LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-JRNL-678917

PIPS-SBB: A Parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs

L. Munguia, G. Oxberry, D. Rajan

November 2, 2015

Proceedings of the 2016 International Parallel and Distributed Processing Symposium (IPDPS) Workshops

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

PIPS-SBB: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs

Lluís-Miquel Munguía
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
Email: lluis.munguia@gatech.edu

Geoffrey Oxberry
Computational Engineering Division
Lawrence Livermore National Laboratory
Livermore, California 94550
Email: oxberry1@llnl.gov

Deepak Rajan
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California 94550
Email: rajan3@llnl.gov

***Abstract*—Stochastic mixed-integer programs (SMIPs) deal with optimization under uncertainty at many levels of the decision-making process. When solved as extensive formulation mixed-integer programs, problem instances can exceed available memory on a single workstation. To overcome this limitation, we present PIPS-SBB: an exact distributed-memory parallel stochastic MIP solver that takes advantage of parallelism at multiple levels of the optimization process. We show promising results on instances from the SIPLIB benchmark by combining methods known for accelerating Branch and Bound (B&B) methods with new ideas that leverage the structure of SMIPs. We expect the performance of PIPS-SBB to improve further as more functionality is added in the future.**

I. Introduction

Stochastic mixed-integer programs (SMIPs) are a generalization of mixed-integer Programs (MIPs) to deal with optimization under uncertainty. Consider the MIP

$$\min_{x\in\mathbb{R}^n}\{c^{\mathrm{T}}x : Ax=b,\, l\le x\le u,\ x_j\in\mathbb{Z},\forall j\in I\subseteq[n]\},\tag{MIP}$$

where $c\in\mathbb{R}^n$, $A\in\mathbb{R}^{m\times n}$, $b\in\mathbb{R}^m$, and I is the set of integer variable indices. Throughout this paper, we use $[n]$ to denote the set $\{1,\dots,n\}$. For ease of exposition, we assume that x is bounded below by $l\in\mathbb{R}^n$ and above by $u\in\mathbb{R}^n$. We assume, as is typical, that all problem data belong to the set of rationals.

Typically, stochastic optimization problems are formulated as multi-stage optimization problems where some model parameters are random variables (with known probability distributions). In each stage, a decision has to be made, and after each decision is made, one learns the realization of some of the random variables. Usually, the goal is to minimize the expected total cost, where the expectation is over all realizations; see [1] for a detailed discussion.

In this work, we focus on two-stage SMIPs, a common variant in which the optimization problem is subdivided into two stages. For two-stage SMIPs, the first-stage variables determine the set of decisions before the uncertainty takes place. Second-stage variables represent the set of decisions to be taken once the uncertainty is revealed, as recourse to the decisions taken in the first stage. A two-stage MIP takes the form

$$\min_{x\in\mathbb{R}^{n_1}}\{c^{\mathrm{T}}x+\mathbb{E}_P[Q(x,\xi)] : Ax=b, l\le x\le u,\ x_j\in\mathbb{Z},\forall j\in I_1\subseteq[n_1]\},\tag{SMIP}$$

where x is the first-stage decision variable, ξ is a random vector with support Ξ sampled from a known probability distribution P, $\mathbb{E}$ is the expectation operator over this probability distribution, and I_1 is the set of first-stage integer variable indices. For a given first-stage solution x, the second-stage optimization problem is of the form

$$Q(x,\xi)=\min_{y\in\mathbb{R}^{n_2}}\{q(\xi)^{\mathrm{T}}y : W(\xi)y=h(\xi)-T(\xi)x, l(\xi)\le y\le u(\xi),\, y_j\in\mathbb{Z},\forall j\in I_2\subseteq[n_2]\},\tag{SS$_\xi^x$}$$

where $W(\cdot),h(\cdot),l(\cdot),u(\cdot),q(\cdot)$ and $T(\cdot)$ may depend on ξ, but not on x. We assume that I_2, the set of second-stage integer variable indices, does not depend on ξ. If (SS$_\xi^x$) is infeasible, then we set $Q(x,\xi)=\infty$. Throughout this paper we assume finite support for ξ.

As defined so far, the main computational challenges in solving SMIPs exactly arise from the difficulty in optimizing the expected cost, which is non-convex. However, SMIPs can be approximated via Sample Average Approximation [2], thus transforming the problem into one large MIP as seen in (EXT) (usually called the *extensive formulation*) and enabling the use of traditional MIP approaches. In (EXT), we use $[s]$ to represent the set of possible sampled realizations, also known as scenarios.

In (EXT), x corresponds to the first-stage decisions of the stochastic mixed-integer program (SMIP). For each $i\in[s]$, the second-stage variable $y_i\in\mathbb{R}^{n_2}$ corresponds to the decision variable $y\in\mathbb{R}^{n_2}$ in the second-stage optimization problem (SS$_\xi^x$) for the realization of ξ sampled in scenario i. The matrix A models the constraints relative to the first stage, while matrices T_i and W_i model the second-stage constraints for scenario i. Vector b_0 represents the right-hand side of the first-stage constraints, and vector b_i represents the right-hand

side of the second stage for scenario i. In creating (EXT), the we have increased the number of second-stage decision variables and constraints by a factor of s with respect to (SS_ξ^x), the second-stage problems for a single scenario.

$$
\begin{aligned}
&\min_{x\in\mathbb{R}^{n_1}, y\in\mathbb{R}^{n_2\times s}} c^{\mathrm{T}}x + \sum_{i=1}^{s} p_i q_i^{\mathrm{T}} y_i, \\
&\text{subject to:} \\
&\begin{array}{lllll}
Ax & & & & = b_0, \\
T_1 x & +W_1 y_1 & & & = b_1, \\
T_2 x & & +W_2 y_2 & & = b_2, \\
\vdots & & & \ddots & \vdots \\
T_s x & & & +W_s y_s & = b_s,
\end{array} \\
&\qquad l \le x \le u, \\
&\qquad l_i \le y_i \le u_i, \quad \forall i \in [s], \\
&\qquad x_j \in \mathbb{Z}, \quad \forall j \in I_1, \\
&\qquad y_{i,j} \in \mathbb{Z}, \quad \forall i \in [s], \forall j \in I_2.
\end{aligned}
\tag{EXT}
$$

Extensive formulations can be solved with a general purpose state-of-the-art MIP solver such as CPLEX [3], Xpress [4], SCIP [5], or GUROBI [6]. General purpose MIP solvers use an enumerative tree search algorithm known as Branch and Bound (B&B) in which linear programming (LP) relaxations are solved at each node of the B&B tree; see Section II-A for more details. For the remainder of this work, we do not distinguish between SMIPs and their extensive formulations; in general, by "SMIP", we refer to the extensive formulation in (EXT).

General purpose solvers do not recognize and/or leverage the dual block-angular structure of the extensive formulation. Furthermore, SMIPs present additional computational challenges. For example, SMIPs become harder to solve efficiently as the number of scenarios grows, mainly due to the increase in problem size. In addition to increased solution times, the problem may not fit in memory at all. Shared-memory and distributed-memory versions of B&B algorithms have been implemented by these state-of-the-art MIP solvers and other MIP solvers such as ParaSCIP ([7], [8]), BLIS ([9], [10]), and PICO [11]. However, all of these efforts focus on parallel B&B, and not on parallelizing the underlying LP relaxation solved at each node of the B&B algorithm. Though state-of-the-art MIP solvers are highly optimized when run in sequential mode, prior studies have shown that the B&B search algorithm does not scale well beyond modest amounts of parallelism [12] and thus these solvers do not leverage recent advances in HPC architectures. Scalability is also limited by memory when optimizing extended formulations. Given that these parallel implementations do not support data distribution, highly detailed approximations are computationally intractable and coarse-grained versions (with fewer scenarios) must be used instead, resulting in lower quality solutions to the original SMIP.

Much of the work in the SMIP literature avoids solving extensive formulations and alternative problem decompositions are devised instead. For an expanded literature review, we refer the readers to [13]. One approach is to derive convex approximations of the expected second-stage cost within an iterative algorithm, such as Benders' decomposition. Such iterative schemes are collectively known as stage-wise decomposition schemes; see [14] for a detailed survey. Alternatively, one can rely on scenario-based decomposition schemes, where stages are decoupled by the replication of first-stage variables for each scenario. To our knowledge, the first exact scenario-wise decomposition scheme was presented by [15], in which the authors solve Lagrangian duals at each node of a B&B procedure. In addition, many heuristics based on scenario-wise decomposition strategies have been developed for stochastic mixed-integer programs.

A large number of stage-wise and scenario-wise decomposition schemes have been implemented sequentially. A list of software may be found in [16]. However, there exist few parallel software libraries that model and solve mixed-integer stochastic programs, with the exception of PySP [17] and DSP [18].

A. Contributions and Overview

In contrast to previous approaches, we leverage the dual block-angular structure of SMIPs to solve LP relaxations using a parallel simplex method. Despite its limited scalability, there is recent work on parallelizing it; see [19] for a review of the challenges involved. PIPS-S is a parallel implementation of primal and dual simplex that has been recently developed to solve LP problems with dual block-angular structure [20], such as the LP relaxation of the extensive formulation. We incorporate PIPS-S in our approach as the primary LP solver. The product is a distributed-memory B&B-based solver for Stochastic MIPs called PIPS-SBB (PIPS - Simple Branch and Bound)[1] PIPS-SBB incorporates new methods for preprocessing, branching, and heuristics that maintain the decomposable structure of SMIPs. These methods also apply to any MIP with dual block-angular structure; we focus on the SMIP case due to its prevalence in the literature and in applications.

The main contributions of PIPS-SBB, a novel B&B algorithm for general two-stage stochastic mixed-integer programs, are the following.

- PIPS-SBB is the first B&B algorithm to solve LP relaxations using a distributed-memory simplex algorithm that leverages the structure of SMIPs.
- By distributing SMIP data, PIPS-SBB can leverage the distributed-memory architecture of supercomputers to address more memory and optimize larger problems. In contrast, existing MIP solvers must replicate the entire extensive formulation within each process, thus limiting input problem size.
- Adapting methods known to accelerate MIP solvers (Presolve [21] and Primal Heuristics [22]) to a distributed-memory setting with dual block-angular data structure,

[1]The name PIPS-SBB deliberately alludes both to COIN-OR's now defunct Sbb (simple branch-and-bound) MIP solver, developed by John Forrest, and PIPS-S.

we present initial results on benchmark SIPLIB instances that show the effectiveness of our method.

The remainder of this work proceeds as follows: In Section II, we describe B&B algorithms, present the main ideas behind PIPS-SBB, and the special-purpose distributed-memory algorithms implemented in PIPS-SBB that leverage the dual block-angular data structure of stochastic MIPs. We present computational results in Section III using instances from SIPLIB, a stochastic programming library that illustrate the effectiveness of algorithms. Finally, we look forward to the future, presenting next steps and ideas in Section IV.

II. PIPS-SBB: A specialized parallel distributed-memory Branch & Bound solver for large-scale stochastic MIP problems

PIPS-SBB is a parallel Branch and Bound (B&B) framework for MIPs that feature a dual block-angular structure, such as the extensive formulation (EXT). This dual block-angular structure offers opportunities for parallelism at many levels of the optimization process, which will eventually enable PIPS-SBB to solve significantly larger extensive formulations than existing technologies. Exploiting these opportunities for parallelism also has the potential to reduce significantly computation times. In this work, we leverage this dual block-angular structure to induce task parallelism[2] by distributing MIP data across multiple processors. LP relaxations are then solved in parallel using PIPS-S within the MIP infrastructure provided by PIPS-SBB.

In this section, we overview the main ingredients of PIPS-SBB. First, in Section II-A, we overview the B&B algorithm. Then, in Section II-B, we discuss how to expose parallelism in solving the LP relaxation. This discussion segues into a description of MIP data distribution in Section II-C. After a brief description of PIPS-SBB implementation in Section II-D, we conclude with a discussion of PIPS-SBB's structure-aware B&B algorithms such as developing branching rules, primal heuristics, and presolve in Section II-E.

A. Branch and Bound

In Branch and Bound (B&B) [23], a mixed-integer program (MIP) is solved to optimality by systematically partitioning and searching the solution space using a tree data structure called a B&B tree to enumerate feasible integer solutions. In LP-relaxation based B&B, an LP relaxation (formed by relaxing all integrality constraints) is solved at each node of this tree[3]. The objective value of the solution to the resulting LP relaxation (*fractional solution*) provides a lower bound on the MIP solution value. If an optimal solution of the LP relaxation is *integer feasible*, then this point is also a feasible solution to the MIP. The objective value of this feasible solution provides an upper bound on the MIP optimal solution value. However, if the calculated LP relaxation solution is not integer feasible, either the corresponding node is deleted (pruned) or the node is divided into two or more nodes with the use of additional inequalities. The former step is *bounding*, one of the main steps of the B&B algorithm, and can be done if the LP relaxation solution objective value is larger than the best upper bound (from the objective value of all integer feasible solutions found so far). The latter step is *branching*, in which inequalities are added to eliminate the fractional solution (which is LP-feasible but not integer feasible) and divide the solution space such that no feasible solutions to (MIP) are cut.

During the search process, let L be the current best lower bound and let U be the current best upper bound (also the objective value of the current best integer-feasible solution). Progress in the B&B algorithm is measured in terms of the *relative gap*, defined by

$$\frac{U - L}{10^{-10} + |U|}, \quad \text{(RelGap)}$$

as in CPLEX. The B&B algorithm terminates when the relative gap is less than a given tolerance, or when there are no nodes remaining in the B&B tree.

State-of-the-art MIP solvers build upon this branch and bound scheme and enhance it with many additional algorithmic practices to improve its performance, primarily by focusing on improving the upper and lower bounds. Primal heuristics [22], [24] are essential for finding high quality integer feasible solutions (better upper bounds) early in the search and reducing the solution space by pruning. Better lower bounds are obtained by developing stronger formulations. One method for strengthening formulations adds cutting planes (inequalities) [25] that strengthen the LP relaxation by eliminating parts of its feasible space without eliminating any integer feasible solutions. Another method for strengthening formulations is pre-processing [21], in which additional information about the problem structure can be derived from the constraints, potentially improving coefficients, eliminating redundant constraints, tightening variable bounds, and even fixing the value of some of the variables. The effectiveness of a MIP B&B tree search algorithm also depends on tree creation algorithms (branching rules) that determine how to partition the feasible space [26]. State-of-the-art MIP solvers are highly optimized in all these aspects, representing over two decades of research [27]. The current version of PIPS-SBB contains a subset of these methods; we plan on implementing more in the future.

Beyond established methods for MIPs, the structure of extensive formulations enables additional algorithmic improvements. The two-stage hierarchical organization in (EXT) suggests that first-stage information may be more important than second-stage information, as first-stage variables may affect multiple scenarios simultaneously, while the impact of second-stage variables is restricted to a single scenario. For this reason, branching rules and primal heuristics in PIPS-SBB prioritize first-stage variables over second-stage variables; examples that illustrate this prioritization are presented in section II-E. Leveraging the dual block-angular problem structure is a critical

[2]Aside from reductions, the algorithms presented in [20] operate on second-stage blocks of dual block-angular LPs independently.

[3]The first node in the tree is referred to as the *root* node.

feature of PIPS-SBB. In Section III-C, we see how specialized branching rules and heuristics allow to reduce the relative gap faster.

B. Parallelism in the LP relaxation

At the very heart of a B&B algorithm, the LP relaxation provides a lower bound on the best MIP solution at every node of the B&B tree. Given its central importance, it is essential that LP relaxations are solved as efficiently as possible. The decomposable nature of the extensive formulation for stochastic MIPs incentivizes the use of interior-point methods to speed up solution of LP relaxations. These algorithms are highly parallelizable, as shown in [28]. Despite their scalability and ability to tackle big problem instances, interior-point methods are not typically used because these methods warm start less efficiently (requiring 50-60% fewer interior-point iterations [29]) than simplex methods (usually requiring a few pivots when used in a B&B algorithm). The enumerative nature of B&B makes warm-starting crucial for performance. For that specific reason, B&B algorithms typically favor the simplex algorithm to solve LP relaxations, since this algorithm can be warm-started for an LP relaxation from the optimal solution of its parent node.

Even though this is an area of active research [19], parallel simplex implementations have been unable to outperform substantively an efficient modern sequential simplex solver for general, unstructured LPs. However, it is possible to develop parallel algorithms that exploit the dual block-angular structure exhibited by stochastic LPs in the extensive form and outperform efficient modern sequential simplex solvers. PIPS-S [20] implements such an algorithm. PIPS-SBB builds upon PIPS-S and uses it as its core LP solver. Thus, PIPS-SBB is able to exploit parallelism in the LP relaxation of every B&B node. Exploiting parallelism within each node of the B&B tree is a novel departure from general purpose MIP solvers, which reserve parallelism to solving LP relaxations of multiple B&B nodes simultaneously.

C. Parallel data distribution

For scalability, PIPS-SBB is designed for distributed-memory parallel computer architectures. Distributed-memory paradigms assume the addressable memory space is segmented and distributed among individual processes, as depicted in Figure 1a. Due to this decentralized memory space, communications libraries such as MPI [30] are required in order to coordinate among processes. PIPS-SBB uses only MPI collectives to communicate efficiently among processes, both in the B&B algorithm and while solving the LP relaxations.

In conjunction with a distributed-memory parallel simplex solver, data is also distributed across processes. PIPS-SBB distributes the data representing each scenario to different processes while first-stage information is replicated. In other words, W_i, T_i, q_i and b_i are allocated on a single process for each $i \in [s]$, while c, A and b_0 are replicated on all processes; see Figure 1b. This data distribution can be scaled to as many processes as scenarios specified in the input problem, which enables PIPS-SBB to solve large SMIPs that would not otherwise fit in memory. Every component of PIPS-SBB conforms to this data distribution policy, including PIPS-S. This data distribution policy also extends to other data stored by PIPS-SBB, such as cutting planes, variable bound updates (as a result of branching), and LP warm-start information.

D. PIPS-SBB implementation details

PIPS-SBB is written in C++ and is designed to provide users with a flexible parallel framework suitable for solving any mixed-integer program with dual block-angular structure, which includes all two-stage stochastic mixed-integer programs. PIPS-SBB uses COINUtils as an auxiliary library for much of its basic functionality.

In designing the architecture of PIPS-SBB, care is taken to minimize the memory footprint, allowing PIPS-SBB to solve large problem instances. For example, information related to relaxations such as LP warm-start information and branching decisions are stored incrementally with respect to the parent problem. Without incremental storage, storing tree information would quickly exhaust available memory, as is the case for the state-of-the-art solver CPLEX when solving certain problem instances; see Section III-B.

E. Parallelism in structure-aware algorithms

The current version of PIPS-SBB features branching rules, primal heuristics, and presolve, all of which are designed and adapted to leverage dual block-angular problem structure and parallelism. There are two major design assumptions. First, every algorithm within PIPS-SBB must conform to the data distribution imposed in Section II-C. Second, this data representation must remain distributed throughout the entire algorithm, and thus every MPI process is responsible for performing all operations on the data it owns. Algorithms 1, 2, and 3 show examples on how these design assumptions are maintained in PIPS-SBB. For ease of exposition, in these examples we assume that each MPI process owns one scenario.

1) Branching Rules: The current version of PIPS-SBB features three branching rules: minimum infeasible index branching, most infeasible branching, and a more complex pseudo-cost branching. Algorithm 1 illustrates the most infeasible branching rule. It proceeds by identifying the most infeasible first-stage variable and returning its index if one is found. If no such variable is found, it searches in parallel for the most infeasible second-stage variable in each scenario. An all-to-all reduction is then required in order to find the most infeasible second-stage variable among all scenarios and communicate it to all processes.

2) Primal Heuristics: In addition, it incorporates ten primal heuristics, ranging from simple rounding and diving schemes to more computationally expensive large neighborhood search schemes such as RENS [31]. Algorithm 2 shows a simple heuristic diving strategy for finding feasible integer solutions, where an input fractional solution is iteratively rounded and bounded. After a variable rounding takes place, the LP relaxation is re-optimized. Once all first-stage variables become

Algorithm 1 PIPS-SBB Most infeasible branching rule

```
function BlockAngularMostInfeasibleBranching(x, y, comm)
    ▷ Input: Integer infeasible LP-feasible solution, MPI communicator

    scen := −1                  ▷ Scenario number to be branched on; -1 is sentinel value for first-stage
    if I_1 ∩ F ≠ ∅ then         ▷ F is the index set of all fractional-valued first-stage variables
        return [scen, argmax_j{|x_j − ⌈x_j⌋|, j ∈ I_1}]     ▷ Return first-stage "scenario", and variable index
    end if

    idx_i := −1                 ▷ In this line and the following, −1 is a sentinel indicating integer feasibility
    frac_i := −1
    scen := i                   ▷ Scenario i owned by process (MPI rank) i
    if I_2 ∩ F^i ≠ ∅ then       ▷ F^i is the index set of all fractional second-stage variables of scenario i
        idx_i := argmax_j{|y_{i,j} − ⌈y_{i,j}⌋|, j ∈ I_2}
        frac_i := |y_{i,idx_i} − ⌈y_{i,idx_i}⌋|
    end if

    ▷ All-to-all reduce process number of maximum second-stage fractional value
    MPI_Allreduce([frac_i, scen], MPI_IN_PLACE, 1, MPI_DOUBLE_INT, MPI_MAXLOC, comm)

    ▷ Broadcast index idx_i of maximum fractional value on process scen to all processes
    MPI_Bcast(idx_i, MPI_IN_PLACE, 1, MPI_INT, scen, comm)

    return [scen, idx_i]        ▷ Returns scenario number and index of variable to branch on
    ▷ If idx_i is -1, solution is integer-feasible
end function
```

Algorithm 2 PIPS-SBB Simple Parallel Diving Heuristic

```
function BlockAngularParallelDivingHeuristic(x, y, comm)
    ▷ Input: Integer infeasible LP-feasible solution, MPI communicator

    Using x, compute F, the index set of all fractional-valued first stage variables
    while I_1 ∩ F ≠ ∅ do
        idx := argmax_j{|x_j − ⌈x_j⌋|, j ∈ I_1}        ▷ idx is index of most fractional variable in I_1
        Fix x_idx to the nearest integer value by modifying bounds: l_idx = u_idx = ⌈x_idx⌋
        Solve LP relaxation of modified problem        ▷ This LP relaxation is solved in parallel using PIPS-S
        if LP relaxation infeasible then
            return Failure
        else
            [x, y] := optimal value of the LP relaxation
        end if
        Recalculate F using new x, index idx ∉ F since x_idx is fixed to integral value
    end while
    Fix all first-stage variables x by modifying bounds: l = u = x     ▷ All x are currently integer-valued

    ▷ Scenario i owned by process (MPI rank) i
    Using y_i, compute F^i, the index set of all fractional-valued second-stage variables of scenario i
    ▷ Compute total number of fractional-valued second-stage variables over all processes
    MPI_Allreduce(|I_2 ∩ F^i|, totalFracVars, 1, MPI_INT, MPI_SUM, comm)
    while totalFracVars > 0 do
        if I_2 ∩ F^i ≠ ∅ then
            idx_i := argmax_j{|y_{i,j} − ⌈y_{i,j}⌋|, j ∈ I_2 ∩ F^i}   ▷ idx_i is index of most fractional variable in I_2 ∩ F^i
            Fix y_{i,idx_i} to the nearest integer value by modifying bounds: l_{i,idx_i} = u_{i,idx_i} = ⌈y_{i,idx_i}⌋
        end if
        Solve LP relaxation of modified problem        ▷ This LP relaxation is solved in parallel using PIPS-S
        if LP relaxation infeasible then
            return Failure
        else
            [x, y] := optimal value of the LP relaxation   ▷ x does not change in this line; it has been fixed
        end if
        Recalculate F^i using new y_i, index idx_i ∉ F_i since y_{i,idx_i} is fixed to integral value
        ▷ Recompute total number of fractional-valued second-stage variables over all processes
        MPI_Allreduce(|I_2 ∩ F^i|, totalFracVars, 1, MPI_INT, MPI_SUM, comm)
    end while

    return [x, y]                                  ▷ Returns integer feasible solution
end function
```

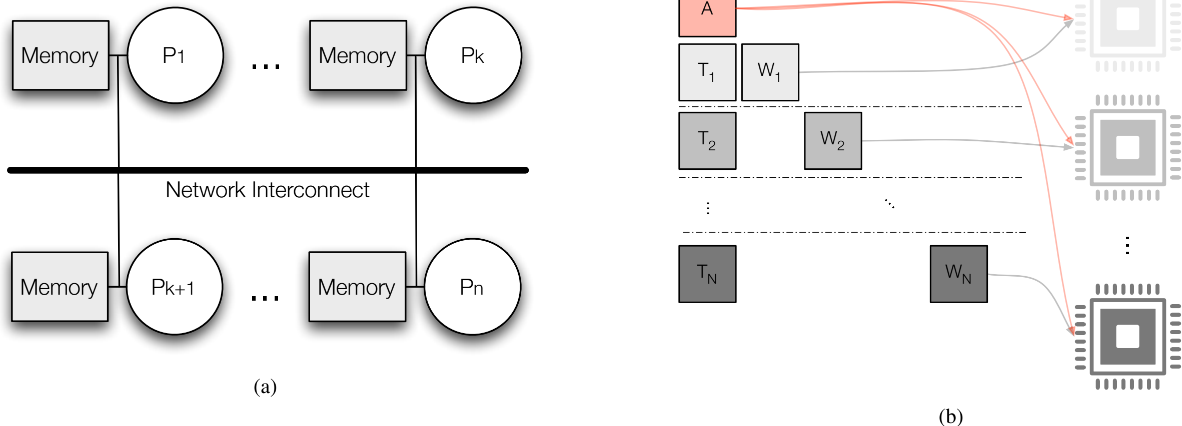

Fig. 1: (a) Schematic depiction of a parallel system with a distributed-memory configuration. It has a segmented memory space, which is distributed among different processes. (b) Data parallelism in PIPS-SBB.

integer, each MPI process independently rounds one locally owned second-stage variable in each iteration. The procedure terminates when an integer solution is found or when the fixings render the LP relaxation infeasible.

3) Presolving: MIP presolve is implemented in PIPS-SBB following Savelsbergh [21, Section 1], with the exception of deleting redundant constraints. In order to accommodate the distributed nature of MIP problem data in PIPS-SBB, the presolve algorithm operates as in Algorithm 3. While presolve continues to modify the MIP, presolve first updates first-stage variable bounds, updates first-stage constraints, and assesses MIP feasibility. Then, because second-stage data is distributed by scenario, presolve processes second-stage constraints in parallel. This step updates first- and second-stage variable bounds, updates second-stage constraints, and assesses MIP feasibility. Since information on first-stage variable bounds and MIP feasibility may be different on different MPI processes due to preprocessing second-stage constraints in distributed fashion, this information must be synchronized across all processes via appropriate all-to-all reductions, as depicted in Algorithm 3.

III. Experimental Results

We illustrate the performance of PIPS-SBB using the Stochastic Server Location Problem (SSLP) [32] instances from SIPLIB [33], a testbed of stochastic mixed-integer programs. We chose SSLP since this test set contains the largest variation in the number of scenarios, with instances ranging from 5 to 2000 scenarios. The SSLP instances model server location problems using a pure binary first-stage and mixed-binary second-stage. They are written in the form sslp.$m.n.s$, where m is the number of potential server locations, n is the number of potential clients, and s is the number of scenarios.

All computations were performed on the Sierra Cluster at Lawrence Livermore National Laboratory. This cluster consists of 1,944 nodes, with nodes connected using InfiniBand QDR interconnects. Each individual node consists of 2 Intel 6-core Xeon X5660 processors and 24GB of memory. In all PIPS-SBB experiments, we bind 1 MPI process per core to ensure that cores are not over-subscribed with multiple processes. For our experiments, we built PIPS-SBB with MVAPICH2 version 1.7.

We compare PIPS-SBB against the state-of-the-art general purpose MIP solver CPLEX 12.6.2 running on a single node and using 12 threads (1 per processor). For this paper, we chose the SSLP instances from SIPLIB since they are relatively small in size, and can be solved on a single node by CPLEX with no memory restrictions. Even then, CPLEX ran out of memory on a few instances due to a rapid growth in the B&B tree size.

A. Scaling Experiments

First, we present results that demonstrate the scaling performance of PIPS-SBB. In particular, we show that PIPS-SBB scales as well as PIPS-S, the underlying distributed-memory LP solver. Note that both PIPS-S and PIPS-SBB can use no more MPI processes than the number of scenarios.

To measure the strong scaling performance of PIPS-S, we calculate the speedup in solving the LP relaxation at the root node of the PIPS-SBB B&B tree of the instances sslp.10.50.*. Defining speedup (a function of the number of cores N) as the ratio of (LP relaxation solution time with N cores) to (LP relaxation solution time with 5 cores), we see in Figure 2a that PIPS-S scales up to 25-50 cores for the large instances. In particular, it strong scales at 90% efficiency up to 10 cores for sslp.10.50.2000, and then strong scaling efficiency drops off quickly, with speedup peaking at 50 cores. For the smaller instances, such as sslp.10.50.100, the speedup peaks at 10 cores. This speedup curve is typical of PIPS-S, illustrating the scaling limitations of PIPS-S [20, Table 3].

Based on these results, the current algorithms in PIPS-SBB will not strong scale to a large number of cores. Some opportunities for exposing additional parallelism are proposed in Section IV. Since PIPS-SBB solves an LP relaxation

Algorithm 3 PIPS-SBB Presolve

```
function BLOCKANGULARPRESOLVE(A, T_i, W_i, b_0, b_i, I_1, I_2, l, u, l_i, u_i, comm)
    ▷ Input: Coefficient matrices, right-hand side vectors, index sets, variable bounds, MPI communicator

    while true do
        [isFeasible, isMIPchanged1, A, b_0, l, u] := FIRSTSTAGEPRESOLVE(A, b_0, I_1, l, u)
        if isFeasible is false then          ▷ Feasibility information is stored in a boolean variable isFeasible
            return MIP is infeasible
        end if

        ▷ Scenario i owned by process (MPI rank) i
        [isFeasible, isMIPchanged2, A, T_i, W_i, b_0, b_i, l, u, l_i, u_i] :=
            SECONDSTAGEPRESOLVE(A, T_i, W_i, b_0, b_i, l, u, l_i, u_i)

        ▷ Synchronize infeasibility information.
        MPI_Allreduce(isFeasible, MPI_IN_PLACE, 1, MPI_INT, MPI_LAND, comm)
        if isFeasible is false then
            return MIP is infeasible
        end if

        ▷ Synchronize whether MIP was modified
        isMIPchanged := isMIPchanged1 or isMIPchanged2
        MPI_Allreduce(isMIPchanged, MPI_IN_PLACE, 1, MPI_INT, MPI_LOR, comm)
        if isMIPchanged is false then          ▷ Exit loop if MIP was not modified
            break
        end if

        ▷ Synchronize upper and lower bounds on x, which may be tighter from second-stage presolve
        MPI_Allreduce(u, MPI_IN_PLACE, n, MPI_DOUBLE, MPI_MIN, comm)
        MPI_Allreduce(l, MPI_IN_PLACE, n, MPI_DOUBLE, MPI_MAX, comm)
    end while

    return A, T_i, W_i, b_0, b_i, l, u, l_i, u_i          ▷ Returns modified coefficent matrix
end function
```

for each node of the B&B tree, one possible metric is its *throughput*, or the number of B&B nodes it can process per unit time. PIPS-SBB speedup (a function of the number of cores N) is therefore measured as the ratio of (Number of B&B Nodes Processed per second with N cores) to (Number of B&B Nodes Processed per second with 5 cores). For this experiment, we turned off all the computationally expensive branching rules and primal heuristics, tuning PIPS-SBB to process B&B nodes as quickly as possible. We see in Figure 2b that the speedup curves are very similar in shape to that of PIPS-S, with peak speedups occurring around 25-50 cores for the larger instances. This experiment illustrates that a stripped-down PIPS-SBB implementation continues to process nodes (and therefore LP relaxations) at roughly the same rate as PIPS-S.

Interestingly, PIPS-SBB shows super-linear throughput scaling for large problem instances. While this result seems surprising and counter-intuitive, it can be explained by a careful analysis of the experimental data. Consider an experiment that processes more than one B&B node within the prescribed time limit. Among all of these nodes, the root node LP relaxation takes the longest, while the rest are typically solved within a few simplex iterations, since the LPs at all other nodes can be warm-started from the optimal solution of the LP relaxation at their parent node. As we increase the number of cores available, the LP relaxation solves faster (by a factor given by PIPS-S speedup) enabling PIPS-SBB to process many more nodes in the time limit. Since all these extra nodes are lightweight nodes (in terms of LP relaxation solution time), this results in a super-linear increase in the number of nodes processed per unit time, skewing the speedup numbers. Using the performance of a single core as the baseline accentuated this effect, and hence 5 cores were used instead. This skew suggests that throughput (as measured in this experiment) is not an accurate indicator of PIPS-SBB's ability to process nodes. Nevertheless, the scaling results presented in Figure 2 indicate that PIPS-SBB throughput scales in a manner consistent with that of PIPS-S performance.

We are primarily interested in how PIPS-SBB *wall clock time* scales. To this end, we consider the default PIPS-SBB algorithm (wherein primal heuristics and other features are enabled), and measure the time required by PIPS-SBB time required to close the relative gap (RelGap) to less than 2%. Presented in Figure 3, we see performance curves analogous to the throughput speedup curves in Figure 2b. As before, performance peaks around 25-50 cores for the larger instances.

B. Overall Performance

For experiments that illustrate the overall performance of PIPS-SBB we present results for a representative parallel processor configuration, where the number of cores is chosen as a function of the number of scenarios, based on our scaling experiments presented in Section III-A. We report the processor configuration as "Cores" for all our experiments.

The instances are solved to a relative gap of 10^{-4} (CPLEX default). Each experiment is given a time limit of 1 hour (3600

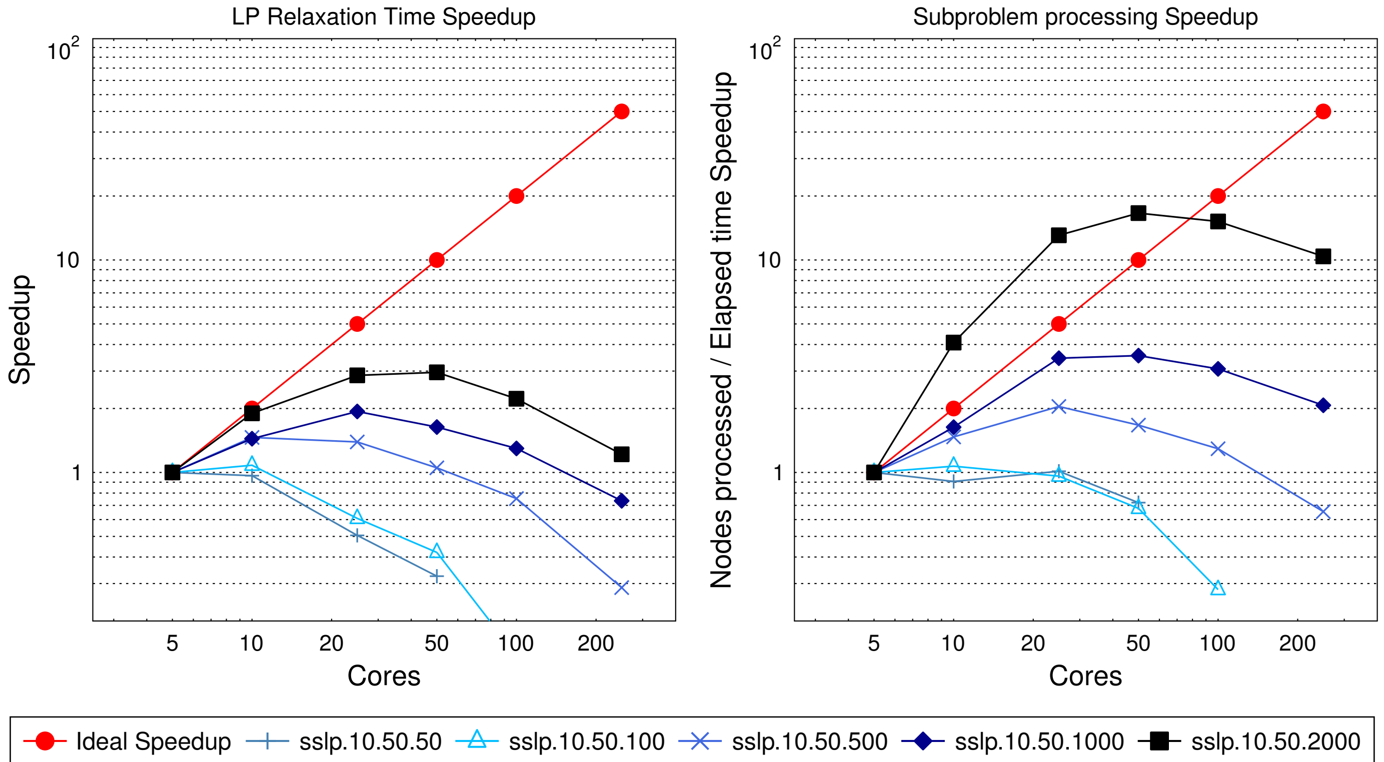

Fig. 2: (a) Strong scaling performance results of PIPS-S. (b) Strong scaling throughput results of PIPS-SBB.

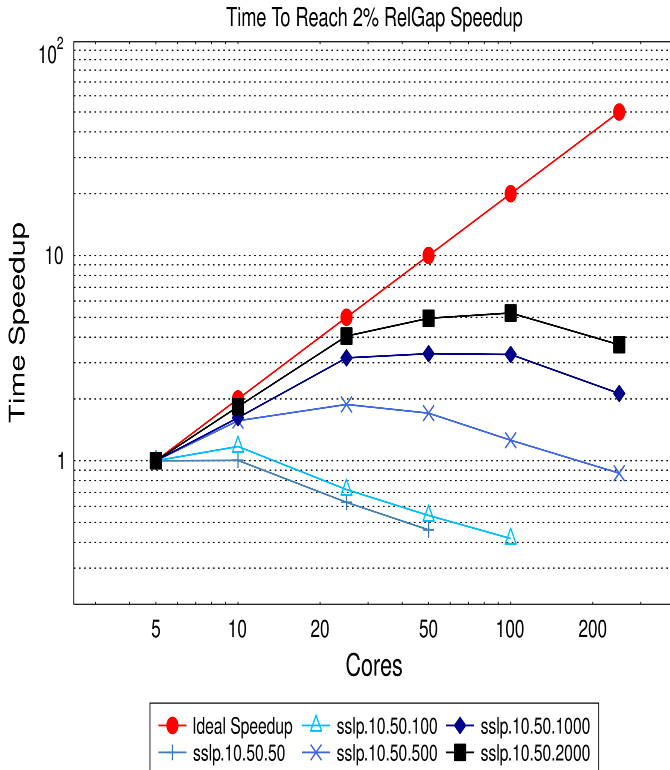

Fig. 3: Strong scaling performance results of PIPS-SBB

seconds), and the performance results are reported as "(Time)" in seconds. If an optimal solution is not provably obtained within the time limit, then the performance results are reported in terms of relative gap, denoted by "RelGap", and computed as in (RelGap). We also report the time (in seconds) at which PIPS-SBB found the best solution, as "Best Solution Time". To measure the quality of U, the best solution found by PIPS-SBB, we present the percentage gap between the best upper bound found by PIPS-SBB and the best upper bound found by CPLEX, denoted as "Best Solution Quality". This number could be negative if PIPS-SBB got a better quality solution than CPLEX at termination; such instances are marked in **bold**. For the instances solved to optimality by CPLEX but not by PIPS-SBB, this number indicates the quality of the solution obtained by PIPS-SBB - it could still be 0%. For such instances (solved to optimality by CPLEX, but not by PIPS-SBB), the difference between Best Solution Quality and PIPS-SBB RelGap indicates how far the PIPS-SBB lower bound L is from the optimal solution. We also present the performance results of CPLEX in the "CP RelGap (Time)" column. The instances where CPLEX ran out of memory are denoted with (M) next to the GAP at termination.

From Table I, we see that PIPS-SBB outperforms CPLEX in 3 out of 10 instances. The first set of rows correspond to the sslp.15.* instances, which have a small number of scenarios. We see that CPLEX shows better performance - it is able to solve the instances while PIPS-SBB is not. The second set of rows correspond to the easy sslp.5.* instances,

which both CPLEX and PIPS-SBB solve to optimality, though CPLEX is significantly faster then PIPS-SBB. The next two rows are instances that CPLEX solves, but PIPS-SBB does not. However, comparing the RelGap and Best Solution Quality entries, we see that the lower bounds obtained by PIPS-SBB at termination are close to the optimal solution, but its upper bound is poor. Furthermore, as the problems get more difficult as the number of scenarios increases (last three rows), PIPS-SBB is able to obtain better quality solutions than the primal heuristics implemented by CPLEX. Note that CPLEX has no knowledge that it is solving an extensive formulation, which results in its poor performance when the number of scenarios is large. In Section III-C, we show that leveraging stochastic MIP problem structure significantly improves the performance of PIPS-SBB. CPLEX runs out of memory in the B&B tree search for sslp.10.50.500.

TABLE I: SSLP instance set results

Problem Instance	Cores	RelGap (Time)	Best Solution Time	Best Solution Quality	CP RelGap (Time)
15.45.5	2	1.36%	1488s	1.07%	(4s)
15.45.10	2	7.93%	2129s	7.26%	(1s)
15.45.15	2	5.25%	2392s	4.84%	(12s)
5.25.50	1	(12.34s)	12s	0%	(1s)
5.25.100	1	(41.63s)	41s	0%	(1s)
10.50.50	5	1.48%	923s	1.31%	(81s)
10.50.100	10	1.74%	194s	1.56%	(442s)
10.50.500	50	1.57%	2792s	**-7.32%**	(M) 10.13%
10.50.1000	100	1.60%	2397s	**-11.19%**	14.47%
10.50.2000	100	24.00%	2384s	**-0.73%**	20.33%

C. Specialized Structure-Aware algorithms

As explained in Section II-E, PIPS-SBB leverages the dual block-angular problem structure during the B&B tree search by prioritizing decisions on first-stage variables over second-stage variables. To show the effectiveness of specialized branching rules and heuristics, we consider a structure-oblivious version of PIPS-SBB where decisions in primal heuristics and branching rules are performed regardless of the variable structure, so that all variables (first- and second-stage) have an equal priority of being chosen within the algorithm. We refer to this version of PIPS-SBB as General PIPS-SBB, and compare its performance against the structure-aware version of PIPS-SBB (referred to as Stochastic PIPS-SBB) in Table II. We see that Stochastic PIPS-SBB is able to deliver better performance in every test instance, which shows that these specializations are critical to the success of the primal heuristics and branching rules.

IV. Conclusions and Future Directions

In this paper, we have presented PIPS-SBB, a new exact distributed-memory parallel Branch-and-Bound (B&B) based solver specialized for dual block-angular MIPs, which include all two-stage stochastic mixed-integer programs (SMIPs). We have shown that leveraging the problem structure of SMIPs leads to three natural advantages. The first is data distribution, allowing us to potentially solve much larger instances than before, as demonstrated by PIPS-S and by the PIPS-SBB infrastructure. Second, operating on the rows of each scenario block independently is a natural source of task parallelism for PIPS-SBB. Last but not least, we see in Section III-C that a B&B code that distinguishes between first- and second-stage data in its algorithms can result in vastly improved performance.

TABLE II: Comparison specialized stochastic and general structure heuristics

Problem Instance	Cores	RelGap (Time) Stochastic PIPS-SBB	RelGap (Time) General PIPS-SBB
15.45.5	2	1.36 %	4.23%
15.45.10	2	7.93 %	8.39%
15.45.15	2	5.25 %	8.26%
5.25.50	1	(12.34s)	289.71%
5.25.100	1	(41.63s)	65.42%
10.50.50	5	1.48%	27.13%
10.50.100	10	1.74 %	28.60%
10.50.500	50	1.57 %	29.13%
10.50.1000	100	1.60 %	∞
10.50.2000	100	24.00 %	∞

It is clear from Section III-B that PIPS-SBB has a long way to go before it is competitive with commercial MIP solvers. Nevertheless, as we continue to work on the algorithms and add more functionality to the PIPS-SBB codebase, we expect the performance to improve significantly. We propose three natural directions of future work.

- **Adding B&B methods:** To improve our performance, we will implement a variety of cutting-plane methods and a stronger presolve, followed by other methods to accelerate the B&B algorithm.
- **Developing specialized stochastic MIP methods:** The effectiveness of our algorithm can be further improved by developing specialized methods for converging the bounds. These potentially include new Benders-like cuts and Lagrangian-like heuristics.
- **Exposing additional parallelism:** We will extend the PIPS-SBB code to search the B&B tree in parallel. This extended framework will have two inherent levels of parallelism — parallelizing the MIP tree and parallelizing the LP relaxation for each node of the B&B tree (already done by PIPS-S) – and can potentially utilize a large number of cores, due to the multiplicative effect of the levels of parallelism.

Acknowledgment

This work is performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. We would like to acknowledge gratefully the authors of the PIPS software suite (Miles Lubin, Cosmin Petra, Nai-Yuan Chiang, Feng Qiang, and others) for graciously making their software available under an open-source license. In particular, we would like to thank Cosmin Petra at Argonne National Laboratory for many

fruitful discussions on algorithms that leverage dual block-angular problem structure, and for adding new features to PIPS-S.

REFERENCES

[1] W. K. Klein Haneveld and M. H. van der Vlerk, "Stochastic integer programming: General models and algorithms," *Annals of Operations Research*, vol. 85, pp. 39–57, 1999.

[2] A. J. Kleywegt, A. Shapiro, and T. Homem-de Mello, "The sample average approximation method for stochastic discrete optimization," *SIAM Journal on Optimization*, vol. 12, no. 2, pp. 479–502, 2002.

[3] "IBM CPLEX optimizer," 2015, http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[4] "FICO Xpress Optimization Suite," http://www.fico.com/en/products/fico-xpress-optimization-suite.

[5] T. Achterberg, "SCIP: Solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.

[6] "Gurobi optimizer," 2015, http://www.gurobi.com.

[7] Y. Shinano and T. Fujie, "ParaLEX: A parallel extension for the CPLEX mixed integer optimizer," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2007, pp. 97–106.

[8] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch, "ParaSCIP: A parallel extension of SCIP," in *Competence in High Performance Computing 2010*. Springer, 2012, pp. 135–148.

[9] T. Ralphs, L. Ládanyi, and M. Saltzman, "A library hierarchy for implementing scalable parallel search algorithms," *J. Supercomput.*, vol. 28, no. 2, pp. 215–234, May 2004.

[10] T. Ralphs, L. Ladányi, and M. Saltzman, "Parallel branch, cut, and price for large-scale discrete optimization," *Mathematical Programming*, vol. 98, pp. 253–280, 2003.

[11] C. A. Phillips, J. Eckstein, and W. Hart, *Massively Parallel Mixed-Integer Programming: Algorithms and Applications*. SIAM Books, 2006, ch. 17, pp. 323–340.

[12] T. Koch, T. Ralphs, and Y. Shinano, "Could we use a million cores to solve an integer program?" *Mathematical Methods of Operations Research*, vol. 76, no. 1, pp. 67–93, 2012.

[13] L.-M. Munguía, G. Oxberry, and D. Rajan, "PIPS-SBB: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs," *Optimization Online*, 2015.

[14] S. Sen, "Algorithms for stochastic mixed-integer programming models," *Handbooks in operations research and management science*, vol. 12, pp. 515–558, 2005.

[15] C. C. Carøe and R. Schultz, "Dual decomposition in stochastic integer programming," *Operations Research Letters*, vol. 24, no. 1, pp. 37–45, 1999.

[16] "Stochastic programming software and test sets," http://stoprog.org/index.html?software.html.

[17] J.-P. Watson, D. L. Woodruff, and W. E. Hart, "PySP: Modeling and solving stochastic programs in Python," *Mathematical Programming Computation*, vol. 4, no. 2, pp. 109–149, 2012.

[18] K. Kim and V. M. Zavala, "Algorithmic innovations and software for the dual decomposition method applied to stochastic mixed-integer programs," *Optimization Online*, 2015.

[19] J. Hall, "Towards a practical parallelisation of the simplex method," *Computational Management Science*, vol. 7, no. 2, pp. 139–170, 2010.

[20] M. Lubin, J. Hall, C. G. Petra, and M. Anitescu, "Parallel distributed-memory simplex for large-scale stochastic LP problems," *Computational Optimization and Applications*, vol. 55, no. 3, pp. 571–596, 2013.

[21] M. W. Savelsbergh, "Preprocessing and probing techniques for mixed integer programming problems," *ORSA Journal on Computing*, vol. 6, no. 4, pp. 445–454, 1994.

[22] M. Fischetti and A. Lodi, "Heuristics in mixed integer programming," *Wiley Encyclopedia of Operations Research and Management Science*, 2011.

[23] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica: Journal of the Econometric Society*, pp. 497–520, 1960.

[24] T. Berthold, "Primal heuristics for mixed integer programs," 2006.

[25] H. Marchand, A. Martin, R. Weismantel, and L. Wolsey, "Cutting planes in integer and mixed integer programming," *Discrete Applied Mathematics*, vol. 123, no. 13, pp. 397 – 446, 2002.

[26] T. Achterberg, T. Koch, and A. Martin, "Branching rules revisited," *Operations Research Letters*, vol. 33, no. 1, pp. 42–54, 2005.

[27] R. Bixby and E. Rothberg, "Progress in computational mixed integer programming—a look back from the other side of the tipping point," *Annals of Operations Research*, vol. 149, pp. 37–41, 2007.

[28] M. Lubin, C. G. Petra, M. Anitescu, and V. Zavala, "Scalable stochastic optimization of complex energy systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 2011, pp. 1–10.

[29] J. Gondzio and A. Grothey, "A new unblocking technique to warmstart interior point methods based on sensitivity analysis," *SIAM Journal of Optimization*, vol. 19, no. 3, pp. 1184–1210, 2008.

[30] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[31] T. Berthold, "RENS: The optimal rounding," *Mathematical Programming Computation*, vol. 6, no. 1, pp. 33–54, 2014.

[32] L. Ntaimo and S. Sen, "The million-variable march for stochastic combinatorial optimization," *Journal of Global Optimization*, vol. 32, no. 3, pp. 385–400, 2005.

[33] S. Ahmed, R. Garcia, N. Kong, L. Ntaimo, G. Parija, F. Qiu, and S. Sen, "SIPLIB: A stochastic integer programming test problem library," 2013. [Online]. Available: http://www.isye.gatech.edu/~sahmed/siplib