Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

YUJI SHINANO, TOBIAS ACHTERBERG, TIMO BERTHOLD, STEFAN HEINZ,
THOSTEN KOCH, MICHAEL WINKLER

Solving Previously Unsolved MIP Instances with ParaSCIP on Supercomputers by using up to 80,000 Cores

*Supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin

ZIB-Report 20-16 (May 2020)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

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

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

Abstract

Mixed-integer programming (MIP) problem is arguably among the hardest classes of optimization problems. This paper describes how we solved 21 previously unsolved MIP instances from the MIPLIB benchmark sets. To achieve these results we used an enhanced version of ParaSCIP, setting a new record for the largest scale MIP computation: up to 80,000 cores in parallel on the Titan supercomputer. In this paper, we describe the basic parallelization mechanism of ParaSCIP, improvements of the dynamic load balancing and novel techniques to exploit the power of parallelization for MIP solving. We give a detailed overview of computing times and statistics for solving open MIPLIB instances.

Solving Previously Unsolved MIP Instances with ParaSCIP on Supercomputers by using up to 80,000 Cores

Yuji Shinano, Tobias Achterberg, Timo Berthold,
Stefan Heinz, Thosten Koch, Michael Winkler

Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany,

Gurobi GmbH, Takustr. 7, 14195 Berlin, Germany,

Fair Isaac Germany GmbH, Takustr. 7, 14195 Berlin, Germany,

shinano@zib.de, achterberg@gurobi.com, TimoBerthold@fico.com,
heinz@gurobi.com, koch@zib.de, winkler@gurobi.com

May 25, 2020

1 Introduction

This paper deals with solving Mixed Integer Programming (MIP) problems in parallel. Throughout this paper we assume, without loss of generality, that a MIP is given in the following general form:

$$\begin{aligned} \min\{c^\top x : Ax \leq b, \\ l \leq x \leq u, x_j \in \mathbb{Z}, \text{ for all } j \in I\}, \end{aligned} \tag{1}$$

with matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$ and $c, l, u \in \mathbb{R}^n$, and a subset $I \subseteq \{1, \ldots, n\}$.

MIP is a very important problem class. Many optimization problems arising in practice can be modeled as a MIP, see, e.g., [1]. Well-established standards for data formats have made it possible to collect a variety of real-world problem instances and make them publicly available in problem libraries, such as MIPLIB. The first version of MIPLIB was created in 1992 [2]. Its latest release is MIPLIB2017[3]. These libraries are key to the evolution of the MIP field of research because new ideas and algorithms can be evaluated against data sets arising from real-world problems. Moreover, researchers can directly compare their results to previous studies, and unsolved models from these libraries provide research challenges to advance the field. MIPLIB has always motivated researchers to compete for solving its challenge instances by utilizing the latest and fastest algorithms and hardware or, ideally, a combination of both.

Supercomputers with more than 10,000 cores first appeared on the Top500 supercomputer list[1] in November 2004. After June 2018, this list has contained no entry that has less than 10,000 cores. When utilizing such a huge amount of computing resources, we expect to obtain valuable and tangible results from the computations on them. This leads to the aim of this research: to solve previously intractable MIP instances on supercomputers.

State-of-the-art MIP solvers are based on the branch-and-cut paradigm, which is a mathematically supercharged mixture of a branch-and-bound tree search combined with a cutting plane approach, employing a large number of sophisticated algorithms to keep the enumeration effort small. This includes a large number of heuristic methods to devise primal feasible solutions, and many cutting plane separation algorithms to increase the lower bound value obtained by the Linear Programming (LP) relaxation, see, e.g., [1].

Tree search algorithms are generally considered easy to parallelize. However, to the best of our knowledge, there have been only two implementations of a large scale parallelized MIP solver that succeeded in solving open benchmarking instances. One is GAMS/CPLEX/Condor by Bussieck et al. [4] who solved three instances from MIPLIB2003 by a GRID computing approach. The other is ParaSCIP [5], extensions of which are presented in this paper. Both solvers use a state-of-the-art MIP solver as a black box to exploit the latest MIP solving technology; the tree search based solving process is parallelized externally. For a recent survey about parallel MIP solving see [6].

In this paper, we first briefly introduce ParaSCIP and explain its parallelization features. Next, we describe new techniques that we developed recently to tackle some of the hardest MIPLIB challenge instances. Finally, we show computational results for solving open instances over a period of seven years by using up to 80,000 cores.

2 ParaSCIP – A Distributed Memory MIP Solver

ParaSCIP has been developed by using the *Ubiquity Generator (UG) framework* [7, 8]. Figure 1 shows the design structure of UG. UG is written in C++. It consists of a set of base classes to instantiate parallel branch-and-bound based solvers. The solver and the parallelization library used for communications are abstract classes. The branch-and-bound based solver is treated as a black box, i.e., UG can be used with different state-of-the-art MIP solvers. As a consequence, improvements of the basic solver technology can immediately be utilized in the parallel case. Also, the parallelization library can be exchanged, which makes the parallel solver more portable. ParaSCIP is an instantiated parallel solver where SCIP is used as the black box MIP solver and MPI is used as the parallelization library.

As shown in Figure 2, two types of processes exist when running ParaSCIP on a supercomputer. There is a single LoadCoordinator (abbreviated to LC throughout of this paper), which makes all decisions concerning the dynamic load balancing and distributes subproblems of MIP instances to be solved (so-called sub-MIPs). All other processes are Solvers that solve the distributed sub-MIPs.

[1] http://www.top500.org

In general, a parallel branch-and-bound algorithm consists of three phases [6]. The *ramp-up phase* is the phase that lasts until all solvers have become busy. A natural way to distribute sub-MIPs to all solvers in the ParaSCIP configuration is the following. SOLVERs that are already solving a sub-MIP transfer every second child node back to the LC. The LC maintains a *node pool* from which it assigns nodes to idle SOLVERs. If no idle SOLVER exists, the LC keeps collecting nodes from SOLVERs until it has p "heavy" (promising to have a large subtree underneath see 3.1.1 for a definition) unassigned nodes in its node pool. Here, p is a run-time parameter, which is set to a value between 10 and 2,000 in our experiments. As soon as the LC's node pool has accumulated p "heavy" nodes, it sends a message to all SOLVERs to stop sending nodes. This is called *normal ramp-up*.

The second phase is the *primary phase*, during which the algorithm operates in steady state. This state is followed by the *ramp-down phase*, at which there is not enough work left to keep all SOLVERs busy and during which termination procedures are executed. In all these phases, the key is how to realize the dynamic load balancing. A *supervisor-worker coordination mechanism* is used to realize the dynamic load balancing in ParaSCIP, which is presented in the next section. A detailed description of how ParaSCIP works can be found in [5] and [7].

3 The key techniques

For tackling unsolved MIP instances, one should not rely on parallelization alone, even if it is on a supercomputer level. Since MIP is an NP-hard problem, a linear speed-up cannot help to improve solvability. The success of ParaSCIP for solving previously unsolvable instances relies on two factors: One is that the supervisor-worker coordination mechanism has been tuned to work up to 80,000 SOLVERs keeping search direction properly. It can last until the end of computation with an application oriented light weight check-pointing and restarting mechanism, even though all jobs on supercomputers always have a time limit. The coordination mechanism is described in this section. More importantly, the other factor is that a parallel solver instantiated by using UG causes algorithmic changes to that of the base solver, that is, ParaSCIP works algorithmically differently from SCIP. As prime examples, *layered presolving* and *racing ramp-up* are presented in this section.

3.1 Supervisor-Worker coordination mechanism

ParaSCIP realizes a parallelization of MIP solvers for a distributed memory computing environment without a centralized global search tree data structure. An idea of Supervisor-Worker is that the Supervisor functions only to coordinate workload, but does not actually store the data associated with the whole search tree. In ParaSCIP, LC represents the Supervisor and the SOLVERs represent the Workers. Each node transferred through the system—called a PARANODE—acts as the root of a subtree. The information sent to a PARANODE only consists of variable bound changes. The SOLVER that receives a new branch-and-bound node instantiates the corresponding sub-MIP using the instance that

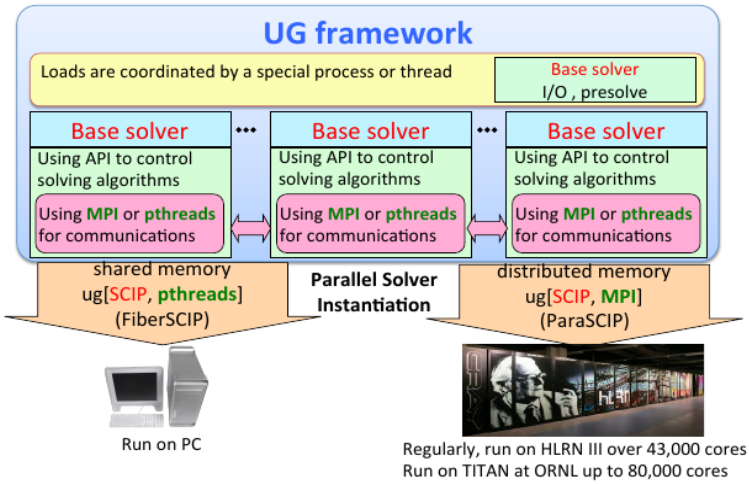

Figure 1: Design structure of Ubiquity Generator Framework.

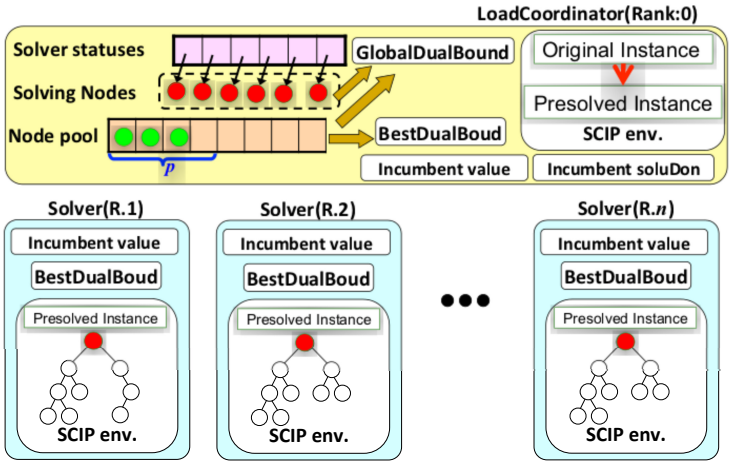

Figure 2: Process composition and data arrangement of ParaSCIP.

was distributed in the initialization step (see also 3.2), and the received bound changes. Therefore, a ParaNode is considered as a representation of a sub-MIP in ParaSCIP. The terminal nodes of the search tree in the Solvers are collected on demand and a set ParaNode in the LC works as a buffer to ensure sub-MIPs are available to idle Solvers as needed. Algorithms 1 and 2 show a parallel algorithm with a simplified Supervisor-Worker coordination mechanism.

In UG, the load balancing is accomplished mainly by toggling the collection mode flag in the Solver. Turning collecting mode on results in additional "heavy" sub-MIPs being sent to the LC as ParaNodes, which can then be distributed to Solvers. Naturally, the method of selecting which Solver to collect from is crucial to the effectiveness of the approach. In the following subsections, detailed dynamic load balancing features implemented in ParaSCIP are presented.

Algorithm 1 Supervisor

Input: Single MIP solver, set of N processors $i \in S = \{1, \ldots, N\}$ and an MIP instance to be solved
Output: An optimal solution

```
Spawn N Workers with the MIP solver on processors 1 to N
collectMode ← false
x* ← NULL
I ← N \ {1}
A ← {1}
Q ← ∅
R ← {(1, 0)} // Subproblems currently being processed, 0 is the index of the root problem
Send the root problem to processor 1
while Q ≠ ∅ and R ≠ ∅ do
  (i, tag) ← Wait for message // Returns processor identifier and message tag
  if tag = solutionFound then
    Receive solution x̂ from processor i
    if x* = NULL or c^⊤ x̂ < c^⊤ x* then
      x* ← x̂
    end if
  else if tag = subproblem then
    Receive a subproblem indexed by k from processor i
    Q ← Q ∪ {k}
  else if tag = terminated then
    R ← R \ {(i, j)} // j is the index of the terminated subproblem
    A ← A \ {i}, I ← I ∪ {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 processors in collecting mode.
        collectMode ← false
      end if
    else
      // collectMode = false
      if there are not enough heavy subproblems in Q then
        Select processors which have heavy subproblems
        Send message with tag = startCollecting to the selected processors
        collectMode ← true
      end if
    end if
  end if
  while I ≠ ∅ and Q ≠ ∅ do
    i ∈ I, I ← I \ {i}, A ← A ∪ {i}
    subproblem j ∈ Q, Q ← Q \ {j}, R ← R ∪ {(i, j)}
    Send subproblem j and x* to processor i
  end while
end while
∀i ∈ S : Send message with tag = termination to processor i
Output x*
```

Algorithm 2 Worker

Input: An MIP solver and an original MIP instance to be solved

```
collectMode ← false
terminate ← false
while terminate = false do
  (i, tag) ← Wait for message from Supervisor // Returns Supervisor identifier 0 and message tag
  if tag = subproblem then
    Receive subproblem and solution from Supervisor
    Solve the subproblem, periodically communicating with supervisor as follows
     - Send message with tag solutionFound anytime 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 ← true
  end if
end while
```

3.1.1 Trigger of the toggling collecting mode

The LC aims to keep at least p "heavy" nodes in the node pool Q in Algorithm 1. We call a node *heavy* if the lower bound value of its subtree (NodeBound) is sufficiently close to the lower bound value of the overall search tree (GlobalBound). This is evaluated by the expression

$$\frac{\textsc{NodeBound} - \textsc{GlobalBound}}{\max\{|\textsc{GlobalBound}|, 1.0\}} < \textsc{Threshold}. \tag{2}$$

If a Solver receives the message to switch into the collecting mode, it changes the search strategy to "best bound order" (see [9]). Similar to normal ramp-up, the Solver alternates between solving nodes and transferring them to the LC.

Solvers switch to collecting mode in ascending order of the minimum lower bound of their open nodes. The collecting mode is stopped as soon as the number of heavy nodes in the pool is larger than $m_p \cdot p$ (1.5 is used for m_p usually). When the collecting mode has been stopped, the search strategy of the Solver is changed back to its original one.

3.1.2 Dynamic tuning of parameters and bulk sending of ParaNodes

The frequency in which a Solver sends ParaNodes to the LC depends not only on the computing environment but also on the instance to be solved. The processing of a node involves the execution of different algorithms such as node preprocessing or LP solving that may have significant runtimes. At some dedicated nodes, expensive primal heuristics or cutting plane separation loops might be run. The time spent for an individual node can range between a fraction of a second and several minutes, even within the same MIP

instance. This time is difficult to estimate in advance. Therefore, the number of SOLVERs that can be in collecting mode at a certain point needs to be adjusted dynamically to reduce the idle time ratio.

Sending PARANODEs, i. e., sub-MIPs, from a subtree to other SOLVERs means that parts of the subtree will be explored more aggressively by using the other SOLVERs. It is beneficial to keep the number of SOLVERs in collecting mode small at any point in time, since this will focus the tree exploration on the hard part of the search tree, compare [4]. On the other hand, it is necessary that enough SOLVERs are in collecting mode in order to collect enough PARANODEs to keep all SOLVERs busy. Therefore, the number of SOLVERs that can be in collecting mode at a point of time is restricted to one at the beginning of the computation and is increased by one whenever the node pool in the LC has stayed empty for a period of time as specified by a run-time parameter (the default setting is ten seconds). The value p is also changed dynamically. It is not only increased, but also decreased depending on how fast the LC switches into collecting mode. In the default setting, if the interval time between collecting modes is less than ten seconds, the p value is doubled. This helps to keep the number of collecting mode SOLVERs small without increasing the idle times.

The synchronization protocol between a SOLVER and the LC renders sending individual PARANODEs comparatively slow. To avoid this, we implemented a fast bulk sending mechanism. The message that requests a SOLVER to switch into collecting mode additionally includes the number of PARANODEs expected to be sent from the SOLVER. That is, when the LC switches into collecting mode it determines how many PARANODEs are to be collected from which SOLVERs by using information from the SOLVER's status messages. If a SOLVER has sufficiently many open nodes, it sends exactly the number of PARANODEs specified by the LC without synchronization in between. If a SOLVER does not have enough open nodes, it sends as many as possible by bulk. Afterwards, it switches to the normal PARANODE sending mechanism.

3.1.3 Special treatment of the ramp-down

The *ramp-down* phase is reached at the end of the computation when it becomes difficult to keep all SOLVERs busy. Typically, at the end of the computation only a few SOLVERs have a significant search tree remaining. At the same time, most of the PARANODEs are solved extremely fast and the SOLVERs send their completion messages to the LC. In the worst case, this can lead to a congestion in the communication network and it may even prevent the LC from collecting PARANODEs. When the LC recognizes such a strongly imbalanced situation, it changes the PARANODE sending mechanism such that

1. it solely collects PARANODEs without redistributing them until a sufficient number had been collected,

2. afterwards, the collected PARANODEs are redistributed to idle SOLVERs.

This change is triggered if for a period of time as specified by a run-time parameter (10 seconds is specified in our experiments) less than 90% of the solvers are active and the

number of open nodes within the SOLVERs exceeds the number of SOLVERs by more than a factor of 100.

3.1.4 Restarting the collecting mode

ParaSCIP can control how frequently the SOLVER statuses are updated by using the *notification interval time* parameter. This parameter indicates the interval of time between status messages from a SOLVER. Each message contains very little data, but all SOLVERs send these messages periodically. When supercomputers with huge amounts of parallel cores are used, this communication eventually becomes a bottleneck and a longer updating interval is required. As a consequence, the LC schedule is based on slightly outdated information. If this leads to the node pool running empty, the collecting mode is restarted immediately. When the number of collecting mode SOLVERs reaches its limit (normally this situation occurs in the ramp-down phase), restarting the collecting mode is the only way to accelerate the collection of PARANODEs. In ramp-down, it occurs frequently that the collecting mode is restarted several times in a row.

3.1.5 Branch node selection in the collecting mode Solver

In SCIP it is possible to customize the node selection strategy by adding a node selector plugin. We implemented a node selector that is designed to select nodes that are expected to have a large search tree underneath. This special node selector is used while a SOLVER is in collecting mode. In the node selector for the collecting mode, a node is selected by the best (i.e., lowest) lower bound as aforementioned, with a lower number of variable bound changes as a tie breaker. The number of bound changes is a rough estimate of the volume of the feasible region for a sub-MIP. The PARANODE with the largest feasible region is transferred.

3.1.6 Checkpointing and restarting

ParaSCIP implements a checkpointing mechanism to write out an intermediate search state in order to restart the parallel search procedure from that state. Therefore, ParaSCIP saves only *primitive* nodes, which are nodes that have no ancestor nodes in the LC. This strategy requires much less effort for the I/O system than to save all open nodes to a disk, in particular in large scale parallel computing environments, but potentially creates a computational overhead after the restart. However, the effort to regenerate the search tree is often outweighed by the benefits of re-applying a global presolving procedure during the restart (see [10]).

A neat feature of our checkpointing mechanism is that the number of nodes in the checkpoint file is bounded by

$$\max\{\#\text{ of open nodes after racing stage},\ \#\text{ of SOLVERs} + p + \alpha\}.$$

Here, α is the number of nodes sent from the SOLVERs after the LC sent a message to stop sending nodes to all SOLVERs owing to buffer messages.

The restart involves ParaSCIP reading the nodes saved in the checkpoint file and restoring them into the node pool of the LC. The LC subsequently distributes these nodes to the SOLVERs in an order determined by their lower bounds.

3.2 Layered presolving

In current state-of-the-art MIP solvers, the instance being solved is not the original one, but a reformulated presolved one and the presolving technique has a big impact on solving it, see, e.g., [11] or [9] for an overview on MIP presolving techniques. Since in ParaSCIP, eachPARANODE is presolved from scratch, it holds that more SOLVERs being employed will lead to more frequent in-tree presolving.

3.2.1 Initial presolving in LC

At the initialization, the LC reads the instance data of the MIP to solve. We refer to the resulting instance as the *original instance*. This instance is presolved directly inside the LC, We call the resulting instance the *presolved instance*. The presolved instance is broadcasted to all available SOLVER processes only once, and is embedded into the (local) SCIP environment of each SOLVER. During the solving process, PARANODEs, which only contain differences between a sub-MIP and the presolved instance, are communicated. This means that a feasible solution found by a SOLVER needs to be transformed back to the original instance. This is done by using the SCIP environment in the LC.

The number of variables and constraints reduced in the presolving depends on the SCIP version used. The Online supplement of [7] shows original instances and the presolved ones generated by SCIP version 2.1.1 for all instances from the MIPLIB 2010 benchmark set.

3.2.2 Recursive application of presolving

ParaSCIP applies presolving when a newly received sub-MIP is started to be solved. When a PARANODE is retrieved, that is, a set of bound changes for variables is applied to the SCIP environment in a SOLVER, the sub-MIP will be presolved from scratch. This recursive presolving does not occur in SCIP. This is one of several examples how parallelization scale can introduce more algorithmic changes to SCIP. In current state-of-the-art MIP solvers, cuts are applied more aggressively at the root node after applying presolving.

The effectiveness of recursive application of presolving depends on the instances to be solved. Small scale, intensive computational experiments were conducted for the MIPLIB 2010 benchmark instance set by using FiberSCIP, which is a shared memory version of ParaSCIP, in [7]. The paper shows that the number of nodes solved decreases when adding more SOLVERs on average. This is not the case, in general, for naive implementations of parallel branch-and-bound. The instances for which we could expect recursive presolving to work well are the ones that we consider in the computational section of this paper.

3.3 Racing ramp-up

In this mechanism, after initialization, the LC sends the root branch-and-bound node to all SOLVERs simultaneously and each SOLVER starts solving the root node of the presolved instance immediately. In order to generate different search trees, even though they work on the same problem, each SOLVER uses different parameter settings and permutations of variables and constraints. As shown in [12], the latter can have a considerable impact on the performance of a solver due to imperfect tie breaking. Due to these variations, we expect that the SOLVERs will generate vastly different search trees. After a specified amount of time, one SOLVER is chosen as the "winner" of this *racing stage*. The winning criterion is a combination of the lower bound and the number of open nodes of the sub-MIP. All open nodes of the "winner" are then collected by the LC and a termination message is sent to all other SOLVERs. The search trees of the other SOLVERs are discarded. Only the feasible solutions found during their solving process are kept. The collected nodes are then redistributed to the now idle SOLVERs. Once a "winner" is selected that provides less nodes than the number of available SOLVERs, ParaSCIP performs normal ramp-up.

4 Node merging and deep probing

This section introduces novel techniques for parallel MIP search that helped us tackle unsolved instances and improve the solving time on hard instances.

4.1 Merging ParaNodes at restart

The choice of the branching variables has a big impact on MIP search, see [14]. This holds in particular for branchings that are performed early in the branch-and-bound process. In MIP, branching decisions are typically based on statistical information derived from previous branchings, so-called *pseudo-costs*. These pseudo-cost statistics are often weak at the beginning of the search. Therefore, it seems beneficial to try to correct "bad" branching decisions later on. On supercomputers, usually a hard time limit for every computation is imposed and we often need to restart the whole solution process multiple times from a checkpoint file when solving very challenging MIP instances. The restart is a natural point to re-organize the branch-and-bound tree by using the branching statistics stored in the checkpoint file. In this subsection, we present an algorithm to merge PARANODEs (from a checkpoint file) at a restart to re-arrange the search tree.

Let

$$\begin{aligned}\text{sub-MIP}_i := \min\{c^\top x : Ax \le b,\\ l^i \le x \le u^i, x_j \in \mathbb{Z} \text{ for all } j \in I\}\end{aligned}$$

be the sub-MIP with local bounds l^i and u^i corresponding to PARANODE i. Let O be the set of such sub-MIPs that corresponds to the set of open PARANODEs, and let $M \subseteq O$ be some subset of these nodes. For a given $j \in I$ and $v \in \mathbb{Z}$ let $S_j^v(M) := \{i \in M : l_j^i = u_j^i = v\}$

be the set of sub-MIPs in M that share the same fixing of integer variable x_j to value v. For a given set of sub-MIPs $\breve{M} \subset M$ we define a *merged sub-MIP* as:

$$\begin{aligned}\text{sub-MIP}_{\breve{M}} := \min\{c^\top x : Ax \leq b,\\ l^{\breve{M}} \leq x \leq u^{\breve{M}}, x_j \in \mathbb{Z} \text{ for all } j \in I\}\end{aligned}$$

with bounds $l_j^{\breve{M}} := \min_{i \in \breve{M}}\{l_j^i\}$ and $u_j^{\breve{M}} := \max_{i \in \breve{M}}\{u_j^i\}$ for all $j \in \{1, \ldots, n\}$. Merging ParaNodes is performed by Algorithm 3 and Algorithm 4.

Algorithm 3 Solve all open sub-MIPs with merging

Input: O
Output: Solve all sub-MIPs in O

```
M ← O
C ← Algorithm 4(M)
T ← C
while T ≠ ∅ do
   // This loop can be performed in parallel
   Select P̂_i ∈ T
   T ← T \ {P̂_i}
   if P̂_i is a merged-node then
      // P̈_i := P̂_i
      // P_i := original sub-MIP of P̈_i
      Perform root node procedure for P̈_i
      if lower bound of P̈_i <
         (lower bound of P_i) · (1 − δ) then
         // δ is a parameter: 0(current default)
         Recover a set of sub-MIPs M from P̈_i
         M ← M \ {P_i}
         C ← Algorithm 4(M)
         T ← T ∪ {P_i} ∪ C
      else
         Keep solving P̈_i(*)
      end if
   else // P_i := P̂_i
      Solve P_i(*)
   end if
end while
```

For the merging of nodes, similar considerations hold as for checkpointing by storing primitive nodes. It potentially loses information because it relaxes already fixed variables. Also, merging is likely to worsen the lower bound of the corresponding sub-MIPs. However, since a merged ParaNode will be solved like a stand-alone problem, namely from scratch by use of the full power of presolving and cutting planes, the lower bound can even improve. This is taken into account during the merging procedure: merging will not be performed if the lower bound decreases too much, see Algorithm 3.

Our empirical observations indicate that the main advantage of merging is a more balanced rearranged tree. Also, in our experiments we noticed that merged nodes increase the chance of finding better solutions earlier in the search.

Algorithm 4 Generate merge-nodes candidate set

Input: $M \subset O$
Output: C // C is merge-nodes candidate set
  $C \leftarrow \emptyset$
  while $M \neq \emptyset$ **do**
    Select $P \in M$ s.t. P is a sub-MIP having the best lower bound in M
    $\breve{M} \leftarrow M$
    $\breve{J} \leftarrow \emptyset$
    $n \leftarrow 0$
    while $\max_{j \in I \setminus \breve{J}, v \in \mathbb{Z}, P \in S_j^v(\breve{M})} |S_j^v(\breve{M})| \geq 2$ **do**
      // At least two nodes can be merged
      $(\hat{j}, \hat{v}) = \underset{j \in I \setminus \breve{J}, v \in \mathbb{Z}, P \in S_j^v(\breve{M})}{\arg\max} |S_j^v(\breve{M})|$
      $\breve{M} \leftarrow S_{\hat{j}}^{\hat{v}}(\breve{M})$ // Note: $P \in \breve{M}$
      $\breve{J} \leftarrow \breve{J} \cup \{\hat{j}\}$
      $n \leftarrow n + 1$
    end while
    if $\frac{n}{|\{j | l_j = u_j \text{ in sub-MIP} P\}|} > \tau$ **then**
      // τ is a parameter: 0.9 (current default)
      Create a merged sub-MIP $P_{\breve{M}}$ from $\breve{M}$
      $C \leftarrow C \cup \{P_{\breve{M}}\}$
      $M \leftarrow M \setminus \breve{M}$
    else
      $C \leftarrow C \cup \{P\}$
      $M \leftarrow M \setminus \{P\}$
    end if
  end while

The current ParaSCIP version also provides a feature to perform the merge procedure off-line (i.e., on a desktop machine in between two supercomputer runs) and to update the checkpoint file accordingly. We are currently investigating under which conditions automatically enforcing such restart might be overall beneficial to the solution process.

4.2 Deep probing

In SCIP, for each variable several statistics are stored that have been collected during the solution process. In particular, branching statistics are used to select a branching variable. The racing stage of racing ramp-up is a good opportunity to tentatively collect these variable statistics for different possible search trees for the MIP instance at hand. This means the LC collects all branching statistics not only from the racing winner, but also from all SOLVERs that participated in racing. This information is then aggregated and used to initialize the branching statistics of all SOLVERs after racing, compare [15]. We refer to this strategy as *deep probing*, since it resembles the ideas of probing and strong branching, with the difference that instead of single nodes whole subtrees are explored tentatively. We expect that initializing branching statistics will help to improve branching decisions and decrease the likelihood of "bad" initial branchings. The effect of deep probing is not yet fully investigated, but our experiments so far have been promising. In order to use deep probing, all PARANODEs need to store (and communicate) this information. Hence, the PARANODE data size increases. Therefore, this technique is best suited for medium scale computing environments and for MIP instances that contain relatively few integer variables.

5 Computational results

In this section, we report on open instances that were solved for the first time by using ParaSCIP. Furthermore, we present the biggest and the longest MIP computation conducted so far, and also summarize the status of open instances from MIPLIB 2003/MIPLIB 2010/MIPLIB 2017 until the end of 2019.

5.1 Open instances solved by ParaSCIP

In 2009, six problem instances of MIPLIB2003 were still unsolved. In April 2010, `ds` and `stp3d` were solved by ParaSCIP, see [5, 10]; the remaining four instances are still open (see [13]). Figure 3 shows the status of MIPLIB 2003 until 2011. In the meantime, MIPLIB2010 [12] has been published, the original paper listed 134 unsolved instances. In the following, we present details of our ParaSCIP runs that solved 12 of these formerly unsolved instances to proven optimality for the first time.

Table 1 gives a short overview on how the instances were solved. In the Table, "Rows" and "Cols" show m and n of the matrix A in (1), and "Int", "Bin" and "Con" show the number of general integer, binary, and continuous variables, respectively. For each instance,

Table 1: Open instances from MIPLIB2010 solved by ParaSCIP

Date	Name	Rows	Cols	Int	Bin	Con	SCIP	CPLEX	Computer	Runs	Cores	Time(h.)	Optimal value
March 2011	rmatr200-p20	29406	29605		200	29405	2.0.1	12.2	Alibaba	1	160	2	837
March 2011	50v-10	233	2013	183	1464	366	2.0.1	12.2	HLRN II	1	1024	5	3313.18
March 2011	probportfolio	302	320		300	20	2.0.1	12.2	HLRN II	1	1024	12	16.7342
									HLRN II	2	2048	36	
March 2011	reblock354	19906	3540		3540		2.0.1	12.2	HLRN II	2	1024	24	-39280521.2281657
									HLRN II	3	2048	209	
Jun 2012	dg012142	6310	2080		640	1440	2.1.1	12.4	ISM	1	256	42	2300867
July 2012	dc1c	1649	10039		8380	1659	2.1.1	12.4	ISM	8	256	400	1767903.6501
									ISM	7	512	700	
August 2012	germany50-DBM	2526	8189	88		8101	2.1.1	12.4	ISM	15	256	590	473840
March 2014	dolom1*	1803	11612		9720	1892	3.0.1	12.5	HLRN III	2	12288	16	6609253
January 2015	set3-10	3747	4019		1424	2595	3.1.1	12.6	HLRN III	3	6144	33	185179.043049708
									HLRN III	2	3072	24	
January 2015	set3-20	3747	4019		1424	2595	3.1.1	12.6	HLRN III	1	6144	12	159462.572721458
									HLRN III	3	3072	36	
November 2015	triptim3	14939	28440	6812	21621	7	3.1.2	12.6	ISM2	1	864	10	13.5311
December 2015	rmine10	65274	8439		8439		3.1.2	12.6	HLRN III	6	6144	54	-1913.88062
									HLRN III	10	12288	112	
									HLRN III	4	18432	85	
									TITAN	1	80000	13	
									HLRN III	2	18432	48	
									HLRN III	1	36000	27	
									HLRN III	9	43200	356	
									HLRN III	11	43344	477	
									HLRN III	3	12000(HT)	611	
									HLRN III	1	6168(HT)	4	

Table 2: Open instances from MIPLIB2017 solved by ParaSCIP (72 Cores of ISM3 were used)

Name	Time to first sol.(sec.)	Value of first sol.	Time to solve(sec.)	Optimal value
fhnw-sq2	46,863	0	46,883	0
neos-4409277-trave	181,150	8	441,575	3
supportcase3	1,539	0	1,551	0
woodlands09	218,528	8	245,205	0
neos-5251015-ogosta	523,602	0.1058	525,353	0.1058
neos-3211096-shag	-	-	42,484	Infeasible
neos-3631363-vilnia	-	-	4,632	Infeasible

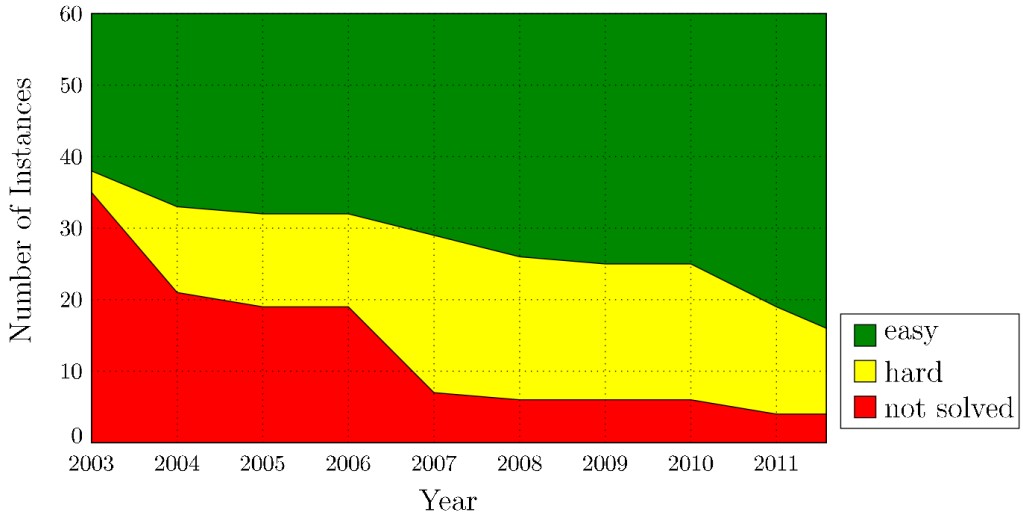

Figure 3: Comparison of the number of solved MIPLIB 2003 instances at the beginning of each year. 'Easy' means, that the instance could be solved within one hour using a commercial MIP-solver, 'hard' stands for instances that were solved, but not in the previous conditions.

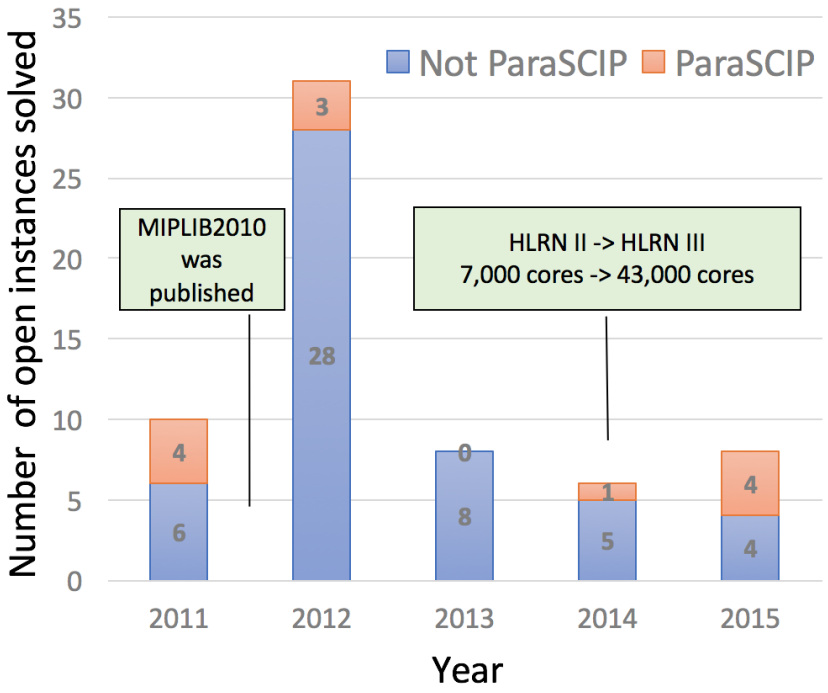

Figure 4: The number of open instances in MIPLIB2010 solved by ParaSCIP and the others

the date of solving, the SCIP and the CPLEX version (the latter was used as an LP solver in SCIP), the supercomputer(s) that we used, and the optimal solution value are presented. The number of runs performed to prove optimality indicates if and how often we restarted the computation from a checkpoint file, with 1 meaning that the initial run without restart was already able to solve the problem instance. In the table, "Alibaba" is a PC cluster with 40 PowerEdgeTM 2950 computers connected by Infiniband, each equipped with two Quad-Core Xeon E5420 CPUs at 2.5 GHz and 16 GB RAM, "HLRN II" is an SGI Altix ICE 8200EX (Xeon QC E5472 3.0 GHz/X5570 2.93 GHz), "HLRN III" is a Cray XC30 (Intel Xeon E5-2695v2 12C 2.400GHz, Aries interconnect), "ISM" is the ISM supercomputer Fujitsu PRIMERGY RX200S5. "ISM2" is the updated ISM supercomputer SGI ICE X (Intel Xeon E5-2697v2 2.7GHz 2CPU, 128GB) 400nodes Total: 207TFLOPS, 50TB. The "(TH)" shown in the Cores column means that we used hyper-threading and the number of processes on the computing nodes are double the number of cores. The computing time shows an accumulated approximate computing time for the number of runs executed with the same number of cores.

The results for solving the four instances `rmatr200-p20`, `50v-10`, `probportfolio` and `reblock354` can already be found in the MIPLIB2010 paper [12]. For these experiments, we initialized the search with the best known solutions. All other instances were solved from scratch. All instances that are solved with more than one run are restarted from the checkpoint file of its previous run, except `dolom1`. For `dolom1`, the second run was solved without a checkpoint file, while we used the incumbent solution of the first run as an initial solution.

Figure 4 shows the status of MIPLIB 2010 and the contribution of ParaSCIP for solving open instances summarized until 2015. Right after MIPLIB 2010 was published, many open instances were solved by mainly commercial MIP solvers. However, the number of open instances solved by the commercial MIP solvers has constantly decreased year by year. However, updating the HLRN II to HLRN III combining with algorithmic improvement of SCIP, ParaSCIP has solved more instances from the previous years. It is not only its scale, but the algorithmic improvement of SCIP combined with algorithmic changes caused by UG would have had more impact to solve the instances.

Before publishing MIPLIB 2017 [3], we tried to solve open instances that did not have any feasible solutions during the instances selection processes of MIPLIB 2017. Table 2 shows the ones solved by using ParaSCIP for the first time. The "ISM3" is the updated ISM supercomputer HPE SGI 8600 (Intel Xeon Gold 6154 18 cores 3.0GHz 2CPU, 384GB) 376 nodes.

An important point is that ParaSCIP has kept solving open instances.

5.2 The biggest and the longest computation

The biggest and the longest computation used to solve a single instance is for solving the open instance `rmine10` by using the supercomputers HLRN III and Titan. 48 jobs were executed with the previous checkpoint files and it took about 75 days and about 5660 years of CPU core time in total. For runs on HLRN III, the number of SOLVERs used is,

in many cases, the number of cores minus 1. That is, one for the LC except runs using hyper-threading, in which the number of Solvers used double the number of cores. For the Titan run, we used one computing node dedicated to the LC process and the number of Solvers was 79,984.

An intermediate result was presented in [16]. From the intermediate results, the idle time ratio was extremely low in general (less than 2% in many cases) while the biggest one was 27.5%. The latter was reached in a run that was aborted due to a hardware error and terminated after 4.4 out of the planned 12 hours. In the primary phase, just an assigned sub-MIP was consistently being solving, therefore, the idle time comes from the ramp-up and ramp-down phases.

In Figures 5, 6 and 7, the results of all 48 runs are arranged by accumulating computing times of the previous runs. Figure 5 shows how the upper and lower bounds evolved. At the end of the first run, the relative gap was already 0.15%, still it was really hard to solve the remaining part to optimality.

Figure 6 shows how the number of open nodes and the number of active Solvers evolved together with the number of ParaNodes in the checkpoint file. All Solvers were active most of the time. The number of ParaNodes in the checkpoint file was very stable at around 10,000. Figure 7 shows how the limit of the collecting mode Solvers parameter value is changed during the computation and the ratio in duration of collecting mode in the computing time. Once the lower bound converges closer to optimality, we see more solvers going into collecting mode for a longer duration, which indicates that the search is getting closer to termination.

Altogether, the results show that ParaSCIP controlled search direction appropriately with handling up to 79,984 Solvers with a single LC. This makes it a new record for the largest number of cores involved in a parallel MIP search.

6 Discussion

It is difficult to evaluate the performance of parallel MIP solvers. This difficulty is summarized in [6, 17]. A general way to compare MIP solvers is to use carefully designed benchmark sets like MIPLIB. However, the performance of commercial MIP solvers has been improving much faster than that of computer hardware. Although caching effects could lead to have super-linear speed-ups, we can expect at most linear speed-up by using parallel solvers in general. Therefore, even if a scalable parallel branch-and-bound framework is used, it is hard to solve instances that cannot be solved by commercial solvers [18].

For parallel branch-and-bound, there are well-known anomalies in speed-ups [19]. This is originated from slight algorithmic changes that lead to a different solution path, the tree search may amplify those differences almost arbitrarily. In ParaSCIP case, UG explicitly introduces more algorithmic changes by running more Solvers as described in this paper. Therefore, evaluating performance by (strong) scalability would not be appropriate. An alternative way to evaluate ParaSCIP performance as a MIP solver in small scale is presented in [7]. It would be good to have some reasonable way to evaluate performance taking into

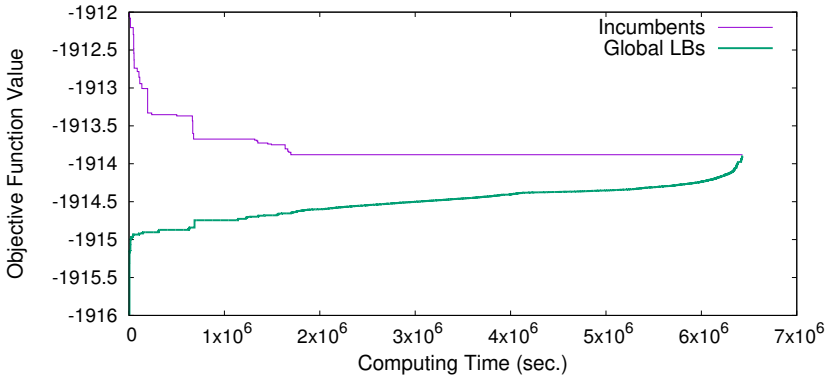

Figure 5: Lower and upper bounds evolution (`rmine10`).

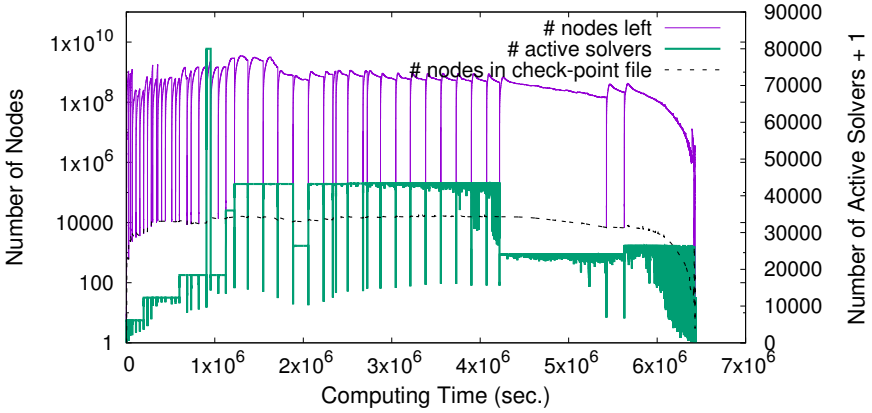

Figure 6: Active solvers and # of nodes left (`rmine10`).

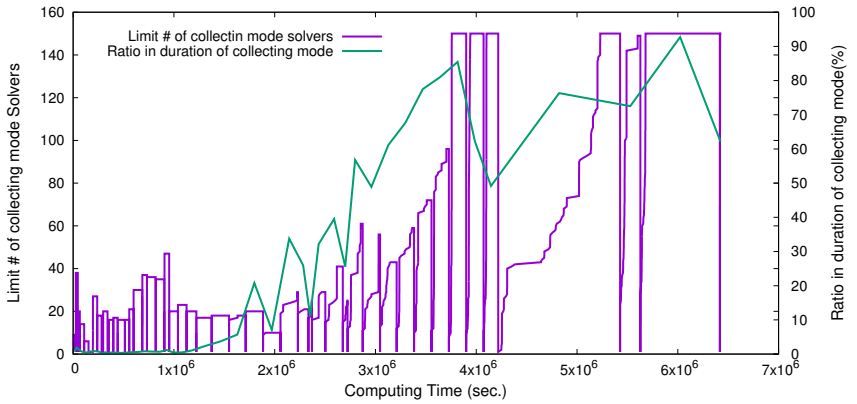

Figure 7: Runtime behavior of collecting mode (`rmine10`).

account the algorithmic changes. The paper [17] is intended to initiate such a research.

7 Concluding remarks

In this paper we have shown that running ParaSCIP on some of the largest supercomputers can be utilized to solve difficult, previously unsolved MIP instances. ParaSCIP can stably handle over 40,000 cores, even in situations where a huge amount of branch-and-bound nodes is constantly distributed. The biggest scale computational experiments conducted use 80,000 cores on Titan. This gives rise to the expectation that ParaSCIP will be capable of handling even larger scale computing environments. Our first design approach of ParaSCIP had a two-layered LC. However, the presented results do not indicate the need for a two-layered LC. To utilize an even higher number of cores, it seems more beneficial to design a combined system that additionally uses the internal shared-memory parallelization of the MIP solver[20].

ParaSCIP can be used by researchers to conduct their own experiments, it is available in source code and distributed as a part of the SCIP Optimization Suite[2]. As we described in this paper, the key is to develop new algorithms that do not work in sequential SCIP, but work in parallel. ParaSCIP is a good tool to test such an algorithm, for example, distributed domain propagation [21]. One of the biggest advantages of SCIP is that it can be extended to build a customized solver by adding user plugins. The latest distribution of ParaSCIP has a feature to parallelize customized SCIP solvers by implementing a small interface [22]. A successful example of such an expansion is the parallel Steiner Tree Problems solver introduced in [23]. It participated in the 11th DIMACS Implementation Challenge in Collaboration with ICERM[3]. In this competition, ParaSCIP was the only solver that was capable of running on distributed memory computing environments. In the release of the SCIP Optimization Suite 5.0, the MISDP solver SCIP-SDP was also parallelized the same way [24].

Given that major MIP software vendors such as IBM Cplex, Gurobi and FICO Xpress have started to integrate distributed computing capabilities, the topic will become even more significant in the future. Important questions are the balancing of ramp-down and ramp-up phases and a proper handling of subproblems—subtrees and individual nodes—that show very different runtime behaviors. We believe that the present paper gives some first clues on how to address these challenges.

Acknowledgments

The work was supported by the North-German Supercomputing Alliance (HLRN). We are grateful to the HLRN III supercomputer staff, especially Matthias Läuter and Guido Laubender and to the ISM supercomputer staff in Tokyo, especially Tomonori Hiruta. This

[2]http://scip.zib.de/#scipoptsuite

[3]http://dimacs11.cs.princeton.edu/

research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. The work for this article has been conducted within the Research Campus Modal funded by the German Federal Ministry of Education and Research (fund numbers 05M14ZAM,05M20ZBM). The work has been supported by a Google Faculty Research award.

References

[1] G. L. Nemhauser and L. A. Wolsey, *Integer and combinatorial optimization*. Wiley, 1988.

[2] R. E. Bixby, E. A. Boyd, and R. R. Indovina, "MIPLIB: A test set of mixed integer programming problems," *SIAM News*, vol. 25, p. 16, 1992.

[3] "MIPLIB 2017," 2018, http://miplib.zib.de.

[4] M. R. Bussieck, M. C. Ferris, and A. Meeraus, "Grid-enabled optimization with GAMS," *IJoC*, vol. 21, no. 3, pp. 349–362, Jul. 2009.

[5] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch, "ParaSCIP – a parallel extension of SCIP," in *Competence in High Performance Computing 2010*, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, Eds. Springer, 2012, pp. 135–148.

[6] T. K. Ralphs, Y. Shinano, T. Berthold, and T. Koch, "Parallel solvers for mixed integer linear optimization," in *Handbook of Parallel Constraint Reasoning*, Y. Hamadi and L. Sais, Eds. Springer International Publishing, 2018, pp. 283–336. [Online]. Available: https://doi.org/10.1007/978-3-319-63516-3_8

[7] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler, "Fiberscip—a shared memory parallelization of scip," *INFORMS Journal on Computing*, vol. 30, no. 1, pp. 11–30, 2018. [Online]. Available: https://doi.org/10.1287/ijoc.2017.0762

[8] Y. Shinano, "The ubiquity generator framework: 7 years of progress in parallelizing branch-and-bound," in *Operations Research Proceedings 2017*, 2018, accepted for publication.

[9] T. Achterberg, "Constraint integer programming," Ph.D. dissertation, Technische Universität Berlin, 2007.

[10] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch, and M. Winkler, "Solving hard MIPLIB2003 problems with ParaSCIP on supercomputers: An update," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 1552–1561.

[11] M. W. P. Savelsbergh, "Preprocessing and probing techniques for mixed integer programming problems," *ORSA Journal on Computing*, vol. 6, pp. 445–454, 1994.

[12] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. Steffy, and K. Wolter, "MIPLIB 2010," *Mathematical Programming Computation*, vol. 3, pp. 103–163, 2011.

[13] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch, and M. Winkler, "Solving hard miplip2003 problems with parascip on supercomputers: An update," in *IPDPSW'14 Proceedings of the 2014 IEEE, International Parallel & Distributed Processing Symposium Workshops*, IEEE, Ed., 2014, pp. 1552 – 1561.

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

[15] T. Berthold, T. Feydy, and P. J. Stuckey, "Rapid learning for binary programs," in *Proc. of CPAIOR 2010*, ser. LNCS, A. Lodi, M. Milano, and P. Toth, Eds., vol. 6140. Springer, June 2010, pp. 51–55.

[16] 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 *Proc. of 30th IEEE International Parallel & Distributed Processing Symposium*, 2016.

[17] S. Maher, T. Ralphs, and Y. Shinano, "Assessing the effectiveness of (parallel) branch-and-bound algorithms," ZIB, Takustr. 7, 14195 Berlin, Tech. Rep. 19-03, 2019.

[18] J. Eckstein, W. E. Hart, and C. A. Phillips, "Pebbl: an object-oriented framework for scalable parallel branch and bound," *Mathematical Programming Computation*, vol. 7, no. 4, pp. 429–469, 2015. [Online]. Available: http://dx.doi.org/10.1007/s12532-015-0087-1

[19] T.-H. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *Communications of the ACM*, vol. 27, no. 6, pp. 594–602, 1984. [Online]. Available: http://doi.acm.org/10.1145/358080.358103

[20] Y. Shinano, T. Berthold, and S. Heinz, "Paraxpress: An experimental extension of the fico xpress-optimizer to solve hard mips on supercomputers," *Optimization Methods & Software*, 2018, accepted for publication.

[21] R. L. Gottwald, S. J. Maher, and Y. Shinano, "Distributed domain propagation," in *16th International Symposium on Experimental Algorithms (SEA 2017)*, vol. 75, 2017, pp. 6:1 – 6:11.

[22] Y. Shinano, D. Rehfeldt, and T. Gally, "An easy way to build parallel state-of-the-art combinatorial optimization problem solvers: A computational study on solving steiner tree problems and mixed integer semidefinite programs by using ug[scip-*,*]-libraries," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019, pp. 530–541.

[23] G. Gamrath, T. Koch, S. Maher, D. Rehfeldt, and Y. Shinano, "Scip-jack – a solver for stp and variants with parallelization extensions," *Mathematical Programming Computation*, vol. 9, no. 2, pp. 231 – 296, 2017.

[24] A. Gleixner, L. Eifler, T. Gally, G. Gamrath, P. Gemander, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, F. Serrano, Y. Shinano, J. M. Viernickel, S. Vigerske, D. Weninger, J. T. Witt, and J. Witzig, "The scip optimization suite 5.0," ZIB, Takustr.7, 14195 Berlin, Tech. Rep. 17-61, 2017.