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, revised version February 2016)

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

February 12, 2016

Abstract

This paper describes how we solved 12 previously unsolved mixed-integer programming (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.

1 Introduction

Supercomputers with more than 10,000 cores first appeared on the Top500 supercomputer list[1] in November 2004. As of June 2015, this 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 we assume, without loss of generality, that a MIP is given in the following general form:

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

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 $I \subseteq \{1, \dots, n\}$.

[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 make them publicly available in problem libraries, such as MIPLIB. The first version of MIPLIB was created in 1992 [2]. Its latest iteration is MIPLIB2010[3]. The availability of such libraries are key to the evolution of the MIP research field, since they allow the evaluation of 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 studies, 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 many cutting plane separation algorithms to increase the lower bound value obtained by the Linear Programming (LP) relaxation, see, e.g., [1].

Tree search is 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 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 parallelization features. Next, we describe the 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 using the *Ubiquity Generator (UG) framework* [6]. 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 the instantiated parallel solver where SCIP is used as the black box MIP solver and MPI is used as the parallelization library.

In this section we briefly explain how ParaSCIP works, more details can be found in [5] and [6]. 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

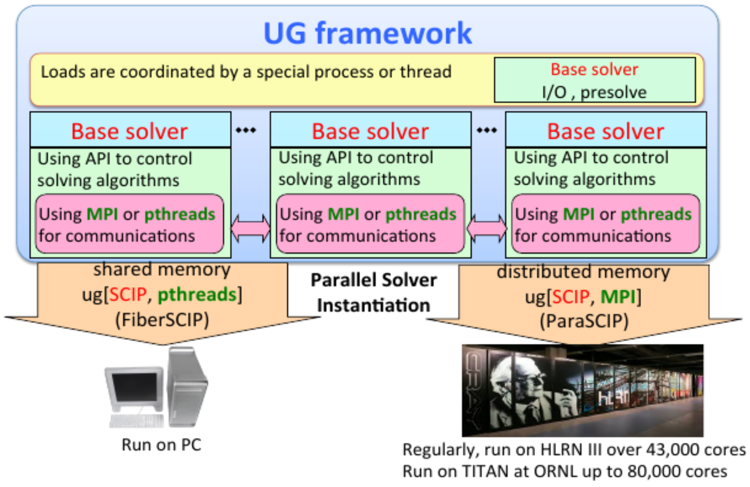
UG framework
Loads are coordinated by a special process or thread
Base solver
I/O , presolve
Base solver
Using API to control solving algorithms
Using MPI or pthreads for communications
shared memory
ug[SCIP, pthreads]
(FiberSCIP)
Parallel Solver Instantiation
distributed memory
ug[SCIP, MPI]
(ParaSCIP)
Run on PC
Regularly, run on HLRN III over 43,000 cores
Run on TITAN at ORNL up to 80,000 cores

Figure 1: Design structure of Ubiquity Generator Framework.

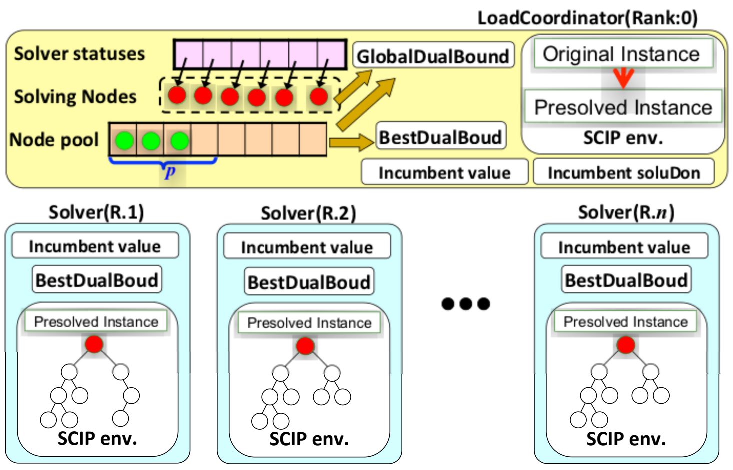

Figure 2: Process composition and data arrangement of ParaSCIP.

subproblems of MIP instances to be solved (so-called sub-MIPs). All other processes are Solvers that solve the distributed sub-MIPs.

2.1 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, see, e.g., [7] or [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.

After the initialization step, the LC creates the root node of the branch-and-bound tree. 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 presolved instance, which was distributed in the initialization step, and the received bound changes. Therefore, ParaNode is considered as a representation of a sub-MIP in ParaSCIP.

2.2 Ramp-up

Ramp-up is the phase that lasts until all solvers have become busy. 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 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) 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.

Racing ramp-up In this mechanism 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 [3], 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 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, if it provides less nodes than the number of available Solvers, ParaSCIP performs normal ramp-up.

Now, all commercial MIP solvers like CPLEX, Gurobi, or Xpress are concurrently solving the same problem that performs only racing stage in ParaSCIP and ParaSCIP can do it by setting a termination criteria of the racing stage as a run-time parameter.

2.3 Dynamic load balancing

Periodically, each Solver notifies the LC 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 LC sends one of the nodes from the pool to the idle Solver. In order to keep all Solvers busy, the LC aims to always have a sufficient number of unprocessed nodes left in its node pool. Further, the LC aims to keep at least p "heavy" nodes in the node pool by employing a *collecting mode*, similar to the one introduced in [9]. We call a node *heavy*, if the lower bound value of its subtree (NodeBound) is sufficiently close to the lower bound value of the complete 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 [8]). 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 $1.5 \cdot p$.

2.4 Termination

The termination phase starts when the node pool is empty and all Solvers are idle. In this phase, the LC 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, 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]).

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 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

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 may have significant runtimes. 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 the other Solvers means that part of the subtree is 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 10 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 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 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 that is 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.

3.2 Improving 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 will be 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 when 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 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. 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.

3.4 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 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.

4 Additional Techniques Applied Externally From SCIP

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 [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 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 variable x_j to value v. For a given set of sub-MIPs $\check{M} \subset M$ we define a *merged sub-MIP* as:

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

with bounds $l_j^{\check{M}} := \min_{i \in \check{M}}\{l_j^i\}$ and $u_j^{\check{M}} := \max_{i \in \check{M}}\{u_j^i\}$ for all $j \in \{1, \dots, n\}$. Merging ParaNodes is performed by Algorithm 1 and Algorithm 2.

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 1.

Our empirically 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.

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

Algorithm 1 Solve all open sub-MIPs with merging

```
Input: $O$
Output: Solve all sub-MIPs in $O$
  $M \leftarrow O$
  $C \leftarrow$ Algorithm 2$(M)$
  $T \leftarrow C$
  while $T \neq \emptyset$ do
    // This loop can be performed in parallel
    Select $\hat{P}_i \in T$
    $T \leftarrow T \setminus \{\hat{P}_i\}$
    if $\hat{P}_i$ is a merged-node then
      // $\dddot{P}_i := \hat{P}_i$
      // $P_i :=$ original sub-MIP of $\dddot{P}_i$
      Perform root node procedure for $\dddot{P}_i$
      if lower bound of $\dddot{P}_i <$
        (lower bound of $P_i) \cdot (1 - \delta)$ then
        // $\delta$ is a parameter: 0(current default)
        Recover a set of sub-MIPs $M$ from $\dddot{P}_i$
        $M \leftarrow M \setminus \{P_i\}$
        $C \leftarrow$ Algorithm 2$(M)$
        $T \leftarrow T \cup \{P_i\} \cup C$
      else
        Keep solving $\dddot{P}_i(*)$
      end if
    else // $P_i := \hat{P}_i$
      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
  $C \leftarrow \emptyset$
  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
      // $\tau$ 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
```

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 [12]. 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, see the previous section. 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

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, which took place 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 based on SCIP 3.0.1 with CPLEX 12.5 as underlying linear programming solver. The experiments were run 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 that all solvers can start right away instead of waiting until enough nodes have been created. Also, in the beginning of computation, it does not need to take care of dynamic load balancing, for which the LC 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`. This decision has 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

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

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	

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 becomes 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 overall. 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. Figure 6 demonstrates that the dynamic load balancing works in two different ways. In the first part, the adaptive parameter tuning tried to balance the workload among Solvers. The number of Solvers operating in collecting mode at the same time is restricted to 100 and the collecting mode is restarted after it reaches the limit. After 4,322 seconds, this is the point in time where the big green area is interrupted by a small white space, 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 version. We used moderate values for the collecting mode parameters, but the values were not tuned well enough. Careful tuning could achieve better performance in the new load balancing.

5.2 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. In the meantime, MIPLIB2010 [3] has been published, the original paper listed 134 unsolved instances. In the following, we present details of our ParaSCIP runs that solved ten 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"

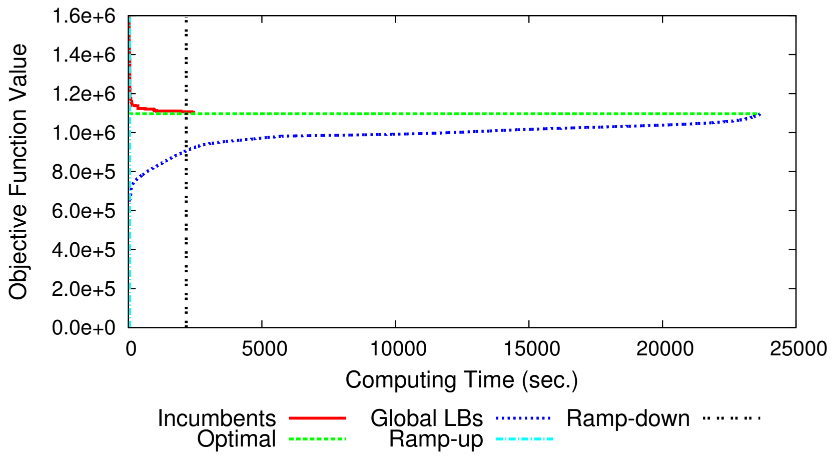

Figure 3: Lower and upper bounds evolution (`timtab2`, Original)

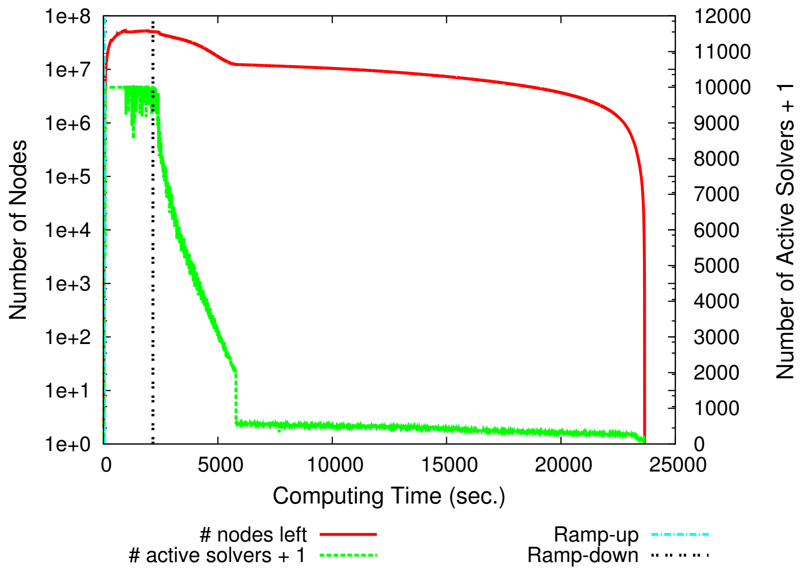

Figure 4: Active solvers and the number of nodes (`timtab2`, Original)

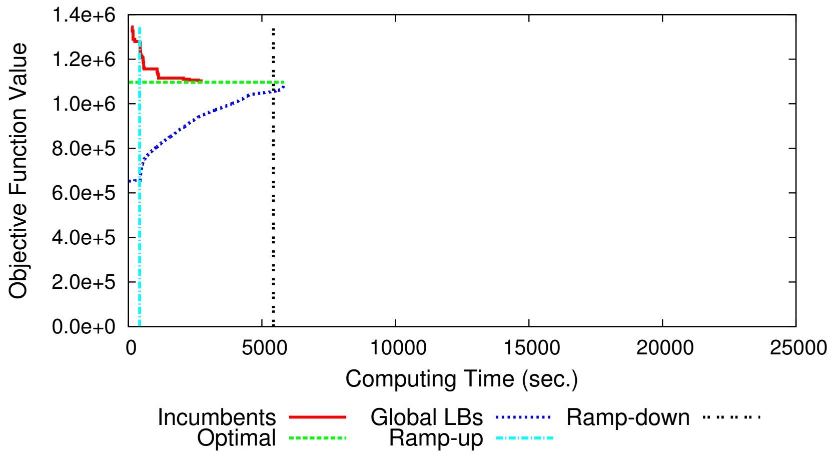

Figure 5: Lower and upper bounds evolution (`timtab2`, improved)

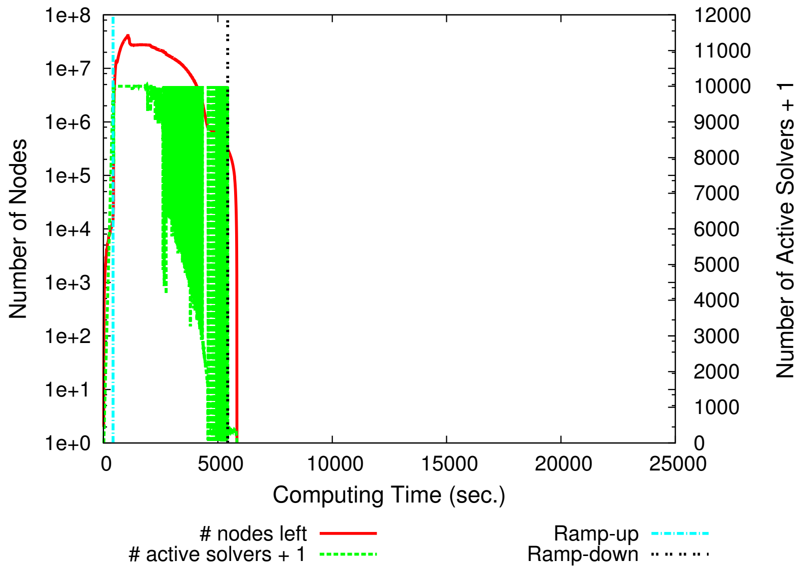

Figure 6: Active solvers and the number of nodes left (`timtab2`, improved)

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, 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), 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 that 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 solution 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: i) the solvers that kept solving only one single sub-MIP during a run and ii) the solvers that solved more than one sub-MIP. This illustrates the ratio between solvers that are working on a single hard sub-MIP and those that get assigned easier sub-MIPs. 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 the LC. For the Titan run, we used one computing node dedicated to the LC process and the number of SOLVERs was 79,984. Figure 8 shows the number of nodes solved and the number that remained at the end of the 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 the 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 the 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. 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 time, the lower bound of the MIP grew 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 at 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. 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.

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 LC 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. During this run, we hardly switched to collecting mode. The maximum number of nodes saved in the checkpoint file was 774 in 15 sec. In the Titan run, shown in Figure 13, the interval time between collecting modes increases. This indicates that the branch-and-bound tree becomes balanced as more Solvers receive reasonably hard sub-MIPs. In the 41st run, shown in Figure 14, most of the computing time is spent 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 closer to the ramp-down phase. The maximum number of nodes saved in the checkpoint file was 16,764 in 51 sec.

Altogether, the results show that ParaSCIP is able to handle up to 80,000 Solvers with a single LC. 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 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

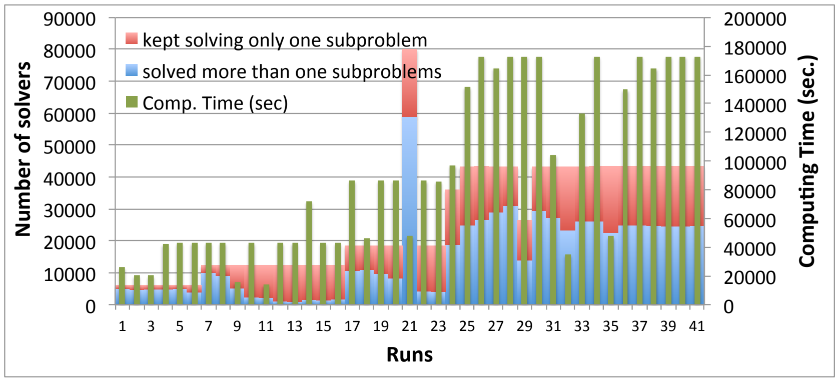

Figure 7: # of SOLVERS and computing times (`rmine10`).

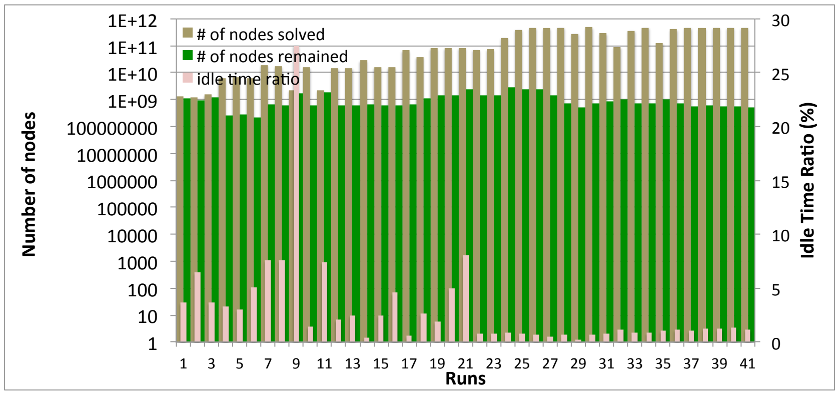

Figure 8: # of nodes and lower bounds of idle time ratios (`rmine10`).

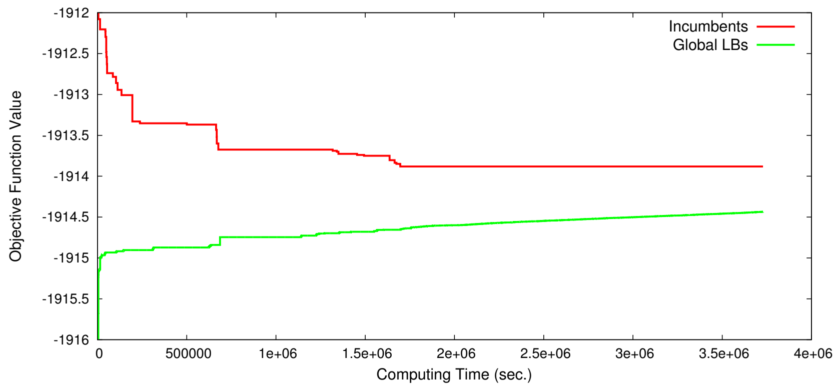

Figure 9: Lower and upper bounds evolution (`rmine10`).

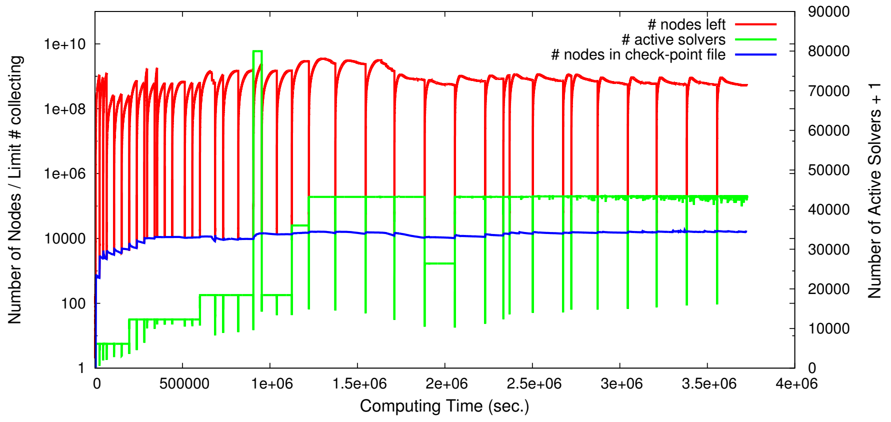

Figure 10: Active solvers and # of nodes left (`rmine10`).

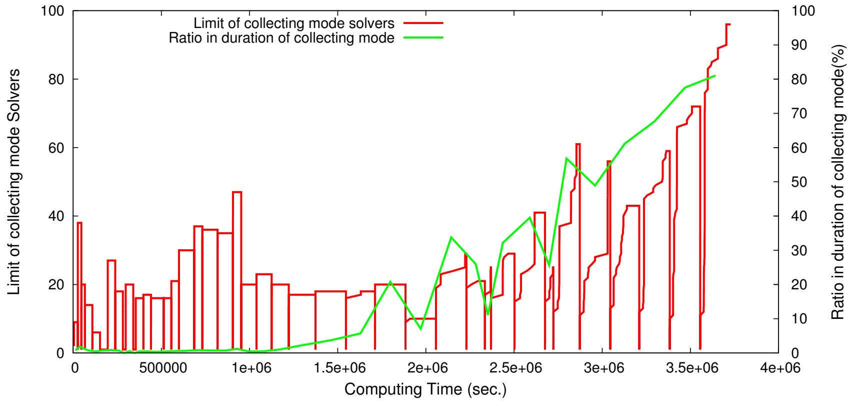

Figure 11: Runtime behavior of collecting mode (`rmine10`).

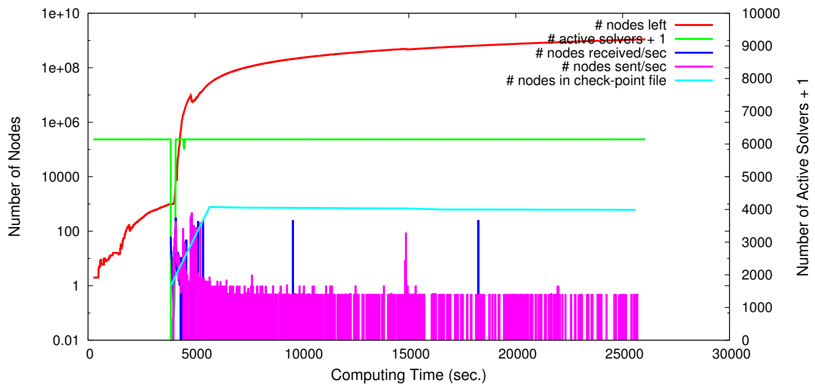

Figure 12: Active solvers and # of nodes left (Run 1: 6,144 cores).

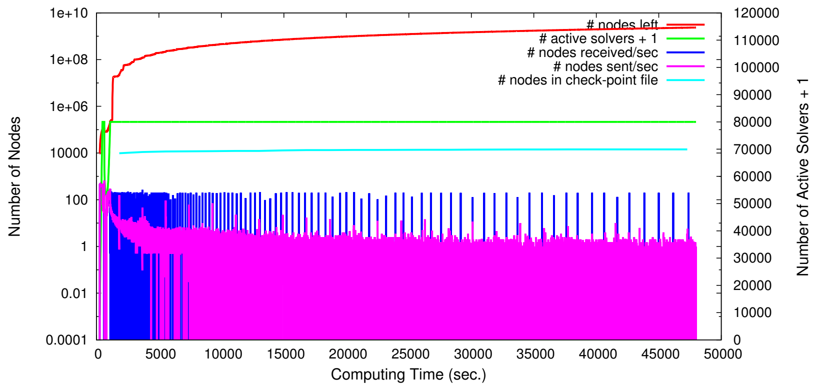

Figure 13: Active solvers and # of nodes left (Run 21: 80,000 cores).

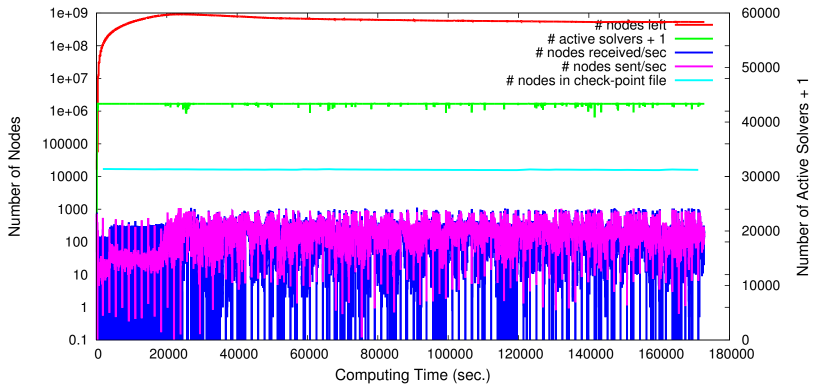

Figure 14: Active solvers and # of nodes left (Run 41: 43,344 cores).

had a two-layered LC of ParaSCIP. 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.

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 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.

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). The work has been supported by a Goggle 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] 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.

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

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

[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] 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.

[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.