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

Lluís-Miquel Munguía*[1], Geoffrey Oxberry[2] and Deepak Rajan[3]

[1]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332
[2]Computational Engineering Division, Lawrence Livermore National Laboratory, Livermore, CA 94550
[3]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550

December 22, 2015

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: a distributed-memory parallel stochastic MIP solver that takes advantage of parallelism at multiple levels of the optimization process. We show promising results on 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.

1 Introduction

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

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

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 $\{1, \ldots, n\}$. Without loss of generality, x is bounded below by $l \in \mathbb{R}^n$ and above by $u \in \mathbb{R}^n$.

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 under uncertainty. After each decision is made, one learns

*lluis.munguia@gatech.edu

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 [21] 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

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

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

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

where W, h, l, u, q and T may depend on ξ, but not on x, and I_2 is the set of second-stage integer variable indices. We assume this set does not depend on ξ. If (SS^x_ξ) 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 [22]; thus transforming the problem into one large MIP (usually called the *extensive formulation*) and enabling the use of traditional MIP approaches. We use $[s]$ to represent the set of possible sampled realizations, also known as scenarios. Suppose scenario i has probability p_i. The extensive formulation is:

$$\text{(EXT)} \qquad \begin{array}{llllll}
\multicolumn{6}{l}{\displaystyle\min_{x\in\mathbb{R}^{n_1}, y\in\mathbb{R}^{n_2\times s}} c^{\mathrm{T}}x + \sum_{i=1}^{s} p_i q_i^{\mathrm{T}} y_i,} \\
\multicolumn{6}{l}{\text{subject to:}} \\
Ax & & & & & = b_0, \\
T_1x & +W_1y_1 & & & & = b_1, \\
T_2x & & +W_2y_2 & & & = b_2, \\
\vdots & & & \ddots & & \vdots \\
T_sx & & & & +W_sy_s & = b_s, \\
\multicolumn{6}{r}{l \le x \le u,} \\
\multicolumn{6}{r}{l_i \le y_i \le u_i, \quad \forall i \in [s],} \\
\multicolumn{6}{r}{x_j \in \mathbb{Z}, \quad \forall j \in I_1,} \\
\multicolumn{6}{r}{y_{i,j} \in \mathbb{Z}, \quad \forall i \in [s], \forall j \in I_2.}
\end{array}$$

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^x_ξ) 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 number of second-stage decision variables and constraints has increased by a factor of s with respect to (SS^x_ξ).

Extensive formulations can be solved with a general purpose state-of-the-art MIP solver such as CPLEX [12], Xpress [47], SCIP [1] or GUROBI [33]. 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 2.1 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) except in reviewing previous work in Section 1.2, where the distinction between SMIPs and their extensive formulations can be inferred from context.

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 [43, 42], BLIS [36, 38], and PICO [34]. However, all of these efforts focus on parallel B&B, and not on parallelization of the simplex algorithm used to solve the LP relaxation solved at each node of the B&B algorithm. Though MIP solvers (especially the state-of-the-art 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 [23] and thus have been unable to leverage recent advances in HPC architectures. Furthermore, 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.

1.1 Contributions and Overview

In our approach, we leverage the dual block-angular structure of SMIPs to solve LP relaxations at each node of the B&B tree using a parallel algorithm. The simplex method is the default LP algorithm. Despite its limited parallel 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 [27], such as the LP relaxation of the extensive formulation. In our approach, we use the PIPS-S solver to solve the LP relaxations at the nodes of the B&B tree. This allows us to build a distributed-memory B&B-based solver for Stochastic MIPs called PIPS-SBB (PIPS - Simple Branch and Bound)[1]. To improve the performance of PIPS-SBB, we also developed new (and computationally efficient) methods for pre-processing, cut generation 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 such that each (A, b_0, T_i, W_i, b_i) tuple for $i \in [s]$ is stored

[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. The Cbc solver replaced Sbb in COIN. PIPS abbreviates "Parallel Interior Point Solvers"; the name was chosen prior to the work in [27] on parallel simplex algorithms for solving dual block-angular LPs.

in memory on only one MPI process, PIPS-SBB can leverage the distributed-memory architecture of supercomputers to address more memory to store and solve LP relaxations. In contrast, existing MIP solvers must load the entire extensive formulation in memory on each process that solves LP relaxations; these solvers parallelize the branch-and-bound tree, not the solution of LP relaxations.

- Adapting methods known to accelerate MIP solvers (Presolve [39] and Primal Heuristics [13]) to a distributed-memory setting with dual block-angular data structure, we present initial results on benchmark SIPLIB instances that show the effectiveness of our method.
- Building PIPS-SBB as a modular, expandable codebase, we enable easy implementation of features, modifications, and extensions as plug-ins.

The remainder of this work proceeds as follows: First, we review related work on SMIP decomposition schemes to conclude Section 1. Then, in Section 2, we describe B&B algorithms, and present the main ideas behind PIPS-SBB. In particular, we focus our design choices for data distribution and parallelism. In Section 3, we describe the structure of PIPS-SBB in detail, in particular its extensible software implementation. We also describe 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 4 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 5.

1.2 Related Work: Solving SMIPs using decomposition schemes

Most of the work in the literature avoids solving extensive formulations and alternative problem decompositions are devised instead.

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 [40] and [26] for a detailed survey. Among its strengths, stage-wise decomposition can leverage the dual block-angular problem structure. However, these schemes are computationally impractical due to their slow convergence. Furthermore, they typically cannot handle the most general case where both stages are mixed-integer and when the data in the second-stage problem is allowed to depend on the scenario. In contrast, our scheme is general and applicable to any two-stage SMIP, as it allows binary, continuous and discrete variables in both stages. To our knowledge, the only stage-wise scheme without variable type limitations has been presented by [37]. In this work, the authors present a novel convergent generalization of Benders' algorithm, albeit in a theoretical context. However, their approach does not allow second-stage data W and q to depend on the scenario. Other works based on stage-wise decomposition schemes such as [48, 41, 6] present more limitations.

Alternatively, one can rely on scenario-based decomposition schemes. In such schemes, stages are decoupled by the replication of first-stage variables for each scenario. Then, non-anticipativity constraints are used to ensure these first-stage variables remain identical across scenarios. The decomposition is achieved by relaxing such constraints. Many heuristic scenario-wise decomposition strategies have been developed for stochastic mixed-integer programs. For example, Progressive Hedging iteratively solves single-scenario relaxations until convergence. Progressive Hedging has been used as a successful primal heuristic [46] and to obtain lower bounds [14]. To our knowledge, the first exact scenario-wise decomposition scheme was presented by [11], in which the authors solve Lagrangian duals at each node of a B&B procedure. In [28], the authors

expanded the same work by presenting a parallel algorithm, which showed potential for parallel speedup, as well as some barriers to scalability. However, the authors only attempt to solve a single relaxation and do not address the challenges of incorporating such schemes within a B&B framework. In combination with Progressive Hedging, a recent parallel implementation of the dual decomposition scheme looks highly promising [18]. Other scenario-based decomposition schemes with variable type limitations include [4], where cover inequalities are used to solve SMIPs with pure binary first-stage variables.

Additional implementations of B&B algorithms using decomposition-based relaxations such as Generic Column Generation (GCG) [15], BapCod [45] and Dip [35] are typically not competitive when compared to LP-relaxation based MIP solvers since they rely on relaxations that cannot be warm-started within a B&B scheme.

There exists very few parallel software libraries that model and solve mixed-integer stochastic programs, with the exception of PySP [46]. In [20], the authors introduce DSP - a parallel implementation based on a Lagrangian scheme in combination with Benders-type cuts. In [25], the authors introduce a parallel implementation of Benders decomposition for stochastic MIPs with first-stage integer variables. There are many sequential implementations of stage-wise and scenario-wise decomposition schemes; a list of software is maintained at [44].

2 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 data 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 2.1, we overview the B&B algorithm. Then, in Section 2.2, we discuss how to expose parallelism in solving the LP relaxation. This discussion naturally segues into a description of MIP data distribution in Section 2.3. We defer discussion of structure-aware MIP infrastructure details such as developing branching rules, primal heuristics, and presolve to Section 3.

2.1 Branch and Bound

In Branch and Bound (B&B) [24], 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[3] 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 [4]. The solution to each LP relaxation supplies information to the partitioning-and-search algorithm. 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

[2] Aside from reductions, the algorithms presented in [27] operate on second-stage blocks of dual block-angular LPs independently.

[3] Despite the nomenclature, this data structure is frequently implemented as a heap.

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

point is also a feasible solution to the MIP. The objective value of this feasible solution provides an upper bound to the MIP optimal solution value. However, if the LP relaxation solution calculated 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 solution value of LP relaxation is larger than the overall upper bound (from the solution value of all integer feasible solutions found so far). The latter step is *branching*, in which inequalities are added in order to eliminate the fractional solution (which is LP-feasible but not integer infeasible) 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

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

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 [13, 8] 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 developing stronger formulations. One method for strengthening formulations adds cutting planes (inequalities) [30] 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 [39], 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 [2]. State-of-the-art MIP solvers are highly optimized in all these aspects, representing over two decades of research making these strategies work synergistically [10]. The current version of PIPS-SBB contains a subset of these methods. We plan on implementing more of these methods in the future to improve performance.

Beyond the 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 because 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 are presented in section 3.2. Leveraging the dual block-angular problem structure is a critical feature of PIPS-SBB; in Section 4.3, we see that with specialized branching rules PIPS-SBB reduces the relative gap faster with a smaller number of nodes and with specialized primal heuristics PIPS-SBB finds high quality feasible solutions early in the search process.

2.2 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 [29]. 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 [16]) 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 [27] 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 at 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 the LP relaxations of multiple B&B nodes simultaneously.

2.3 Parallel data distribution

For scalability, PIPS-SBB is designed for distributed-memory parallel computer architectures. Distributed-memory paradigms assume that 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 [17] are required in order to enable coordination between processes. PIPS-SBB relies strictly on collective MPI communication primitives to communicate efficiently among processes, both in the B&B algorithm and while solving the LP relaxations.

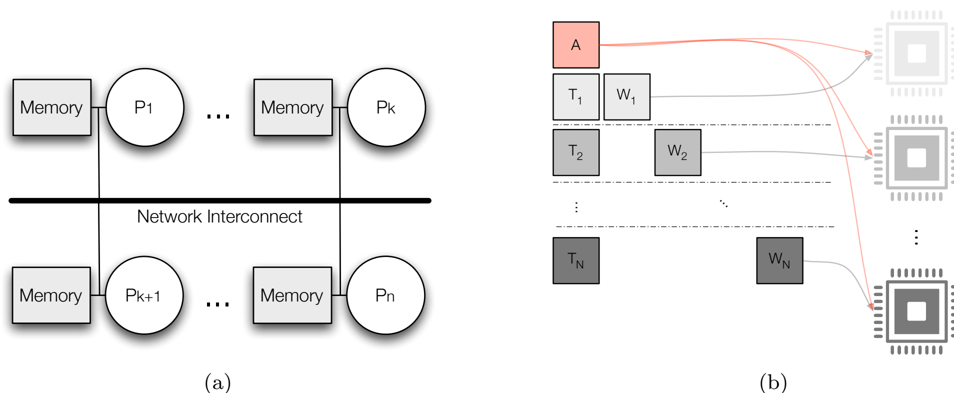

Figure 1: (a) Schematic depiction of a parallel system with a distributed-memory configuration. It has a segmented memory space, which is distributed among different processes. (b) Data parallelism in PIPS-SBB.

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.

3 Implementation

In this section, we present the structure of PIPS-SBB in detail, describing both its software architecture and its features. Section 3.1 discusses the main software components of PIPS-SBB and their concerns, including where users can add their own functionality, such as branching rules. Section 3.2 discusses how existing MIP algorithms are adapted to versions that are dual block-angular structure-aware, exploiting this structure to expose parallelism and increase performance.

3.1 Software architecture of PIPS-SBB

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.

PIPS-SBB code is distributed in two main software components, presented in Figure 2 using a schematic UML representation of the components as well as the interactions between them. The Solver Component manages the distributed problem data. It includes all algorithms that must directly access the distributed data, such as the presolver and the interface to the PIPS-S simplex solver. The Search State Component coordinates the B&B tree search and contains the current B&B algorithm state. It includes tree search algorithms such as branching rules and primal heuristics, and tree node data such as warm-start and branching information.

The problem formulation is read, stored and managed within the Solver Component of PIPS-SBB. The `BBSMPSSolver` class is one of the most critical components, as it acts as a proxy for outer software abstractions that may require LP relaxations or access to the solution pool. Due to its ubiquitous access from other classes, it is implemented as a singleton class (represented by the ^{1} next to the class name in Figure 2). The associated `BBSMPSPresolver` class performs all data presolving operations and `PIPSSInterface` serves as the interface for the LP relaxation solver.

In the Search State Component, the `BBSMPSTree` stores the collection of open nodes and controls the B&B tree search through a set of optimization managers. Currently, we have implemented PIPS-SBB Managers that provide users with a flexible and extendable framework to coordinate the execution of primal heuristics and branching rules within the search process. For instance, these managers enable users to determine which primal heuristics are executed and the frequency with which they do so. Under the manager paradigm, the creation of additional heuristics and branching rules becomes a simple process, consisting of generating a new class by extending the generic `BBSMPSHeuristic` or `BBSMPSBranchingRule` and registering it in the appropriate manager. Observe from Figure 2 that there may be no `BBSMPSHeuristic` defined (0..*), but there must be at least one `BBSMPSBranchingRule` defined (1..*). In future versions of PIPS-SBB, other managers will coordinate the execution of other features, such as cutting-plane algorithms and tree search strategies.

In designing the architecture of PIPS-SBB, care is taken to minimize the memory footprint, allowing PIPS-SBB to solve large problems instances. For example, information related to relax-

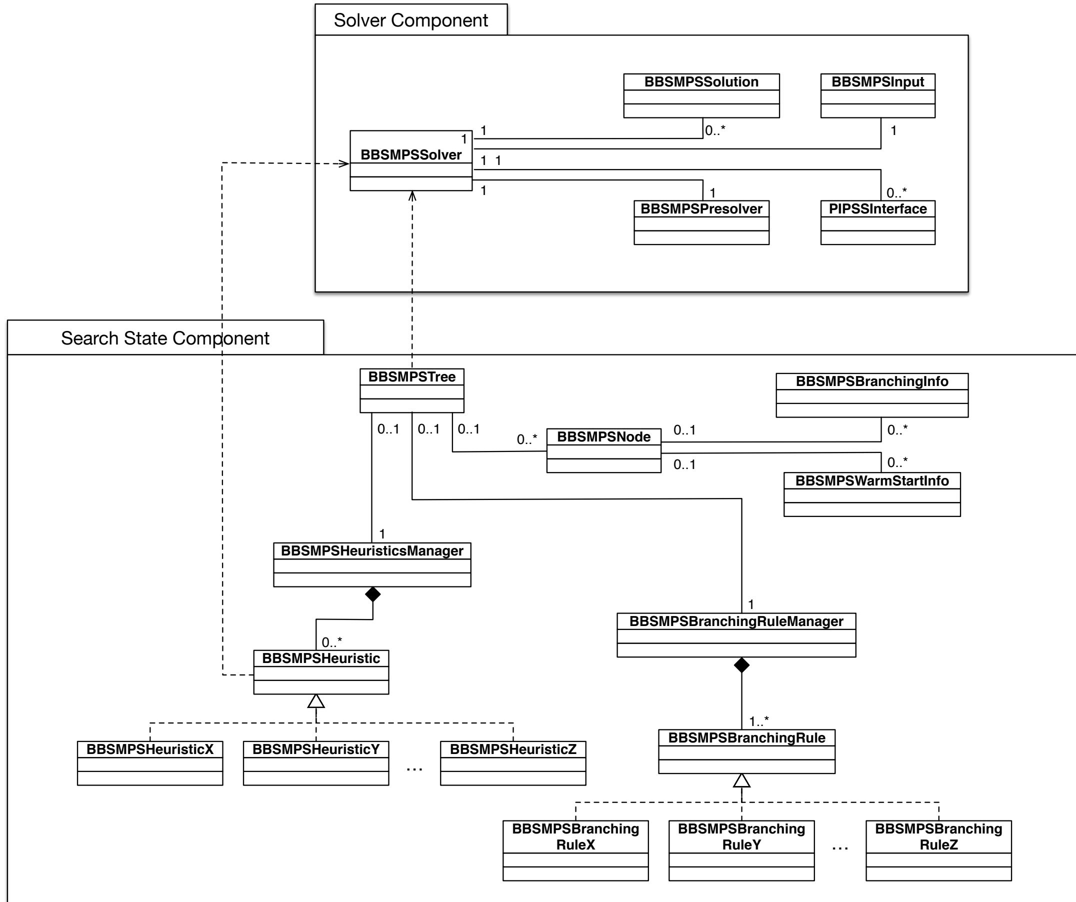

Figure 2: Class diagram of PIPS-SBB. Classes are shown as boxes with the top section as the name of the class. Interactions between classes are depicted as lines between classes, and indicate multiplicity indicators at each end, for example (1..*) representing "one or more".

ations such as LP warm-start information and branching decisions are stored incrementally with respect to the parent problem. Otherwise, the algorithm would quickly run out of memory to store tree information, as is the case for the state-of-the-art solver CPLEX when solving certain problem instances; see Section 4.2.

3.2 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 2.3. 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.

3.2.1 Branching Rules

The current version of PIPS-SBB features three branching rules: a simple 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, the search for the most infeasible second-stage variable is parallelized. An all-to-all reduction primitive is then required in order to find the most infeasible second-stage variable and communicate it to all processes.

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 $I_1 \cap F \neq \emptyset$ **then** ▷ F is the index set of all fractional-valued first-stage variables
    return $[scen, \text{argmax}_j\{|x_j - \lceil x_j \rfloor|,\ j \in I_1\}]$ ▷ 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 $I_2 \cap F^i \neq \emptyset$ **then** ▷ F^i is the index set of all fractional second-stage variables of scenario i
    $idx_i := \text{argmax}_j\{|y_{i,j} - \lceil y_{i,j} \rfloor|,\ j \in I_2\}$
    $frac_i := |y_{i,idx_i} - \lceil y_{i,idx_i} \rfloor|$
  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

3.2.2 Primal Heuristics

The current version of PIPS-SBB features ten primal heuristics, ranging from simple rounding and diving schemes to more computationally expensive neighborhood search schemes such as RENS [9]. 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 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.

3.2.3 Presolving

MIP presolve is implemented in PIPS-SBB following Savelsbergh [39, Section 1], with the exception of deleting redundant constraints. In order to accommodate the distributed nature of

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
    ▷ Compute total number of fractional-valued second-stage variables over all processes
    Using y_i, compute F^i, the index set of all fractional-valued second-stage variables of scenario i
    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
```

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 to update first- and second-stage variable bounds, update second-stage constraints, and assess MIP feasibility. Since information on first-stage variable bounds and MIP feasibility may be different on different MPI processes (ranks) due to preprocessing second-stage constraints in distributed fashion, this information must be synchronized across all processes via appropriate all-to-all reduction operations, as depicted in Algorithm 3.

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 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
```

4 Experimental Results

We illustrate the performance of PIPS-SBB using instances from SIPLIB [3], a testbed of stochastic mixed-integer programs. In particular, we solve instances from the following test suites: Stochastic Server Location Problem (SSLP), Stochastic Server Location Problem Replication (SSLPRep), Stochastic Multiple Knapsack Problem (SMKP), and Dynamic CAPacity acquisition and allocation under uncertainty (DCAP). All our 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 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.

4.1 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. We present scaling results on instances from SSLP [32], 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, and 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.

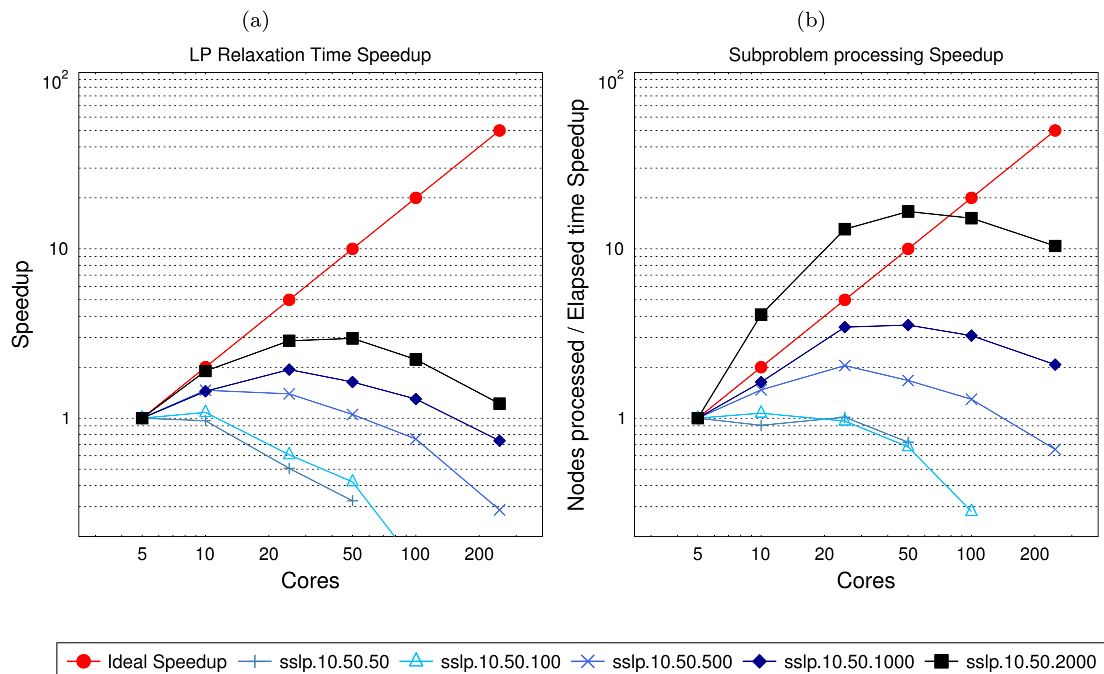

Figure 3: (a) Strong scaling performance results of PIPS-S. (b) Strong scaling throughput results of PIPS-SBB.

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 3a 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 [27, 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 5. Since PIPS-SBB solves an LP relaxation 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 3b 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 speedup is superior to PIPS-S, and even shows super-linear scaling for large problem instances. While this 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) allowing PIPS-SBB to process many more nodes in the time limit. As 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. 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 3 indicate that PIPS-SBB throughput scales in a manner consistent with that of PIPS-S performance.

We are primarily interested in how PIPS-SBB *performance* scales, as we should be. 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 4, we see analogous performance curves, but with a decrease in speedup relative to the results shown in Figure 3b. This reduction is mainly due to the time spent by PIPS-SBB in computationally expensive primal heuristics in an attempt to find good feasible solutions. As before, performance peaks around 25-50 cores for the larger instances.

4.2 Overall Performance

We illustrate the overall performance of PIPS-SBB on many instances from the SIPLIB library. For these experiments, 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 4.1. We report the number of scenarios and processor configuration as "Scenarios (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 seconds), and the performance results are reported as "(Time)"

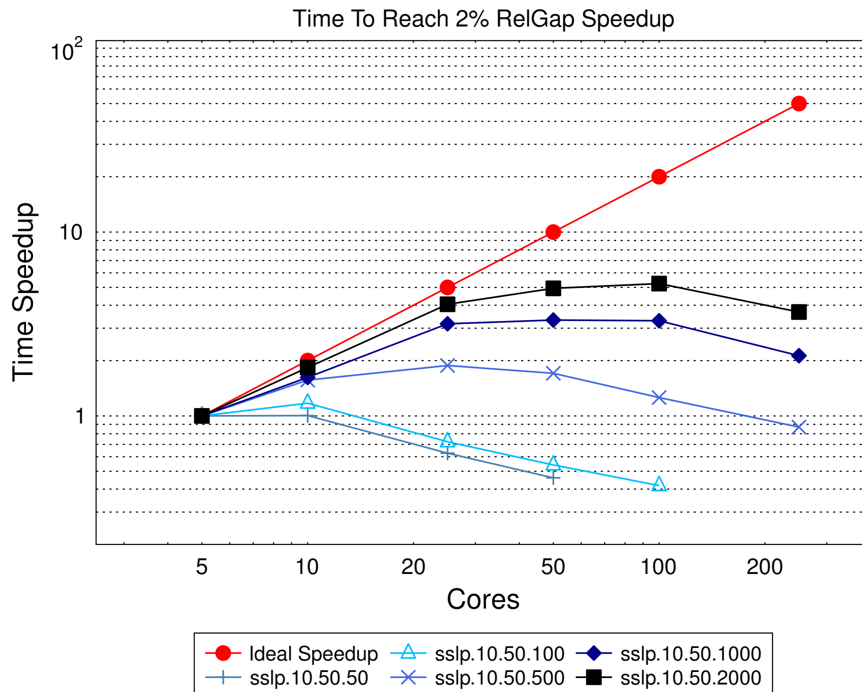

Figure 4: Strong scaling performance results of PIPS-SBB

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 "CPLEX RelGap (Time)" column. The instances where CPLEX ran out of memory are denoted with (M) next to the GAP at termination.

4.2.1 SSLP

The SSLP instance set is formed by 12 model server location problems [32]. As mentioned earlier, this set contains the largest variation in the number of scenarios (ranging from 5 to 2000), a pure binary first-stage and mixed-binary second-stage.

From Table 1, 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 4.3, 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 1: SSLP instance set results

Problem Instance	Scenarios (Cores)	RelGap (Time)	Best Solution Time	Best Solution Quality	CPLEX RelGap (Time)
sslp.15.45.5	5 (2)	1.36%	1488s	1.07%	(4s)
sslp.15.45.10	10 (2)	7.93%	2129s	7.26%	(1s)
sslp.15.45.15	15 (2)	5.25%	2392s	4.84%	(12s)
sslp.5.25.50	50 (1)	(12.34s)	12s	0%	(1s)
sslp.5.25.100	100 (1)	(41.63s)	41s	0%	(1s)
sslp.10.50.50	50 (5)	1.48%	923s	1.31%	(81s)
sslp.10.50.100	100 (10)	1.74%	194s	1.56%	(442s)
sslp.10.50.500	500 (50)	1.57%	2792s	**-7.32%**	(M) 10.13%
sslp.10.50.1000	1000 (100)	1.60%	2397s	**-11.19%**	14.47%
sslp.10.50.2000	2000 (100)	24.00%	2384s	**-0.73%**	20.33%

4.2.2 SSLPRep

SSLPRep instances are slight variations of the SSLP set, available at [31]. The results displayed in Table 2 show a performance analogous to the SSLP instances. As before, while PIPS-SBB is not able to close the RelGap for the smaller instances, performance is comparable to CPLEX for the larger instances. As in the SSLP case, from column Best Solution Quality, we see that PIPS-SBB obtains better upper bounds than CPLEX for some large instances. Overall, PIPS-SBB outperforms CPLEX on 6 out of 50 instances.

It is interesting to note that the problem structure has more of an impact on solution time (for both PIPS-SBB and CPLEX) for instances with a small number of scenarios as shown by the variability in solution time among the sslp.15.45.* instances as opposed to the solution times for the sslp.10.50.* instances.

4.2.3 DCAP

The DCAP instances consist of a set of 12 two-stage stochastic integer programs with mixed-integer first-stage variables and pure binary second-stage variables. They model dynamic capacity acquisitions and allocations under uncertainty [5]. As seen in Table 3, PIPS-SBB shows a substantially inferior performance in comparison to CPLEX on finding improvements in both the

Table 2: SSLPRep instance set results

Problem Instance	Scenarios (Cores)	RelGap (Time)	Best Solution Time	Best Solution Quality	CPLEX RelGap (Time)
sslp.15.45.5a	5 (2)	0.77%	2956s	0.6%	(2s)
sslp.15.45.5b	5 (2)	(38.05s)	38s	0%	(1s)
sslp.15.45.5c	5 (2)	6.94%	3288s	6%	(5s)
sslp.15.45.5d	5 (2)	4.14%	1378s	3.92%	(2s)
sslp.15.45.5e	5 (2)	5.18%	92s	4.4%	(4s)
sslp.15.45.10a	10 (2)	6.24%	606s	5.78%	(23s)
sslp.15.45.10b	10 (2)	6.46%	2336s	5.58%	(10s)
sslp.15.45.10c	10 (2)	0.51%	3200s	0.51%	(5s)
sslp.15.45.10d	10 (2)	7.30%	3600s	6.36%	(13s)
sslp.15.45.10e	10 (2)	0.43%	1311s	0.39%	(1s)
sslp.15.45.15a	15 (2)	8.93%	2290s	7.91%	0.04%
sslp.15.45.15b	15 (2)	7.58%	11s	6.23%	(34s)
sslp.15.45.15c	15 (2)	9.06%	1326s	7.37%	(41s)
sslp.15.45.15d	15 (2)	12.32%	1362s	10.33%	(279s)
sslp.15.45.15e	15 (2)	4.07%	3215s	3.69%	(6s)
sslp.5.25.50a	50 (1)	(19.27s)	15s	0%	(1s)
sslp.5.25.50b	50 (1)	(14.88s)	15s	0%	(1s)
sslp.5.25.50c	50 (1)	(13.9s)	11s	0%	(1s)
sslp.5.25.50d	50 (1)	(13.59s)	14s	0%	(1s)
sslp.5.25.50e	50 (1)	(13.63s)	13s	0%	(1s)
sslp.5.25.100a	100 (1)	(2255.2s)	2068s	0%	(20s)
sslp.5.25.100b	100 (1)	(198.52s)	195s	0%	(1s)
sslp.5.25.100c	100 (1)	(44.14s)	44s	0%	(1s)
sslp.5.25.100d	100 (1)	(45.07s)	45s	0%	(1s)
sslp.5.25.100e	100 (1)	(43.33s)	41s	0%	(1s)
sslp.10.50.50a	50 (5)	2.36%	1451s	2.11%	(102s)
sslp.10.50.50b	50 (5)	1.67%	1708s	1.53%	(99s)
sslp.10.50.50c	50 (5)	2.31%	891s	2.02%	(765s)
sslp.10.50.50d	50 (5)	2.34%	3369s	2.16%	(15s)
sslp.10.50.50e	50 (5)	2.59%	2493s	2.32%	(251s)
sslp.10.50.100a	100 (10)	2.03%	1234s	1.70%	(233s)
sslp.10.50.100b	100 (10)	1.83%	3266s	1.64%	(161s)
sslp.10.50.100c	100 (10)	2.19%	1206s	1.96%	(248s)
sslp.10.50.100d	100 (10)	2.71%	2531s	2.45%	(52s)
sslp.10.50.100e	100 (10)	2.96%	3490s	2.60%	(267s)
sslp.10.50.500a	500 (50)	2.25%	2330s	0.53%	1.88%
sslp.10.50.500b	500 (50)	2.35%	2550s	2.03%	0.2%
sslp.10.50.500c	500 (50)	2.40%	2801s	0.63%	(M) 1.69%
sslp.10.50.500d	500 (50)	2.75%	2395s	**-3.69%**	(M) 6.33%
sslp.10.50.500e	500 (50)	3.26%	2456s	2.57%	0.5%
sslp.10.50.1000a	1000 (100)	2.41%	3565s	**-2.21%**	6.1%
sslp.10.50.1000b	1000 (100)	2.52%	3451s	**-2.05%**	5.07%
sslp.10.50.1000c	1000 (100)	2.60%	3611s	2.24%	0.28%
sslp.10.50.1000d	1000 (100)	3.00%	3233s	2.28%	0.43%
sslp.10.50.1000e	1000 (100)	3.34%	3547s	**-1.98%**	5.32%
sslp.10.50.2000a	2000 (100)	24.12%	2438s	1.31%	18.62%
sslp.10.50.2000b	2000 (100)	24.72%	3610s	6.69%	12.44%
sslp.10.50.2000c	2000 (100)	21.66%	2183s	**-0.12%**	18.46%
sslp.10.50.2000d	2000 (100)	9.72%	2094s	**-5.85%**	14.46%
sslp.10.50.2000e	2000 (100)	20.83%	947s	1.64%	14.57%

lower bound and the upper bound. Advanced preprocessing and cutting-plane methods enable CPLEX to solve all instances. As we suggest in Section 5, the addition of cutting-plane methods in future releases of PIPS-SBB will narrow the current performance gap between PIPS-SBB and CPLEX.

Table 3: DCAP instance set results

Problem Instance	Scenarios (Cores)	RelGap (Time)	Best Solution Time	Best Solution Quality	CPLEX RelGap (Time)
dcap233_200	200 (20)	58.89%	2s	21.50%	(1s)
dcap233_300	300 (20)	68.75%	3s	47.59%	0.01%
dcap233_500	500 (50)	65.15%	8s	32.26%	(2s)
dcap243_200	200 (20)	50.15%	2s	26.46%	(1s)
dcap243_300	300 (20)	49.43%	4s	23.43%	(10s)
dcap243_500	500 (50)	53.96%	9s	31.57%	(12s)
dcap332_200	200 (20)	84.48%	1482s	63.66%	0.01%
dcap332_300	300 (20)	81.79%	248s	35.00%	(26s)
dcap332_500	500 (50)	87.36%	345s	38.85%	(78s)
dcap342_200	200 (20)	68.93%	104s	36.21%	(33s)
dcap342_300	300 (20)	69.31%	585s	30.86%	(88s)
dcap342_500	500 (50)	68.30%	536s	25.59%	(405s)

4.2.4 SMKP

The SMKP instance set is formed by 30 instances of a stochastic multiple knapsack problem. Each problem contains binary variables in both stages and knapsack constraints [7]. As seen in Table 4, Compared to DCAP, we see in Table 4 that PIPS-SBB performs much better in terms of the RelGap at termination. CPLEX is unable to solve two instances due to memory limitations in the B&B tree search. For the remaining ones, it is able to obtain smaller RelGap than PIPS-SBB, mainly due to finding better feasible solutions.

4.3 Specialized Structure-Aware algorithms

As explained in Section 3.2, 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 randomly, so that all variables (first- and second-stage) have an equal probability 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 5. 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.

5 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

Table 4: SMKP instance set results

Problem Instance	Scenarios (Cores)	RelGap (Time)	Best Solution Time	Best Solution Quality	CPLEX RelGap (Time)
smkp_1	20 (2)	0.50%	2921s	0.35%	0.14%
smkp_2	20 (2)	0.43%	1672s	0.28%	0.13%
smkp_3	20 (2)	0.57%	2080s	0.41%	0.14%
smkp_4	20 (2)	0.51%	3299s	0.34%	0.15%
smkp_5	20 (2)	0.59%	1650s	0.39%	0.12%
smkp_6	20 (2)	0.72%	3318s	0.50%	0.14%
smkp_7	20 (2)	0.70%	952s	0.46%	(M) 0.11%
smkp_8	20 (2)	0.57%	523s	0.36%	0.15%
smkp_9	20 (2)	0.62%	3584s	0.42%	0.17%
smkp_10	20 (2)	0.66%	3538s	0.33%	0.22%
smkp_11	20 (2)	0.55%	694s	0.26%	0.25%
smkp_12	20 (2)	0.68%	122s	0.38%	0.17%
smkp_13	20 (2)	0.64%	254s	0.30%	0.26%
smkp_14	20 (2)	0.72%	1029s	0.09%	(M) 0.17%
smkp_15	20 (2)	0.59%	3080s	0.34%	0.11%
smkp_16	20 (2)	0.62%	259s	0.33%	0.20%
smkp_17	20 (2)	0.62%	857s	0.36%	0.14%
smkp_18	20 (2)	0.61%	340s	0.34%	0.17%
smkp_19	20 (2)	0.77%	111s	0.45%	0.16%
smkp_20	20 (2)	0.66%	126s	0.38%	0.17%
smkp_21	20 (2)	0.84%	3148s	0.50%	0.17%
smkp_22	20 (2)	0.66%	2209s	0.29%	0.26%
smkp_23	20 (2)	0.71%	3177s	0.42%	0.18%
smkp_24	20 (2)	0.71%	1992s	0.44%	0.14%
smkp_25	20 (2)	0.60%	1552s	0.27%	0.16%
smkp_26	20 (2)	0.73%	1441s	0.48%	0.14%
smkp_27	20 (2)	0.69%	439s	0.35%	0.19%
smkp_28	20 (2)	0.67%	507s	0.35%	0.18%
smkp_29	20 (2)	0.89%	1941s	0.55%	0.20%
smkp_30	20 (2)	0.82%	1802s	0.40%	0.31%

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 4.3 that a B&B code that distinguishes between first- and second-stage data in its algorithms can result in vastly improved performance.

It is clear from Section 4.2 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 four natural directions of future work.

- **Adding B&B methods:** It is well known that the success of a B&B scheme is dependent on the optimized implementation of a variety of schemes for converging MIP bounds, including cuts, presolve, and heuristics. By specializing heuristics, branching strategies, and a very simple presolve for stochastic MIPs, our experiments show promise, but at the same time indicate how far we still have to go. Next, 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.

Table 5: Comparison specialized stochastic and general structure heuristics

Problem Instance	Scenarios (Cores)	RelGap (Time) Stochastic PIPS-SBB	RelGap (Time) General PIPS-SBB
sslp_15_45_5	5 (2)	1.36 %	4.23%
sslp_15_45_10	10 (2)	7.93 %	8.39%
sslp_15_45_15	15 (2)	5.25 %	8.26%
sslp_5_25_50	50 (1)	(12.34s)	289.71%
sslp_5_25_100	100 (1)	(41.63s)	65.42%
sslp_10_50_50	50 (5)	1.48%	27.13%
sslp_10_50_100	100 (10)	1.74 %	28.60%
sslp_10_50_500	500 (50)	1.57 %	29.13%
sslp_10_50_1000	1000 (100)	1.60 %	∞
sslp_10_50_2000	2000 (100)	24.00 %	∞

These potentially include new Benders-like cuts and Lagrangian-like heuristics.

- **Exposing additional parallelism:** Currently, PIPS-SBB can utilize as many cores in parallel as the number of scenarios in the stochastic MIP. However, our experiments in Section 4 show that the performance of PIPS-S is best when using fewer cores than the number of scenarios. To expose additional parallelism, especially when the number of available cores is far larger than the number of scenarios, 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). Since many relaxations are being solved simultaneously in this two-level framework (one for each node of the B&B tree), the available task parallelism will be limited by the amount of computational resources and not by the number of scenarios. Such a framework can potentially utilize a large number of cores, due to the multiplicative effect of the levels of parallelism. For instance, a 100-scenario stochastic MIP solved by using 10 parallel B&B tree searches, and 100 cores for solving each LP relaxation scales to 1000 cores overall.

  The current implementation of PIPS-SBB, as described in Section 3, can be easily extended to instantiate multiple distributed `BBSMPSTree` objects that search separate parts of the B&B tree and are managed by a centralized coordinator. While exposing more parallelism by parallelizing the B&B search will use more cores and increase performance, it must be noted that parallel efficiency will probably decrease due to the well-known loss of parallel efficiency in B&B search [23].

- **Improving usability:** Even though PIPS-SBB is released as open-source code, allowing users to modify the algorithm as needed, future versions of PIPS-SBB will provide a callback mechanism to allow users to influence or even override its methods in a more convenient fashion. These would include callbacks for various components of a B&B tree search, such as node selection, adding cutting planes, and heuristics to find feasible solutions. A command-line interface will enable easy modification and parameterization of the PIPS-SBB solver.

6 Acknowledgements

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] Tobias Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[2] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[3] S. Ahmed, R. Garcia, N. Kong, L. Ntaimo, G. Parija, F. Qiu, and S. Sen. SIPLIB: A stochastic integer programming test problem library, 2013.

[4] Shabbir Ahmed. A scenario decomposition algorithm for 0–1 stochastic programs. *Operations Research Letters*, 41(6):565–569, 2013.

[5] Shabbir Ahmed and Renan Garcia. Dynamic capacity acquisition and assignment under uncertainty. *Annals of Operations Research*, 124(1-4):267–283, 2003.

[6] Shabbir Ahmed, Mohit Tawarmalani, and Nikolaos V Sahinidis. A finite branch-and-bound algorithm for two-stage stochastic integer programs. *Mathematical Programming*, 100(2):355–377, 2004.

[7] Gustavo Angulo, Shabbir Ahmed, and Santanu S Dey. Improving the integer L-shaped method. 2014.

[8] Timo Berthold. Primal heuristics for mixed integer programs. 2006.

[9] Timo Berthold. RENS: The optimal rounding. *Mathematical Programming Computation*, 6(1):33–54, 2014.

[10] Robert Bixby and Edward Rothberg. Progress in computational mixed integer programming—a look back from the other side of the tipping point. *Annals of Operations Research*, 149:37–41, 2007.

[11] Claus C Carøe and Rüdiger Schultz. Dual decomposition in stochastic integer programming. *Operations Research Letters*, 24(1):37–45, 1999.

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

[13] Matteo Fischetti and Andrea Lodi. Heuristics in mixed integer programming. *Wiley Encyclopedia of Operations Research and Management Science*, 2011.

[14] Dinakar Gade, Gabriel Hackebeil, Sarah Ryan, Jean-Paul Watson, Roger Wets, and David Woodruff. Obtaining lower bounds from the progressive hedging algorithm for stochastic mixed-integer programs. 2014.

[15] Gerald Gamrath and Marco E. Lübbecke. Experiments with a generic dantzig-wolfe decomposition for integer programs. In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 239–252. Springer Berlin Heidelberg, 2010.

[16] Jacek Gondzio and Andreas Grothey. A new unblocking technique to warmstart interior point methods based on sensitivity analysis. *SIAM Journal of Optimization*, 19(3):1184–1210, 2008.

[17] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[18] Ge Guo, Gabriel Hackebeil, Sarah M Ryan, Jean-Paul Watson, and David L Woodruff. Integration of progressive hedging and dual decomposition in stochastic integer programs. *Operations Research Letters*, 43(3):311–316, 2015.

[19] J. Hall. Towards a practical parallelisation of the simplex method. *Computational Management Science*, 7(2):139–170, 2010.

[20] Kibaek Kim and Victor M Zavala. Algorithmic innovations and software for the dual decomposition method applied to stochastic mixed-integer programs. *Optimization Online*, 2015.

[21] Willem K. Klein Haneveld and Maarten H. van der Vlerk. Stochastic integer programming: General models and algorithms. *Annals of Operations Research*, 85:39–57, 1999.

[22] Anton J Kleywegt, Alexander Shapiro, and Tito Homem-de Mello. The sample average approximation method for stochastic discrete optimization. *SIAM Journal on Optimization*, 12(2):479–502, 2002.

[23] Thorsten Koch, Ted Ralphs, and Yuji Shinano. Could we use a million cores to solve an integer program? *Mathematical Methods of Operations Research*, 76(1):67–93, 2012.

[24] Ailsa H Land and Alison G Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[25] A. Langer, R. Venkataraman, U. Palekar, and L.V. Kale. Parallel branch-and-bound for two-stage stochastic integer optimization. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 266–275, Dec 2013.

[26] François V Louveaux and Rüdiger Schultz. Stochastic integer programming. *Handbooks in Operations Research and Management Science*, 10:213–266, 2003.

[27] Miles Lubin, Julian Hall, Cosmin G. Petra, and Mihai Anitescu. Parallel distributed-memory simplex for large-scale stochastic LP problems. *Computational Optimization and Applications*, 55(3):571–596, 2013.

[28] Miles Lubin, Kipp Martin, Cosmin G. Petra, and Burhaneddin Sandıkçı. On parallelizing dual decomposition in stochastic integer programming. *Operations Research Letters*, 41(3):252–258, 2013.

[29] Miles Lubin, Cosmin G. Petra, Mihai Anitescu, and Victor Zavala. Scalable stochastic optimization of complex energy systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–10. IEEE, 2011.

[30] Hugues Marchand, Alexander Martin, Robert Weismantel, and Laurence Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1–3):397 – 446, 2002.

[31] Lewis Ntaimo. Stochastic mixed-integer programming test problems. `http://ise.tamu.edu/people/faculty/ntaimo/personal_web/test_instances.htm`, 2015.

[32] Lewis Ntaimo and Suvrajeet Sen. The million-variable "march" for stochastic combinatorial optimization. *Journal of Global Optimization*, 32(3):385–400, 2005.

[33] Gurobi Optimization. Gurobi optimizer, 2015. `http://www.gurobi.com`.

[34] Cynthia A. Phillips, Jonathan Eckstein, and William Hart. *Massively Parallel Mixed-Integer Programming: Algorithms and Applications*, chapter 17, pages 323–340. SIAM Books, 2006.

[35] T. Ralphs and M. Galati. DIP – Decomposition for integer programming, 2009. `https://projects.coin-or.org/Dip`.

[36] T. K. Ralphs, L. Ládanyi, and M. J. Saltzman. A library hierarchy for implementing scalable parallel search algorithms. *J. Supercomput.*, 28(2):215–234, May 2004.

[37] Ted K Ralphs and Anahita Hassanzadeh. A generalization of benders' algorithm for two-stage stochastic optimization problems with mixed integer recourse. 2014.

[38] T.K. Ralphs, L. Ladányi, and M.J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming*, 98:253–280, 2003.

[39] Martin WP Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6(4):445–454, 1994.

[40] Suvrajeet Sen. Algorithms for stochastic mixed-integer programming models. *Handbooks in operations research and management science*, 12:515–558, 2005.

[41] Hanif D Sherali and Barbara MP Fraticelli. A modification of benders' decomposition algorithm for discrete subproblems: An approach for stochastic programs with integer recourse. *Journal of Global Optimization*, 22(1-4):319–342, 2002.

[42] Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. ParaSCIP: A parallel extension of SCIP. In *Competence in High Performance Computing 2010*, pages 135–148. Springer, 2012.

[43] Yuji Shinano and Tetsuya Fujie. ParaLEX: A parallel extension for the CPLEX mixed integer optimizer. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–106. Springer, 2007.

[44] Stochastic programming software and test sets. `http://stoprog.org/index.html?software.html`.

[45] F. Vanderbeck. BaPCod – A generic branch-and-price code, 2005. `https://wiki.bordeaux.inria.fr/realopt/pmwiki.php/Project/BaPCod`.

[46] Jean-Paul Watson, David L Woodruff, and William E Hart. PySP: Modeling and solving stochastic programs in Python. *Mathematical Programming Computation*, 4(2):109–149, 2012.

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

[48] Minjiao Zhang and Simge Küçükyavuz. Finitely convergent decomposition algorithms for two-stage stochastic pure integer programs. *SIAM Journal on Optimization*, 24(4):1933–1951, 2014.