Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

LLUÍS-MIQUEL MUNGUÍA[1], GEOFFREY OXBERRY[2], DEEPAK RAJAN[3],
YUJI SHINANO

Parallel PIPS-SBB: Multi-Level Parallelism For Stochastic Mixed-Integer Programs

[1] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA, lluis.munguia@gatech.edu
[2] Computational Engineering Division, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
[3] Computational Engineering Division, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA

Some of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. This work has also been supported by the Research Campus Modal *Mathematical Optimization and Data Analysis Laboratories* funded by the Federal Ministry of Education and Research (BMBF Grant 05M14ZAM), and partially supported by the BMWi project Realisierung von Beschleunigungsstrategien der anwendungsorientierten Mathematik und Informatik für optimierende Energiesystemmodelle - BEAM-ME (fund number 03ET4023DE). All responsibility for the content of this publication is assumed by the authors.

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Parallel PIPS-SBB: Multi-Level Parallelism For Stochastic Mixed-Integer Programs

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

[1]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA, lluis.munguia@gatech.edu
[2]Computational Engineering Division, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
[3]Department of Mathematical Optimization, Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany

November 7, 2017

Abstract

PIPS-SBB is a distributed-memory parallel solver with a scalable data distribution paradigm. It is designed to solve MIPs with a dual-block angular structure, which is characteristic of deterministic-equivalent Stochastic Mixed-Integer Programs (SMIPs). In this paper, we present two different parallelizations of Branch & Bound (B&B), implementing both as extensions of PIPS-SBB, thus adding an additional layer of parallelism. In the first of the proposed frameworks, PIPS-PSBB, the coordination and load-balancing of the different optimization workers is done in a decentralized fashion. This new framework is designed to ensure all available cores are processing the most promising parts of the B&B tree. The second, ug[PIPS-SBB,MPI], is a parallel implementation using the Ubiquity Generator (UG), a universal framework for parallelizing B&B tree search that has been successfully applied to other MIP solvers. We show the effects of leveraging multiple levels of parallelism in potentially improving scaling performance beyond thousands of cores.

1 Introduction

The breakdown of Dennard scaling in CPUs has forced manufacturers to design for increasing parallelism to improve theoretical peak CPU performance, as processor clock speeds cannot be increased using current chip manufacturing processes. Realizing a nontrivial fraction of these theoretical performance improvements requires algorithms that leverage parallelism effectively. For high-performance computing (HPC) systems used in large-scale scientific and engineering applications, this trend has been exacerbated, with recent HPC systems such as Argonne National Laboratory's Mira enabling million-process parallelism, and future systems enabling even larger process counts. Using such massive-scale parallelism – four to five orders of magnitude greater than current workstations – to its full potential in large-scale Mixed Integer Programming (MIP) applications demands MIP algorithms that scale efficiently (e.g., in a weak or strong sense) as process counts increase [10].

Current parallel algorithms for MIPs do not meet this criterion. The scalability of existing approaches, discussed later in this introduction, can vary dramatically depending on the instance

being solved, exposing a significant gap between current HPC hardware and MIP algorithmic capabilities. To begin to address this algorithmic capability gap, this work investigates the parallel scaling benefits of exploiting the problem structure found in dual block-angular MIPs and exposing multiple levels of parallelism. Leveraging the parallelism in each level yields multiplicative effects on speedup: if each of two levels scales efficiently to hundreds of cores, the overall algorithm has the potential to scale efficiently to tens of thousands of cores. Our intention is to exploit this principle for dual block-angular MIPs using PIPS-SBB [16].

In the rest of this introduction, we first provide some background in Sections 1.1-1.3, describing: Stochastic MIPs, Branch & Bound (the primary MIP algorithm), and PIPS-SBB (a parallel B&B framework for solving SMIPs). Subsequently, in Section 1.4, we summarize the main contributions of this work.

1.1 Stochastic Mixed Integer Programs

Deterministic-equivalent SMIPs exhibit dual block-angular structure of the form:

$$\min_{x\in\mathbb{R}^{n_1}, y\in\mathbb{R}^{n_2\times s}} c^{\mathrm{T}}x + \sum_{i=1}^{s} q_i^{\mathrm{T}} y_i,$$

subject to:

$$\begin{array}{llllll}
Ax & & & & & = b_0, \\
T_1 x & +W_1 y_1 & & & & = b_1, \\
T_2 x & & +W_2 y_2 & & & = b_2, \\
\vdots & & & \ddots & & \vdots \\
T_s x & & & & +W_s y_s & = b_s, \\
& & & & l \le x \le u, & \\
& & & & l_i \le y_i \le u_i, \quad \forall i \in [s], & \\
& & & & x_j \in \mathbb{Z}, \quad \forall j \in I_1, & \\
& & & & y_{i,j} \in \mathbb{Z}, \quad \forall i \in [s], \forall j \in I_2. &
\end{array} \tag{1}$$

In (1), $[s]$ is the set of natural numbers from 1 to s inclusive, and $\overline{\mathbb{R}}$ is the extended set of real numbers $\mathbb{R} \cup \{-\infty, \infty\}$. Members of $[s]$ label *scenarios* – instances of the second-stage MIP for given realizations of the random variables on which it depends. *First-stage decision variables* x are bounded from above and below by $l \in \overline{\mathbb{R}}^n$ and $u \in \overline{\mathbb{R}}^n$, respectively, where n is the number of first-stage decision variables. *Second-stage decision variables* y_i for scenario i are bounded from above and below by $l_i \in \overline{\mathbb{R}}^{n_2}$ and $u_i \in \overline{\mathbb{R}}^{n_2}$ respectively, where n_2 is the number of second-stage decision variables in each scenario. These sets of decision variables are constrained by a set of linear constraints, modeled with different independent submatrices. The sets I_1 and I_2 are sets of indices of first-stage and second-stage variables, respectively, that correspond to integer-valued decision variables; for simplicity, we assume that if second-stage decision variable $y_{i,j}$ is integer in scenario i, then j is in I_2, and $y_{k,j}$ is integer for each scenario k from 1 to s inclusive. This formulation is called *block-angular* because it can be permuted into a "block half-arrow" structure, where the A and T_i matrices make up the left half of the arrow head, while the A and W_i matrices form the stem. This formulation is also called *dual block-angular* because columns of this block-angular constraint matrix link primal variables, while the corresponding second-stage dual variables remain uncoupled across scenarios.

SMIPs arise in many important applications, most notably stochastic unit commitment [31]. For the remainder of this work, we will discuss algorithms for dual block-angular MIPs using the

terminology of SMIPs for convenience, due to the prevalence of SMIPs and the well-developed terminology for this problem class. However, dual block-angular MIPs arise in other problem classes besides SMIPs. We emphasize that the algorithms described herein could be applied to any dual block-angular MIP, and the explanations in this work map easily to this case by drawing an analogy to SMIPs.

1.2 Parallel Branch & Bound

The B&B algorithm [11] is the most commonly used method for solving MIPs to optimality. It is a search algorithm that systematically partitions a problem into smaller subproblems and searches the solution space using a dynamically-generated tree data structure called the *B&B tree*. In LP-relaxation-based B&B, the quality and promise of subproblems is evaluated by solving an LP relaxation formed by relaxing all integrality constraints. The optimization concludes once all subproblems are fathomed or solved to optimality. Furthermore, since the B&B algorithm computes upper and lower bounds on the optimal objective function value, it can also be used to provide a guarantee on the quality of the best feasible solution found, if terminated early.

Algorithm 1 Branch and Bound

$UB = \infty$
$LB = -\infty$
priority queue $Q = \emptyset$
Add root subproblem r of original MIP problem to Q
while $Q \neq \emptyset$ and $LB < UB$ **do**
    remove subproblem p from top of Q
    Process p
    UB_p = best solution found within p
    LB_p = lower bound of p, ∞ if infeasible, $-\infty$ if unbounded
    $UB = \min\{UB, UB_p\}$
    if $LB_p < UB$ **then**
        Partition p into a set $\{s_0, \ldots, s_k\}$ of subproblems, each with a lower bound defined to be LB_p
        Add $\{s_0, \cdots, s_k\}$ to Q
    else
        continue ▷ If $LB_p \geq UB$, problem p is discarded (fathomed by bound dominance)
    end if
    LB = minimum lower bound among all open subproblems in Q
end while
if $LB < UB$ **then**
    return infeasible
else
    return optimal solution
end if

Despite being an exact algorithm for solving MIPs, naïve B&B as described in Algorithm 1 is computationally infeasible due to the exponential growth of the search space. State-of-the-art MIP solvers enhance this B&B scheme with many additional methods that prune the search space in order to make the scheme practical. Developing each of these techniques is nontrivial, given that they usually involve solving additional NP-hard problems. Primal heuristics [5, 8] are algorithms, usually with a guarantee of success, that focus on finding high-quality feasible solutions to improve the upper bound. Pre-processing [19, 2, 9] and cutting planes [15] are techniques focused on strengthening the LP relaxation, thus reducing the search space. Other important components include the branching rules (how to partition the problem) [3], and how the priority queue is sorted [12]. The search strategy can have a big impact on how fast high-quality solutions are found and determine greatly the size of the tree required to solve a problem to optimality.

Multiple parallel B&B tree search algorithms have been devised in order to speed up the optimization algorithm, though most of them differ significantly on how they might go about it. For a thorough taxonomy of most B&B parallelizations, we urge the reader to refer to [18]. In theory, parallel B&B tree search is not difficult because the processing of subproblems is independent. However, There are many challenges associated with designing a parallel B&B algorithm, such as maintaining a healthy work balance among all the processors and avoiding significant communication and synchronization costs. In other words, distributed-memory parallelization is not straightforward in practice, in contrast to the relatively straightforward shared-memory parallel algorithms present in most state-of-the-art MIP solvers. In this paper, we classify parallel B&B algorithms based on the smallest transferrable unit of work between the various cores of the parallel search. On one hand, solvers using *fine-grained* parallelizations use a single MIP tree node as the basic unit of work. In contrast, each solver in *coarse-grained* parallelizations focuses on solving an entire MIP subtree at a time. Generally speaking, coarse-grained parallelizations require less communication. On the other hand, it may be challenging for such implementations to supply useful work to all processors.

To keep this discussion self-contained, we mention three notable parallel B&B tree search frameworks developed recently: PEBBL [6, 7], CHiPPS [25, 26, 27, 28, 29, 30], and UG [20, 21, 22, 1].

PEBBL parallelizes B&B at a fine-grained, node level within an MPI-based, shared-nothing parallel programming model using a decentralized hub-worker approach. In this approach, each hub process owns a collection of worker processes that solve subproblems (nodes). Workers and hubs communicate problem metadata to coordinate subproblem transfers between worker processes. "Load factors" are used to estimate the work required to process the B&B subtrees rooted at the subproblems owned by workers in each hub. "Rendezvous" load balancing is then used to decide which hubs donate or receive subproblems to or from other hubs. A hub may also transfer subproblems between worker processes it owns through dynamic load-balancing algorithms specific to PEBBL.

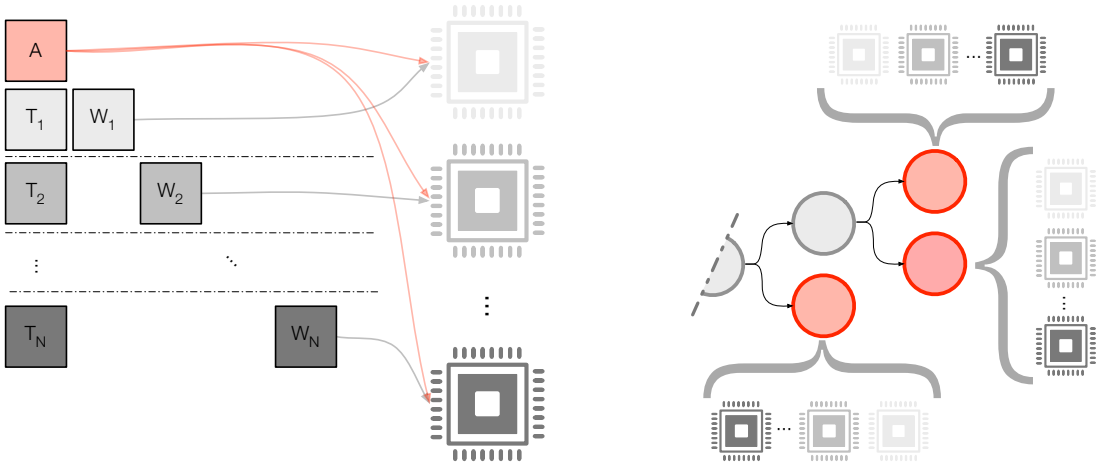
CHiPPS parallelizes B&B using a master-hub-worker approach built upon the ALPS [26] parallel tree search framework. This approach uses coarse-grained, subtree-level parallelism in which workers explore B&B subtrees, and these subtrees are transferred among workers during load-balancing. As with PEBBL, CHiPPS uses hub processes to coordinate worker processes, but instead of employing peer-to-peer coordination among hubs, CHiPPS load-balances hierarchically, using a master process to coordinate and balance work among hub processes, each of which themselves coordinates and balances work among the worker processes it owns.

The Ubiquity Generator (UG) framework [1] was designed to parallelize existing state-of-the-art sequential B&B algorithms for MIPs, referred to as *base solvers*, using a supervisor-worker approach. In this approach, UG wraps around both the API of the base solver (e.g., CPLEX) and the parallel programming model (e.g., MPI, POSIX threads) to enable passing B&B subtrees among worker processes, each of which processes a B&B subtree using a base solver instance. A supervisor process called a "LoadCoordinator" is used to serialize B&B subtrees (e.g., using variable bound changes in the root subproblem of a subtree) and to redistribute them among worker processes during load balancing.

1.3 PIPS-SBB, the base solver

PIPS-SBB [16] is a parallel B&B framework for MIPs that have dual block-angular structure. PIPS-SBB is designed to operate in a parallel distributed-memory environment (i.e., an MPI-based, shared-nothing parallel programming model) and exploits problem structure to offer two unique advantages. First, PIPS-SBB distributes data representing each scenario to different

processes within its MPI communicator, while first-stage information is replicated across all processes. This data distribution enables allocating to a communicator up to as many processes as scenarios specified in the input problem, so PIPS-SBB can solve some large SMIPs that would not otherwise fit in memory. Second, PIPS-SBB uses PIPS-S [13] to solve LP relaxations of subproblems in parallel; PIPS-S also exploits dual block-angular structure exhibited by stochastic LPs in extensive form. Every component of the B&B framework 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.



(a) Data distribution policy of PIPS-SBB and PIPS-S: Data for each scenario is sent to different processor.

(b) Distributed-memory B&B tree search with PIPS-SBB: Each B&B node solved in parallel.

Figure 1: Two of the most prominent features of PIPS-SBB are (a) the ability to distribute problem data across distributed-memory and (b) the capability of processing each node of B&B tree (LP relaxations, cuts, heuristics, etc.) in parallel.

PIPS-SBB aims to achieve speed-up by using PIPS-S as its LP solver to exploit data parallelism when processing each B&B node. This architectural decision avoids scaling bottlenecks associated with parallel node exploration. However, the scalability of the base solver PIPS-S for a given SMIP is limited because it depends on the relative sizes of its first- and second-stage coefficient matrices and the number of scenarios.

The reader is referred to [16] for details about the various features implemented in the initial version of PIPS-SBB. More recently, other features have been added to PIPS-SBB, including newer primal heuristics and best-estimate based node selection strategies. While these features have improved the performance of our base solver, the main focus of this paper is the extension of PIPS-SBB for parallel B&B tree search.

1.4 Contributions and Outline

The premise of this paper is to expose multiple nested levels of parallelism in MIP algorithms. We present two new frameworks for parallelizing the B&B tree, and demonstrate them on SMIPs using PIPS-SBB [16], a distributed-memory parallel Branch and Bound (B&B) solver for SMIPs. In these frameworks, the LP relaxations of SMIPs are solved using PIPS-S [13], a distributed-memory parallel simplex solver. By parallelizing the B&B tree search in PIPS-SBB, we incorpo-

rate two levels of parallelism: parallelism (1) in the LP solver, and (2) in the B&B tree search. This additional level of parallelism to PIPS-SBB with parallel node exploration enables us to increase its scalability.

We briefly describe both frameworks, which differ in the approach used in the management and distribution of work among the parallel workers. The first framework, called PIPS-PSBB, is a new, fine-grained framework that attempts to search aggressively the promising parts of the B&B tree, with the intention of decreasing the amount of *redundant work* performed by the solver. For minimization problems, a *redundant node* can be characterized as a subproblem with an LP relaxation value greater than the optimal value. To limit communication overhead, this framework transfers node metadata (e.g., upper/lower bounds, tree sizes) asynchronously to hide latency until a load imbalance is detected, at which point MIP subproblems are redistributed among processes using synchronous communication. MPI collective operations are used instead of point-to-point operations to further limit communications overhead. The proposed architecture is also decentralized: node exchanges do not pass through a single coordinating process, which avoids a potential communication bottleneck. The second framework is based on UG. We compare the strengths and weaknesses of both approaches, and show that when implemented as extensions of PIPS-SBB, both approaches leverage multi-level parallelism to scale efficiently beyond the natural limitations of each framework in isolation.

The main contributions of this paper are therefore:

- A novel framework for fine-grained parallel B&B tree search for solving mixed-integer programs.
- Two new multi-level distributed-memory parallelisms for solving SMIPs, implemented as extensions of the distributed-memory SMIP solver PIPS-SBB.
  - PIPS-PSBB: B&B parallelism implemented using the new fine-grained parallelism presented in this paper.
  - B&B parallelism implemented using UG, the coarse-grained parallel B&B framework available as part of the SCIP optimization suite [14].
- Detailed computational experiments illustrating the scaling performance of both parallel B&B frameworks.

In general, leveraging the parallelism in each level simultaneously yields multiplicative effects on speedup and parallel scaling efficiency (in a weak or strong sense). In the specific case of B&B parallelizations for practical MIP instances, it is non-trivial to achieve even 20% strong scaling efficiency beyond 100-1000 processors (modest by HPC standards). By providing an extra level of parallelism, available processing power can be partitioned across multiple levels to improve overall efficiency and speedups over allocating the same number of processes to either single level.

The paper is organized as follows. In Section 2, we present PIPS-PSBB: our decentralized, lightweight parallel framework for fine-grained B&B tree search. Additionally, we delve into the details of our implementation, discussing some of the challenges, and our approach to minimizing the communication overhead associated with the distributed-memory algorithm. In Section 3, we discuss the same topics for ug[PIPS-SBB,MPI]. In Section 4, we begin by first comparing the performance of both of our implementations in detail on a few select instances, and then present performance results (comparing with distributed-memory CPLEX) over the complete test set. Finally, we conclude in Section 5 with some directions for future research.

2 PIPS-PSBB: A decentralized lightweight parallel framework for B&B

In this section, we present PIPS-PSBB, a decentralized fine-grained, yet lightweight parallel framework built as an extension to PIPS-SBB. Our approach and ideas are general, and can be applied to any base MIP solver. We first present the main ideas and the implementation details, which are independent of PIPS-SBB. Then, in Section 2.3, we address the challenges specific to extending and adhering to the design principles of PIPS-SBB.

A primary aspect to consider in the design of parallel B&B algorithms is the policy used in the distribution of work, and the degree and frequency of communication used to coordinate optimization workers. On one hand, coarse-grained parallelizations typically require less communication, and allow each worker to focus on processing an entire subtree at a time. However, less frequent coordination between workers may have a detrimental effect on parallel efficiency. Workers may be likely to solve subtrees that would be otherwise fathomed in a sequential execution, especially in the context of large-scale computations with thousands of available solvers. Performing redundant work may seem unavoidable because the information required for fathoming the nodes is only discovered during the optimization and is therefore impossible to know beforehand. Figure 2 highlights this issue. The discovery of a feasible solution (black-colored node) by processor 2 could have resulted in the fathoming of many nodes belonging to other workers, if only this information was known beforehand. In the context of a coarse-grained distribution of work, nodes from the same part of the B&B tree typically belong to the same processor, unlike in fine-grained distributions.

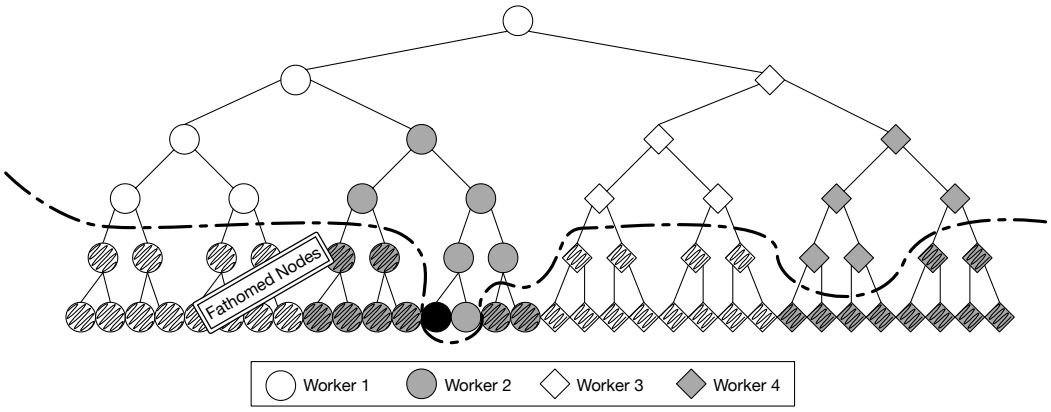


Figure 2: Example depicting the coarse-grained distribution of subtrees among the parallel processors at a given point in the parallel exploration. Knowledge of feasible solution at black-colored node could have fathomed many other nodes on many processors.

When point-to-point communications are used, a coarse-grained distribution of work may seem the only viable option for avoiding large communication overheads without resorting to complex communication schemes, as in PEBBL. Nevertheless, it is possible to develop a practical, effective, and scalable coarse-grained parallelization framework; as has been demonstrated by UG. Some strategies for reducing overheads include restricting the number of workers simultaneously engaging in communication. We explore the integration of PIPS-SBB with UG in Section 3.

On the other side of the spectrum, fine-grained parallelizations have a MIP tree node as their smallest unit of work, which can be transferred among workers. This allows the most promising

parts of the tree to be partitioned and processed in parallel, thus ensuring an effective use of the computing resources. Fine-grained control of the work performed is an adaptive approach where the amount of redundant work can be minimized effectively. At the same time, a large degree of communication is also required, which may cause an excessive overhead. By establishing complex protocols so that communication between processors is structured hierarchically, it is possible to develop scalable fine-grained parallelizations, as demonstrated by PEBBL. In PIPS-PSBB, we take a slightly different approach, overcoming many of these communication challenges using a simpler, light-weight framework using MPI collectives. We note that MPI collectives can also be implemented efficienclty using hierarchical tree-based algorithms [23], enabling us to reap some of the benefits of these schemes with less implementation complexity than PEBBL. As in PEBBL, PIPS-PSBB is a decentralized framework based on asynchronous MPI communications.

2.1 The philosophy behind fine-grained node rebalancing

B&B can be regarded as a graph algorithm, the objective of which is to traverse the different subproblems until the optimal solution is found and proven. Borrowing from graph algorithms, we define the frontier as the collection of open subproblems at a given time. We also define the active nodes as the set of subproblems currently being explored in parallel. To be able to measure the effectiveness of any parallel B&B framework, we use the notion of redundant work (as described in [10, 18]) to refer to the collection of subproblems that would be fathomed if the optimal solution was known.

One way to reduce the amount of redundant work is to have extremely powerful heuristics. By finding good feasible solutions very early in the search, B&B fathoms as many nodes as possible, thus increasing parallel efficiency. In addition, one can try to ensure that all optimization workers focus on the most promising nodes. In the context of parallel B&B tree search, load balancing is done to ensure that all processors have access to a fraction of the *most promising* nodes, rather than an *equal number* of nodes. The quality of a node is determined by the criterion used to order it within the priority work queue. Typically the lower bound of its parent is used, or some form of node estimation as the ones described in [24, 12].

With the objective of minimizing load imbalance in PIPS-PSBB, we define the smallest transferrable unit of work to be one B&B node. As the optimization progresses and the work load is continually rebalanced, every processor gradually collects a set of nodes from different subtrees in its work queue. This feature enables great flexibility in the parallel search strategy. An example depicting the fine-grained distribution of work is shown in Figure 3. For instance, nodes 8-13 belong to the same subree, but are distributed among all the available optimization workers.

The objective of our new parallel B&B implementation is to increase parallel efficiency by minimizing the number of redundant nodes explored. We actively target this goal by ensuring the set of active nodes being explored are always the most promising ones. We present an overview of our fine-grained implementation in Algorithm 2. One of its key components is the work redistribution mechanism, described between lines 9 to 15. The approach is described in greater detail in Section 2.2.

PIPS-PSBB uses a lightweight mechanism for redistributing the most promising nodes among all the optimization workers without the need for a centralized load coordinator. Rather, this alternative scheme seeks to reduce the communication bottlenecks that would be caused by the existence of one. Instead of point-to-point communications, parallel processors exchange subproblems via all-to-all collective MPI asynchronous communications, enabling the framework to rebalance the computational load using a single communication step. Parallel processes proceed to solve subproblems until the problem has been solved to optimality.

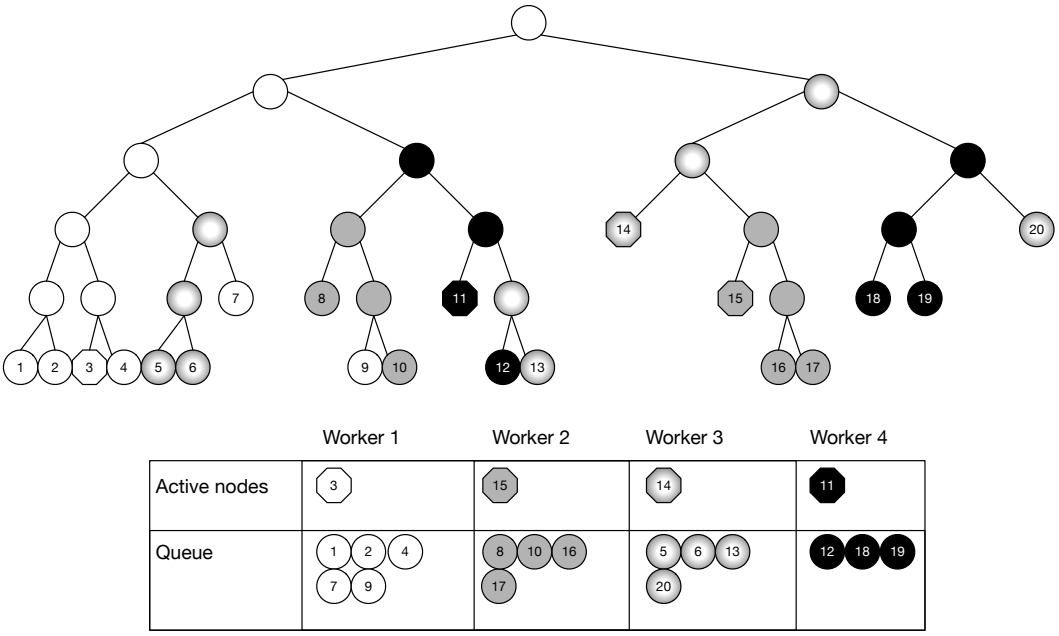


Figure 3: PIPS-PSBB: Example depicting the fine-grained distribution of subtrees among the parallel processes at a given point in the parallel exploration. The frontier and the active nodes in the frontier are also depicted.

2.2 Decentralized node exchange

Load rebalancing among all workers is maintained via a sequence of MPI collective communications. It enables workers to rank the most promising subproblems and to redistribute them in a round robin fashion to improve load balance. A flow chart of the rebalancing process is provided in Figure 4.

The first step in the process consists of identifying the location of the most promising nodes. Assuming the objective is to redistribute K of the most promising nodes for every one of the N available optimization workers, the total nodes to exchange will be $K \cdot N$. In the example presented in Figure 4, we have $K = N = 3$, yielding 9 nodes to exchange. To achieve this exchange, every worker first collects the lower bounds and estimates of its best $K \cdot N$ ($= 9$ in this example) open subproblems, since it is possible that a single worker owns all the nodes to exchange. It is also possible that some worker does not have $K \cdot N$ nodes (as is the case for orkers 1 and 2), the algorithm proceeds with the maximum available. The node bounds/estimates are exchanged through a decentralized all-to-all MPI Allgather communication, collecting a total of $K \cdot N \cdot N$ lower bounds and estimates. Once the $K \cdot N$ most promising nodes have been identified in the next step by each worker (by sorting), and their origin has been determined, the actual node information is prepared to be redistributed in round-robin fashion. In this example, worker 1 is identified to receive nodes 1,4,7. Observe that it may already own some of these nodes, and therefore these nodes do not need to be exchanged. Once it is determined which nodes need to be redistributed, every worker proceeds to serialize the actual node information prior to their exchange. In this example, none of the nodes from worker 2 were deemed promising nodes, but it received three new promising subproblems.

Communication must be used strategically in order to avoid overheads. While node transfers

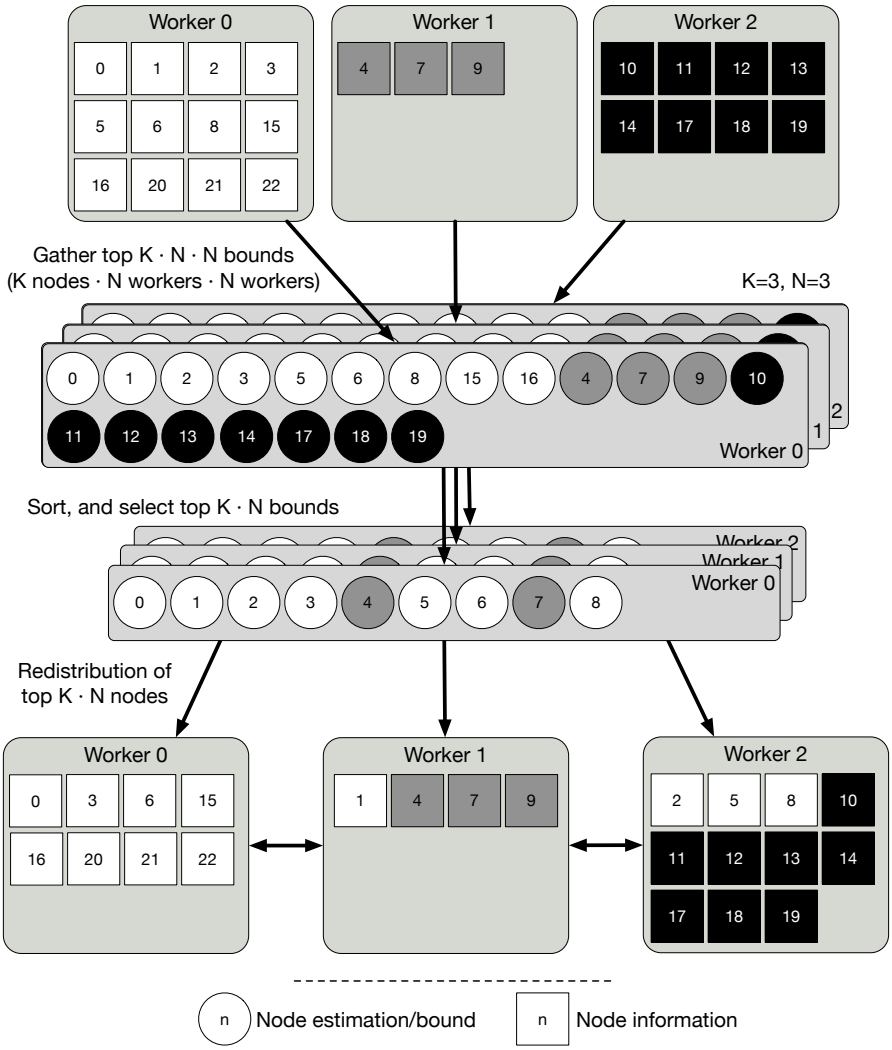


Figure 4: Steps of the node exchange: Node estimates (in circles) are gathered using all-to-all communication and sorted at each worker. When each worker has determined (in parallel) which nodes need to be exchanged, actual node information (in squares) is redistributed using point-to-point communication.

Algorithm 2 Fine-grained parallel Branch & Bound algorithm of PIPS-PSBB

for All processors $t \in [N]$ in Parallel **do**
  $UB^t = \infty$
  $LB^t = -\infty$
  priority queue $Q^t = \emptyset$
  if $t = 1$ **then**
    Add root r of original MIP problem to Q^1
  end if
  while termination conditions are not met **do**
    mustCommunicate = TestConditionsForCommunication(t, Q^t, UB^t)
    if mustCommunicate **then**
      Determine the top $K \cdot N$ candidate subproblems from $Q^t, t \in [N]$ and redistribute them among all processors in a round robin fashion.
      if termination conditions are met **then**
        return
      end if
    end if
    remove subproblem p from top of Q^t
    Process p
    UB_p = best solution found within p
    LB_p = lower bound of p, ∞ if infeasible, $-\infty$ if unbounded
    $UB^t = \min\{UB^t, UB_p\}$
    if $LB_p < UB^t$ **then**
      Partition p into a set $\{s_0, \ldots, s_k\}$ of subproblems, each with a lower bound defined to be LB_p
      Add $\{s_0, \cdots, s_k\}$ to Q^t
    end if
    ▷ If $LB_p > UB^t$, fathom problem p by bound dominance
    LB^t = minimum lower bound among all open subproblems in Q^t
  end while
end for
if $LB < UB$ **then**
  return infeasible
else
  return optimal solution
end if

are carried out synchronously, exchanges of worker statuses such as upper/lower bounds, tree sizes, times, and solutions are performed asynchronously. Nodes are transferred synchronously only after all workers signal that communication is needed. When communicating over a large number of processes, MPI collective communication primitives have been shown to significantly outperform their point-to-point equivalents[23], provided a tuned MPI implementation is used.

We provide details regarding the asynchronous detection of load imbalance in Algorithm 3. The algorithm establishes a threshold λ that determines the number of B&B iterations between asynchronous communication calls. This parameter is adjusted throughout the parallel B&B tree search to adapt load rebalancing as needed. The parameter is modified based on the difference between the minimum and maximum optimality gap (gap between lower and upper bounds) among the workers. A difference in gap greater than a provided threshold δ indicates load imbalance, and therefore the parameter λ is decreased in order to rebalance more aggressively in the future. During ramp-up and ramp-down, frequent rebalancing is expected. Work queues are also rebalanced whenever any worker has no active nodes in its priority queue. Otherwise, if the number of iterations since the last communication is greater than λ, asynchronous all-to-all communication is performed.

2.3 Processor distribution

Every software component and algorithm in PIPS-PSBB is designed to comply with the distributed data representation imposed by PIPS-SBB. Thus, every MPI processor is responsible for performing all operations on the data it owns. With the addition of B&B parallelism to the framework, processes are arranged in a matrix of MPI communicators, as depicted in Figure 5.

Algorithm 3 Asynchronous communication mechanism in PIPS-PSBB

```
1: procedure TestConditionsForCommunication(t, Q^t, UB^t)
2:     if asynchronous communication pending then
3:         Test communication flags
4:         if communication complete then
5:             Update best solutions, UB^t = min_{1≤i≤N}{UB^i}
6:             Update global lower bound, and check termination conditions
7:             Gap_min = smallest optimality gap among workers
8:             Gap_max = largest optimality gap among workers
9:             if |Gap_max − Gap_min| ≥ δ then
10:                Decrease number of iterations threshold λ before next communication
11:                Return true
12:            else
13:                increase number of iterations threshold λ before next communication
14:            end if
15:        end if
16:        Return false
17:    end if
18:    if number of iterations since last communication is greater than λ or Q^t = ∅ then
19:        initiate asynchronous exchange of current best lower bound, upper bound, tree size, solution time, and best
   solution
20:    end if
21:    Return false
22: end procedure
```

MPI processes belonging to a given PIPS-S solver communicator operate as a single worker, conform to the data distribution, and exchange information in order to be able to solve LP relaxations in parallel. In turn, all MPI processes belonging to a B&B communicator own the same data of the problem, and communicate to exchange nodes during the rebalancing process. The process/communicator arrangement is such that with a total of $M \cdot N$ processes, each collective communication only involves either M or N processes.

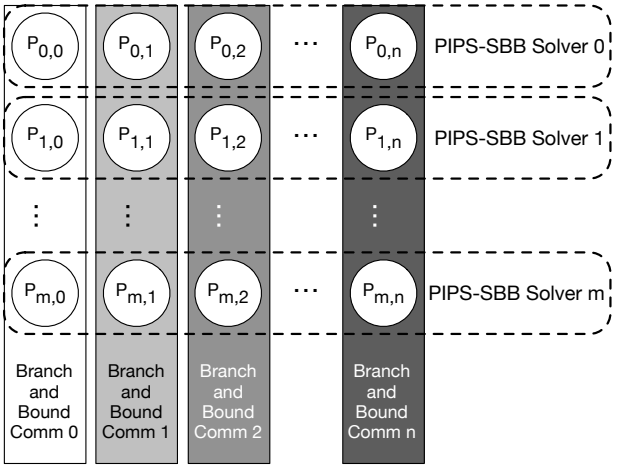


Figure 5: Processor distribution used in PIPS-PSBB: Each row corresponds to the communicator of a single PIPS-SBB solver. Each column corresponds to the communicator used by processes from different PIPS-SBB solvers that own the same data.

3 Parallelizing PIPS-SBB with UG

The main concept of UG is to exploit the performance of a powerful state-of-the-art "*base solver*" by coordinating multiple instances in parallel. The UG framework is depicted in Figure 6. Using the established notation, ug[PIPS-SBB, MPI] is the product of parallelizing the base solver PIPS-SBB using MPI under the framework of UG. UG carefully abstracts all functions related to managing parallel B&B search from the actual processing of the B&B tree itself. It supports many common features needed in parallel B&B, such as multiple communication protocols, ramp-up, dynamic load balancing, check-pointing, and restarting mechanisms.

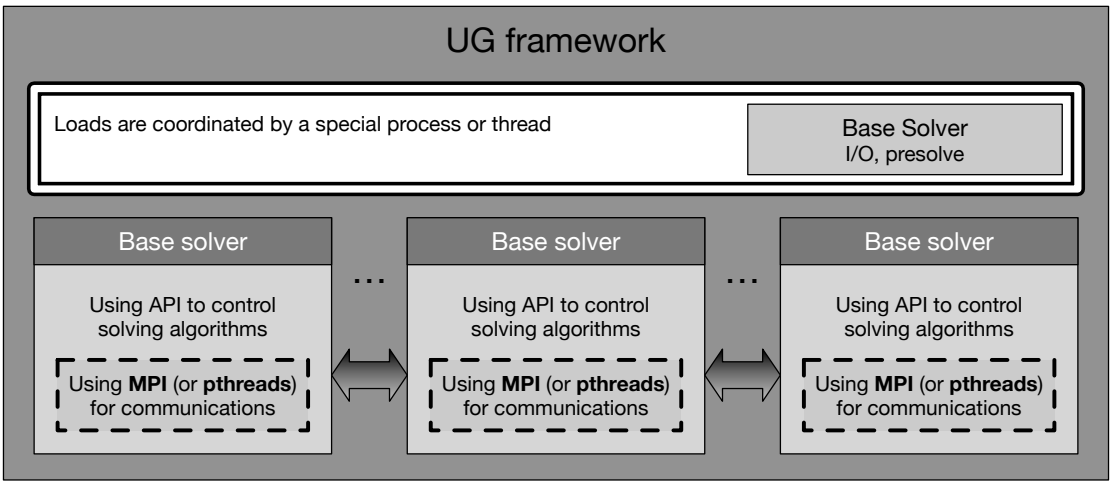


Figure 6: Design of UG and ug[PIPS-SBB, MPI]

This framework has been instrumental in solving to optimality several open instances of the MIPLIB2003 and MIPLIB2010 libraries [21]. Its coordination paradigm has also been used in the distributed-memory implementation of the CPLEX solver. The design of UG is such that its integration with PIPS-SBB is straightforward, only requiring the implementation of the interface between both software components.

3.1 Processor distribution within UG

The ug[PIPS-SBB,MPI] design requires an additional MPI processor to act as coordinator and some additional design compromises. The supervisor-worker paradigm of UG designates a single processor within each PIPS-S communicator to exchange solver information and subproblems. Therefore, only a single communicator is needed in order to enable parallelism at the branch-and-bound level, as depicted in Figure 7. The main downside to this approach is the need to break the data distribution policy imposed by PIPS-SBB when communication takes place. This design limitation limits the scalability of the parallel implementation to large data sizes.

3.2 Supervised workload coordination mechanism

UG follows a supervisor-worker paradigm, in which a supervisor, the LoadCoordinator, monitors and coordinates the workload among multiple optimization workers. The LoadCoordinator coordinates the workload and does not store data associated with the search tree. Load balancing is accomplished mainly by toggling the collection mode flag in the UG solvers. Turning on

Algorithm 4 UG LoadCoordinator (UG workers 1 to N with the PIPS-SBB are spawned)

`collectMode` $\leftarrow$ `False`
$x^* \leftarrow$ NULL
$I \leftarrow N \setminus \{1\}$
$A \leftarrow \{1\}$
$Q \leftarrow \emptyset$
$R \leftarrow \{(1, 0)\}$ ▷ Subproblems currently being processed, 0 is the index of the root problem
Send the root problem to UG workers 1 (see Algorithm 5)
while $Q \neq \emptyset$ and $R \neq \emptyset$ **do**
    (i, tag) $\leftarrow$ Wait for message ▷ Returns UG workers identifier and message tag
    if `tag` = `solutionFound` **then**
        Receive solution $\hat{x}$ from UG worker i
        if x^* = `NULL` **or** $c^\top \hat{x} < c^\top x^*$ **then**
            $x^* \leftarrow \hat{x}$
        end if
    else if `tag` = `subproblem` **then**
        Receive a subproblem indexed by k from UG worker i
        $Q \leftarrow Q \cup \{k\}$
    else if `tag` = `terminated` **then**
        $R \leftarrow R \setminus \{(i, j)\}$ ▷ j is the index of the terminated subproblem
        $A \leftarrow A \setminus \{i\}$, $I \leftarrow I \cup \{i\}$
    else if `tag` = `status` **then**
        if `collectMode` = `True` **then**
            if there are enough heavy subproblems in Q **then**
                ▷ *heavy subproblem* is a subproblem which is expected to generate a large subtree
                Send message with `tag` = `stopCollecting` to UG workers in collecting mode. (see Algorithm 5)
                `collectMode` $\leftarrow$ `False`
            end if
        else
                ▷ `collectMode` = `False`
            if there are not enough heavy subproblems in Q **then**
                Select UG workers which have heavy subproblems
                Send message with `tag` = `startCollecting` to the selected UG workers (see Algorithm 5)
                `collectMode` $\leftarrow$ `True`
            end if
        end if
    end if
    while $I \neq \emptyset$ **do**
        $i \in I$, $I \leftarrow I \setminus \{i\}$, $A \leftarrow A \cup \{i\}$
        subproblem $j \in Q$, $Q \leftarrow Q \setminus \{j\}$, $R \leftarrow R \cup \{(i, j)\}$
        Send subproblem j and x^* to UG worker i (see Algorithm 5)
    end while
end while
$\forall i \in S$: Send message with `tag` = `termination` to UG worker i (see Algorithm 5)
Output x^*

Algorithm 5 UG worker i with the PIPS-SBB ($i = 1, \ldots, N$)

`collectMode` $\leftarrow$ `False`
`terminate` $\leftarrow$ `False`
while `terminate` = `False` **do**
    (i, tag) $\leftarrow$ Wait for message from LoadCoordinator (from Algorithm 4) ▷ Returns LoadCoodinator identifier 0 and message tag
    if `tag` = `subproblem` **then**
        Receive subproblem and solution from LoadCoordinator (from Algorithm 4)
        Solve the subproblem using PIPS-SBB, periodically communicating with LoadCoordinator (from Algorithm 4) as follows
            - Send message with tag `solutionFound` any time a new solution is discovered.
            - Periodically send message with tag `status` to report current lower bound for this subproblem.
            - When messages with tag `startCollecting` or `stopCollecting` are received, toggle `collectMode`.
            - When `collectMode = True`,
                periodically send message with tag `subproblem` containing best candidate subproblem.
        Send a message with `tag` = `terminated`
    else if `tag` = `termination` **then**
        `terminate` $\leftarrow$ `True`
    end if
end while

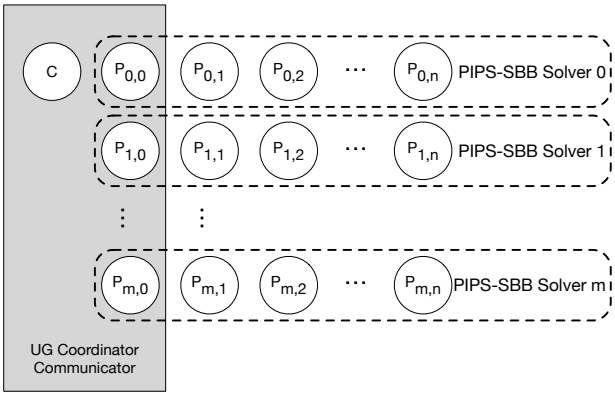


Figure 7: Processor distribution used in ug[PIPS-SBB,MPI]

collecting mode sends additional "high quality" subproblems to other UG solvers via the LoadCoordinator. Algorithms 4 and 5 show a simplified coordination mechanism used in UG. The LoadCoordinator can tune the frequency of the status updates produced by the optimization workers to avoid communication bottlenecks in large-scale parallel optimizations. In addition, the number of workers simultaneously participating in the collection mode can be restricted and selected dynamically.

Naturally, there are tradeoffs among the frequency of communication, the number of UG workers participating in collection mode, and the degree to which the parallel search order replicates the sequential one. As the number of processes increases, these tradeoffs must be navigated carefully.

An exchanged subproblem that contains additional bound changes of variables and warmstart information different from the original problem must be solved from scratch on the receiver side. This is a price to pay for externalization. However, there is a potential benefit from this practice, as better performance can be obtained by applying additional presolving, cutting planes and heuristics in the subproblem. In the case of ug[PIPS-SBB,MPI], the impact of externalization is minimal because the information exchange level is the same as the one produced in PIPS-PSBB.

4 Experimental Results

We test the performance of our parallel implementations on the Stochastic Server Location Problem (SSLP) [17] instances from the SIPLIB library [4]. The SSLP instances model server location problems with pure binary variables in the first-stage and mixed-binary variables in the second-stage. The problems are encoded as $sslp\_m\_n\_s$, 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 `cab` Cluster at Lawrence Livermore National Laboratory, which consists of 1,296 computing nodes. Each computing node features two Intel Xeon E5-2670 8-core processors and 32 GB of RAM. In all experiments, bindings of MPI processes were configured to prevent over-subscription. We evaluate the performance of our frameworks from the perspective of parallel scaling, as well as the overall performance when compared against CPLEX 12.6.2 on a distributed-memory parallel environment.

We evaluate the scaling performance of our methods using four metrics: time to optimality, tree size, communication overhead, and node inefficency. We define communication overhead as the fraction of the total time spent by a process performing communication, being idle due

to parallel synchronization, or waiting to receive work from the central coordinator. In other words, it is the fraction of time a process spends not performing computation. Recall that for minimization problems, a redundant node can be characterized as a subproblem with an LP relaxation value greater than the optimal solution. If the optimal solution is known at the beginning of the B&B tree search, all redundant nodes would be fathomed immediately, and therefore never processed. Hence, the number of redundant nodes processed is a measure of the extra work performed by the solver in order to prove optimality. We define node inefficiency as the fraction of the total number of nodes processed that are redundant nodes. In a serial B&B implementation, node inefficiency is purely a measure of how quickly the optimal solution is found. In parallel B&B implementations, there will be additional redundant work because some processes may not be working on the most promising nodes. Therefore, node inefficiency is a good surrogate for how well the parallel B&B framework ensures that the processors are doing useful work.

4.1 Scaling performance: B&B parallelization

We first present results that demonstrate the scaling performance of both parallel implementations. For these results, we use *sslp*_15_45_5, a problem instance with 5 scenarios, 3390 binary variables, and 301 constraints. Because of the relatively small number of scenarios, the LP solver PIPS-S is unable to scale beyond a handful of cores. Therefore, The focus of these experiments is to understand the benefits from B&B parallelization, the trade-offs involved, and the impact of certain algorithmic parameters.

Figure 8 shows the scaling performance of PIPS-PSBB (black) and ug[PIPS-SBB,MPI] (gray). For both frameworks, we use as many solvers as the number of available MPI processes. We see that PIPS-PSBB is able to scale up to 200 processes with a speedup of 66x with respect to the baseline serial execution (2920s). This represents a parallel efficiency of 33%. Further progress (increasing the number of processes to 400) is hampered by the overhead caused by the communication required to synchronize the solvers. The total number of nodes explored remains fairly constant, under 400000 nodes. The proportion of redundant nodes is below 15% for configurations under 50 processes, but increases as the number of solvers is increased further. Later, in Section 4.2, we will see that decreasing the number of solvers (by giving more processes to each PIPS-S solver) can reduce significantly the communication overhead as well as the node inefficiency. We next look at the scaling performance of ug[PIPS-SBB,MPI], presented in gray in Figure 8. This parallel solver is able to scale up to 200 processes, with a speedup of 33x with respect to the baseline serial execution (4491s). This represents a parallel efficiency of 16.5%. Compared to PIPS-PSBB, ug[PIPS-SBB,MPI] shows slightly worse performance. This performance decrease is caused by a larger communication overhead and a larger node inefficiency.

Figure 9 provides further information regarding the origin of the overhead for both frameworks. For ug[PIPS-SBB,MPI], the overhead coming from the ramp down gains importance as the number of processes increases. This overhead is due to the centralized nature of the load coordinator, which fails to provide all available solvers with open subproblems to process. As a result, most processes remain idle during ramp down. On the other hand, PIPS-PSBB has much lower overhead in general. However, its overhead in the primal phase increases as the number of processors increases, whereas ug[PIPS-SBB,MPI] has relatively stable overhead in this phase. These experiments suggest that the two parallelization frameworks have different strengths, and may perform differently depending on problem instance.

Finally, we analyze the behavior of PIPS-PSBB in more detail. As described in Section 2.2, the control of communication between solvers in PIPS-PSBB is dictated primarily by the communication frequency parameter λ. In the experiments presented in Figure 8, the parameter λ

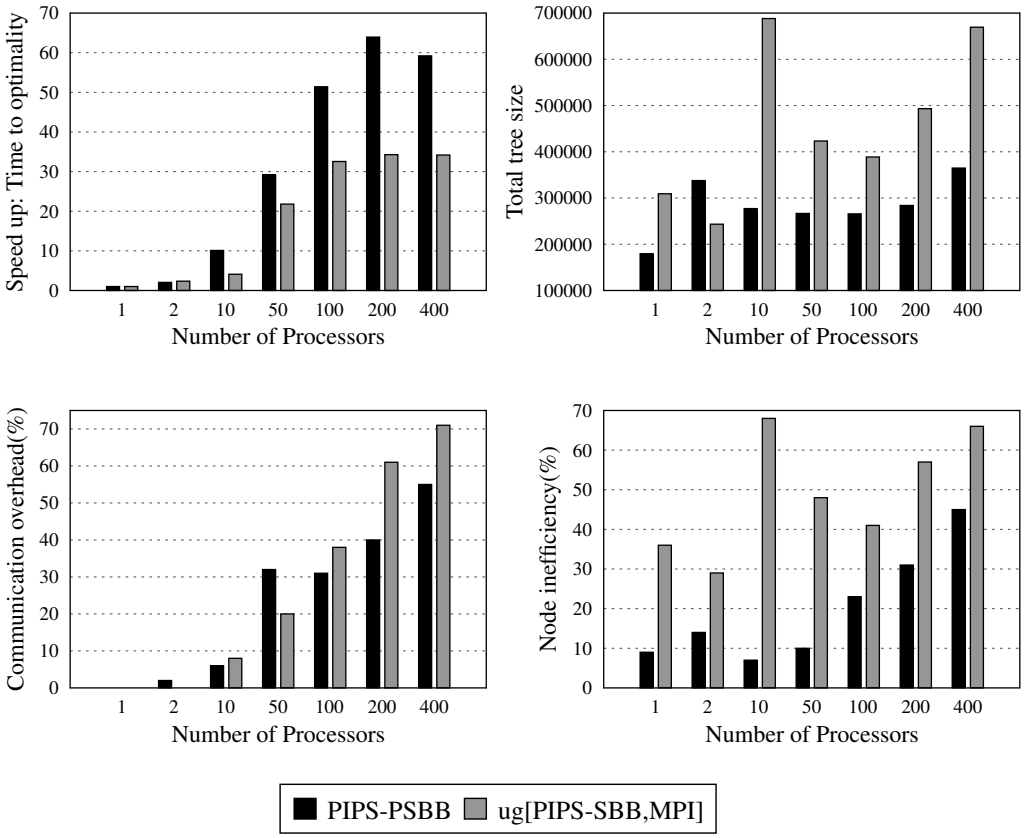


Figure 8: Scaling performance of PIPS-PSBB and ug[PIPS-SBB,MPI] when solving *sslp*_15_45_5: Speedup (time to optimality), branch and bound nodes processed, parallel communication overhead, and proportion of redundant nodes

is set to fluctuate within a minimum of 50 iterations and a maximum of 1000. In Figure 10, we study the effects on performance when the communication frequency is altered. When solvers are forced to communicate more frequently, PIPS-PSBB suffers from an increased communication overhead and a corresponding decrease in performance, especially when the number of processes is increased to 400. The positive side-effect from more frequent communication is a comparatively smaller tree size. When synchronization is less frequent, we see the opposite effect: a decrease in overhead but a significantly large number of nodes processed.

4.2 Scaling performance: Multiplicative effects of two levels of parallelism

Parallel scaling results on small problems such as *sslp*_15_45_5 provide an interesting picture for understanding the behavior and the interactions between the different solvers in the B&B parallelization frameworks. From a practical standpoint, it is more valuable to test the effects of parallelism when optimizing larger problems, especially those with a larger number of scenarios, because the UG framework was designed with large-scale parallel optimization in mind [21].

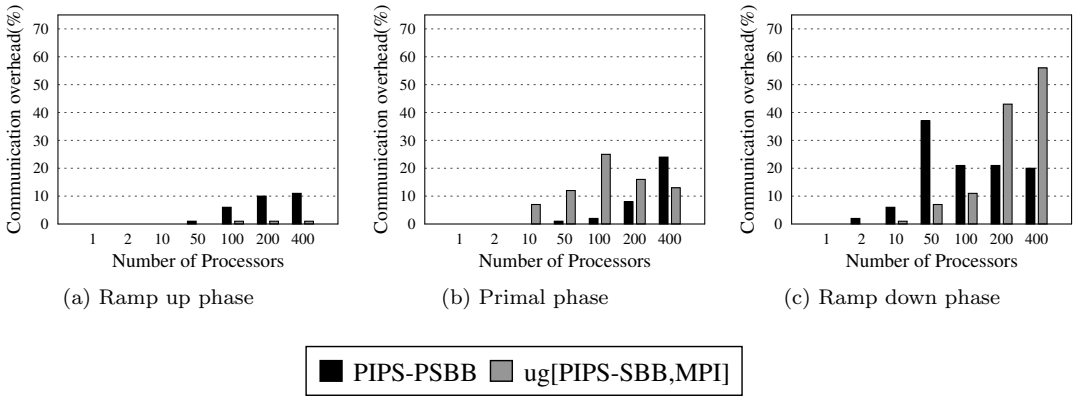


Figure 9: Communication overhead for PIPS-PSBB and ug[PIPS-SBB,MPI] when solving *sslp*_15_45_5 broken down by stage in the optimization process

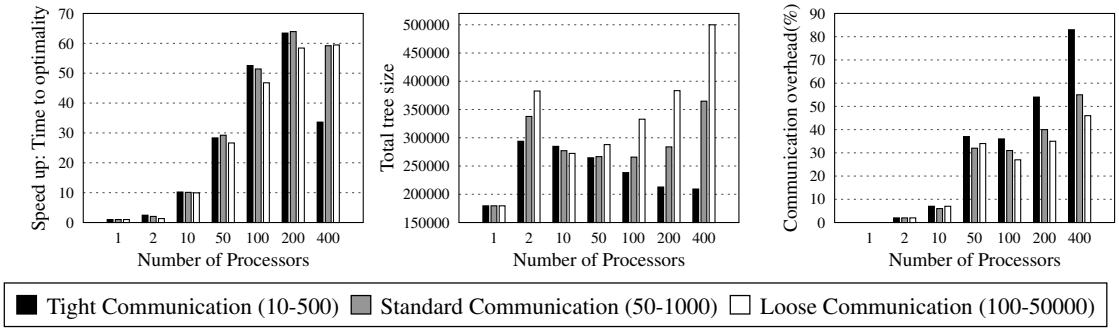


Figure 10: Performance of PIPS-PSBB for different communication frequencies λ. $(x-y)$ denotes the minimum (x) and the maximum (y) number of solver iterations before communication is attempted.

The following set of results describe the performance of PIPS-PSBB and ug[PIPS-SBB,MPI] when optimizing *sslp*_10_50_500, a problem with 500 scenarios, 250010 variables, and 504510 constraints. We particularly take a look at the combined effect of the two levels of paralellism by splitting a budget of 500 parallel cores and testing the solvers' behavior under different configurations.

In Figure 11, we first analyze the performance of the LP solver in solving the root node of the B&B tree. When a single core per solver is used (last column), the performance of the LP solver is significantly inferior to other configurations. However, its speed improves when more processors are used (moving to the left), achieving a speedup of 6x when using 10 cores per solver. As the number of processors increase, the returns for using more cores per solver diminishes. The trend is inverted after the mark of 20 cores per PIPS-S solver because the time needed to solve the LP relaxation grows and the efficiency of PIPS-S drops off quickly. This dropoff in PIPS-S efficiency suggests that the total available parallelism should be divided between the two levels of parallelism. In this case, the best configuration is to distribute the 500 available processors among 20-25 PIPS-SBB solvers, with each solver getting 20-25 processors. This timing result regarding multilevel parallelism is an important point to make, and further confirmed by the

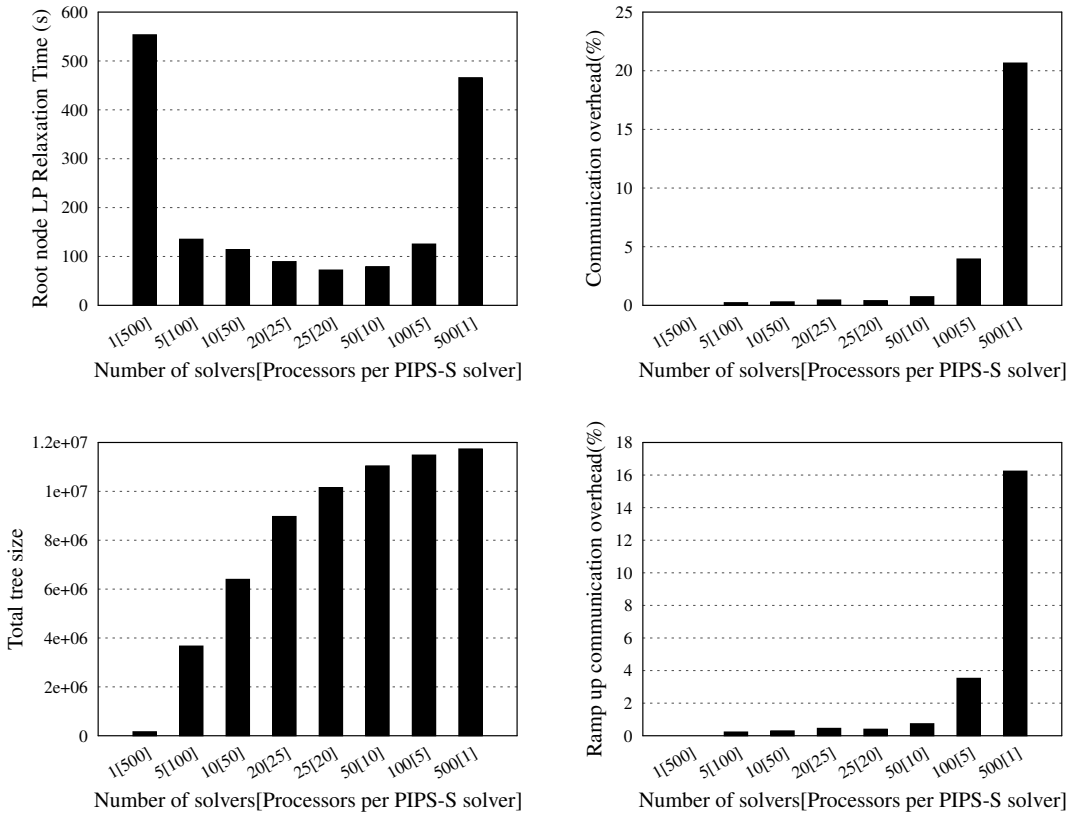


Figure 11: Scaling performance of PIPS-PSBB when solving *sslp*_10_50_500: Time to solve the LP relaxation, communication overhead, number of nodes processed, and communication overhead at ramp-up.

remaining charts in Figure 11: When large process counts are available, performance can be significantly improved by distributing them among different levels of parallelism.

The remaining charts display the performance of PIPS-PSBB in terms of total tree size and communication overhead. When solving larger problems, PIPS-PSBB is able to keep a low communication overhead until 100 solvers are used. The overhead spikes significantly when using 500 solvers. This spike in overhead is due to the slow performance of PIPS-S when solving the initial LP relaxations, and the problems PIPS-PSBB faces when generating work for all solvers quickly at the beginning of the optimization. As a result, most solvers remain idle at ramp-up, causing the significant overhead. This finding can be confirmed in the last plot, where the communication overhead at ramp up is plotted for the different processor arrangements, and is highly correlated with the overall communication overhead. In general, the node throughput increases as more solvers are used, though its progress is hampered by the overhead created by slow LP relaxations.

ug[PIPS-SBB,MPI] displays a similar performance behavior, as seen in Figure 12.

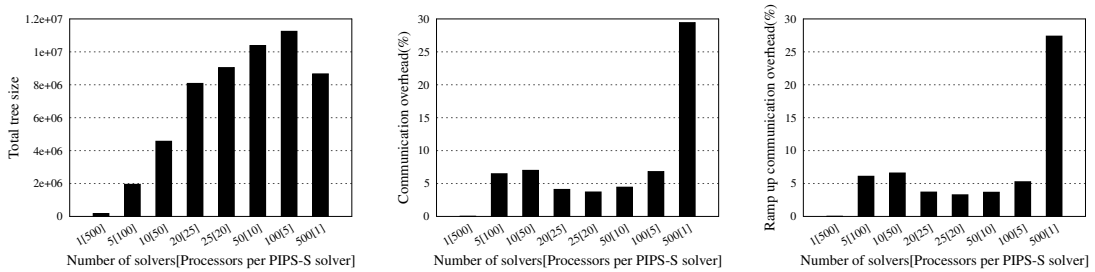


Figure 12: Scaling performance of ug[PIPS-SBB,MPI] when solving *sslp_10_50_500*: Time to solve the LP relaxation, communication overhead, number of nodes processed, and communication overhead at ramp-up.

4.3 Overall performance and comparison to CPLEX

We illustrate the overall performance of PIPS-PSBB and ug[PIPS-SBB, MPI] by testing them on all instances from the SSLP set. For these experiments, we present results for a representative parallel processor configuration, where the number of cores used for each PIPS-S LP solver is chosen as a function of the number of scenarios. In turn, a configuration of 200 solvers is used for all but the trivial problems. We also add CPLEX 12.6.2 to the comparison, both in its shared-memory and distributed-memory implementations using a comparable number of computing cores.

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)" in seconds needed to prove optimality. If an optimal solution is not provably obtained within the time limit, then the performance results are reported in terms of the relative gap, which is calculated based on the best found upper and lower bounds, z_{UB} and z_{LB}:

$$Gap = \frac{z_{UB} - z_{LB}}{z_{UB}} \tag{2}$$

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

From Table 1, we see that PIPS-PSBB and ug[PIPS-SBB,MPI] perform comparably through the entire problem set. The first set of rows correspond to the trivial *sslp_5_∗* instances, which all solvers are able solve to optimality, though CPLEX is significantly faster than all PIPS-SBB variants. The next set consists of three easy *sslp_15_∗* instances, which have a small numer of scenarios. CPLEX is able to solve all instances in this case, while PIPS-SBB implementations are able to solve only one of the three instances, and at a significantly slower pace. In the case of *sslp_10_∗* instances, the problem difficulty increases as the number of scenarios grows. Distributed-memory CPLEX runs out of memory before solving the instances with 50, 100 and 500 scenarios to optimality. We also suspect the same would have happened for the larger instances if the solver was allowed more time. The performance for *sslp_10_∗* is similar in all distributed-memory parallel algorithms, with PIPS-SBB implementations taking a significant advantage for instances with more than 500 scenarios. 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. It is also worth mentioning the poor performance of distributed-memory CPLEX compared to its shared-memory counterpart, which is only able to provide a better performance for the two largest problems in the set.

Table 1: Performance comparison for all SSLP instances

Problem Instance	Configuration Solvers	PIPS-S procs	PARBB Gap(%) (time)(s)	ug[PIPS-SBB,MPI] Gap(%) (time)(s)	CPLEX SM Procs	Gap(%) (time)(s)	CPLEX DM Procs	Gap(%) (time)(s)
sslp_5_25_50	2	2	(7.45s)	(8.03s)	4	(0.27s)	4	(0.27s)
sslp_5_25_100	2	2	(22.37s)	(17.7951s)	4	(0.64s)	4	(0.64s)
sslp_15_45_5	200	2	(107.11s)	(163.53s)	16	(1.97s)	400	(6.26s)
sslp_15_45_10	200	2	0.09	0.16	16	(1.81s)	400	(15.94s)
sslp_15_45_15	200	2	0.25	0.30	16	(7.8s)	400	(15.75s)
sslp_10_50_50	200	10	0.13	0.21	16	(43.88s)	2000	0.15(M)
sslp_10_50_100	200	10	0.17	0.20	16	(221.69s)	2000	0.16(M)
sslp_10_50_500	200	10	0.24	0.24	16	4.91(M)	2000	1.25(M)
sslp_10_50_1000	200	10	0.24	0.24	16	9.21	2000	6.08
sslp_10_50_2000	200	10	0.26	0.26	16	19.93	2000	8.11

In Table 2, we show the performance improvements made from the version of PIPS-SBB introduced in [16] to the current parallel implementations presented in this paper. Significant performance improvements are made possible, not only with the introduction of parallel B&B, but also with the addition of further specialized heuristics and branching methods.

Table 2: Version-to-version performance comparison of PIPS-SBB solvers

Problem Instance	PIPS-SBB presented in [16]		Parallel versions in current paper			
			Configuration		PARBB	ug[PIPS-SBB,MPI]
	Procs	Gap(time)	Solvers	PIPS-S procs	Gap(time)	Gap(time)
sslp_5_25_50	1	(12.34s)	2	2	(7.45s)	(8.03s)
sslp_5_25_100	1	(41.63s)	2	2	(22.37s)	(17.7951s)
sslp_15_45_5	2	1.36	200	2	(107.11s)	(163.53s)
sslp_15_45_10	2	7.93	200	2	0.09	0.16
sslp_15_45_15	2	5.25	200	2	0.25	0.30
sslp_10_50_50	5	1.48	200	10	0.13	0.21
sslp_10_50_100	10	1.74	200	10	0.17	0.20
sslp_10_50_500	50	1.57	200	10	0.24	0.24
sslp_10_50_1000	100	1.60	200	10	0.24	0.24
sslp_10_50_2000	100	24.00	200	10	0.26	0.26

5 Conclusions

In this paper, we present PIPS-PSBB and ug[PIPS-SBB,MPI]: two implementations of parallel distributed-memory Branch & Bound. The first of the proposed methods, PIPS-PSBB, is a new fine-grained algorithm for parallelizing the tree search. The coordination and load-balancing of the different parallel solvers is done in a decentralized fashion, and designed to ensure that all available cores are processing the most promising parts of the B&B tree. The second method is ug[PIPS-SBB,MPI]: a parallel implementation using UG, a generic framework for parallelizing B&B tree search that is relatively coarse-grained in its approach. The UG framework has been effectively used to parallelize other MIP solvers such as Xpress and SCIP.

We implement both frameworks for parallelizing B&B tree search as extensions of PIPS-SBB, a distributed memory solver for Stochastic MIPs (SMIPs). Therefore, both our implementations leverage two levels of nested parallelism in order to improve parallel scalability. We study the effects of leveraging multiple levels of parallelism in improving scaling performance. We also compare our algorithms against the distributed-memory B&B implementation of the state-of-the-art commercial solver CPLEX. The latter proves to be the best performer for small problems.

However, the specialized nature of the methods present in PIPS-SBB-based solvers enable them to outperform CPLEX in large SMIP instances.

PIPS-SBB has seen a dramatic performance improvement since its inception. As new features get added, it will become a more viable option when solving generic two-stage SMIPs. A natural extension of this work is to improve the performance of the base solver. In particular, further specialized methods to improve the convergence of bounds, such as specialized cuts and preprocessing can be added. Another natural extension of this work is to incorporate *three* levels of parallelism by hierarchically incorporating both frameworks of parallelism presented in this paper. Since UG can already handle a distributed-memory base solver, it can be integrated with the fine-grained parallel B&B implementation PIPS-PSBB, resulting in ug[*PIPS-PSBB*,MPI]. Adding a third level of parallelism would enable more opportunities for parallel speedup with large process counts.

6 Acknowledgements

Some of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. This work has also been supported by the Research Campus Modal *Mathematical Optimization and Data Analysis Laboratories* funded by the Federal Ministry of Education and Research (BMBF Grant 05M14ZAM), and partially supported by the BMWi project Realisierung von Beschleunigungsstrategien der anwendungsorientierten Mathematik und Informatik für optimierende Energiesystemmodelle - BEAM-ME (fund number 03ET4023DE).

References

[1] UG: Ubiquity Generator framework. `http://ug.zib.de/`.

[2] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. Technical report, Technical Report 16-44, ZIB, Takustr. 7, 14195 Berlin, 2016.

[3] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

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

[5] T. Berthold. Primal heuristics for mixed integer programs. Master's thesis, TU Berlin, 2006.

[6] J. Eckstein, W. E. Hart, and C. A. Phillips. PEBBL: an object-oriented framework for scalable parallel branch-and-bound. *Mathematical Programming Computation*, 7(4):429–469, 2015.

[7] J. Eckstein, C. A. Phillips, and W. E. Hart. PEBBL 1.0 User Guide. `https://software.sandia.gov/acro/releases/votd/acro/packages/pebbl/doc/uguide/user-guide.pdf`, 2007.

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

[9] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, and D. Weninger. Progress in presolving for mixed integer programming. *Mathematical Programming Computation*, 7(4):367–398, December 2015.

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

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

[12] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.

[13] M. Lubin, J. Hall, C. Petra, and M. Anitescu. Parallel distributed-memory simplex for large-scale stochastic LP problems. *Computational Optimization and Applications*, 55(3):571–596, 2013.

[14] S. J. Maher, T. Fischer, T. Galley, G. Gamrath, A. Gleixner, Robert L. Gottwald, G. Hendel, T. Koch, M. E. Lübbecke, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, D. Weninger, J. T. Witt, and J. Witzig. The SCIP optimization suite 4.0. Technical Report ZIB-Report 17-12, Zuse Institute Berlin, March 2017.

[15] H. Marchand, A. Martin, R. Weismantel, and L. Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1–3):397 – 446, 2002.

[16] L. M. Munguía, G. Oxberry, and D. Rajan. PIPS-SBB: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 730–739, May 2016.

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

[18] T. Ralphs, Y. Shinano, T. Berthold, and T. Koch. Parallel solvers for mixed integer linear programming. Technical Report 16-74, ZIB, Takustr.7, 14195 Berlin, 2016.

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

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

[21] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch, and M. Winkler. Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 770–779, Los Alamitos, CA, USA, 2016. IEEE Computer Society.

[22] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler. FiberSCIP – a shared memory parallelization of SCIP. *INFORMS Journal on Computing*, 30(1):11–30, 2018.

[23] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[24] D. T. Wojtaszek and J. W. Chinneck. Faster MIP solutions via new node selection rules. *Computers & Operations Research*, 37(9):1544–1556, 2010.

[25] Y. Xu. *Scalable algorithms for parallel tree search*. Phd thesis, Lehigh University, 2007.

[26] Y. Xu, T. K. Ralphs, L. Ladányi, and M. Saltzmann. ALPS version 1.5. `https://github.com/coin-or/CHiPPS-ALPS`, 2016.

[27] Y. Xu, T. K. Ralphs, L. Ladányi, and M. Saltzmann. BiCePs version 0.94. `https://github.com/coin-or/CHiPPS-BiCePS`, 2017.

[28] Y. Xu, T. K. Ralphs, L. Ladányi, and M. Saltzmann. BLIS version 0.94. `https://github.com/coin-or/CHiPPS-BLIS`, 2017.

[29] Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzmann. ALPS: a framework for implementing parallel search algorithms. The Proceedings of the Ninth INFORMS Computing Society Conference, pages 319–334, 2005.

[30] Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzmann. Computational expereicne with a software framework for parallel integer programming. *The INFORMS Journal on Computing*, 21:383–397, 2009.

[31] Q. P. Zheng, J. Wang, and A. L. Liu. Stochastic optimization for unit commitment - a review. *IEEE Transactions on Power Systems*, 30(4):1913–1924, July 2015.