Computational Optimization and Applications manuscript No.
(will be inserted by the editor)

Parallel distributed-memory simplex for large-scale stochastic LP problems

Miles Lubin · J. A. Julian Hall · Cosmin G. Petra · Mihai Anitescu

Received: date / Accepted: date

Abstract We present a parallelization of the revised simplex method for large extensive forms of two-stage stochastic linear programming (LP) problems. These problems have been considered too large to solve with the simplex method; instead, decomposition approaches based on Benders decomposition or, more recently, interior-point methods are generally used. However, these approaches do not provide optimal basic solutions, which allow for efficient hot-starts (e.g., in a branch-and-bound context) and can provide important sensitivity information. Our approach exploits the dual block-angular structure of these problems inside the linear algebra of the revised simplex method in a manner suitable for high-performance distributed-memory clusters or supercomputers. While this paper focuses on stochastic LPs, the work is applicable to all problems with a dual block-angular structure. Our implementation is competitive in serial with highly efficient sparsity-exploiting simplex codes and achieves significant relative speed-ups when run in parallel. Additionally, very large problems with hundreds of millions of variables have been successfully solved to optimality. This is the largest-scale parallel sparsity-exploiting revised simplex implementation that has been developed to date and the first truly distributed solver. It is built on novel analysis of the linear algebra for dual block-angular LP problems when solved by using the revised simplex method and a novel parallel scheme for applying product-form updates.

Keywords Simplex method · Parallel computing · Stochastic optimization · Block-angular

Mathematics Subject Classification (2000) 90C05 · 90C15 · 68W10

Miles Lubin · Cosmin G. Petra · Mihai Anitescu
Mathematics and Computer Science Division, Argonne National Laboratory
Argonne, IL 60439-4844, USA
E-mail: mlubin@mcs.anl.gov, petra@mcs.anl.gov, anitescu@mcs.anl.gov

J. A. Julian Hall
School of Mathematics, University of Edinburgh
JCMB, King's Buildings, Edinburgh EH9 3JZ, UK
E-mail: J.A.J.Hall@ed.ac.uk

1 Introduction

In this paper, we present a parallel solution procedure based on the revised simplex method for linear programming (LP) problems with a special structure of the form

$$\begin{array}{lrcccccl} \text{minimize} & c_0^T x_0 & + & c_1^T x_1 & + & c_2^T x_2 & + \ldots + & c_N^T x_N \\ \text{subject to} & Ax_0 & & & & & & = b_0, \\ & T_1 x_0 & + & W_1 x_1 & & & & = b_1, \\ & T_2 x_0 & & & + & W_2 x_2 & & = b_2, \\ & \vdots & & & & & \ddots & \vdots \\ & T_N x_0 & & & & & + & W_N x_N = b_N, \\ & x_0 \geq 0, & & x_1 \geq 0, & & x_2 \geq 0, & \ldots, & x_N \geq 0. \end{array} \tag{1}$$

The structure of such problems is known as *dual block angular* or block angular with linking columns. This structure commonly arises in stochastic optimization as the *extensive form* or *deterministic equivalent* of two-stage stochastic linear programs with recourse when the underlying distribution is discrete or when a finite number of samples have been chosen as an approximation [4]. The dual problem to (1) has a *primal* or row-linked block-angular structure. Linear programs with block-angular structure, both primal and dual, occur in a wide array of applications, and this structure can also be identified within general LPs [2].

Borrowing the terminology from stochastic optimization, we say that the vector x_0 contains the *first-stage* variables and the vectors $x_1, \ldots, x_N$ the *second-stage* variables. The matrices $W_1, W_2, \ldots, W_N$ contain the coefficients of the second-stage constraints, and the matrices $T_1, T_2, \ldots, T_N$ those of the linking constraints. Each diagonal block corresponds to a *scenario*, a realization of a random variable. Although we adopt this specialist terminology, our work applies to any LP of the form (1).

Block-angular LPs are natural candidates for decomposition procedures that take advantage of their special structure. Such procedures are of interest both because they permit the solution of much larger problems than could be solved with general algorithms for unstructured problems and because they typically offer a natural scope for parallel computation, presenting an opportunity to significantly decrease the required time to solution. Our primary focus is on the latter motivation.

Existing parallel decomposition procedures for dual block-angular LPs are reviewed by Vladimirou and Zenios [32]. Subsequent to their review, Linderoth and Wright [24] developed an asynchronous approach combining ℓ_∞ trust regions with Benders decomposition on a large computational grid. Decomposition inside interior-point methods applied to the extensive form has been implemented in the state-of-the-art software package OOPS [14] as well as by some of the authors in PIPS [25].

These parallel decomposition approaches, based on either Benders decomposition or specialized linear algebra inside interior-point methods, have successfully demonstrated both parallel scalability on appropriately sized problems and capability to solve very large instances. However, each has algorithmic drawbacks. Neither of the approaches produces an optimal basis for the extensive form (1), which would allow for efficient hot-starts when solving a sequence of related LPs, whether in the context

of branch-and-bound or real-time control, and may also provide important sensitivity information.

While techniques exist to warm-start Benders-based approaches, such as in [24], as well as interior-point methods to a limited extent, in practice the simplex method is considered to be the most effective for solving sequences of related LPs. This intuition drove us to consider yet another decomposition approach, which we present in this paper, one in which the simplex method itself is applied to the extensive form (1) and its operations are parallelized according to the special structure of the problem. Conceptually, this is similar to the successful approach of linear algebra decomposition inside interior-point methods.

Exploiting primal block-angular structure in the context of the primal simplex method was considered in the 1960s by, for example, Bennett [3] and summarized by Lasdon [23, p. 340]. Kall [20] presented a similar approach in the context of stochastic LPs, and Strazicky [29] reported results from an implementation, both solving the dual of the stochastic LPs as primal-block angular programs. These works focused solely on the decrease in computational and storage requirements obtained by exploiting the structure. As serial algorithms, these specialized approaches have not been shown to perform better than efficient modern simplex codes for general LPs and so are considered unattractive and unnecessarily complicated as solution methods; see, for example, the discussion in [4, p. 226]. Only recently (with a notable exception of [28]) have the opportunities for parallelism been considered, and so far only in the context of the primal simplex method; Hall and Smith [18] have developed a high-performance shared-memory primal revised simplex solver for primal block-angular LP problems. To our knowledge, a successful parallelization of the revised simplex method for dual block-angular LP problems has not yet been published. We present here our design and implementation of a distributed-memory parallelization of both the primal and dual simplex methods for dual block-angular LPs.

2 Revised simplex method for general LPs

We review the primal and dual revised simplex methods for general LPs, primarily in order to establish our notation. We assume that the reader is familiar with the mathematical algorithms. Following this section, the computational components of the primal and dual algorithms will be treated in a unified manner to the extent possible.

A linear programming problem in standard form is

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0, \end{array} \tag{2}$$

where $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$. The matrix A in (2) usually contains columns of the identity corresponding to *logical* (slack) variables introduced to transform inequality constraints into equalities. The remaining columns of A correspond to *structural* (original) variables. It may be assumed that the matrix A is of full rank.

In the simplex method, the indices of variables are partitioned into sets $\mathscr{B}$, corresponding to m *basic* variables x_B, and $\mathscr{N}$, corresponding to $n-m$ nonbasic variables

x_N, such that the *basis matrix* B formed from the columns of A corresponding to $\mathscr{B}$ is nonsingular. The set $\mathscr{B}$ itself is conventionally referred to as the basis. The columns of A corresponding to $\mathscr{N}$ form the matrix N. The components of c corresponding to $\mathscr{B}$ and $\mathscr{N}$ are referred to as, respectively, the basic costs c_B and non-basic costs c_N.

When the simplex method is used, for a given partition $\{\mathscr{B}, \mathscr{N}\}$ the values of the primal variables are defined to be $x_N = 0$ and $x_B = B^{-1}b =: \hat{b}$, and the values of the nonbasic dual variables are defined to be $\hat{c}_N = c_N - N^T B^{-T} c_B$. The aim of the simplex method is to identify a partition characterized by primal feasibility ($\hat{b} \geq 0$) and dual feasibility ($\hat{c}_N \geq 0$). Such a partition corresponds to an optimal solution of (2).

2.1 Primal revised simplex

The computational components of the primal revised simplex method are illustrated in Figure 1. At the beginning of an iteration, it is assumed that the vector of reduced costs $\hat{c}_N$ and the vector $\hat{b}$ of values of the basic variables are known, that $\hat{b}$ is feasible (nonnegative), and that a representation of B^{-1} is available. The first operation is `CHUZC` (choose column), which scans the (weighted) reduced costs to determine a good candidate q to enter the basis. The *pivotal column* $\hat{a}_q$ is formed by using the representation of B^{-1} in an operation referred to as `FTRAN` (forward transformation).

The `CHUZR` (choose row) operation determines the variable to leave the basis, with p being used to denote the index of the row in which the leaving variable occurred, referred to as the *pivotal row*. The index of the leaving variable itself is denoted by p'. Once the indices q and p' have been interchanged between the sets $\mathscr{B}$ and $\mathscr{N}$, a *basis change* is said to have occurred. The vector $\hat{b}$ is then updated to correspond to the increase $\alpha = \hat{b}_p / \hat{a}_{pq}$ in x_q.

`CHUZC:` Scan $\hat{c}_N$ for a good candidate q to enter the basis.
`FTRAN:` Form the pivotal column $\hat{a}_q = B^{-1}a_q$, where a_q is column q of A.
`CHUZR:` Scan the ratios $\hat{b}_i / \hat{a}_{iq}$ for the row p of a good candidate to leave the basis.
Update $\hat{b} := \hat{b} - \alpha \hat{a}_q$, where $\alpha = \hat{b}_p / \hat{a}_{pq}$.
`BTRAN:` Form $\pi_p = B^{-T} e_p$.
`PRICE:` Form the pivotal row $\hat{a}_p = N^T \pi_p$.
Update reduced costs $\hat{c}_N := \hat{c}_N - \beta \hat{a}_p$, where $\beta = \hat{c}_q / \hat{a}_{pq}$.
`If` {growth in representation of B} `then`
`INVERT:` Form a new representation of B^{-1}.
`else`
`UPDATE:` Update the representation of B^{-1} corresponding to the basis change.
`end if`

Fig. 1 Operations in an iteration of the primal revised simplex method

Before the next iteration can be performed, one must update the reduced costs and obtain a representation of the new matrix B^{-1}. The reduced costs are updated by computing the *pivotal row* $\hat{a}_p^T = e_p^T B^{-1} N$ of the standard simplex tableau. This is obtained in two steps. First the vector $\pi_p^T = e_p^T B^{-1}$ is formed by using the representation of B^{-1} in an operation known as `BTRAN` (backward transformation), and then the vector $\hat{a}_p^T = \pi_p^T N$ of values in the pivotal row is formed. This sparse matrix-vector

product with N is referred to as PRICE. Once the reduced costs have been updated, the UPDATE operation modifies the representation of B^{-1} according to the basis change. Note that, periodically, it will generally be either more efficient or necessary for numerical stability to find a new representation of B^{-1} using the INVERT operation.

2.2 Dual revised simplex

While primal simplex has historically been more important, it is now widely accepted that the dual variant, the dual simplex method, generally has superior performance. Dual simplex is often the default algorithm in commercial solvers, and it is also used inside branch-and-bound algorithms.

Given an initial partition $\{\mathscr{B},\mathscr{N}\}$ and corresponding values for the basic and nonbasic primal and dual variables, the dual simplex method aims to find an optimal solution of (2) by maintaining dual feasibility and seeking primal feasibility. Thus optimality is achieved when the basic variables $\hat{b}$ are non-negative.

The computational components of the dual revised simplex method are illustrated in Figure 2, where the same data are assumed to be known at the beginning of an iteration. The first operation is CHUZR which scans the (weighted) basic variables to determine a good candidate to leave the basis, with p being used to denote the index of the row in which the leaving variable occurs. A candidate to leave the basis must have a negative primal value $\hat{b}_p$. The pivotal row $\hat{a}_p^T$ is formed via BTRAN and PRICE operations. The CHUZC operation determines the variable q to enter the basis. In order to update the vector $\hat{b}$, it is necessary to form the pivotal column $\hat{a}_q$ with an FTRAN operation.

```
CHUZR: Scan b̂ for the row p of a good candidate to leave the basis.
BTRAN: Form π_p = B^{-T} e_p.
PRICE: Form the pivotal row â_p^T = π_p^T N.
CHUZC: Scan the ratios ĉ_j/â_pj for a good candidate q to enter the basis.
       Update ĉ_N := ĉ_N − β â_p, where β = ĉ_q/â_pq.
FTRAN: Form the pivotal column â_q = B^{-1} a_q, where a_q is column q of A.
       Update b̂ := b̂ − α â_q, where α = b̂_p/â_pq.
If {growth in representation of B} then
   INVERT: Form a new representation of B^{-1}.
else
   UPDATE: Update the representation of B^{-1} corresponding to the basis change.
end if
```

Fig. 2 Operations in an iteration of the dual revised simplex method

2.3 Computational techniques

Today's highly efficient implementations of the revised simplex method are a product of over 60 years of refinements, both in the computational linear algebra and in the mathematical algorithm itself, which have increased its performance by many orders of magnitude. A reasonable treatment of the techniques required to achieve an

efficient implementation is beyond the scope of this paper; the reader is referred to the works of Koberstein [21] and Maros [27] for the necessary background. Our implementation includes these algorithmic refinements to the extent necessary in order to achieve serial execution times comparable with existing efficient solvers. This is necessary, of course, for any parallel speedup we obtain to have practical value. Our implementation largely follows that of Koberstein, and advanced computational techniques are discussed only when their implementation is nontrivial in the context of the parallel decomposition.

3 Parallel computing

This section provides the necessary background in the parallel-computing concepts relevant to the present work. A fuller and more general introduction to parallel computing is given by Grama et al. [15].

3.1 Parallel architectures

When classifying parallel architectures, an important distinction is between *distributed memory*, where each processor has its own local memory, and *shared memory*, where all processors have access to a common memory. We target distributed-memory architectures because of their availability and because they offer the potential to solve much larger problems. However, our parallel scheme could be implemented on either.

3.2 Speedup and scalability

In general, success in parallelization is measured in terms of speedup, the time required to solve a problem with more than one parallel process compared with the time required with a single process. The traditional goal is to achieve a speedup factor equal to the number of cores and/or nodes used. Such a factor is referred to as *linear speedup* and corresponds to a *parallel efficiency* of 100%, where parallel efficiency is defined as the percentage of the ideal linear speed-up obtained empirically. The increase in available processor cache per unit computation as the number of parallel processes is increased occasionally leads to the phenomenon of *superlinear speedup*.

3.3 MPI (Message Passing Interface)

In our implementation we use the MPI (Message Passing Interface) API [16]. MPI is a widely portable standard for implementing distributed-memory programs. The *message-passing* paradigm is such that all communication between *MPI processes*, typically physical processes at the operating-system level, must be explicitly requested by function calls. Our algorithm uses only the following three types of communication operations, all *collective* operations in which all MPI processes must participate.

- *Broadcast* (`MPI_Bcast`) is a simple operation in which data that are locally stored on only one process are broadcast to all.
- *Reduce* (`MPI_Allreduce`) combines data from each MPI process using a specified operation, and the result is returned to all MPI processes. One may use this, for example, to compute a sum to which each process contributes or to scan through values on each process and determine the maximum/minimum value and its location.
- *Gather* (`MPI_Allgather`) collects and concatenates contributions from all processes and distributes the result. For example, given P MPI processes, suppose a (row) vector x_p is stored in local memory in each process p. The gather operation can be used to form the combined vector $\begin{bmatrix} x_1 & x_2 & \dots & x_P \end{bmatrix}$ in local memory in each process.

The efficiency of these operations is determined by both their implementation and the physical communication network between nodes, known as an *interconnect*. The cost of communication depends on both the *latency* and the *bandwidth* of the interconnect. The former is the startup overhead for performing communication operations, and the latter is the rate of communication. A more detailed discussion of these issues is beyond the scope of this paper.

4 Linear algebra overview

Here we present from a mathematical point of view the specialized linear algebra required to solve, in parallel, systems of equations with basis matrices from dual block-angular LPs of the form (1). Precise statements of algorithms and implementation details are reserved for Section 5.

To discuss these linear algebra requirements, we represent the basis matrix B in the following *singly bordered block-diagonal* form,

$$B = \begin{bmatrix} W_1^B & & & T_1^B \\ & W_2^B & & T_2^B \\ & & \ddots & \vdots \\ & & W_N^B & T_N^B \\ & & & A^B \end{bmatrix}, \tag{3}$$

where the matrices W_i^B contain a subset, possibly empty, of the columns of the original matrix W_i, and similarly for T_i^B and A^B. This has been achieved by reordering the linking columns to the right and the rows of the first-stage constraints to the bottom, and it is done so that the block A^B will yield pivotal elements later than the W_i^B blocks when performing Gaussian elimination on B. Note that, within our implementation, this reordering is achieved implicitly.

In the linear algebra community, the scope for parallelism in solving linear systems with matrices of the form (3) is well known and has been used to facilitate parallel solution methods for general unsymmetric linear systems [8]. However, we are not aware of this form having been analyzed or exploited in the context of solving

dual block-angular LP problems. For completeness and to establish our notation, we present the essentials of this approach.

Let the dimensions of the W_i^B block be $m_i \times n_i^B$, where m_i is the number of rows in W_i and is fixed while n_i^B is the number of basic variables from the ith block and varies. Similarly let A^B be $m_0 \times n_0^B$, where m_0 is the number of rows in A and is fixed and n_0^B is the number of linking variables in the basis. Then T_i^B is $m_i \times n_0^B$.

The assumption that the basis matrix is nonsingular implies particular properties of the structure of a dual block-angular basis matrix. Specifically, the diagonal W_i^B blocks have full column rank and therefore are *tall* or square ($m_i \geq n_i^B$), and the A^B block has full row rank and therefore is *thin* or square ($m_0 \leq n_0^B$). The structure implied by these two observations is illustrated on the left of Figure 3.

A special case occurs when A^B is square, since it follows from the observations above and the fact that B is square that all diagonal blocks are square and nonsingular. In this case a solution procedure for linear systems with the basis matrix B is mathematically trivial since

$$\begin{bmatrix} W & T \\ & A \end{bmatrix}^{-1} = \begin{bmatrix} W^{-1} & -TA^{-1} \\ & A^{-1} \end{bmatrix},$$

for any square, nonsingular A and W. In our case, W would be block-diagonal. We refer to a matrix with this particular dual block-angular structure as being *trivial* because of the clear scope for parallelism when computing a representation of W^{-1}.

In the remainder of this section, we will prove the following result, which underlies the specialized solution procedure for singly bordered block-diagonal linear systems.

Result Any square, nonsingular matrix with singly bordered block-diagonal structure can be reduced to *trivial* form, up to a row permutation, by a sequence of invertible transformations that may be computed independently for each diagonal block.

Proof We may apply a sequence of Gauss transformations [13], together with row permutations, to eliminate the lower-triangular elements of W_i^B. Denote the sequence of operations as G_i, so that $G_i W_i^B = \begin{bmatrix} U_i \\ 0 \end{bmatrix}$, where U_i is square $(n_i^B \times n_i^B)$, upper triangular, and nonsingular because W_i^B has full rank. One may equivalently consider this as an LU factorization of a "tall" matrix,

$$P_i W_i^B = L_i U_i', U_i' = \begin{bmatrix} U_i \\ 0 \end{bmatrix}, \tag{4}$$

where L_i is square $(m_i \times m_i)$, lower triangular, and invertible, P_i is a permutation matrix, and $G_i = L_i^{-1} P_i$.

For notational purposes, we write the block form $G_i = \begin{bmatrix} X_i \\ Z_i \end{bmatrix}$, although these blocks are generally inaccessible, so that $X_i W_i^B = U_i$ and $Z_i W_i^B = 0$. We let $D :=$

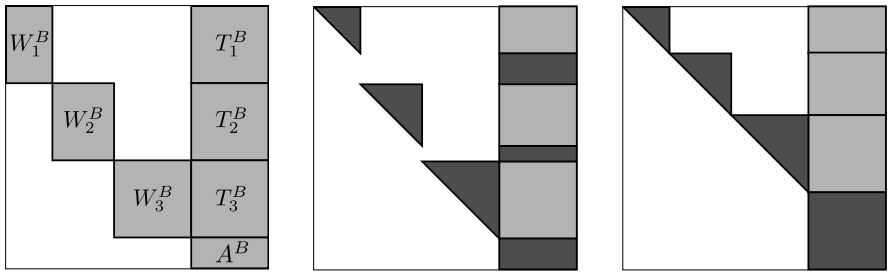

Fig. 3 Illustration of the structure of a dual block-angular basis matrix before and after elimination of lower-triangular elements in the diagonal W_i^B blocks. *Left*: A basis matrix with the linking columns on the right. The W_i^B blocks contain the basic columns of the corresponding W_i blocks of the constraint matrix. The W_i^B blocks in the basis cannot have more columns than rows. *Center*: The matrix after eliminating the lower-triangular elements in the W_i^B blocks. *Right*: The matrix with the linearly independent rows permuted to the bottom. The darkened elements indicate square, nonsingular blocks. This linear system is now trivial to solve.

$\mathrm{diag}(G_1, G_2, \ldots, G_n, I)$ be a block-diagonal matrix with the G_i transformations on the diagonal. Then,

$$DB = \begin{bmatrix} U_1 & & & X_1 T_1^B \\ 0 & & & Z_1 T_1^B \\ & U_2 & & X_2 T_2^B \\ & 0 & & Z_2 T_2^B \\ & & \ddots & \vdots \\ & & U_N & X_N T_N^B \\ & & 0 & Z_N T_N^B \\ & & & A^B \end{bmatrix}. \tag{5}$$

After permuting the rows corresponding to the $Z_i T_i^B$ blocks to the bottom, we obtain a *trivial* singly bordered block-diagonal matrix whose square, invertible, bottom-right block is

$$M := \begin{bmatrix} Z_1 T_1^B \\ Z_2 T_2^B \\ \vdots \\ Z_N T_N^B \\ A^B \end{bmatrix}. \tag{6}$$

□

This procedure is illustrated in Figure 3. If we then perform an *LU* factorization of the *first-stage block M*, this entire procedure could be viewed as forming an *LU* factorization of *B* through a restricted sequence of pivot choices. Note that the sparsity and numerical stability of this *LU* factorization are expected to be inferior to those of a structureless factorization resulting from unrestricted pivot choices. Thus it is important to explore fully the scope for maintaining sparsity and numerical stability within the scope of the structured *LU* factorization. This topic is discussed in Section 5.2.

5 Design and Implementation

Here we present the algorithmic design and implementation details of our code, PIPS-S, a new simplex code base for dual block-angular LPs written in C++. PIPS-S implements both the primal and dual revised simplex methods.

5.1 Distribution of data

In a distributed-memory algorithm, as described in Section 3, we must specify the distribution of data across parallel processes. We naturally expect to distribute the second-stage blocks, but the extent to which they are distributed and how the first stage is treated are important design decisions. We arrived at the following design after reviewing the data requirements of the parallel operations described in the subsequent sections.

Given a set of P MPI processes and $N \geq P$ scenarios or second-stage blocks, on initialization we assign each second-stage block to a single MPI process. All data, iterates, and computations relating to the first stage are duplicated in each process. The second-stage data (i.e., W_i, T_i, c_i, and b_i), iterates, and computations are only stored in and performed by their assigned process. If a scenario is *not* assigned to a process, this process stores *no* data pertaining to the scenario, not even the basic/nonbasic states of its variables. Thus, in terms of memory usage, the approach scales to an arbitrary number of scenarios.

5.2 Factorizing a dual block-angular basis matrix

In the `INVERT` step, one forms an invertible representation of the basis matrix B. This is performed in efficient sparsity-exploiting codes by forming a sparse LU factorization of the basis matrix. Our approach forms these factors implicitly and in parallel.

Sparse LU factorization procedures perform both row and column permutations in order to reduce the number of nonzero elements in the factors [7] while achieving acceptable numerical stability. Permutation matrices P and Q and triangular factors L and U are identified so that $PBQ = LU$.

In order to address issues of sparsity and numerical stability to the utmost within our structured LU factorization, the elimination of the lower-triangular elements in the W_i^B blocks, or equivalently, the "tall" LU factorization (4), must be modified. A pivot chosen within W_i^B may be locally optimal in terms of sparsity and numerical stability but have adverse consequences for fill-in or numerical growth in $G_i T_i^B$. This drawback is avoided by maintaining the active submatrix corresponding to the augmented matrix $\left[W_i^B \; T_i^B \right]$ during the factorization, even though the pivots are limited to the columns of W_i^B. This approach yields $P_i, L_i, U_i, X_i T_i^B$, and $Z_i T_i^B$, as previously defined, and Q_i, an $n_i^B \times n_i^B$ permutation matrix, which satisfy

$$P_i \left[W_i^B Q_i \; T_i^B \right] = L_i \begin{bmatrix} U_i & X_i T_i^B \\ 0 & Z_i T_i^B \end{bmatrix}. \tag{7}$$

As a benefit, the $X_i T_i^B$ and $Z_i T_i^B$ terms do not need to be formed separately by what would be multiple triangular solves with the L_i factor.

We implemented this factorization (7) by modifying the `CoinFactorization` C++ class, written by John Forrest, from the open-source CoinUtils[1] package. The methods used in the code are undocumented; however, we determined that it uses a Markowitz-type approach [26] such as that used by Suhl and Suhl [31]. After the factorization is complete, we extract the $X_i T_i^B$ and $Z_i T_i^B$ terms and store them for later use.

After the factorization is computed for each scenario or block i, we must also form and factor the first-stage block M (6). This is a square matrix of dimension $n_0^B \times n_0^B$. The size of n_0^B (the number of basic first-stage variables) and the density of the $Z_i T_i^B$ terms determine whether M should be treated as dense or sparse. In problems of interest, n_0^B ranges in the thousands and the $Z_i T_i^B$ terms are sufficiently sparse that treating M as sparse is orders of magnitude faster than treating it as dense; hence, this is the form used in our implementation. We duplicate M in the local memory of each MPI process and factorize it using the unmodified `CoinFactorization` routines. The nontrivial elements of the $Z_i T_i^B$ terms are collected and duplicated in each process by `MPI_Allgather`. We summarize the `INVERT` procedure in Figure 4.

The Markowitz approach used by `CoinFactorization` is considered numerically stable because the pivot order is determined dynamically, taking into account numerical cancellation. Simpler approaches that fix a sequence of pivots are known to fail in some cases; see [31], for example. Because our `INVERT` procedure couples Markowitz factorizations, we expect it to share many of the same numerical properties.

INVERT

1. Perform partial sparse Gaussian elimination on each $\begin{bmatrix} W_i^B & T_i^B \end{bmatrix}$, forming (7).
2. Collect and duplicate $Z_i T_i^B$ terms across processes.
3. In each process, form and factor the first-stage block M (6).

Fig. 4 Procedure for factoring a dual block-angular basis matrix. Step 1 may be performed in parallel for each second-stage block. In Step 2 the results are collected and duplicated in each parallel process by using `MPI_Allgather`, and in Step 3 each process factors its local copy of the first-stage block.

[1] https://projects.coin-or.org/CoinUtils

5.3 Solving linear systems with B

Given $r_0, r_1, \ldots, r_N$, suppose we wish to solve

$$B \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \\ x_0 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \\ r_0 \end{bmatrix} \tag{8}$$

for $x_0, x_1, \ldots, x_N$, where the partitioning of the vectors conforms to the structure of the dual block-angular basis matrix. The linear system (8) is solved by the procedure in Figure 5, which can be derived following the mathematical developments of Section 4.

Solving linear systems with B

1. Compute $\begin{bmatrix} X_i r_i \\ Z_i r_i \end{bmatrix} = G_i r_i = L_i^{-1} P_i r_i$, $i = 1, 2, \ldots, N$.
2. Collect $Z_i r_i$ terms and form $\hat{r} := \begin{bmatrix} Z_1 r_1 \\ Z_2 r_2 \\ \vdots \\ Z_N r_N \\ r_0 \end{bmatrix}$ in local memory in each process.
3. In each process, solve the first-stage system $M x_0 = \hat{r}$.
4. Form $q_i := X_i r_i - (X_i T_i^B) x_0$, $i = 1, \ldots, N$.
5. Compute $x_i = Q_i U_i^{-1} q_i$, $i = 1, \ldots, N$.

Fig. 5 Procedure to solve linear systems of the form (8) with a dual block-angular basis matrix B for arbitrary right-hand sides. Parallelism may be exploited within Steps 1, 4, and 5. Step 2 is a collective communication operation performed by `MPI_Allgather`, and Step 3 is a calculation duplicated in all processes.

Efficient implementations must take advantage of sparsity, not only in the matrix factors, but also in the right-hand side vectors, exploiting *hyper-sparsity* [17], when applicable. We reused the routines from `CoinFactorization` for solves with the factors. These include hyper-sparse solution routines based on the original work in [11], where a symbolic phase computes the sparsity pattern of the solution vector and then a numerical phase computes only the nonzero values in the solution.

The solution procedure exhibits parallelism in the calculations that are performed per block, that is, in Steps 1, 4, and 5. An `MPI_Allgather` communication operation is required at Step 2, and the first-stage calculation in Step 3 is duplicated, in serial in each process.

This computational pattern changes somewhat when the right-hand side vector is structured, as is the case in the most common FTRAN step, which computes the pivotal column $\hat{a}_q = B^{-1}a_q$, where a_q is a column of the constraint matrix. If the entering column in FTRAN is from second-stage block j, then r_i has nonzero elements only for $i = j$. Since Step 1 is then trivial for all other second-stage blocks, it becomes a serial bottleneck. Fortunately we can expect Step 1 to be relatively inexpensive because the right-hand side will be a column from W_j, which should have few non-zero elements. Additionally, Step 2 becomes a broadcast instead of a gather operation.

5.4 Solving linear systems with B^T

Given $r_0, r_1, \ldots, r_N$, suppose we wish to solve

$$B^T \begin{bmatrix} \pi_1 \\ \pi_2 \\ \vdots \\ \pi_N \\ \pi_0 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \\ r_0 \end{bmatrix} \tag{9}$$

for $\pi_0, \pi_1, \ldots, \pi_N$, where the partitioning of the vectors conforms to the structure of the dual block-angular basis matrix. The linear system (9) is solved by the procedure in Figure 6.

Solving linear systems with B^T

1. Compute $\alpha_i := U_i^{-T} Q_i^T r_i$, $i = 1, 2, \ldots, N$.
2. Form $q_i := (X_i T_i^B)^T \alpha_i$, $i = 1, 2, \ldots, N$.
3. Reduce q_i's, forming $q_0 := \sum_{i=1}^N q_i$ in local memory in each process.
4. In each process, solve the first-stage system $M^T \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_N \\ \pi_0 \end{bmatrix} = r_0 - q_0$.
5. Compute $\pi_i = G_i^T \begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix} = P_i^T L_i^{-T} \begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix}$, $i = 1, 2, \ldots, N$.

Fig. 6 Procedure to solve linear systems of the form (9) with a dual block-angular basis B for arbitrary right-hand sides. Parallelism may be exploited within Steps 1, 2, and 5. Step 3 is a collective communication operation performed by MPI_Allreduce, and Step 4 is a calculation duplicated in all processes. Intermediate vectors β_i have $m_i - n_i^B$ elements, the difference between the number of rows and columns of the W_i^B block.

Hyper-sparsity in solves with the factors is handled as previously described. Note that at Step 3 we must sum an arbitrary number of sparse vectors across MPI processes. This step is performed by using a dense buffer and `MPI_Allreduce`, then passing through the result to build an index of the nonzero elements in q_0. The overhead of this pass-through is small compared with the overhead of the communication operation itself.

As for linear systems with B, there is an important special case for the most common `BTRAN` step in which the right-hand side to `BTRAN` is entirely zero except for a unit element in the position corresponding to the row/variable that has been selected to leave the basis. If a second-stage variable has been chosen to leave the basis, Steps 1 and 2 become trivial for all but the corresponding second-stage block. Fortunately, as before, we can expect these steps to be relatively inexpensive because of the sparsity of the right-hand sides. Similarly, Step 3 may be implemented as a broadcast operation instead of a reduce operation.

5.5 Matrix-vector product with the block-angular constraint matrix

The `PRICE` operation, which forms the pivotal row at every iteration, is a matrix-vector product between the transpose of the nonbasic columns of the constraint matrix and the result of the `BTRAN` operation. Let W_i^N, T_i^N, and A^N be the nonbasic columns of W_i, T_i, and A, respectively. Then, given $\pi_1, \pi_2, \ldots, \pi_N, \pi_0$, we wish to compute

$$\begin{bmatrix} W_1^N & & & T_1^N \\ & W_2^N & & T_2^N \\ & & \ddots & \vdots \\ & & W_N^N & T_N^N \\ & & & A^N \end{bmatrix}^T \begin{bmatrix} \pi_1 \\ \pi_2 \\ \vdots \\ \pi_N \\ \pi_0 \end{bmatrix} = \begin{bmatrix} (W_1^N)^T \pi_1 \\ (W_2^N)^T \pi_2 \\ \vdots \\ (W_N^N)^T \pi_N \\ (A^N)^T \pi_0 + \sum_{i=1}^N (T_i^N)^T \pi_i \end{bmatrix}. \tag{10}$$

The procedure to compute (10) in parallel is immediately evident from the right-hand side. The terms involving W_i^N and T_i^N may be computed independently by column-wise or row-wise procedures, depending on the sparsity of each π_i vector. Then a reduce operation is required to form $\sum_{i=1}^N (T_i^N)^T \pi_i$.

5.6 Updating the inverse of the dual block-angular basis matrix

Multiple approaches exist for updating the invertible representation of the basis matrix following each iteration, in which a single column of the basis matrix is replaced. Updating the *LU* factors is generally considered the most efficient, in terms of both speed and numerical stability. Such an approach was considered but not implemented. We provide some brief thoughts on the issue, both for the interested reader and to indicate the difficulty of the task given the specialized structure of the representation of the basis inverse. The factors from the second-stage block could be updated by extending the ideas of Forrest and Tomlin [10] and Suhl and Suhl [30]. The first-stage matrix M, however, is significantly more difficult. With a single basis update, large

blocks of elements in M could be changed, and the dimension of M could potentially increase or decrease by one. Updating this block with external Schur-complement-type updates as developed in [5] is one possibility.

Given the complexity of updating the basis factors, we first implemented product-form updates, which do not require any modification of the factors. While this approach generally has larger storage requirements and can exhibit numerical instability with long sequences of updates, these downsides are mitigated by invoking `INVERT` more frequently. Early reinversion is triggered if numerical instability is detected. Empirically, a reinversion frequency on the order of 100 iterations performed well on the problems tested, although the optimal choice for this number depends on both the size and numerical difficulty of a given problem. Product-form updates performed sufficiently well in our tests that we decided to not implement more complex procedures at this time. We now review briefly the product-form update and then describe our parallel implementation.

5.6.1 Product-form update

Suppose that a column a_q replaces the pth column of the basis matrix B in the context of a general LP problem. One may verify that the new basis matrix $\overline{B}$ may be expressed as $\overline{B} = B(I + (\hat{a}_q - e_p)e_p^T)$, where $\hat{a}_q = B^{-1}a_q$ is the pivotal column computed in the `FTRAN` step. Let $E = (I + (\hat{a}_q - e_p)e_p^T)^{-1}$. Now, $\overline{B}^{-1} = EB^{-1}$. The *elementary transformation* ("eta") matrix E may be represented as $E = (I + \eta e_p^T)$, where η is a vector derived from $\hat{a}_q$. Given a right-hand side x, we have

$$Ex = (x + x_p\eta).$$

That is, a multiple of the η vector is added, and the multiple is determined by the element in the pivotal index p. Additionally,

$$E^T x = x + (\eta^T x)e_p.$$

That is, the dot product between η and x is added to the element in the pivotal index. After K iterations, one obtains a representation of the basis inverse of the form $\overline{B}^{-1} = E_K \ldots E_2 E_1 B^{-1}$, and in transpose form, $\overline{B}^{-T} = B^{-T} E_1^T E_2^T \ldots E_K^T$. This is the essence of the product-form update, first introduced in [6].

We introduce a small variation of this procedure for dual block-angular bases. An implicit permutation is applied with each eta matrix in order to preserve the structure of the basis (an entering variable is placed at the end of its respective block). With each η vector, we store both an entering index and a leaving index. We refer to the leaving index as the pivotal index.

5.6.2 Product-form update for parallel computation

Consider the requirements for applying an eta matrix during `FTRAN` to a structured right-hand side vector with components distributed by block. If the pivotal index for the eta matrix is in a second-stage block, then the element in the pivotal index is

stored only in one MPI process, but it is needed by all and so must be broadcast. Following this approach, one would need to perform a communication operation for each eta matrix applied. The overhead of these broadcasts, which are synchronous bottlenecks, would likely be significantly larger than the cost of the floating-point operations themselves. Instead, we implemented a procedure that, at the cost of a small amount of extra computation, requires only one communication operation to apply an arbitrary number of eta matrices. The procedure may be considered a parallel product-form update. The essence of the procedure has been used by Hall in previous work, although the details below have not been published before.

Every MPI process stores all of the pivotal components of each η vector, regardless of which block they belong to, in a rectangular array. After a solve with B is performed (as in Figure 5), `MPI_Allgather` is called to collect the elements of the solution vector in pivotal positions. Given all the pivotal elements in both the η vectors and the solution vector, each process can proceed to apply the eta matrices restricted to only these elements in order to compute the sequence of multipliers. Given these multipliers, the eta matrices can then be applied to the local blocks of the entire right-hand side. The communication cost of this approach is the single communication to collect the pivotal elements in the right-hand side, as well as the cost of maintaining the pivotal components of the η vectors.

The rectangular array of pivotal components of each η vector also serves to apply eta matrices in the `BTRAN` operation efficiently when the right-hand side is the unit vector with a single nontrivial element in the leaving index; see Hall and McKinnon [17]. This operation requires no communication once the leaving index is known to all processes and after the rectangular array has been updated correspondingly.

5.7 Algorithmic refinements

In order to improve the iteration count and numerical stability, it is valuable to use a weighted edge-selection strategy in `CHUZC` (primal) and `CHUZR` (dual). In PIPS-S, the former operation uses the DEVEX scheme [19], and the latter is based on the exact steepest-edge variant of Forrest and Goldfarb [9] described in [22]. Algorithmically, weighted edge selection is a simple operation with a straightforward parallelization. Each process scans through its local variables (both the local second-stage blocks and the first-stage block) and finds the largest (weighted) local infeasibility. An `MPI_Allreduce` communication operation then determines the largest value among all processes and returns to all processes its corresponding index.

Further contributions to improved iteration count and numerical stability come from the use of a two-pass EXPAND [12] ratio test, together with the shifting techniques described by [27] and [22]. We implement the two-pass ratio test in its canonical form, inserting the necessary `MPI_Allreduce` operations after each pass.

5.8 Updating iterates

After the pivotal column and row are chosen and computed, iterate values and edge weights are updated by using the standard formulas to reflect the basis change. We

note here that each MPI process already has the sections of the pivotal row and column corresponding to its local variables (both the local second-stage blocks and the first-stage block). The only communication required is a broadcast of the primal and dual step lengths by the processes that own the leaving and entering variables, respectively. If edge weights are used, the pivotal element in the simplex tableau and a dot product (reduce) are usually required as well, if not an additional linear solve.

6 Numerical experiments

Numerical experiments with PIPS-S were conducted on two distributed-memory architectures available at Argonne National Laboratory. *Fusion* is a 320-node cluster with an InfiniBand QDR interconnect; each node has two 2.6 GHz Xeon processors (total 8 cores). Most nodes have 36 GB of RAM, while a small number offer 96 GB of RAM. A single node of Fusion is comparable to a high-performance workstation. *Intrepid* is a Blue Gene/P (BG/P) supercomputer with 40,960 nodes with a custom high-performance interconnect. Each BG/P node, much less powerful in comparison, has a quad-core 850 MHz PowerPC processor with 2 GB of RAM.

Using the stochastic LP test problems described in Section 6.1, we present results from problems of three different scales by varying the number of scenarios in the test problems. The smallest instances considered (Section 6.2) are those that could be solved on a modern desktop computer. The next set of instances (Section 6.3) are large-scale instances that demand the use of the Fusion nodes with 96 GB of RAM to solve in serial. The largest instances considered (Section 6.4) would require up to 1 TB of RAM to solve in serial; we use the Blue Gene/P system for these. At the first two scales, we compare our solver, PIPS-S, with the highly efficient, open-source, serial simplex code Clp[2]. At the largest scale, no comparison is possible. These experiments aim to demonstrate both the scalability and capability of PIPS-S.

In all experiments, primal and dual feasibility tolerances of 10^{-6} were used. It was verified that the optimal objective values reported in each run were equal for each instance. A reinversion frequency of 150 iterations was used in PIPS-S, with earlier reinversion triggered by numerical stability tests. Presolve and internal rescaling, important features of LP solvers that have not yet been implemented in PIPS-S, were disabled in Clp to produce a fair comparison. Otherwise, default options were used. Commercial solvers were unavailable for testing on the Fusion cluster because of licensing constraints. The number of cores reported used corresponds to the total number of MPI processes.

6.1 Test problems

We take two two-stage stochastic LP test problems from the stochastic programming literature as well as two stochastic power-grid problems of interest to the authors. Table 1 lists the problems and their dimensions. “Storm” and “SSN” were used by

[2] https://projects.coin-or.org/Clp

Table 1 Dimensions of stochastic LP test problems.

Test	1st Stage		2nd-Stage Scenario		Nonzero Elements		
Problem	Vars.	Cons.	Vars.	Cons.	A	W_i	T_i
Storm	121	185	1,259	528	696	3,220	121
SSN	89	1	706	175	89	2,284	89
UC12	3,132	0	56,532	59,436	0	163,839	3,132
UC24	6,264	0	113,064	118,872	0	327,939	6,264

Linderoth and Wright [24] and are publicly available in SMPS format[3]. We refer the reader to [24] for a description of these problems. Scenarios were generated by simple Monte-Carlo sampling.

The "UC12" and "UC24" problems are *stochastic unit commitment* problems developed at Argonne National Laboratory by Victor Zavala. See [25] for details of a *stochastic economic dispatch* model with similar structure. The problems aim to choose optimal on/off schedules for generators on the power grid of the state of Illinois over a 12-hour and 24-hour horizon, respectively. The stochasticity considered is that of the availability of wind-generated electricity, which can be highly variable. In practice each scenario would be the result of a weather simulation. For testing purposes only, we generate these scenarios by normal perturbations. Each second-stage scenario incorporates (direct-current) transmission constraints corresponding to the physical power grid, and so these scenarios become very large. We consider the LP relaxations, in the context of what would be required in order to solve these problems using a branch-and-bound approach.

In these test problems, only the right-hand side vectors $b_1, b_2, \ldots, b_N$ vary per scenario; the matrices T_i and W_i are identical for each scenario. This special structure, common in practice, is not currently exploited by PIPS-S.

6.2 Solution *from scratch*

We first consider instances that could be solved on a modern desktop *from scratch*, that is, from an all-slack starting basis. These instances, with 1-10 million total variables, could be considered large scale because they take many hours to solve in serial, although they are well within the capabilities of modern simplex codes. These tests serve both to compare the serial efficiency of PIPS-S with that of a modern simplex code and to investigate the potential for parallel speedup on problems of this size. The results are presented in Table 2.

We observe that Clp is faster than PIPS-S in serial on all instances; however, the total number of iterations performed by PIPS-S is consistent with the number of iterations performed by Clp, empirically confirming our implementation of pricing strategies. Significant parallel speedups are observed in all cases, and PIPS-S is 5 and 8 times faster than Clp for SSN and Storm respectively when using four nodes (32 cores). Parallel speedups obtained on the UC12 and UC24 instances are smaller,

[3] http://pages.cs.wisc.edu/~swright/stochastic/sampling

Table 2 Solves from scratch (all-slack basis) using dual simplex. Storm instance has 8,192 scenarios, 10,313,849 variables, and 4,325,561 constraints. SSN instance has 8,192 scenarios, 5,783,651 variables, and 1,433,601 constraints. UC12 instance has 32 scenarios, 1,812,156 variables, and 1,901,952 constraints. UC24 instance has 16 scenarios, 1,815,288 variables, and 1,901,952 constraints. Runs performed on nodes of the Fusion cluster.

Test Problem	Solver	Nodes	Cores	Iterations	Solution Time (Sec.)	Iter./ Sec.
Storm	Clp	1	1	6,706,401	133,047	50.4
	PIPS-S	1	1	6,353,593	385,825	16.5
	"	1	4	6,357,445	108,517	58.6
	"	1	8	6,343,352	52,948	119.8
	"	2	16	6,351,493	28,288	224.5
	"	4	32	6,347,643	15,667	405.2
SSN	Clp	1	1	1,175,282	12,619	93.1
	PIPS-S	1	1	1,025,279	58,425	17.5
	"	1	4	1,062,776	16,511	64.4
	"	1	8	1,055,422	7,788	135.5
	"	2	16	1,051,860	3,865	272.1
	"	4	32	1,046,840	1,931	542.1
UC12	Clp	1	1	2,474,175	39,722	62.3
	PIPS-S	1	1	1,968,400	236,219	8.3
	"	1	4	2,044,673	86,834	23.5
	"	1	8	1,987,608	39,033	50.9
	"	2	16	2,063,507	27,902	74.0
	"	4	32	2,036,306	16,255	125.3
UC24	Clp	1	1	2,441,374	41,708	58.5
	PIPS-S	1	1	2,142,962	543,272	3.9
	"	1	4	2,204,729	182,370	12.1
	"	1	8	2,253,199	101,893	22.1
	"	2	16	2,270,728	60,887	37.3

possibly because of the smaller number of scenarios and the larger dimensions of the first stage.

6.3 Larger instances with advanced starts

We next consider larger instances with 20-40 million total variables. The high-memory nodes of the Fusion cluster with 96 GB of RAM were required for these tests. Given the long times to solution for the smaller instances solved in the previous section, it is impractical to solve these larger instances from scratch. Instead, we consider using advanced or near-optimal starting bases in two different contexts. In Section 6.3.1 we generate starting bases by taking advantage of the structure of the problem. In Section 6.3.2 we attempt to simulate the problems solved by dual simplex inside a branch-and-bound node.

6.3.1 Advanced starts from exploiting structure

We provide a minimal description of an approach we developed for generating advanced starting bases to solve extensive-form LPs. Given an extensive-form LP with a set of scenarios $\{1,\dots,N\}$, fix a subset of the scenarios $S \subset \{1,\dots,N\}$, and solve the extensive-form LP corresponding to S. Given the first-stage solution $\hat{x}_0$ for this subset of scenarios, for $i \in \{1,\dots,N\} \setminus S$ solve the second-stage problem

$$\min c_i x_i \text{ s.t. } W_i x_i = b_i - T_i \hat{x}_0,\ x_i \geq 0. \qquad \text{LP2}(i,\ \hat{x}_0)$$

Then, concatenate the optimal bases from the extensive-form LP for S with those of LP2(i, $\hat{x}_0$). For some i, LP2(i, $\hat{x}_0$) may be infeasible. If this situation occurs, one may take a slack basis or, if W_i does not vary per scenario, use the optimal basis from another subproblem. It may be shown that this procedure produces a valid (but potentially infeasible) basis for the original extensive-form LP. If all LP2(i, $\hat{x}_0$) problems are feasible, then this basis is in fact primal feasible (and so is appropriate for primal simplex). We call this procedure *stochastic basis bootstrapping*, because one uses the solution to a smaller problem to "bootstrap" or warm-start the solution of a larger problem; this has no relationship to the common usage of the term bootstrapping in the field of statistics. A more detailed description and investigation of this procedure are warranted but are beyond the scope of this paper.

In the following tests, we consider these advanced starting bases as given. The time to generate the starting basis is not included in the execution times; these two times are typically of the same order of magnitude. Results with the Storm and SSN problems are given in Table 3. These two problems have the property that all second-stage subproblems are feasible, and so the starting basis is primal feasible. We apply this approach to the UC12 problem in Section 6.4.

Clp remains faster in serial than PIPS-S on these instances, although by a smaller factor than before. The parallel scalability of PIPS-S is almost ideal (>90% parallel efficiency) up to 4 nodes (32 cores) and continues to scale well up to 16 nodes (128 cores). Scaling from 16 nodes to 32 nodes is poor. On 16 nodes, the iteration speed of PIPS-S is nearly 100x that of Clp for Storm and 70x that of Clp for SSN. A curious instance of *superlinear* scaling is observed within a single node. This could be caused by properties of the memory hierarchy (e.g. processor cache size).

We observe that the advanced starting bases are indeed near-optimal, particularly for Storm, where approximately ten thousand iterations are required to solve a problem with 41 million variables.

6.3.2 Dual simplex inside branch and bound

The dual simplex method is generally used inside branch-and-bound algorithms because the optimal basis obtained at the parent node remains dual feasible after variable bounds are changed.

For the UC12 and UC24 problems, we obtained optimal bases for the LP relaxation and then selected three binary variables to fix to zero. Because of the nature of the model, the subproblems resulting from fixing variables to one typically require very few iterations, and these problems are not considered because the initial `INVERT`

Table 3 Solves from advanced starting bases using primal simplex. Both instances have 32,768 scenarios. Starting bases were generated by using a subset of 16,386 scenarios. Storm instance has 41,255,033 variables and 17,301,689 constraints. SSN instance has 23,134,297 variables and 5,734,401 constraints. Runs performed on nodes of Fusion cluster. Asterisk indicates that high-memory nodes were required.

Test Problem	Solver	Nodes	Cores	Iterations	Solution Time (Sec.)	Iter./ Sec.
Storm	Clp	1*	1	16,247	7537	2.2
	PIPS-S	1*	1	9,026	7184	1.3
	"	1*	4	6,598	662	10.0
	"	1*	8	10,899	486	22.4
	"	2*	16	6,519	137	47.6
	"	4	32	5,776	61.5	93.9
	"	8	64	7,509	47.3	158.8
	"	16	128	7,691	35.5	216.6
	"	32	256	6,572	25.2	260.4
SSN	Clp	1*	1	99,303	50,737	2.0
	PIPS-S	1*	1	353,354	427,648	0.8
	"	1*	4	239,882	58,621	4.1
	"	1*	8	235,039	22,485	10.5
	"	2	16	219,050	9,550	22.9
	"	4	32	193,565	4,134	46.8
	"	8	64	219,560	2,365	92.8
	"	16	128	212,269	1,481	143.3
	"	32	256	200,979	1,117	180.0

operation, which may not required in a real branch-and-bound setting, dominates their execution time. For each of the three variables chosen, all subproblems were (primal) feasible. These subproblems are intended to simulate the work performed inside a branch-and-bound node, with the aim of both investigating parallel scalability and indicating how many simplex iterations may be needed for such problems. The results are presented in Table 4.

For UC12 and UC24, we obtain 71% and 81% scaling efficiency, respectively, up to 4 nodes (32 cores) and approximately 50% scaling efficiency on both instances up to 16 nodes (128 cores). On 16 nodes, the iteration speed of PIPS-S is slightly over 25x that of Clp. PIPS-S requires fewer iterations than Clp on these problems, although we do not claim any general advantage. Comparing solution times, we observe relative speedups of over 100x in some cases.

These results, while inconclusive because only three subproblems were considered, suggest that with sufficient resources, branch-and-bound subproblems for these instances may be solved in minutes instead of hours. With these same resources, a parallel branch-and-bound approach is also possible; however, for instances of the sizes considered it will likely be necessary to distribute the subproblems to some extent because of memory constraints.

Table 4 Iteration counts and solution times for three reoptimization problems intended to simulate the work performed at a branch-and-bound node. UC12 instance has 512 scenarios, 28,947,516 variables, and 30,431,232 constraints. UC24 instance has 256 scenarios, 28,950,648 variables, and 30,431,232 constraints. Runs performed on Fusion cluster. Asterisk indicates that high-memory nodes were required.

Test Problem	Solver	Nodes	Cores	Iterations	Solution Time (Sec.)	Avg. Iter./Sec
UC12	Clp	1*	1	10,370/13,495/5,888	15,205/19,782/ 7,022	0.73
	PIPS-S	1*	1	5,030/ 6,734/4,454	14,033/20,762/12,749	0.34
	"	1*	8	5,031/ 6,793/4,454	1,963/ 2,955/ 1,788	2.5
	"	2*	16	5,031/ 6,794/4,451	1,015/ 1,537/ 929	4.7
	"	4	32	5,031/ 6,738/4,454	548/ 810/ 503	8.8
	"	8	64	5,031/ 6,738/4,454	321/ 476/ 300	14.9
	"	16	128	5,031/ 6,793/4,454	226/ 346/ 214	20.9
	"	32	256	5,031/ 6,794/4,454	180/ 296/ 169	25.8
UC24	Clp	1*	1	5,813/ 9,386/2,909	6,818/10,815/ 3,280	0.87
	PIPS-S	1*	1	3,035/ 2,240/2,272	8,031/ 6,049/ 6,841	0.36
	"	1*	8	3,035/ 2,230/2,271	1,247/ 855/ 1,043	2.4
	"	2*	16	3,035/ 2,230/2,272	675/ 487/ 565	4.4
	"	4	32	3,035/ 2,230/2,271	358/ 257/ 300	8.2
	"	8	64	3,035/ 2,230/2,270	198/ 143/ 170	14.8
	"	16	128	3,035/ 2,230/2,272	125/ 90/ 111	23.2
	"	32	256	3,035/ 2,230/2,272	101/ 71/ 92	28.7

6.4 Very large instance

We report here on the solution of a very large instance, UC12 with 8,192 scenarios. This instance has 463,113,276 variables and 486,899,712 constraints. An advanced starting basis was generated from 4,096 scenarios, not included in the execution time. Results are reported in Table 5 for a range of node counts using a Blue Gene/P system. This problem requires approximately 1 TB of RAM to solve, requiring a minimum of 512 Blue Gene nodes; however, results are only available for runs with 1,024 nodes or more because of execution time limits. While scaling performance is poor on these large numbers of nodes, this test demonstrates the capability of PIPS-S to solve instances considered far too large to solve today with commercial solvers.

Table 5 Iteration counts and solution times for UC12 with 8,192 scenarios. Starting basis was generated by using a subset of 4,096 scenarios. Runs performed on *Intrepid* Blue Gene/P system using PIPS-S.

Nodes	Cores	Iterations	Solution Time (Hr.)	Iter./ Sec.
1,024	2,048	82,638	6.14	3.74
2,048	4,096	75,732	5.03	4.18
4,096	8,192	86,439	4.67	5.14

6.5 Performance analysis

For a given operation in the simplex algorithm, a simple model of its *ideal* execution time on P processes is

$$\frac{1}{P}\sum_{i=1}^{P} t_p + t_0, \tag{11}$$

where t_p is the time spent by process p on its local second-stage calculations and t_0 is the time spent on first-stage calculations. A more realistic, but still imperfect, model is

$$\max_p\{t_p\} + c + t_0, \tag{12}$$

where c is the cost of communication.

The magnitude of the *load imbalance*, defined as $\max_p\{t_p\} - \frac{1}{P}\sum_{i=1}^{P} t_p$, and the cost of communication explain the deviation between the observed execution time and the ideal execution time, whereas the magnitude of the serial bottleneck t_0 determines the algorithmic limit of scalability according to Amdahl's law [1]. Evaluating the relative impacts of these three factors; namely, load imbalance, communication cost, and serial bottlenecks; on a given instance provides valuable insight into the empirical performance of PIPS-S.

The `PRICE` operation (Section 5.5), chosen for its relative simplicity, was instrumented to calculate the quantities $t_0, t_1, \ldots, t_P$ and c explicitly. Table 6 displays the magnitudes of the three performance factors identified for the problem instances from Section 6.3 (the "larger" instances).

We observe that for all instances, the cost of the first-stage calculations is small compared with the other factors. For a sufficiently large number of scenarios, this property will always hold for stochastic LPs, but it may not hold for block-angular LPs obtained by using the hypergraph partitioning of [2].

Communication cost is significant in all cases, particularly when more than 4 or 8 nodes are used. Communication cost increases with the number of first-stage variables (from SSN to Storm to UC12 to UC24), indicating the effects of bandwidth. Unsurprisingly, the communication cost increases with the number of nodes.

Load imbalance in `PRICE` is due primarily to exploitation of hyper-sparsity, and UC12 and UC24, potentially because they contain network-like structure, exhibit this property to a greater extent than do Storm and SSN. Interestingly, the load imbalance does not increase on an absolute scale with the number of nodes, although it becomes a larger proportion of the total execution time.

Despite communication overhead and load imbalance, which we had suspected to be insurmountable, significant speedups are possible, as evidenced by the results presented here. Advances in hardware have brought communication times across nodes to as little as tens of microseconds using high-performance interconnects such as InfiniBand. We note that a shared-memory implementation of our approach would have the potential to address load-balancing issues to a greater extent than the present distributed-memory implementation.

Table 6 Inefficiencies in PRICE operation on instances from Sections 6.3.1 and 6.3.2. Times, given in microseconds (μs), are averages over all iterations. Load imbalance is defined as the difference between the maximum and average execution time in the second-stage calculations per MPI process. Total time in PRICE per iteration is given on the right.

Test Problem	Nodes	Cores	Load Imbal. (μs)	Comm. Cost (μs)	Serial Bottleneck (μs)	Total Time/Iter. (μs)
Storm	1	1	0	0	1.0	13,243
	1	8	88	33	0.8	1,635
	2	16	40	68	0.9	856
	4	32	25	105	0.9	512
	8	64	26	112	1.0	326
	16	128	11	102	0.9	205
	32	256	34	253	0.8	333
SSN	1	1	0	0	0.8	2,229
	1	8	18	23	0.8	305
	2	16	25	54	0.8	203
	4	32	14	68	0.7	133
	8	64	12	65	0.7	100
	16	128	10	87	0.6	106
	32	256	8	122	0.6	135
UC12	1	1	0	0	6.8	24,291
	1	8	510	183	6.0	4,785
	2	16	554	274	6.0	2,879
	4	32	563	327	6.0	1,921
	8	64	542	355	6.0	1,418
	16	128	523	547	6.0	1,335
	32	256	519	668	5.8	1,323
UC24	1	1	0	0	11.0	28,890
	1	8	553	259	9.8	5,983
	2	16	543	315	9.7	3,436
	4	32	551	386	9.6	2,248
	8	64	509	367	9.5	1,536
	16	128	538	718	9.5	1,593
	32	256	584	1413	9.5	2,170

7 Conclusions

We have developed the linear algebra techniques necessary to exploit the dual block-angular structure of an LP problem inside the revised simplex method and a technique for applying product form updates efficiently in a parallel context. Using these advances, we have demonstrated the potential for significant parallel speedups. The approach is most effective on large instances that might otherwise be considered very difficult to solve using the simplex algorithm. The number of simplex iterations required to solve such problems is greatly reduced by using advanced starting bases generated by taking advantage of the structure of the problem. The optimal bases generated may be used to efficiently hot-start the solution of related problems, which often occur in real-time control or branch-and-bound approaches. This work paves

the path for efficiently solving stochastic programming problems in these two contexts.

Acknowledgements We acknowledge John Forrest and all other contributors for the open-source CoinUtils library which is used throughout the implementation. This work was supported by the U.S. Department of Energy under Contract DE-AC02-06CH11357. This research used resources of the Laboratory Computing Resource Center and the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. Computing time on *Intrepid* was granted by a 2012 DOE INCITE Award "Optimization of Complex Energy Systems Under Uncertainty," PI Mihai Anitescu.

References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring), pp. 483–485. ACM, New York (1967)
2. Aykanat, C., Pinar, A., Çatalyürek, U.V.: Permuting sparse rectangular matrices into block-diagonal form. SIAM J. Sci. Comput. **25**, 1860–1879 (2004)
3. Bennett, J.M.: An approach to some structured linear programming problems. Operations Research **14**(4), 636–645 (1966)
4. Birge, J., Louveaux, F.: Introduction to Stochastic Programming, 2nd edn. Springer Series in Operations Research and Financial Engineering Series. Springer, New York (2011)
5. Bisschop, J., Meeraus, A.: Matrix augmentation and partitioning in the updating of the basis inverse. Mathematical Programming **13**, 241–254 (1977)
6. Dantzig, G.B., Orchard-Hays, W.: The product form for the inverse in the simplex method. Mathematical Tables and Other Aids to Computation **8**(46), pp. 64–67 (1954)
7. Davis, T.: Direct methods for sparse linear systems. Fundamentals of algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA (2006)
8. Duff, I.S., Scott, J.A.: A parallel direct solver for large sparse highly unsymmetric linear systems. ACM Trans. Math. Softw. **30**, 95–117 (2004)
9. Forrest, J.J., Goldfarb, D.: Steepest-edge simplex algorithms for linear programming. Mathematical Programming **57**, 341–374 (1992)
10. Forrest, J.J.H., Tomlin, J.A.: Updated triangular factors of the basis to maintain sparsity in the product form simplex method. Mathematical Programming **2**, 263–278 (1972)
11. Gilbert, J.R., Peierls, T.: Sparse partial pivoting in time proportional to arithmetic operations. SIAM J. on Scientific and Statistical Computing **9**, 862–874 (1988)
12. Gill, P.E., Murray, W., Saunders, M.A., Wright, M.H.: A practical anti-cycling procedure for linearly constrained optimization. Mathematical Programming **45**, 437–474 (1989)
13. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore, MD (1996)
14. Gondzio, J., Grothey, A.: Exploiting structure in parallel implementation of interior point methods for optimization. Computational Management Science **6**, 135–160 (2009)
15. Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to parallel computing, 2nd edn. Addison-Wesley (2003)
16. Gropp, W., Thakur, R., Lusk, E.: Using MPI-2: Advanced Features of the Message Passing Interface, 2nd edn. MIT Press, Cambridge, MA (1999)
17. Hall, J., McKinnon, K.: Hyper-sparsity in the revised simplex method and how to exploit it. Computational Optimization and Applications **32**, 259–283 (2005)
18. Hall, J.A.J., Smith, E.: A high performance primal revised simplex solver for row-linked block angular linear programming problems. Tech. Rep. ERGO-12-003, School of Mathematics, University of Edinburgh (2012)
19. Harris, P.M.J.: Pivot selection methods of the DEVEX LP code. Mathematical Programming **5**, 1–28 (1973)
20. Kall, P.: Computational methods for solving two-stage stochastic linear programming problems. Zeitschrift für Angewandte Mathematik und Physik (ZAMP) **30**, 261–271 (1979)

21. Koberstein, A.: The dual simplex method, techniques for a fast and stable implementation. Ph.D. thesis, Universität Paderborn, Paderborn, Germany (2005)
22. Koberstein, A.: Progress in the dual simplex algorithm for solving large scale LP problems: techniques for a fast and stable implementation. Computational Optimization and Applications **41**, 185–204 (2008)
23. Lasdon, L.S.: Optimization Theory for Large Systems. Macmillan, New York (1970)
24. Linderoth, J., Wright, S.: Decomposition algorithms for stochastic programming on a computational grid. Computational Optimization and Applications **24**, 207–250 (2003)
25. Lubin, M., Petra, C.G., Anitescu, M., Zavala, V.: Scalable stochastic optimization of complex energy systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pp. 64:1–64:64. ACM, New York (2011)
26. Markowitz, H.M.: The elimination form of the inverse and its application to linear programming. Management Science **3**(3), pp. 255–269 (1957)
27. Maros, I.: Computational Techniques of the Simplex Method. Kluwer Academic Publishers, Norwell, MA (2003)
28. Stern, J., Vavasis, S.: Active set methods for problems in column block angular form. Comp. Appl. Math. **12**, 199–226 (1993)
29. Strazicky, B.: Some results concerning an algorithm for the discrete recourse problem. In: Stochastic programming (Proc. Internat. Conf., Univ. Oxford, Oxford, 1974), pp. 263–274. Academic Press, London (1980)
30. Suhl, L.M., Suhl, U.H.: A fast LU update for linear programming. Annals of Operations Research **43**, 33–47 (1993)
31. Suhl, U.H., Suhl, L.M.: Computing sparse LU factorizations for large-scale linear programming bases. ORSA Journal on Computing **2**(4), 325 (1990)
32. Vladimirou, H., Zenios, S.: Scalable parallel computations for large-scale stochastic programming. Annals of Operations Research **90**, 87–129 (1999)