direct methods for solving symmetric indefinite systems of linear equations*, SIAM Journal on Numerical Analysis 8 (1971), pp. 639–655.

[8] E.M. Constantinescu, V.M. Zavala, M. Rocklin, S. Lee, and M. Anitescu, *A computational framework for uncertainty quantification and stochastic optimization in unit commitment with wind power generation*, IEEE Transactions on Power Systems, in press (2010).

[9] E.M. Gertz and S.J. Wright, *Object-oriented software for quadratic programming*, ACM Transactions on Mathematical Software 29 (2003), pp. 58–81.

[10] G.H. Golub and C.F. Van Loan, *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*, 3rd ed., The Johns Hopkins University Press 1996.

[11] J. Gondzio and A. Grothey, *Parallel interior-point solver for structured quadratic programs: Application to financial planning problems*, Annals of Operations Research 152 (2007), pp. 319–339.

[12] J. Gondzio and A. Grothey, *Exploiting structure in parallel implementation of interior point methods for optimization*, Computational Management Science 6 (2009), pp. 135–160.

[13] J. Gondzio and R. Sarkissian, *Parallel interior point solver for structured linear programs*, Mathematical Programming 96 (2003), pp. 561–584.

[14] S. Mehrotra and M.G. Ozevin, *Decomposition based interior point methods for two-stage stochastic convex quadratic programs with recourse*, Oper. Res. 57 (2009), pp. 964–974.

[15] C.G. Petra and M. Anitescu, *A preconditioning technique for Schur complement systems arising in stochastic optimization*, Tech. rep., Preprint ANL/MCS-P1748-0510, Argonne National Laboratory,, 2010.

[16] J. Poulson, B. Marker, and R.A. van de Geijn, *Elemental: A new framework for distributed memory dense matrix computations (flame working note #44)*, Tech. rep., Institute for Computational Engineering and Sciences, The University of Texas at Austin, 2010.

[17] P.E. Strazdins and J.G. Lewis, *An efficient and stable method for parallel factorization of dense symmetric indefinite matrices*, The 5th International Conference and Exhibition on High Performance Computing in the Asia-Pacific Region (HPC Asia 2001) (2001).

[18] R.A. van de Geijn, *Using PLAPACK*, MIT Press 1997.

[19] V.M. Zavala, C.D. Laird, and L.T. Biegler, *Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems*, Chemical Engineering Science 63 (2008), pp. 4834–4845.