Parallel MIP Solving with Dynamic Task Decomposition

Peng Lin
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China

Shaowei Cai[1]
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China

Mengchuan Zou
Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Shengqi Chen
Department of Computer Science and Technology, Tsinghua University, Beijing, China

Abstract

Mixed Integer Programming (MIP) is a foundational model in operations research. Although significant progress has been made in enhancing sequential MIP solvers through sophisticated techniques and heuristics, remarkable developments in computing resources have made parallel solving a promising direction for performance improvement. In this work, we propose a novel parallel MIP solving framework that employs dynamic task decomposition in a divide-and-conquer paradigm. Our framework incorporates a hardness estimate heuristic to identify challenging solving tasks and a reward decaying mechanism to reinforce the task decomposition decision. We apply our framework to two state-of-the-art open-source MIP solvers, SCIP and HiGHS, yielding efficient parallel solvers. Extensive experiments on the full MIPLIB benchmark, using up to 128 cores, demonstrate that our framework yields substantial performance improvements over modern divide-and-conquer parallel solvers. Moreover, our parallel solvers have established new best known solutions for 16 open MIPLIB instances.

2012 ACM Subject Classification Mathematics of computing → Integer programming; Computing methodologies → Parallel algorithms; Applied computing → Operations research

Keywords and phrases Mixed Integer Programming, Parallel Computing, Complete Search, Task Decomposition

Digital Object Identifier 10.4230/LIPIcs.CP.2025.26

Supplementary Material *Software*: `https://github.com/shaowei-cai-group/PartiMIP`

Funding This work is supported by National Key R&D Program of China (2023YFA1009500).

1 Introduction

Mixed Integer Programming (MIP) is both a foundational model in operations research and a central combinatorial optimization problem [40, 2]. Its importance stems from its ability to model a wide range of problems in diverse fields, from practical applications such as planning [31] and scheduling [13] to theoretical problems in propositional logic [11] and graph theory [25].

[1] Corresponding author

MIP is formulated to optimize a linear objective function subject to linear constraints, where decision variables may be either integer or real-valued. Because solving general MIP instances is NP-hard [20, 19], practical applications require sophisticated algorithms. Since solver performance often represents a critical bottleneck when addressing large-scale real-world instances, improving the efficiency of MIP solvers remains a primary focus for algorithm developers.

During the past decades, significant advances have been made in the algorithmic efficiency of MIP solvers, with substantial research efforts primarily focused on the refinement of sequential solving techniques and heuristic enhancements [21, 2, 23]. In today's computing landscape, multi-core architectures are ubiquitous, making parallel computing a natural approach for improving solver performance and a widely researched topic in constraint programming [24, 34, 15].

Numerous studies on parallel MIP solving can be categorized into two groups as summarized in [41]: those developed basically from scratch [12, 5, 8, 9, 32] and those that integrate existing MIP solvers [26, 38, 37, 7, 36, 39]. However, only a few approaches in the latter group achieved competitive results compared to state-of-the-art sequential solvers [39], highlighting the persistent challenges in parallel MIP solving. A further obstacle in this field is that the most effective solvers, both parallel and sequential, are closed-source commercial products, such as Gurobi [17], CPLEX [29], Xpress [3], and COPT [14]. Consequently, it is difficult for the academic community to analyze the underlying principles of their parallel approaches, and when these solvers are used as black-box base solvers, the potential for parallel interaction is inherently limited [33].

Current academic research on parallel MIP solving can be broadly classified into two approaches: portfolio-based and divide-and-conquer methods. Portfolio solvers concurrently execute multiple solvers, or different configurations of a single solver, to address identical or slightly perturbed MIP instances. Representative examples include the race ramp-up technique employed in FiberSCIP [36, 39], the DC cutting plane algorithm with multiple random starts [30], and portfolio methods based on local search [22, 10]. However, the performance of portfolio methods is inherently constrained by the best possible sequential performance. In contrast, the divide-and-conquer approach has the potential to outperform the best sequential methods by accelerating the solving process through the parallelization of key algorithmic components. For example, FiberSCIP [39] parallelizes the processing of nodes from sequential branch-and-bound solvers, while HiGHS offers a parallel version that accelerates the solving of internal linear programming and symmetry detection [18]. Similar work in the area of stochastic MIP includes PIPS-PSBB [28], which combines parallel linear programming solving with parallel branch-and-bound tree search. Furthermore, portfolio-based and divide-and-conquer approaches can be seamlessly integrated to develop highly effective hybrid solvers. Extensive investigations on hybrid solvers have been predominantly led by the UG framework of the SCIP team and are well documented in [36, 39].

Our research focuses on the divide-and-conquer strategy for parallel MIP solving. Most of the existing approaches in this category are tightly coupled with their underlying sequential algorithms. For instance, FiberSCIP relies on sequential solvers to generate nodes to be distributed and solved in parallel, thereby forcing the base solver to employ branch-and-bound methods to maintain the branch-and-bound tree. Consequently, the parallel performance of such systems is heavily influenced by the search strategies (e.g., branching and node selection) employed by the sequential solver.

In this work, we propose a novel parallel MIP solving framework that supports search strategies tailored for parallel environments through dynamic problem decomposition. In our approach, solving a MIP problem is treated as a task organized within a global task

tree of the parallel system, which is used to schedule these tasks for parallel execution by the underlying base solvers. The expansion of the task tree is entirely governed by our newly designed decomposition algorithms, which integrate the dynamic information from the parallel solving process to enhance decomposition decisions, and are inherently independent of the underlying base solver. Consequently, our framework is loosely coupled with the base solvers, which require only standard input/output interfaces and are not restricted to branch-and-bound methods. This design enables seamless integration with a wide range of MIP solvers.

To efficiently expand the task tree, we propose a dynamic task decomposition method by partitioning selected tasks into smaller subtasks. The decision on which task to decompose, as well as the specific decomposition action used, is critical to the overall solving process. To optimize this, we propose a hardness estimate heuristic to assess the difficulty of task solving, thus prioritizing the decomposition of more challenging tasks and allocating additional computational resources accordingly. Furthermore, we develop a reward-guided mechanism that leverages historical feedback from the task tree construction to reinforce future decomposition decisions. By incorporating a reward decaying strategy, this mechanism strikes a balance between exploration and exploitation, ultimately enhancing the performance of the parallel solving process.

We apply our framework to state-of-the-art open-source MIP solvers, including SCIP [6] and HiGHS [18], as our base solvers. We conducted experiments on the entire MIPLIB benchmark across a wide range of core configurations from 8 to 128 cores to evaluate the performance of the resulting parallel solvers. To the best of our knowledge, the 128-core configuration reported in our work represents the largest core configuration used to test parallel solvers on the entire MIPLIB benchmark. Overall, the experimental results demonstrate that our techniques significantly enhance the performance of sequential MIP solvers, yielding a substantial improvement in the discovery of high-quality solutions and the number of solved instances. We compare our framework with the default divide-and-conquer strategies implemented in SCIP (i.e., FiberSCIP) and HiGHS. The comparison shows that our strategy consistently outperforms the corresponding default parallel approaches, and the parallel solver developed using our framework achieves state-of-the-art performance among open-source parallel solvers. Moreover, our framework establishes 16 new records for MIPLIB open instances by discovering new best known solutions.

To facilitate reproducibility and further research, we have made our solvers, evaluation scripts, experimental results, and new best known solutions available on GitHub [2].

2 Preliminaries

2.1 Mixed Integer Programming

▶ **Definition 1** (Mixed Integer Programming). *Let* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *be a matrix,* $\mathbf{b} \in \mathbb{R}^m$ *a vector,* $\mathbf{c}, \mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ *vectors,* $I \subseteq N = \{1, \ldots, n\}$ *denote a set of indices, and* $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ *be the set of decision variables. The mixed integer programming problem is defined as follows:*

[2] https://github.com/shaowei-cai-group/PartiMIP

$$\min_{\mathbf{x}} \quad \mathbf{c}^\top \mathbf{x} \quad s.t. \quad \begin{aligned} &\mathbf{A}\mathbf{x} \leq \mathbf{b}, \\ &\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \\ &x_j \in \mathbb{Z} \quad \forall j \in I, \\ &\mathbf{x} \in \mathbb{R}^n. \end{aligned} \tag{1}$$

In this formulation, $\mathbf{c}^\top \mathbf{x}$ denotes the objective function, $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ represents the general linear constraints, $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ specifies the global bounds of decision variables, and the integrality constraints require $x_j \in \mathbb{Z}$ for all $j \in I$.

Domain Propagation. Domain propagation is a technique to reformulate a MIP problem into an equivalent formulation that is potentially easier to solve. By iteratively tightening the variable domains (i.e., $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$) through iterative analysis of constraints and neighboring variable domains, this technique reduces the complexity of the MIP model by eliminating redundancies, such as satisfied constraints or fixed variables. Consequently, domain propagation often effectively excludes considerable portions of the search space, occasionally enabling proofs of optimality or infeasibility. In practice, modern solvers efficiently implement domain propagation in their presolving application programming interfaces (APIs) [6, 17, 29]. The fundamental principles of domain propagation are detailed in [35].

2.2 Parallel Divide-and-conquer Strategies in MIP Solving

Here, we review divide-and-conquer-based parallel strategies employed in state-of-the-art solvers. Among today's open-source solvers, SCIP and HiGHS are prominent, each offering a parallel version. For SCIP, the parallel version, known as FiberSCIP [36, 39], adopts a divide-and-conquer strategy to parallelize the processing of nodes in the branch-and-bound tree. In this method, base solvers alternately solve branch-and-bound nodes and transfer the unsolved child nodes to a load coordinator. The load coordinator maintains a pool of unsolved nodes and assigns them to idle solvers as they become available. In FiberSCIP, the generation of nodes is determined by the internal algorithm of the sequential branch-and-bound solver, while the parallel component is primarily responsible for the collection and distribution of nodes from the sequential branch-and-bound tree. In contrast, the parallel version of HiGHS [18] focuses on the parallelization of specific components within its sequential solver, including the dual simplex algorithm, symmetry detection, and querying of clique tables [3]. Notably, parallel HiGHS has been the top-ranked open-source solver in the MIP rankings for several consecutive years [4].

Both these two parallel solvers are tightly coupled with their underlying sequential frameworks, making it challenging to adapt their methods to solvers lacking these specific components. In contrast, our goal is to develop a general parallel framework that does not rely on the base solver's internal algorithms and requires only a standard input/output interface.

[3] `https://ergo-code.github.io/HiGHS/dev/parallel/`
[4] `https://mattmilten.github.io/mittelmann-plots/`

3 Our Parallel Framework for Solving MIP

In this section, we introduce our parallel MIP solving framework with dynamic task decomposition, named PartiMIP. We first present the basic concepts of the main components and how they cooperate together. Afterwards, we dive into the motivation and design of each component. A detailed description of the dynamic task decomposition method is provided in Sect. 4.

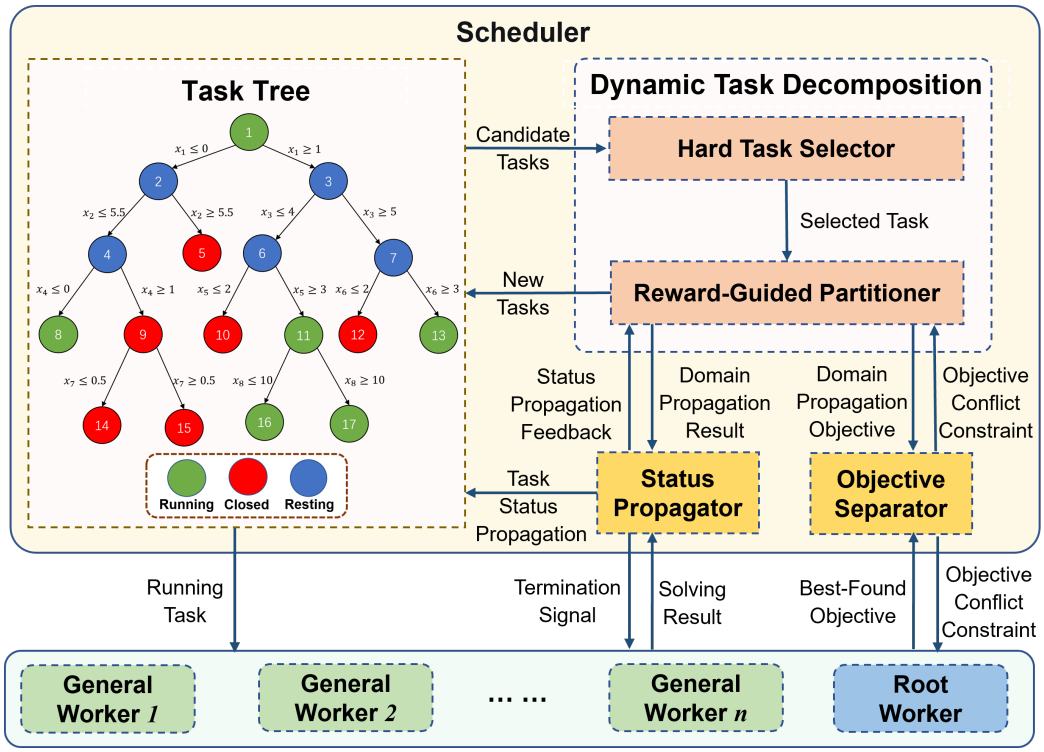

Figure 1 PartiMIP: A parallel MIP solving framework with dynamic task decomposition.

3.1 Basic Concept and Process of Framework

As illustrated in Fig. 1, our framework consists of two roles: the scheduler and the worker. These roles are executed by distinct threads to leverage multiple cores in parallel:

- **Scheduler:** The scheduler maintains a *task tree*, in which the nodes represent the solving tasks for the original problem or subproblems of a given MIP instance. It incorporates a *dynamic task decomposition* to expand the tree, a *status propagator* to deduce task statuses and an *objective separator* to reduce search spaces.
- **Worker:** The workers primarily invoke a base MIP solver to solve the tasks. A *root worker* is designated solely to solve the original problem, while other *general workers* are flexible and can solve any tasks dynamically assigned by the scheduler.

The task tree serves as the central data structure throughout the solving process, where each task can be in one of three possible statuses:

- **Running**: Tasks that are currently being solved by workers.

- **Closed**: Tasks for which the solution result is known (i.e., optimal or infeasible). The solving of closed tasks is completed, and the entire solving process is completed when the root task is closed.
- **Resting**: Tasks that are decomposed into subtasks and remain unassigned to workers. The solution results of resting tasks are inferred from the results of their subtasks.

At a given time, the leaf nodes in the task tree are denoted as *leaf tasks*, each of which contains a distinct search space.

Initially, the scheduler designates the given MIP problem as the root task of the task tree and assigns it to the root worker for solving, enabling quick completion when the instance is simple, and thereby avoiding the overhead of parallel solving.

Subsequently, the scheduler accelerates the solving process in a divide-and-conquer view, decomposing the root task into simpler subtasks. During task decomposition, a leaf task is selected, and the domain of one variable is partitioned into two parts to generate two new subtasks. The search spaces of these subtasks are further reduced through domain propagation, and then the new subtasks are inserted as leaf tasks to update the task tree.

Allowing tasks with overlapping search spaces to be solved simultaneously can lead to the exploration of identical search spaces, which should be avoided as much as possible. In the task tree, each leaf task contains distinct search spaces. To ensure that each general worker explores distinct search spaces at the beginning, task decomposition is divided into two phases:

1. **Initial Decomposition Phase**: Initially, only the root worker is activated to solve the root task, while all general workers remain idle. The scheduler iteratively decomposes the leaf tasks to generate a sufficient number of simpler leaf subtasks to assign to all general workers, during which the statuses of the decomposed tasks transition to resting. The task decomposition process is parallelized to enhance resource utilization and accelerate this phase. Specifically, let $\mathcal{N}$ denote the available CPU cores. $min\{\mathcal{N}/2,\ size(leaf\ tasks)\}$ tasks are selected to be decomposed, and the subtasks are generated and simplified in parallel. Since all selected tasks are leaf tasks, the parallel domain partition and domain propagation processes are independent and consistent. Once sufficient tasks are generated, all general workers are activated to solve the leaf tasks, each with distinct search spaces.
2. **Dynamic Decomposition Phase**: General workers become idle once their assigned tasks are closed. To utilize these idle workers, the scheduler dynamically decomposes ongoing tasks into simpler subtasks and assigns them to idle workers. This strategy ensures that idle workers remain engaged in resolving challenging tasks, thus expediting the whole solving process.

Once a task is solved successfully, the corresponding worker communicates the solution result to the scheduler. Additionally, a worker may receive a termination signal from the scheduler due to task status propagation. In either case, the worker terminates the ongoing task and releases its computational resources to accommodate upcoming tasks.

The simplicity of interaction with the scheduler allows the worker role to be seamlessly integrated into most modern MIP solvers via APIs, without requiring modifications to their internal algorithms. The only requirement for base solvers is the support for a standard input/output interface (e.g., reading problem files and logging solution results), which allows the use of any algorithm, not just those based on branch-and-bound methods. This design offers substantial flexibility in selecting and configuring base solvers, enabling a diverse range of potential solving approaches.

3.2 Task Status Propagation

In the task tree, subtasks are generated by partitioning the domain of a variable in their parent task. Consequently, the union of the search spaces of the subtasks is equivalent to the search space of their parent task. Based on this, the status propagator is designed to exploit the relationship between parent and child tasks to deduce the solution results of related tasks, with the aim of accelerating the solving process. The status propagator monitors signals that signify the solution result of each task, which originate from two sources:

1. During task decomposition, the domain propagation process may directly prove new subtasks to optimality or infeasibility.
2. Workers return the solution results of running tasks.

Once a task is solved, the status propagator tries to deduce the statuses of related tasks through the task tree. The propagation of task status can take place in two directions: upward and downward.

- **Upward Propagation**: This happens when all child tasks are closed. If all child tasks are determined to be infeasible, the parent task is closed as infeasible, and the propagation proceeds upward. If any child task is optimal, the parent task is closed as optimal, and the propagation continues upward.
- **Downward Propagation**: Once the parent task is proven to be optimal or infeasible, all child tasks are closed as optimal or infeasible, and the propagation continues downward.

Furthermore, propagation information reflects the quality of the relationship between parent and subtasks, which is used to improve future task decomposition decisions (Sect. 4.2).

3.3 Objective Conflict Constraint

The differences in search spaces between tasks arise from partitioning variable domains and simplifying through domain propagation. As a result, the original solution for the root task can be reconstructed from any given task solution by reintroducing the eliminated variable assignments. Additionally, the value of the objective function remains unchanged when reconstructing the original root task solution.

Therefore, with respect to the globally updated best-found objective value $\mathcal{O}^*$, finding solutions worse than $\mathcal{O}^*$ for new tasks is meaningless. To eliminate redundant search space that cannot contain a better solution than $\mathcal{O}^*$ for new tasks, we design the objective separator to generate the objective conflict constraint. Specifically, the objective separator dynamically collects the globally updated best-found objective value $\mathcal{O}^*$ in real time. Relying on $\mathcal{O}^*$, for a task $\mathcal{T}$, the objective separator constructs the latest objective conflict constraint:

$$\text{Objective Conflict Constraint: } \mathbf{c}_{\mathcal{T}}\mathbf{x}_{\mathcal{T}} < \mathcal{O}^* - \text{offset}_{\mathcal{T}} \tag{2}$$

where $\mathbf{c}_{\mathcal{T}}\mathbf{x}_{\mathcal{T}}$ represents the product of the variables and their corresponding coefficients in the objective function of $\mathcal{T}$, and $\text{offset}_{\mathcal{T}}$ is the constant term in the objective function of $\mathcal{T}$, which is derived from variable elimination during the domain propagation process.

The objective conflict constraint is in the standard form of a MIP constraint. Incorporating the objective conflict constraint effectively forces the new task to search exclusively for better feasible solutions, significantly reducing the search space. The objective conflict constraint can bring advantages to both the regular solving process and the task decomposition:

- Worker's regular solving: Before solving a newly assigned task, the worker retrieves the latest objective conflict constraint, which is then incorporated into the MIP model to reduce the search space. After task solving begins, the worker synchronizes its best-found objective value in real time with the scheduler, feeding it back to improve the objective conflict constraint.

- Task decomposition: The latest objective conflict constraint is added to the new tasks generated during task decomposition, further enhancing the domain propagation process for greater simplification. Additionally, task decomposition may provide a new best-found objective value once the new task is reduced to optimality via domain propagation.

Providing the best-found objective value as early as possible to the objective separator facilitates the creation of the objective conflict constraint to aid the task decomposition. Therefore, an additional benefit of starting the root worker before the initial decomposition phase is that its regular solving can yield the best-found objective value early in the process.

Additionally, since constructing an objective conflict constraint requires a feasible solution, the application of the objective conflict constraint does not affect the solution result of the original problem, including its feasibility or optimality.

4 Dynamic Task Decomposition

In this section, we propose a dynamic task decomposition method, the key process to maintain the entire task tree. This process decomposes challenging tasks into simpler new tasks for parallel solving. First, we design the *hard task selector*, which selects complex tasks from the candidate tasks to be decomposed. Second, we propose the *reward-guided partitioner* to partition the selected task based on the feedback from historical decomposition actions. These two components both combine static task information with dynamic solving data to enhance decision making. Finally, in Sect. 4.3, we present the complete decomposition algorithm.

4.1 Hard Task Selector

When maintaining the task tree, the elementary operation involves decomposing a complex task into smaller subtasks. The first step in this process is to select a task for decomposition. The strategy used for this selection can influence the structure of the tree by guiding its expansion, such as in a breadth-first search (BFS) or depth-first search (DFS) manner. However, simple strategies like BFS or DFS do not consider the characteristics of the MIP instance in parallel solving, ignoring factors such as load balancing and convergence during the solving process.

Preliminary experiments indicate that distributing relatively easy tasks to workers can incur significant overhead on the scheduler. First, frequent communication between the scheduler and workers is required. Second, continuous decomposition is necessary to generate new tasks for workers that often remain idle. To mitigate this issue, we propose the hard task selector, which prioritizes more challenging tasks for decomposition, aiming to allocate more resources to the complex parts of the task tree.

However, assessing the difficulty of solving a MIP task is NP-hard. To address this, we define a hardness estimate function to heuristically estimate the difficulty of a task. The sparsity of a MIP matrix ($\mathbf{A}$ in $\mathbf{Ax} \leq \mathbf{b}$) is correlated with the difficulty of solving the problem [4]. When the constraint matrix is dense, the solving of large models often becomes computationally intractable, as noted in the HiGHS documentation [5]. Therefore, during the initial decomposition phase, the difficulty of a task is assessed by its non-zero count, defined as follows:

▶ **Definition 2** (nnz). *The **nnz** of a task is the number of non-zero elements in the constraint matrix of the task.*

[5] https://ergo-code.github.io/HiGHS/dev/terminology/#The-constraint-matrix

As the process transitions into the dynamic decomposition phase, tasks are selected among those currently running, and additional runtime information becomes available. At any given moment, tasks that have been running longer are more likely to encounter computational difficulties due to increased elapsed time. Therefore, we define the duration of a task as follows:

▶ **Definition 3** (duration). *Given a specified time point, the* ***duration*** *of a task is the elapsed runtime since the task was assigned to a worker for execution.*

Consequently, the hard task selector incorporates both the non-zero count and the duration. Specifically, for a task $\mathcal{T}$, the hardness estimate function $\mathbf{Hardness}(\mathcal{T})$ is defined as:

$$\text{Hardness}(\mathcal{T}) = \begin{cases} nnz \text{ of } \mathcal{T}, & \text{initial decomposition phase,} \\ nnz \text{ of } \mathcal{T} \times duration \text{ of } \mathcal{T}, & \text{dynamic decomposition phase.} \end{cases} \tag{3}$$

4.2 Reward-Guided Partitioner

Given a task selected by the hard task selector, we design a reward-guided partitioner to choose a specific variable for decomposition. The domain of the selected variable is partitioned and subsequently the domain propagation is applied to simplify the resulting subtasks. The partitioner incorporates a reward-guided mechanism for variable selection and employs a reward decaying strategy to balance exploration and exploitation.

Rewarding Variables. The goal of task decomposition is that the resulting subtasks are more easily solvable than the parent task. If a parent task is closed due to upward propagation triggered by the closure of all its child tasks, this indicates that the selected variable for decomposing the parent task is a favorable choice. This is because the time required to close all subtasks is shorter than the time required to solve the parent task directly. To encourage this desirable outcome, we design a variable reward mechanism to reinforce future task decomposition decisions.

Initially, each variable is associated with a global reward value, denoted $\mathbf{Reward}(x)$, which is initialized to 0. Once the status propagator receives a solution result of a task, it performs task status propagation to update the task tree. Subsequently, the results of this task status propagation are then used to reward the corresponding variables. For simplicity, we denote the variable selected in the decomposition of a parent task $\mathcal{T}$ as $x_{\mathcal{T}}$. Specifically, for each closed task in a task status propagation, the rewards of corresponding selected variables are updated as follows:

$$\text{Reward}(x_{\mathcal{T}}) := \begin{cases} \text{Reward}(x_{\mathcal{T}}) + 1, & \text{if } \mathcal{T} \text{ is closed by upward propagation} \\ \text{Reward}(x_{\mathcal{T}}), & \text{otherwise} \end{cases} \tag{4}$$

These updates ensure that a higher reward for a variable indicates that it has been selected more frequently in parent tasks that are closed via upward propagation.

Reward-Guided Multi-level Selection. For task decomposition, we prioritize selecting the variable with the highest reward, which is dynamically updated as the solving process progresses. However, when multiple variables share the same highest reward, particularly before any upward propagation occurs, tiebreaking is necessary. In such cases, static information about variables is incorporated to guide the task decomposition process.

First, we consider the number of constraints in which each variable appears. The more constraints a variable appears in, the greater the potential to simplify additional constraints through domain propagation resulting from the partitioning of the variable's domain.

Additionally, based on our preliminary experiments, we observe that partitioning variables with very small domains (e.g., less than 0.1) or excessively large domains (e.g., containing infinity) is often ineffective. Therefore, variables with reasonable domain lengths are prioritized.

Algorithm 1 Reward-Guided Multi-level Selection Heuristic.

Input: $\mathcal{T}$: A task to be decomposed
Output: $x_{\mathcal{T}}$: A variable to decompose the task $\mathcal{T}$

```
candidate_variables ← ∅;
normal_variables ← {variables with normal domain lengths | all variables in T};
if normal_variables ≠ ∅ then
    candidate_variables ← normal_variables;
else candidate_variables ← {all variables in T};
best_variables ← arg max_{v∈candidate_variables} {Reward(v)};
x_T ← arg max_{v∈best_variables} {the number of constraints where v appears in T};
return x_T;
```

Based on these considerations, we design a reward-guided multi-level heuristic to select the most promising variable for decomposing a task, as detailed in Algorithm 1.

Reward Decaying. In the reward-guided multi-level selection algorithm, reward is the primary criterion for variable selection. Since subtasks only contain a subset of the original problem's variables, the higher a variable's reward, the greater its chance of becoming the variable with the highest local reward for a specific task. Therefore, as the dynamic decomposition phase progresses, the likelihood that a variable is selected increases with its reward. Being selected, in turn, is a prerequisite for the increase of reward, creating a positive feedback loop within the algorithm. However, such a mechanism can lead to premature convergence in the selection process. To enhance selection diversity and strike a balance between exploration and exploitation, we propose a reward decaying strategy. When a variable is selected, its reward decays, thereby increasing the likelihood that other variables will be selected in subsequent iterations. In this sense, the reward of a variable becomes a global quota, and each time it is selected, a portion of its reward is consumed.

Specifically, for a task $\mathcal{T}$ to be decomposed, the reward of the selected variable $x_{\mathcal{T}}$ is decayed as follows:

$$\text{Reward}(x_{\mathcal{T}}) := \begin{cases} \text{Reward}(x_{\mathcal{T}}) - 1, & \text{if Reward}(x_{\mathcal{T}}) > 0 \\ \text{Reward}(x_{\mathcal{T}}), & \text{otherwise} \end{cases} \tag{5}$$

4.3 Task Decomposition Algorithm

Here, based on the proposed ideas, we present the dynamic task decomposition in Algorithm 2.

Algorithm 2 Task Decomposition Algorithm.

1 $candidate\_tasks \leftarrow$ {the unclosed leaf tasks | leaf tasks from the task tree};
2 $\mathcal{T} \leftarrow \arg\max_{\tau \in candidate\_tasks}$ {hardness estimate function Hardness(τ) by Eq. 3};
3 Update the latest objective conflict constraint of $\mathcal{T}$ to reduce search space by Eq. 2;
4 $\mathcal{V} \leftarrow$ a decomposition variable to be partitioned in $\mathcal{T}$ by Algorithm 1;
5 Perform reward decaying on $\mathcal{V}$ by Eq. 5;
6 $\{lb, ub\} \leftarrow$ the lower bound and upper bound of $\mathcal{V}$ within $\mathcal{T}$;
7 $\{\mathcal{T}_{\text{left}}, \mathcal{T}_{\text{right}}\} \leftarrow$ partition the domain of $\mathcal{V}$ on $\mathcal{T}$ into $[lb, \frac{lb+ub}{2}]$ and $[\frac{lb+ub}{2}, ub]$;
8 $\{\hat{\mathcal{T}}_{\text{left}}, \hat{\mathcal{T}}_{\text{right}}\} \leftarrow$ perform domain propagation on $\{\mathcal{T}_{\text{left}}, \mathcal{T}_{\text{right}}\}$ to simplify tasks;
9 Add $\{\hat{\mathcal{T}}_{\text{left}}, \hat{\mathcal{T}}_{\text{right}}\}$ into the task tree as new leaf tasks;

First, the task and the variable for task decomposition are selected using the hardness estimate function and the reward-guided multi-level selection heuristic, respectively (lines 1-4). Next, the reward decaying is applied to the selected variable (line 5). Two subtasks are generated by partitioning the domain of the selected variable at the midpoint (lines 6-7), followed by domain propagation to simplify the new tasks (line 8). Finally, the newly generated tasks are added to the task tree (line 9). Note that the latest objective conflict constraint is integrated into the task decomposition to facilitate domain propagation, thereby simplifying additional search spaces (line 3).

5 Experimental Evaluations

This section systematically evaluates the PartiMIP framework through four complementary dimensions: (1) advantages over state-of-the-art parallel strategies, (2) breakthroughs in challenging open instances, (3) performance gains over sequential solvers, and (4) ablation studies on the effectiveness of the reward-guided multi-level selection heuristic. Collectively, the experimental results demonstrate our framework's ability to leverage parallel computation to improve both solution quality and solving efficiency for general MIP instances.

5.1 Experiment Preliminaries

This subsection introduces the setup of the experiment, including the implementation, benchmarks, base solvers, running environment, and evaluation metrics.

Implementation. Both the scheduler and the workers in our framework are implemented in C++ and compiled using g++ 9.4.0 with the -O3 optimization flag. Parallelization is achieved using the pthread libraries. During task decomposition, domain propagation is executed by invoking the presolving APIs provided by the HiGHS libraries [18] [6].

Benchmarks. Our experiments are conducted on the entire MIPLIB benchmark, MIPLIB2017 [16], which comprises 240 general MIP instances collected from a wide range of real-world applications [7], which also serves as the criterion for Hans Mittelmann's benchmark [8], which is a widely recognized worldwide MIP solver ranking [27].

[6] https://ergo-code.github.io/HiGHS/dev/guide/further/#guide-presolve
[7] https://miplib.zib.de/downloads/benchmark.zip
[8] https://mattmilten.github.io/mittelmann-plots/

Base Solvers. For our research, we selected two state-of-the-art open-source MIP solvers as base solvers in the workers: SCIP (v9.2.0) [1, 6] and HiGHS (v1.9.0) [18]. SCIP is one of the most widely used MIP solvers in both academia and industry; it has been in continuous development for over 20 years and plays a crucial role in solving various optimization problems. Similarly, HiGHS has consistently demonstrated superior performance among open-source solvers in MIP solver rankings over several consecutive years. For comparison, we also evaluated the parallel versions of these solvers.

Experiment Setup. All experiments were carried out on a server equipped with 2 AMD EPYC 9654 CPUs and 2048 GB of RAM, running Ubuntu 20.04.4. We report the following metrics for each solver on the MIPLIB benchmark:

- **SOLVED**: The total number of instances that the solver successfully solved, either by proving optimality or infeasibility.
- **SOLVED Improvement (S-Imp.)**: The percentage increase in the total number of solved instances relative to sequential solvers or comparative parallel solvers, reflecting the overall enhancement in problem resolution capability.
- **PAR-2**: The penalized runtime score commonly used in SAT competitions [9]. The runtime for an unsolved instance is penalized by twice the time limit, yielding a single measure that reflects both the runtime efficiency and the number of instances solved.
- **PAR-2 Improvement (P-Imp.)**: The percentage reduction in the PAR-2 score relative to sequential solvers or comparative parallel solvers, indicating the overall efficiency gain.
- **FEAS**: The number of instances for which the solver finds at least one feasible solution, thereby assessing its ability to identify feasibility.
- **FEAS Improvement (F-Imp.)**: The percentage increase in the number of instances with at least one feasible solution compared to sequential solving or comparative parallel solvers, reflecting improvements in feasibility detection.
- **WIN**: The number of instances in which the solver achieved the best feasible solution among compared solvers, measuring the effectiveness in obtaining high-quality solutions.
- **WIN Improvement (W-Imp.)**: The percentage increase in the number of instances in which the solver found the best feasible solution relative to sequential solving or comparative parallel solvers, highlighting improvements in solution quality.

We compare our framework with sequential solving and with parallel divide-and-conquer strategies employed in state-of-the-art MIP solvers. The parallel solvers are evaluated on core configurations of 8, 16, 32, 64, and 128 cores; to our knowledge, the 128-core configuration reported herein is the largest used to test parallel solvers on the full MIPLIB benchmark. In our experiments, each instance is solved to a relative gap of 0, consistent with the setting in the Hans Mittelmann's benchmark. Each solver is executed with a 300-second time limit per instance, and the total CPU time consumed by the competitive experiments exceeded 2.3 CPU years. To accelerate experimentation, we run one or more instances concurrently to fully utilize 128 cores; for example, we may run 16 instances on 8 threads each, 4 instances on 32 threads each, or a single instance on 128 threads.

Finally, our solver, evaluation scripts, related experimental results, and new best known solutions have been made available on GitHub [10]. Researchers interested in our work are encouraged to access our solver and explore the experimental details further.

[9] `https://satcompetition.github.io/`
[10] `https://github.com/shaowei-cai-group/PartiMIP`

5.2 Comparison to Parallel Divide-and-conquer Strategies

Table 1 Comparison to the parallel divide-and-conquer strategies in state-of-the-art open-source solvers, where "FiberSCIP_X" refers to the parallel version of SCIP implemented in FiberSCIP with X cores. "Parallel-HiGHS_X" stands for the parallel version of HiGHS with X cores. "PartiMIP-[base solver]_X" is the notation for employing base solver within our framework with X cores.

Solver	WIN	W-Imp.	FEAS	F-Imp.	SOLVED	S-Imp.	PAR-2	P-Imp.
FiberSCIP_8	129	0.0%	198	0.0%	79	0.0%	102421.1	0.0%
PartiMIP-SCIP_8	**159**	**23.3%**	**208**	**5.1%**	**81**	**2.5%**	**100615.9**	**1.8%**
FiberSCIP_16	126	0.0%	200	0.0%	83	0.0%	100803.4	0.0%
PartiMIP-SCIP_16	**163**	**29.4%**	**210**	**5.0%**	**86**	**3.6%**	**97747.0**	**3.0%**
FiberSCIP_32	125	0.0%	202	0.0%	87	0.0%	98630.5	0.0%
PartiMIP-SCIP_32	**168**	**34.4%**	**214**	**5.9%**	**88**	**1.1%**	**96887.0**	**1.8%**
FiberSCIP_64	128	0.0%	202	0.0%	93	0.0%	95876.1	0.0%
PartiMIP-SCIP_64	**167**	**30.5%**	**212**	**5.0%**	**94**	**1.1%**	**94113.6**	**1.8%**
FiberSCIP_128	120	0.0%	201	0.0%	92	0.0%	96415.2	0.0%
PartiMIP-SCIP_128	**168**	**40.0%**	**214**	**6.5%**	**98**	**6.5%**	**92223.4**	**4.3%**
Parallel-HiGHS_8	110	0.0%	192	0.0%	79	0.0%	101955.1	0.0%
PartiMIP-HiGHS_8	**179**	**62.7%**	**200**	**4.2%**	**89**	**12.7%**	**96903.0**	**5.0%**
Parallel-HiGHS_16	107	0.0%	192	0.0%	79	0.0%	101945.6	0.0%
PartiMIP-HiGHS_16	**184**	**72.0%**	**206**	**7.3%**	**89**	**12.7%**	**96480.1**	**5.4%**
Parallel-HiGHS_32	111	0.0%	192	0.0%	79	0.0%	101956.3	0.0%
PartiMIP-HiGHS_32	**186**	**67.6%**	**209**	**8.9%**	**96**	**21.5%**	**93368.3**	**8.4%**
Parallel-HiGHS_64	101	0.0%	192	0.0%	78	0.0%	102273.5	0.0%
PartiMIP-HiGHS_64	**190**	**88.1%**	**209**	**8.9%**	**97**	**24.4%**	**92603.9**	**9.5%**
Parallel-HiGHS_128	101	0.0%	192	0.0%	78	0.0%	102322.3	0.0%
PartiMIP-HiGHS_128	**190**	**88.1%**	**209**	**8.9%**	**100**	**28.2%**	**90516.2**	**11.5%**

We compare our PartiMIP framework with the default parallel divide-and-conquer strategies in state-of-the-art open-source solvers.

- **FiberSCIP** [39]: A famous parallel version of SCIP employing a normal ramp-up phase process, developed by the SCIP team. We use its latest version 1.0.0, which is based on SCIP 9.2.0 as its internal base solver.
- **Parallel HiGHS** (v1.9.0) [18]: The parallel version of HiGHS, featuring parallel dual simplex, symmetry detection, and clique table querying [11]. According to Hans-Mittermann's benchmark, the 8-core HiGHS is the top-ranked open-source solver in the MIP rankings.

Table 1 reports the performance metrics on the MIPLIB benchmark for various configurations. Our experimental results clearly demonstrate the advantages of the PartiMIP framework over conventional parallel strategies. For example, with an 8-core configuration, PartiMIP-SCIP achieves a 23.3% improvement in the WIN metric, a 5.1% increase in the FEAS metric, a 2.5% increase in the SOLVED metric, and a 1.8% reduction in the PAR-2 score compared to FiberSCIP. These performance gains not only persist but also improve with higher core counts; At 128 cores, PartiMIP-SCIP shows a 40.0% improvement in WIN and a 6.5% enhancement in SOLVED relative to FiberSCIP.

Similarly, PartiMIP-HiGHS consistently outperforms the parallel version of HiGHS across all configurations. With 8 cores, PartiMIP-HiGHS records a 62.7% improvement in WIN and a 12.7% in SOLVED. At 128 cores, it attains a 88.1% improvement in WIN and an 28.2% improvement in SOLVED compared to its default counterpart.

[11] `https://ergo-code.github.io/HiGHS/dev/parallel/`

Notably, the superior performance of PartiMIP is not confined to a single solver or configuration. Our framework consistently enhances all metrics, including the total number of solved instances (SOLVED), the quality of feasible solutions (WIN and FEAS), and overall runtime efficiency (PAR-2), across both SCIP and HiGHS. These findings robustly validate our dynamic task decomposition and scheduling strategies, demonstrating that PartiMIP not only improves the efficiency of parallel MIP solving but also facilitates a robust, scalable integration with a wide range of MIP solvers.

5.3 New Best Known Solutions to Open Instances

Table 2 PartiMIP establishes new best known solutions for 16 open instances.

Instance name	#Variable	#Constraint	Previous Best	PartiMIP
dlr1	9142907	1735470	2708148.95990256	2708064.1369803
neos-5151569-mologa	108116	45671	686759699	686750731.344582
bmocbd3	403771	152791	-372986719.737107	-373286017.205902
gmut-76-40	24338	2586	-14169441.78	-14169460.9675000
eva1aprime6x6opt	3514	34872	-16.31528287738903	-18.100995280293
dws012-02	51108	26382	122074.2013795086	121112.055928511
neos-4232544-orira	87060	180600	5557371.400000357	5553207.1245239
neos-4292145-piako	32950	75834	29160.50026450142	28122.4999807616
polygonpack5-15	48163	163429	-55494653.8357854	-55494686.5559904
sct5	37265	13304	-228.1172303718	-228.119492755556
cmflsp40-36-2-10	28152	4266	66452235.08297937	66452234.49456009
adult-regularized	32674	32709	7022.953543477999	7022.953543474559
supportcase23	24275	40502	-12160.6593559088	-12160.6593571676
neos-5045105-creuse	3848	252	20.57142909929996	20.5714105876044
gsvm2rl9	801	600	7438.181167768	7438.181021170049
s82	1690631	87878	-33.78523764658873	-33.7970576238223

In the MIPLIB dataset, open instances are those for which the optimal solution remains unproven. The current best known solutions for these instances are published on the MIPLIB website, and these open instances pose significant research challenges that drive progress in the field of MIP solving. Notably, PartiMIP has established new best known solutions for 16 open instances, and these solutions have been submitted and accepted by MIPLIB.

As shown in Table 2, these 16 instances vary widely in the number of variables and constraints, reflecting a wide range of problem structures. This diversity demonstrates the extensive applicability and robust solveability of our framework.

Overall, these experimental results validate the potential of large-scale parallel solving to overcome difficult MIP instances. By dynamically decomposing tasks and using parallel computation, PartiMIP achieves significant gains in both solution quality and efficiency, providing compelling evidence of its superior performance and extensive applicability on challenging optimization problems.

5.4 Comparison to Sequential Solving

Here, we evaluate the effectiveness of our dynamic task decomposition framework in enhancing and accelerating the solving capabilities of sequential solvers on the MIPLIB benchmark. Table 3 compares sequential solvers with their PartiMIP-enhanced counterparts across various core configurations, clearly demonstrating that our parallel framework consistently improves performance for both SCIP and HiGHS.

Table 3 Comparison to sequential solving in MIPLIB benchmarks. "[base solver]_Sequential" denotes sequential solving of the corresponding base solver, and solvers using our framework are labeled "PartiMIP-[base solver]" with the number of cores indicated.

Solver	WIN	W-Imp.	FEAS	F-Imp.	SOLVED	S-Imp.	PAR-2	P-Imp.
SCIP_Sequential	85	0.0%	198	0.0%	73	0.0%	105616.9	0.0%
PartiMIP-SCIP_8	110	29.4%	208	5.1%	81	11.0%	100615.9	4.7%
PartiMIP-SCIP_16	128	50.6%	210	6.1%	86	17.8%	97747.0	7.5%
PartiMIP-SCIP_32	136	60.0%	214	8.1%	88	20.5%	96887.0	8.3%
PartiMIP-SCIP_64	142	67.1%	212	7.1%	94	28.8%	94113.6	10.9%
PartiMIP-SCIP_128	149	75.3%	214	8.1%	98	34.2%	92223.4	12.7%
HiGHS_Sequential	91	0.0%	191	0.0%	76	0.0%	103461.3	0.0%
PartiMIP-HiGHS_8	108	18.7%	200	4.7%	89	17.1%	96903.0	6.3%
PartiMIP-HiGHS_16	118	29.7%	206	7.9%	89	17.1%	96480.2	6.7%
PartiMIP-HiGHS_32	120	31.9%	209	9.4%	96	26.3%	93368.3	9.8%
PartiMIP-HiGHS_64	138	51.6%	209	9.4%	97	27.6%	92603.9	10.5%
PartiMIP-HiGHS_128	148	62.6%	209	9.4%	100	31.6%	90516.2	12.5%

PartiMIP markedly improves solution quality, as evidenced by substantial improvements in the WIN metric for both solvers[12]. It also improves feasibility detection, demonstrating that our parallelized decomposition strategy facilitates the rapid identification of feasible solutions. Moreover, compared to their sequential counterparts, both PartiMIP-SCIP and PartiMIP-HiGHS achieve a notable increase in the total number of solved instances, accompanied by a consistent reduction in the PAR-2 score, indicating that PartiMIP accelerates the overall solving process.

Across all configurations, from 8 to 128 cores, PartiMIP consistently enhances all performance metrics, demonstrating its robust scalability in leveraging parallel computing resources. The consistent improvements observed across different base solvers and core configurations further validate the robustness and broad applicability of our approach.

5.5 Ablation Study

To evaluate the effectiveness of our proposed reward-guided multi-level selection heuristic (Algorithm 1), we conducted comparative experiments with a modified variant. Specifically, we modify PartiMIP by implementing random variable selection for partition tasks, yielding a modified version designated PartiMIP-R.

As shown in Table 4, PartiMIP consistently outperforms PartiMIP-R in almost all evaluation metrics. These results confirm that the reward-guided selection is effective and essential for improving variable selection during task decomposition.

6 Conclusions

In this work, we propose a novel parallel MIP solving framework with dynamic task decomposition. Our approach is built on two key ideas: a hardness estimate heuristic that identifies challenging subproblems and a reward decaying mechanism that reinforces decomposition decisions. Using our framework, we developed parallel solvers based on two leading open-source

[12] The WIN values here are computed by comparing solvers with the same base solver; thus WIN values are lower than those in Table 1, where only solvers with identical core counts are compared.

Table 4 Comparison between PartiMIP and its modified version PartiMIP-R.

Solver	WIN	W-Imp.	FEAS	F-Imp.	SOLVED	S-Imp.	PAR-2	P-Imp.
PartiMIP-R-SCIP_8	158	0.0%	203	0.0%	78	0.0%	102534.5	0.0%
PartiMIP-SCIP_8	**170**	**7.6%**	**208**	**2.5%**	**81**	**3.8%**	**100615.9**	**1.9%**
PartiMIP-R-SCIP_16	154	0.0%	208	0.0%	82	0.0%	101123.1	0.0%
PartiMIP-SCIP_16	**178**	**15.6%**	**210**	**1.0%**	**86**	**4.9%**	**97747.0**	**3.3%**
PartiMIP-R-SCIP_32	169	0.0%	212	0.0%	79	0.0%	101974.5	0.0%
PartiMIP-SCIP_32	**176**	**4.1%**	**214**	**0.9%**	**88**	**11.4%**	**96887.0**	**5.0%**
PartiMIP-R-SCIP_64	166	0.0%	**213**	0.0%	81	0.0%	101101.9	0.0%
PartiMIP-SCIP_64	**181**	**9.0%**	212	-0.5%	**94**	**16.0%**	**94113.6**	**6.9%**
PartiMIP-R-SCIP_128	162	0.0%	**215**	0.0%	86	0.0%	98563.1	0.0%
PartiMIP-SCIP_128	**181**	**11.7%**	214	-0.5%	**98**	**14.0%**	**92223.4**	**6.4%**
PartiMIP-R-HiGHS_8	154	0.0%	199	0.0%	86	0.0%	98939.8	0.0%
PartiMIP-HiGHS_8	**164**	**6.5%**	**200**	**0.5%**	**89**	**3.5%**	**96903.0**	**2.1%**
PartiMIP-R-HiGHS_16	148	0.0%	204	0.0%	83	0.0%	99885.4	0.0%
PartiMIP-HiGHS_16	**180**	**21.6%**	**206**	**1.0%**	**89**	**7.2%**	**96480.1**	**3.4%**
PartiMIP-R-HiGHS_32	162	0.0%	205	0.0%	86	0.0%	98660.3	0.0%
PartiMIP-HiGHS_32	**177**	**9.3%**	**209**	**2.0%**	**96**	**11.6%**	**93368.2**	**5.4%**
PartiMIP-R-HiGHS_64	155	0.0%	208	0.0%	87	0.0%	98003.5	0.0%
PartiMIP-HiGHS_64	**178**	**14.8%**	**209**	**0.5%**	**97**	**11.5%**	**92603.9**	**5.5%**
PartiMIP-R-HiGHS_128	151	0.0%	206	0.0%	89	0.0%	97166.4	0.0%
PartiMIP-HiGHS_128	**174**	**15.2%**	**209**	**1.5%**	**100**	**12.4%**	**90516.2**	**6.8%**

MIP solvers. Extensive experiments demonstrate that our parallel strategy outperforms the divide-and-conquer approaches employed by existing state-of-the-art MIP solvers. Moreover, our parallel solvers have established new best known solutions for 16 open MIPLIB instances.

For future work, our goal is to generalize our framework to address a broader range of constraint programming problems.

References

1 Tobias Achterberg. SCIP: solving constraint integer programs. *Math. Program. Comput.*, 1(1):1–41, 2009. doi:10.1007/S12532-008-0001-1.

2 Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In *Facets of combinatorial optimization: Festschrift for martin grötschel*, pages 449–481. Springer, 2013.

3 Timo Berthold, James Farmer, Stefan Heinz, and Michael Perregaard. Parallelization of the FICO xpress-optimizer. *Optim. Methods Softw.*, 33(3):518–529, 2018. doi:10.1080/10556788.2017.1333612.

4 Robert E. Bixby. Solving real-world linear programs: A decade and more of progress. *Oper. Res.*, 50(1):3–15, 2002. doi:10.1287/OPRE.50.1.3.17780.

5 Robert E. Bixby, William J. Cook, Alan Cox, and Eva K. Lee. Computational experience with parallel mixed integerprogramming in a distributed environment. *Ann. Oper. Res.*, 90:19–43, 1999. doi:10.1023/A\%3A1018960631213.

6 Suresh Bolusani, Mathieu Besanc$_c$on, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, et al. The scip optimization suite 9.0. *arXiv preprint arXiv:2402.17702*, 2024.

7 Michael R. Bussieck, Michael C. Ferris, and Alexander Meeraus. Grid-enabled optimization with GAMS. *INFORMS J. Comput.*, 21(3):349–362, 2009. doi:10.1287/IJOC.1090.0340.

8 Qun Chen and Michael C. Ferris. FATCOP: A fault tolerant condor-pvm mixed integer programming solver. *SIAM J. Optim.*, 11(4):1019–1036, 2001. doi:10.1137/S1052623499353911.

9 Qun Chen, Michael C. Ferris, and Jeff T. Linderoth. FATCOP 2.0: Advanced features in an opportunistic mixed integer programming solver. *Ann. Oper. Res.*, 103(1-4):17–32, 2001. doi:10.1023/A:1012982400848.

10 Zhihan Chen, Peng Lin, Hao Hu, and Shaowei Cai. Parls-pbo: A parallel local search solver for pseudo boolean optimization. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, volume 307 of *LIPIcs*, pages 5:1–5:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.5.

11 Jessica Davies and Fahiem Bacchus. Exploiting the power of mip solvers in maxsat. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing – SAT 2013 – 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2013. doi:10.1007/978-3-642-39071-5_13.

12 Jonathan Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM J. Optim.*, 4(4):794–814, 1994. doi:10.1137/0804046.

13 Christodoulos A. Floudas and Xiaoxia Lin. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Ann. Oper. Res.*, 139(1):131–162, 2005. doi:10.1007/S10479-005-3446-X.

14 Dongdong Ge, Qi Huangfu, Zizhuo Wang, Jian Wu, and Yinyu Ye. Cardinal optimizer (COPT) user guide. *CoRR*, abs/2208.14314, 2022. doi:10.48550/arXiv.2208.14314.

15 Ian P. Gent, Ian Miguel, Peter Nightingale, Ciaran McCreesh, Patrick Prosser, Neil C. A. Moore, and Chris Unsworth. A review of literature on parallel constraint solving. *Theory Pract. Log. Program.*, 18(5-6):725–758, 2018. doi:10.1017/S1471068418000340.

16 Ambros M. Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff T. Linderoth, Marco E. Lübbecke, Hans D. Mittelmann, Derya B. Özyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Math. Program. Comput.*, 13(3):443–490, 2021. doi:10.1007/S12532-020-00194-3.

17 LLC Gurobi Optimization. Gurobi optimizer ref. manual, 2024.

18 Qi Huangfu and J. A. J. Hall. Parallelizing the dual revised simplex method. *Math. Program. Comput.*, 10(1):119–142, 2018. doi:10.1007/S12532-017-0130-5.

19 Ravindran Kannan and Clyde L Monma. On the computational complexity of integer programming problems. In *Optimization and Operations Research: Proceedings of a Workshop Held at the University of Bonn, October 2–8, 1977*, pages 161–172. Springer, 1978.

20 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

21 Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 – From the Early Years to the State-of-the-Art*, pages 105–132. Springer, 2010. doi:10.1007/978-3-540-68279-0_5.

22 Peng Lin, Mengchuan Zou, Zhihan Chen, and Shaowei Cai. Parailp: A parallel local search framework for integer linear programming with cooperative evolution mechanism. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, pages 6949–6957. ijcai.org, 2024. URL: https://www.ijcai.org/proceedings/2024/768.

23 Andrea Lodi. Mixed integer programming computation. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 – From the Early Years to the State-of-the-Art*, pages 619–645. Springer, 2010. doi:10.1007/978-3-540-68279-0_16.

24 Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. Embarrassingly parallel search in constraint programming. *J. Artif. Intell. Res.*, 57:421–464, 2016. doi:10.1613/JAIR.5247.

25 Rafael A. Melo and Celso C. Ribeiro. MIP formulations for induced graph optimization problems: a tutorial. *Int. Trans. Oper. Res.*, 30(6):3159–3200, 2023. doi:10.1111/ITOR.13299.

26 Gautam Mitra, I. Hai, and M. T. Hajian. A distributed processing algorithm for solving integer programs using a cluster of workstations. *Parallel Comput.*, 23(6):733–753, 1997. doi:10.1016/S0167-8191(97)00016-1.

27 Hans Mittelmann. Latest benchmark results. In *Proceedings of the INFORMS Annual Conference, Phoenix, AZ, USA*, pages 4–7, 2018.

28 Lluís-Miquel Munguía, Geoffrey Oxberry, Deepak Rajan, and Yuji Shinano. Parallel PIPS-SBB: multi-level parallelism for stochastic mixed-integer programs. *Comput. Optim. Appl.*, 73(2):575–601, 2019. doi:10.1007/S10589-019-00074-0.

29 Stefan Nickel, Claudius Steinhardt, Hans Schlenker, and Wolfgang Burkart. Ibm ilog cplex optimization studio—a primer. In *Decision Optimization with IBM ILOG CPLEX Optimization Studio: A Hands-On Introduction to Modeling with the Optimization Programming Language (OPL)*, pages 9–21. Springer, 2022.

30 Yi-Shuai Niu, Yu You, and Wen-Zhuo Liu. Parallel DC cutting plane algorithms for mixed binary linear program. In Hoai An Le Thi, Hoai Minh Le, and Tao Pham Dinh, editors, *Optimization of Complex Systems: Theory, Models, Algorithms and Applications, WCGO 2019, World Congress on Global Optimization, Metz, France, 8-10 July, 2019*, volume 991 of *Advances in Intelligent Systems and Computing*, pages 330–340. Springer, 2019. doi:10.1007/978-3-030-21803-4_34.

31 Yves Pochet and Laurence A Wolsey. *Production planning by mixed integer programming*, volume 149. Springer, 2006.

32 Ted K. Ralphs, Laszlo Ladányi, and Matthew J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Math. Program.*, 98(1-3):253–280, 2003. doi:10.1007/S10107-003-0404-8.

33 Ted K. Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear optimization. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 283–336. Springer, 2018. doi:10.1007/978-3-319-63516-3_8.

34 Jean-Charles Régin and Arnaud Malapert. Parallel constraint programming. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 337–379. Springer, 2018. doi:10.1007/978-3-319-63516-3_9.

35 Martin W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *INFORMS J. Comput.*, 6(4):445–454, 1994. doi:10.1287/IJOC.6.4.445.

36 Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. Parascip: A parallel extension of SCIP. In Christian H. Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010 – Proceedings of an International Conference on Competence in High Performance Computing, Schloss Schwetzingen, Germany, June 2010*, pages 135–148. Springer, 2010. doi:10.1007/978-3-642-24025-6_12.

37 Yuji Shinano and Tetsuya Fujie. Paralex: A parallel extension for the CPLEX mixed integer optimizer. In Franck Cappello, Thomas Hérault, and Jack J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 – October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*, pages 97–106. Springer, 2007. doi:10.1007/978-3-540-75416-9_19.

38 Yuji Shinano, Tetsuya Fujie, and Yuusuke Kounoike. Effectiveness of parallelizing the ILOG-CPLEX mixed integer optimizer in the PUBB2 framework. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, volume 2790 of *Lecture Notes in Computer Science*, pages 451–460. Springer, 2003. doi:10.1007/978-3-540-45209-6_67.

39 Yuji Shinano, Stefan Heinz, Stefan Vigerske, and Michael Winkler. Fiberscip – A shared memory parallelization of SCIP. *INFORMS J. Comput.*, 30(1):11–30, 2018. doi:10.1287/IJOC.2017.0762.

40 Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 1999.

41 Y. Xu, Ted K. Ralphs, Laszlo Ladányi, and Matthew J. Saltzman. Computational experience with a software framework for parallel integer programming. *INFORMS J. Comput.*, 21(3):383–397, 2009. doi:10.1287/IJOC.1090.0347.