Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

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

Solving Open MIP Instances with ParaSCIP on Supercomputers using up to 80,000 Cores

The work for this article has been conducted within the Research Campus Modal funded by the German Federal Ministry of Education and Research (fund number 05M14ZAM).

ZIB Report 15-53 (October 2015)

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

Solving Open MIP Instances with ParaSCIP on Supercomputers using up to 80,000 Cores

Yuji Shinano*, Tobias Achterberg†, Timo Berthold‡, Stefan Heinz‡,
Thorsten Koch*, Michael Winkler†

*Zuse Institute Berlin, †Gurobi GmbH, ‡Fair Issac Europe Ltd

Takustr. 7, 14195 Berlin, Germany

*{shinano,koch}@zib.de, †{achterberg,winkler}@gurobi.com,

‡{timoberthold,stefanheinz}@fico.com

October 27, 2015

Abstract

This paper shows how we solved 12 previously unsolved mixed-integer programming (MIP) instances from the MIPLIB benchmark sets. Therefore, 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 solving statistics for solving open MIPLIB instances.

1 Introduction

The first time that supercomputers with more than 10,000 cores appeared on the Top500 supercomputing list [1] was in November 2004. As of June 2015, the list contains only five entries that have 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 paper deals with solving Mixed Integer Programming (MIP) problems in parallel. Throughout this paper without loss of generality, we assume that the MIP is given in the following, general form:

$$\min\{\langle c, x\rangle : Ax \leq b,\ x_I \in \mathbb{Z}^{|I|}\}, \tag{1}$$

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

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

Many optimization problems arising in practice can be modeled as MIP, see, e.g., [1]. Due to well-established data format standards it was possible to collect a variety of real-world problem instances and to make them publicly available in problem libraries like MIPLIB. The first version of MIPLIB has been created in 1992 [2], its latest update is MIPLIB2010[3]. Such libraries are key to the evolution of the MIP research field, as they allow to evaluate new ideas and algorithms on data sets that are similar to those that really matter in practice. Moreover, researchers can directly compare their results to previous work in the area, and unsolved models from these libraries provide research challenges to advance the field.

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 a number of cutting plane separation algorithms to increase the lower bound value obtained by the Linear Programming(LP) relaxation, see, e.g., [1].

Tree search generally is considered as 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 to solve 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, extensions of which we present in this paper. Both solvers used a state-of-the-art MIP solver as a black box to exploit the latest MIP solving technology; the tree search based solving process was parallelized externally.

In this paper, we first briefly introduce ParaSCIP and explain its different parallelization features. Next, we explain two novel techniques of merging search nodes and exploring history information for branching prior to search. Finally, we highlight some results from recent computational experiments on up to 80,000 cores.

2 ParaSCIP – A Distributed Memory MIP Solver

ParaSCIP has been developed by using the *Ubiquity Generator (UG) framework* [5]. 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 communication library can be exchanged, which makes the parallel solver more portable. ParaSCIP is the instantiated parallel solver, in which SCIP is used as the black box MIP solver and MPI is used as the communication library.

In this subsection, we briefly explain how ParaSCIP works, more details can be found in [6, 5]. As shown in Figure 2, two types of processes exist when running ParaSCIP on a supercomputer. There is a single LoadCoordinator which makes all decisions concerning the dynamic load balancing and distributes subproblems of the MIP instances to be solved (so-called sub-MIPs). All other processes are Solvers that solve the distributed sub-MIPs.

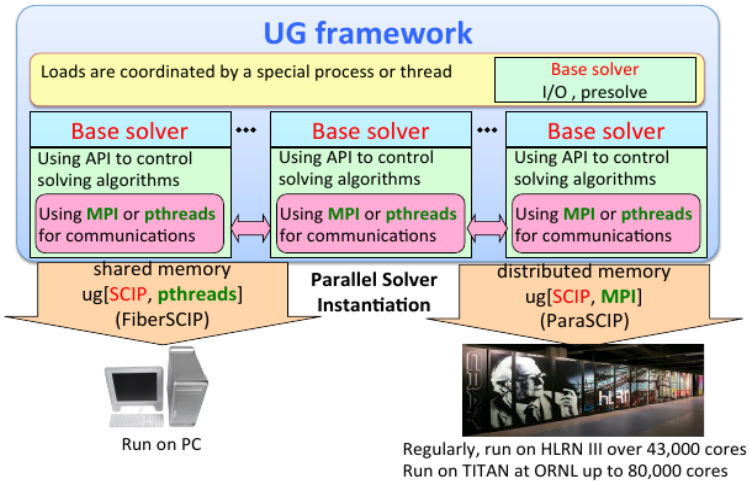

Figure 1: Design structure of Ubiquity Generator Framework.

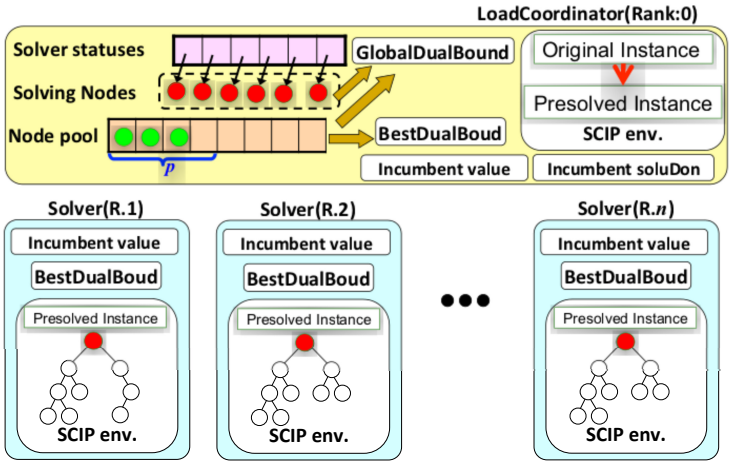

Figure 2: Process composition and data arrangement of ParaSCIP.

2.1 Initialization

The LoadCoordinator reads the instance data of the MIP to solve. We refer to the result as the *original instance*. This instance is presolved directly inside the LoadCoordinator, see, e.g., [7, 8] for an overview on MIP presolving techniques. 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. Later, only differences between a sub-MIP and the presolved instance will be communicated.

2.2 Ramp-up

Ramp-up is the phase until all solvers become busy. After the initialization step, the LoadCoordinator creates the root node of the branch-and-bound tree. Each node transferred through the system – we call such a node ParaNode– acts as the root of a subtree. The information that has to be sent consists only of bound changes of variables. The Solver which receives a new branch-and-bound node instantiates the corresponding sub-MIP using the presolved instance (which was distributed in the initialization step) and the received bound changes. ParaSCIP provides two ramp-up mechanisms.

Normal ramp-up Solvers, that are already solving a sub-MIP, transfer every second child node back to the LoadCoordinator. The LoadCoordinator maintains a *node pool*, from which it assigns nodes to idle Solvers. If no idle Solver exists, the LoadCoordinator keeps collecting nodes from Solvers until it has p "good" (promising to have a large subtree underneath) unassigned nodes in its node pool. Here, p is a run-time parameter. As soon as the LoadCoordinator's node pool accumulated p "good" nodes, it sends a message to all Solvers to stop sending nodes.

Racing ramp-up In this mechanism, the LoadCoordinator 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, each Solver uses a different parameter setting and a different permutation of variables and constraints. As shown in [3], the latter can have a considerable impact on the performance of a solver due to imperfect tie breaking. Due to these variations, we can expect that the Solvers generate different search trees. After a certain 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 LoadCoordinator and a termination message is sent to all other Solvers. Afterwards, the collected nodes are redistributed to the now idle Solvers. If the "winner" provides less nodes than there are Solvers, ParaSCIP changes its strategy to normal ramp-up.

2.3 Dynamic load balancing

Periodically, each Solver notifies the LoadCoordinator about the number of unexplored nodes in its SCIP environment and the lower bound of its subtree; we call this information the *solver status*. If a Solver becomes idle, the LoadCoordinator sends one of the nodes from the pool to the idle Solver. In order to keep all Solvers busy, the LoadCoordinator should always have a sufficient amount of unprocessed nodes left in its node pool. In order to keep at least p "good" nodes in the LoadCoordinator, we employ the *collecting mode*, similar to the one introduced in [9]. We call a node *good*, if the lower bound value of its subtree (NodeBound) is sufficiently close to the lower bound

value of the complete search tree (GlobalBound), in formula, if

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

If a Solver receives the message to switch into the collecting mode, it changes the search strategy to "best bound order" (see [8]). Similar to normal ramp-up, the Solvers alternately solve nodes and transfer them to the LoadCoordinator.

Solvers are switched 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 good nodes in the pool is larger than $1.5 \cdot p$.

2.4 Termination

The termination phase starts when the node pool is empty and all Solvers are idle. In this phase, the LoadCoordinator collects statistical information from all Solvers and outputs the optimal solution and statistics.

2.5 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, i. e., nodes for which no ancestor nodes are in the LoadCoordinator. This strategy requires much less effort for the I/O system, 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 weighed out by the benefits of re-applying a global presolving procedure at the restart (see [10]).

To restart, ParaSCIP reads the nodes saved in the checkpoint file and restores them into the node pool of the LoadCoordinator. After that, the LoadCoordinator distributes these nodes to the Solvers ordered by their lower bounds.

3 Improving the Dynamic Load Balancing

ParaSCIP realizes a parallelization of MIP solvers for a distributed memory computing environment without a centralized global search tree data structure. Due to the latter, the dynamic load balancing among Solvers is extremely important to improve scalability.

3.1 Dynamic tuning of parameters and bulk sending of ParaNodes

How frequently a Solver can send ParaNodes to the LoadCoordinator depends not only on the computing environment but also on the instance to be solved. The time needed to process individual nodes includes different algorithms with significant runtimes, such as

node preprocessing or LP solving. The time spent for an individual node can vary between a fraction of a second and several minutes, even within the same MIP instance. This time can hardly be estimated in advance. Therefore, although the number of SOLVERs which can be in collecting mode at a certain point of time is specified by a run-time parameter, this number needs to be adjusted dynamically to reduce the idle time ratio.

Sending PARANODEs, i. e., sub-MIPs, from a subtree means that the part of the subtree is explored more aggressively by using the other SOLVERs. On the one hand, it is beneficial to keep the number of the collecting mode SOLVERs small at any point in time, since this will focus the tree exploration to 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 only one at the beginning of the computation and is increased by one whenever the node pool in the LOADCOORDINATOR stays empty for a time period specified by a run-time parameter (the default setting is 10 seconds). The value p is also changed dynamically. It is not only increased but also is decreased depending on how fast the LOADCOORDINATOR switches into collecting mode. In the default setting, if the interval time between collecting modes is less than 10 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 LOADCOORDINATOR makes sending individual PARANODEs comparatively slow. To avoid this, we implemented a fast bulk sending mechanism. To realize this, the message which requests a SOLVER to switch into collecting mode additionally includes the number of PARANODEs that is expected to be sent from the SOLVER. That is, when the LOADCOORDINATOR 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 LOADCOORDINATOR without synchronization in between. If a SOLVER does not have enough open nodes, it sends as many as possible as a bulk. Afterwards, it switches to the normal PARANODE sending mechanism.

3.2 Improving the ramp-down

The *ramp-down* phase is reached at the end of the computation when it gets difficult to keep all SOLVERs busy. At the end of the computation of the MIP solver, typically only a few SOLVERs have a significant search tree left. Contrarily, most of the PARANODEs will be solved extremely fast and the SOLVERs send their completion messages to the LOADCOORDINATOR. In worst case, this can lead to a congestion in the communication network and it may even prevent the LOADCOORDINATOR from collecting PARANODEs. When the LOADCOORDINATOR 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 when for a longer period of time less than 90% of the solvers are active and the number of open nodes exceeds the number of Solvers by more than a factor of one hundred.

3.3 Restarting the collecting mode

ParaSCIP can control how frequently the Solver statuses are updated by using the *notification interval time* parameter that indicates the interval time between status messages from a Solver. Each message contains very little data, but all Solvers send these message periodically. When the number of parallel cores grows larger and larger, eventually this communication becomes a bottleneck and a longer updating interval is chosen. As a consequence, the LoadCoordinator 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 reached 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.

3.4 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 promising to have a larger 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, using 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 and the ParaNode with the larger feasible region is transferred.

4 Additional Techniques Applied Externally From SCIP

This section introduces novel techniques for parallel MIP search that helped us tackle unsolved instances and further improved 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 [11]. 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 about previous

branchings. Naturally, these statistics are often weak in the beginning of 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 need to restart the search from a checkpoint file. This seems like a natural point to re-organize the branch-and-bound tree, base on 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 $V := \{v_k | k = 1, \ldots, l\}$ be a set of values assigned to variables and let $P_i := (F_i^V, F_i^{\overline{V}})$ be a representation of a sub-MIP i, that is it corresponds to a ParaNode, by the following variable statuses:

$$F_i^V := \{j_i | x_{j_i} \text{ is fixed to } v \in V, j_i \in J\},$$
$$F_i^{\overline{V}} := \{j_i | x_{j_i} \text{ is not fixed}, j_i \in J\}.$$

Here, J denotes the index set of problem variables and x_{j_i} is fixed to $v \in V$means that$|x_{j_i} - v| < \epsilon$. Let $O := \{P_i | i = 1, \ldots, m\}$ be a set of sub-MIPs and let $S_j^k := \{P_i \in O | x_{j_i} = v_k, v_k \in V, x_{j_i} \in F_i^V\}$ be a set of sub-MIPs that share the same fixed value variable. We call a subset of S_j^k, namely $S_j^k(M) := \{P_i \in M \subset O | x_{j_i} = v_k, v_k \in V, x_{j_i} \in F_i^V\}$ a candidate for merging of ParaNodes. Merging ParaNodes is performed by Algorithm 1 and Algorithm 2.

For the merging of nodes, similar considerations holds as for checkpointing by storing primitive nodes. It potentially looses information because it relaxes already fixed variables. Also, merging is likely to decrease the lower bound of the corresponding sub-MIPs. However, since the merged ParaNode will be solved like a stand-alone problem, namely from scratch using 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 drops too much, see Algorithm 1.

The main advantage of merging, actually its main purpose, is that the search tree gets re-arranged such that is better balanced. By this, merging also raises the chance to find improving solutions in previously unexploited search regions. The current ParaSCIP version also includes a feature to perform the merge procedure off-line (i.e., on a desktop machine in between two supercomputer runs) and update the checkpoint file correspondingly.

4.2 Deep probing

In SCIP, for each variable several statistics are stored that were collected during the solution process. In particular, branching statistics are used to select a branching variable. The racing stage is a good opportunity to tentatively collect these variable statistics for different possible search trees for the MIP instance at hand. This means, the LoadCoordinator collect all branching statistics not only from the racing winner, but from all Solvers that participated in racing. These information are then aggregated and used to initialize the branching statistics of all Solvers after racing, compare [12]. We refer to this strategy as *deep probing* since it resembles the ideas of probing and strong branching, just that

Algorithm 1 Solve all open sub-MIPs with merging

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

```
M ← O
C ← Algorithm 2(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 P⃛_i}
    Perform root node procedure for P⃛_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 2(M)
      T ← T ∪ P_i ∪ C
    else
      Keep solving P⃛_i(*)
    end if
  else {P_i := P̂_i}
    Solve P_i(*)
  end if
end while
```

Algorithm 2 Generate merge-nodes candidate set

Input: $M \subset O$
Output: C {C is merge-nodes candidate set}
  while $M \neq \emptyset$ **do**
    Select $P_i \in M$ s.t. P_i is a sub-MIP having the best lower bound in M
    $\breve{M} \leftarrow M$
    $n \leftarrow 0$
    while $\max_{j_i \in F_i^V, k:x_{j_i}=v_k} |S^k_{j_i}(\breve{M})| > 2$ **do**
      {At least two nodes can be merged}
      $\hat{j} = \underset{j_i \in F_i^V, k:x_{j_i}=v_k}{\arg\max} |S^k_{j_i}(\breve{M})|$
      $\breve{M} \leftarrow S^{k:x_{\hat{j}}=v_k}_{\hat{j}}(\breve{M})$ {NOTE: $P_i \in \breve{M}$}
      $n \leftarrow n + 1$
    end while
    if $\frac{n}{|F_i^V|} > \tau$ **then**
      {τ is a parameter: 0.9(current default)}
      Make a merged-sub-MIP $\ddot{\breve{P}}_i$ from $\breve{M}$
      $C \leftarrow C \cup \ddot{\breve{P}}_i$
      $M \leftarrow M \setminus \breve{M}$
    else
      $C \leftarrow C \cup P_i$
      $M \leftarrow M \setminus P_i$
    end if
  end while

instead of single nodes whole subtrees are explored tentatively. We expect that initializing branching statistics helps to improve branching and decreases the likelihood of "bad" initial branchings, see the previous section. The effect of deep probing is not yet fully investigated, but our experiments so far were promising. In order to use deep probing, all ParaNodes need to store (and communicate) these 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

Our computational experiments are split into three parts. First, we demonstrate the improvements of the ramp-down process. Second, we summarize computational results for challenging MIP instances from the MIPLIB collection that have been solved for the first time using ParaSCIP. Third, we report on the largest (up to our knowledge) MIP solver run that has ever been conducted in our attempt to solve the `rmine10` instance.

5.1 Improvements of load balancing process

For the computational results presented in this subsection, we used ParaSCIP using SCIP 3.0.1 with Cplex 12.5 as underlying linear programming solver and ran it on Titan[2]: Cray XK7, Opteron 6274 16C 2.2GHz, Cray Gemini interconnect, NVIDIA K20x with 10,000 cores. There is a genuine advantage that racing ramp-up has over normal ramp-up, namely all solvers can start right away instead of waiting until enough nodes have been created. Also, it does not need to take care of dynamic load balancing, for which the LoadCoordinator needs to collect open nodes and distribute them trying to keep all solvers busy. For this experiment we used normal ramp-up, since our interest was to improve the dynamic load balancing. The effect of load balancing can be best seen during (normal) ramp-up and during ramp-down. As our show case test instance, we used `timtab2`, for two reasons: first, for this instance both, finding the optimal solution and proving its optimality is really hard and second, the instance is solvable on a supercomputer within a reasonable amount of time.

Figures 3 and 4 show how upper and lower bounds evolve and how the number of open nodes and the number of active Solvers change during the computation in our first trial run. One can observe that when a good feasible solution is found, huge parts of the search tree are pruned and the workload among Solvers gets imbalanced. The original dynamic load balancing did not recover well in this situation; the number of active Solvers decreased to about 280, even though there were enough open nodes. About 90% of the computing time was spent for the ramp-down process, using only 3% of the computing resources.

Figures 5 and 6 show the same information as in the previous two figures for the improved load balancing described in Section 3. When comparing the course of the graphs until the optimal solution is found, we see that it is almost the same as for the original load balancing, Note that the horizontal scales are different. Figure 6 demonstrates that the dynamic load balancing works in two different ways. In the first part, the adaptive parameter tuning tried to balance workloads among Solvers. The number of collecting mode Solvers at the same time is restricted to 100 and the collecting mode is restarted after it reached the limit. After 4322, the restarted collecting mode detects a huge imbalance and it switches to the bulk sending mode. The ParaSCIP version that uses improved load balancing is about four times faster than the original one. We used ad-hoc values for the collecting mode parameters, a careful tuning could even the performance if the new load balancing.

5.2 Open instances solved by ParaSCIP

Until 2008, the number of unsolved instances of MIPLIB2003 could be reduced to six. In April 2010, `ds` and `stp3d` have been solved by ParaSCIP, see [6, 10]; the remaining four instances are still open. In the meantime, MIPLIB2010 [3] has been published, the original

[2]http://www.olcf.ornl.gov/titan/

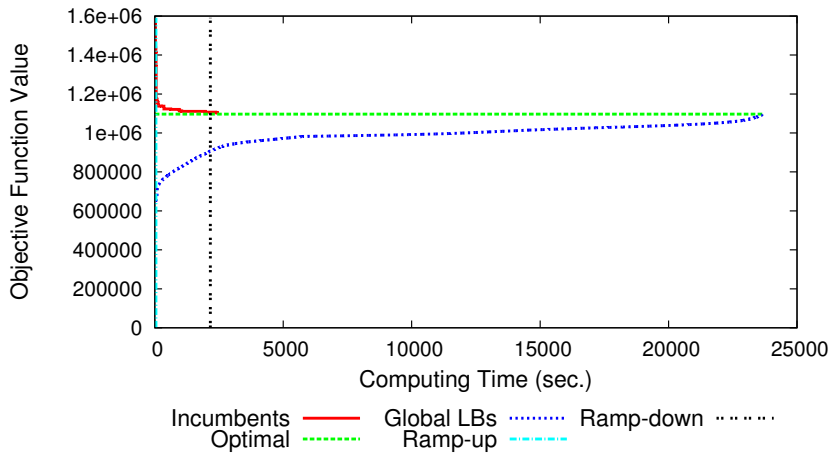

Figure 3: Lower and upper bounds evolution (`timtab2`, Original)

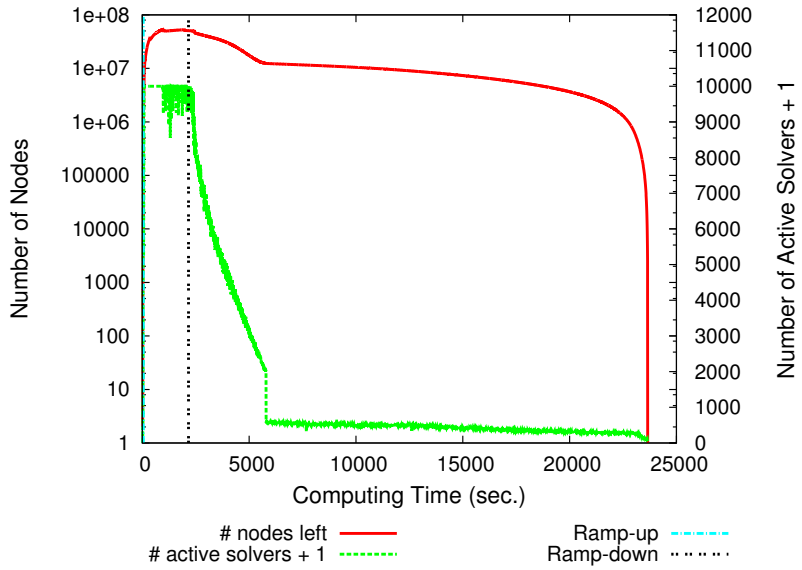

Figure 4: Active solvers and the number of nodes (`timtab2`, Original)

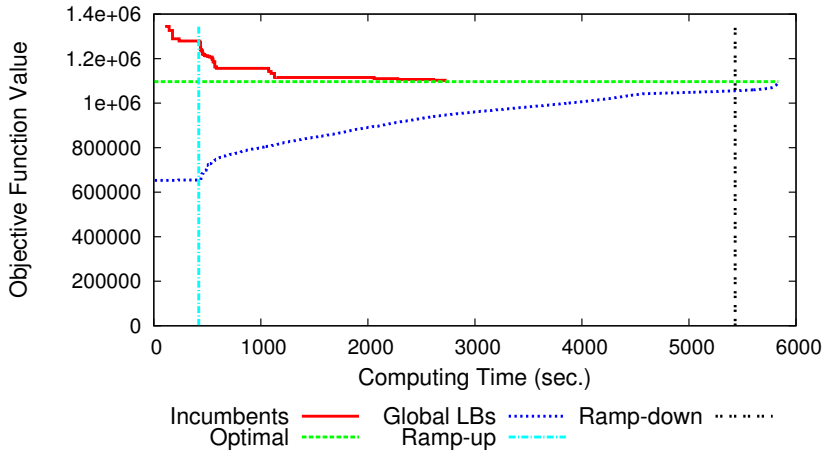

Figure 5: Lower and upper bounds evolution (`timtab2`, improved)

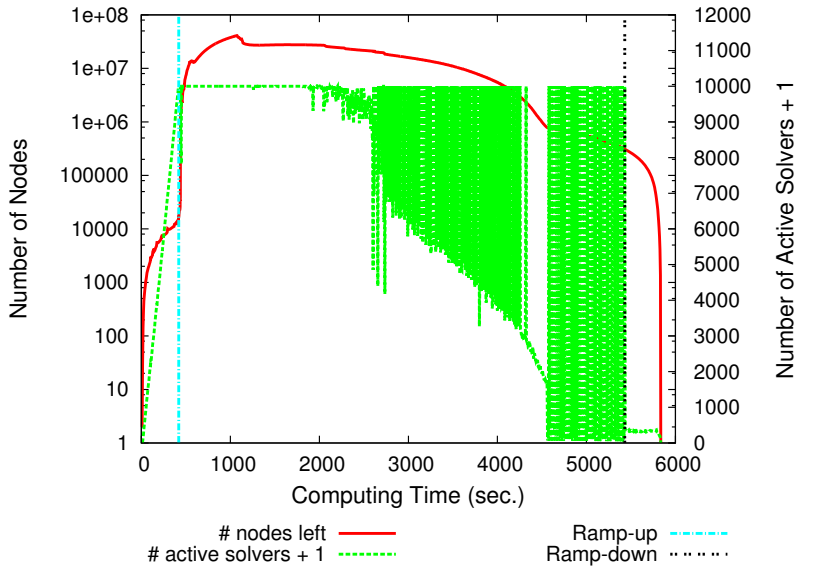

Figure 6: Active solvers and the number of nodes left (`timtab2`, improved)

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 2013	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	

paper listed 134 unsolved instances. In the following, we present details of our ParaSCIP runs that solved ten of them to proven optimality for the first time.

Table 1 gives a short overview on how the instances were solved. For each instance, 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. 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), and "ISM" is the ISM supercomputer Fujitsu PRIMERGY RX200S5. 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[3]. For these experiments, we initialized the search with the best known solutions. All other instances were solved from scratch.All instances which are solved using 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 of the first run as an initial solution.

5.3 The biggest and the longest computation

Currently, we aim at solving the open instance `rmine10` using the supercomputers HLRN III and Titan. Figure 7 shows the computing time and the number of Solvers used for each run. The Solvers are categorized by two types: one is solvers that kept solving only one single sub-MIP during a run, the other is solvers that solved more than one sub-MIPs. This illustrates the ratio between solvers that are working on a single hard subproblem and those that get assigned easier subproblems. We see that by now, this ratio converged to

roughly fifty-fifty.

For all runs on HLRN III, the number of Solvers used is exactly the number of cores minus 1, that is one for LoadCoordinator. For the Titan run, we used one computing node dedicated to LoadCoordinator process and the number of Solvers was 79,984. Figure 8 shows the number of nodes solved and that remained at the end of computation, together with the idle time ratio. The idle time ratio is calculated by using a log file which contains Solver statistics of solved sub-MIPs. These statistics are only obtained when at least one sub-MIP was solved, the data for the Solvers that kept solving one sub-MIP until the end of computation is missing. Figure 8 shows that 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 hours out of planned 12 hours.

In Figures 9, 10 and 11, the results of all 41 runs are arranged by accumulating computing times of the previous runs.Figure 9 shows how upper and lower bounds evolved. At the end of the first run, the relative gap was already 0.15%, still it is really hard to solve the remaining part to optimality. Now, at the end of the 41st run, it is less than 0.03%. As described above, important performance measures for parallel MIP solving, such as the ratio of Solvers that solve hard/easy problems, and the idle time ratio, have been almost constant over the last 20 runs. At the same, the lower bound of the MIP grows steadily. It seems likely that the solution process can be finished with a reasonable number of additional runs.

Figure 10 shows how the number of open nodes and the number of active Solvers evolved together with the number of ParaNodes in the checkpoint file. Also it is clear from the idle time ratio that all Solvers are active most of the time. Again, the number of ParaNodes in the checkpoint file is very stable around 10,000. Figure 11 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. We see that now, that the lower bound converges closer to optimality, more solvers go into collecting mode for a longer time; this is an indicator that the search is getting closer to termination.

Figures 12, 13 and 14 give detailed results for three particular runs. The very first run, the last run so far (run 41), and the only run that has been conducted on Titan (run 21). The diagrams show how the number of open nodes and the number of active Solvers evolved together with how many ParaNodes the LoadCoordinator received from and sent to the Solvers per second. A big number for nodes received per second, i.e., a blue bar in the diagram, indicates that ParaSCIP switched to collecting mode.

The first run, Figure 12, initially performed racing ramp-up. In this run, we hardly switched to collecting mode. In the Titan run shown in Figure 13, the interval time between collecting modes is increasing. This indicates that the branch-and-bound tree gets more and more balanced with more Solvers receiving reasonably hard sub-MIPs.In the 41st run, shown in Figure 14, most of the computing time is spend in collecting mode and the number of remaining nodes starts decreasing at the very beginning of the computation. In this run, often Solvers become idle, but they also recover quite well, recall also the small solver idle ratio. The high ratio of collecting mode times also indicates that we are getting

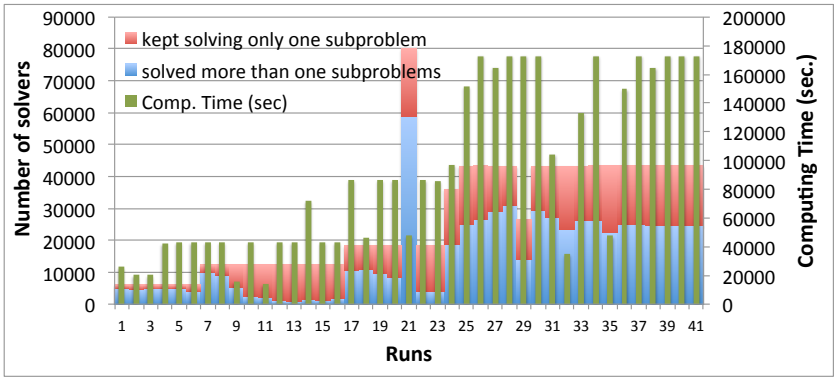

Figure 7: # of SOLVERs and computing times (`rmine10`).

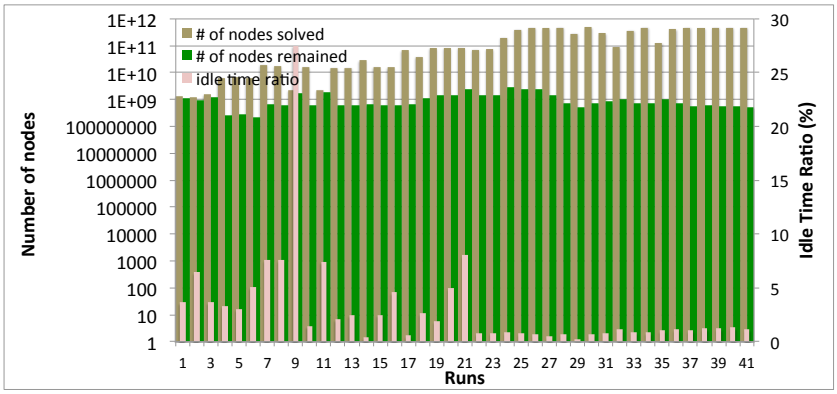

Figure 8: # of nodes and lower bounds of idle time ratios (`rmine10`).

closer to the ramp-down phase.

Altogether, the results show that ParaSCIP is able to handle up to 80,000 SOLVERs with a single LOADCOORDINATOR. This makes it a new record for the largest number of cores involved in a parallel MIP search.

6 Concluding remarks

In this paper we showed that running ParaSCIP on some of the largest supercomputers can be a utilized to solve hard, open 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, as shown in the Figure 14. In the biggest scale we conducted computational experiments using 80,000 cores on Titan without any problem. This gives rise to the assumption that ParaSCIP will be capable of handling even larger scale computing environments. Our first design approach of ParaSCIP had a two layered LOADCOORDINATORs of ParaSCIP. However, the presented results do not indicate a need for a two layered LOADCOORDINATOR and it seems more beneficial to design a system combined with the internal parallelization of the MIP solvers.

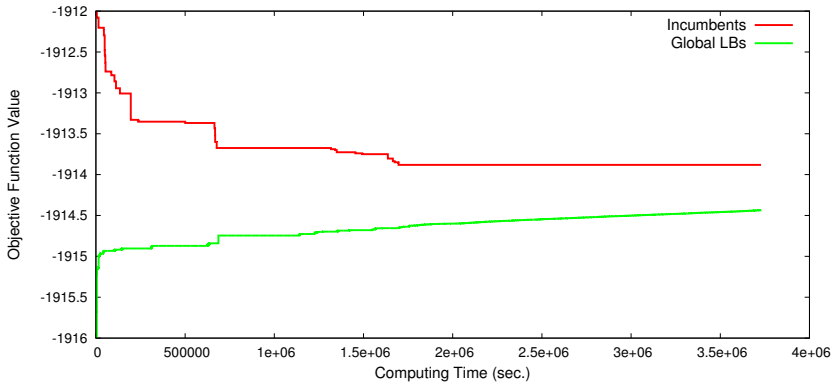

Figure 9: Lower and upper bounds evolution (`rmine10`).

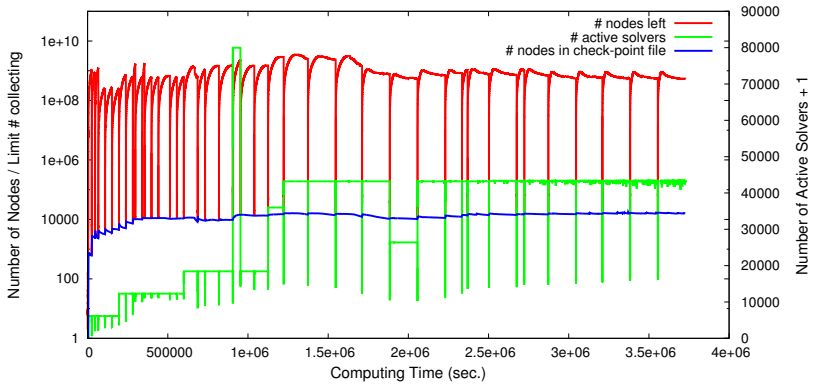

Figure 10: Active solvers and # of nodes left (`rmine10`).

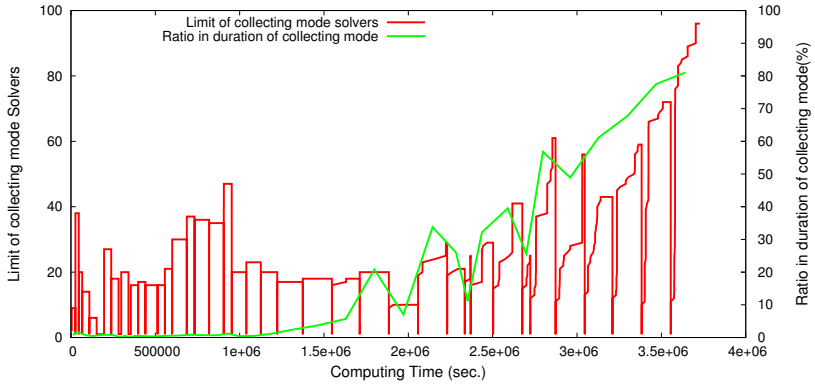

Figure 11: Runtime behavior of collecting mode (`rmine10`).

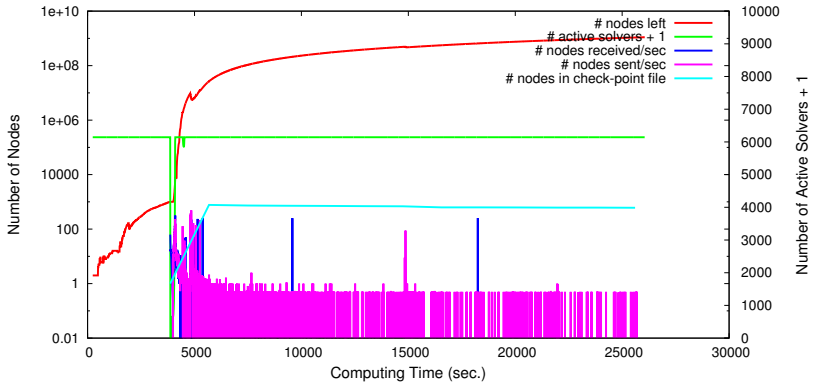

Figure 12: Active solvers and # of nodes left (Run 1).

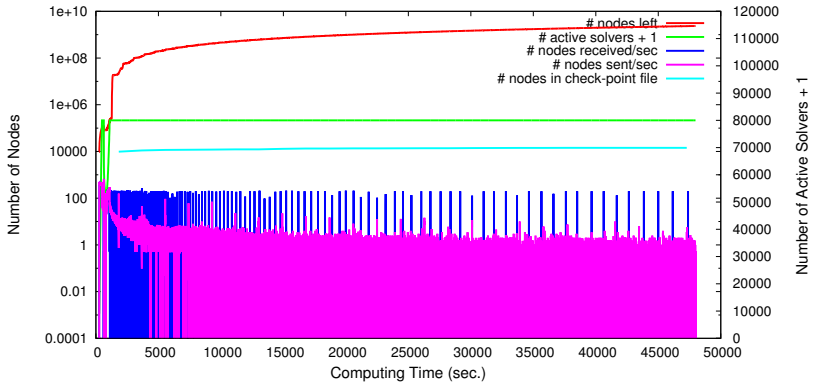

Figure 13: Active solvers and # of nodes left (Run 21).

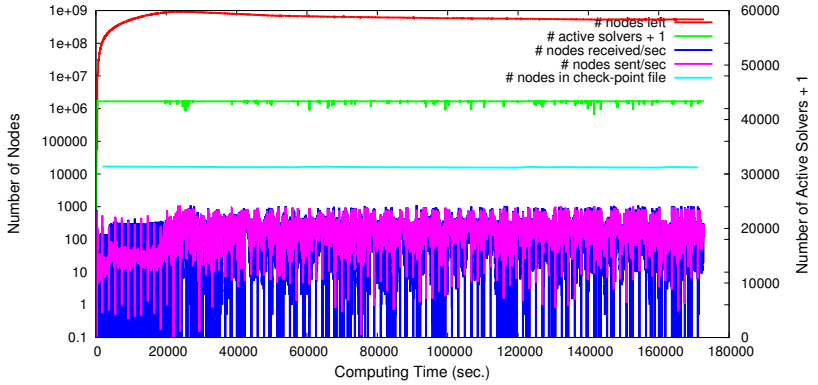

Figure 14: Active solvers and # of nodes left (Run 41).

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[3].

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. A successful example of such an expansion is the parallel Steiner Tree Problems solver introduced in [13]. It participated at the 11th DIMACS Implementation Challenge in Collaboration with ICERM[4]. In this competition, ParaSCIP was the only solver, that was capable of running on distributed memory computing environments.

Given that major MIP software vendors such as IBM Cplex, Gurobi and FICO Xpress have recently started to investigate distributed computing capabilities, it seems that this is an important future research topic. 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 how to address these challenges.

Acknowledgment

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 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 number 05M14ZAM).

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] 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.

[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, S. Heinz, S. Vigerske, and M. Winkler, "FiberSCIP – a shared memory parallelization of SCIP," Zuse Institute Berlin, Tech. Rep. ZR 13-55, 2013.

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

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

[6] 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.

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

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

[9] Y. Shinano, T. Achterberg, and T. Fujie, "A dynamic load balancing mechanism for new ParaLEX," in *In: Proceedings of ICPADS 2008*, 2008, pp. 455–462.

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

[12] 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.

[13] G. Gamrath, T. Koch, S. Maher, D. Rehfeldt, and Y. Shinano, "SCIP-Jack - a solver for STP and variants with parallelization extensions," ZIB, Takustr.7, 14195 Berlin, Tech. Rep. 15-27, 2015.