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 hard MIPLIB2003 problems with ParaSCIP on Supercomputers: An update

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

ZIB-Report 13-66 (October 2013, revised version April 2014)

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 hard MIPLIB2003 problems with ParaSCIP on Supercomputers: An update

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

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

shinano@zib.de, achterberg@de.ibm.com, berthold@zib.de, heinz@zib.de, koch@zib.de, michael.winkler@zib.de

April 2, 2014

Abstract

Contemporary supercomputers can easily provide years of CPU time per wall-clock hour. One challenge of today's software development is how to harness this vast computing power in order to solve really hard mixed-integer programming instances. In 2010, two out of six open MIPLIB2003 instances could be solved by ParaSCIP in more than ten consecutive runs, restarting from checkpointing files. The contribution of this paper is threefold: For the first time, we present computational results of single runs for those two instances. Secondly, we provide improved upper and lower bounds for all of the remaining four open MIPLIB2003 instances. Finally, we explain which new developments led to these results and discuss the current progress of ParaSCIP. Experiments were conducted on HLRN II, on HLRN III, and on the Titan supercomputer, using up to 35,200 cores.

1 Introduction

SCIP [1, 2] is a state-of-the-art framework for solving *constraint integer programs* (CIP). CIP is formally defined as follows:

Definition 1 (constraint integer program)**.** A *constraint integer program* (CIP) is a tuple $(\mathfrak{C}, I, c)$ that encodes the task of solving

$$\min\{\langle c, x\rangle : \mathfrak{C}(x),\ x \in \mathbb{R}^n,\ x_I \in \mathbb{Z}^{|I|}\},$$

where $c \in \mathbb{R}^n$ is the objective function vector, $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ specifies the constraints $\mathcal{C}_j : \mathbb{R}^n \to \{0, 1\}$, $j \in [m]$, and $I \subseteq [n]$ specifies the set of variables that have to take integral

values. Further, a CIP has to fulfill the condition

$$\begin{aligned} \forall \hat{x}_I \in \mathbb{Z}^{|I|}\, \exists (A', b') \in \mathbb{R}^{k \times n} \times \mathbb{R}^k : \\ \{x \in \mathbb{R}^n : \mathfrak{C}(x), x_I = \hat{x}_I\} \\ = \{x \in \mathbb{R}^n : A'x \leq b'\}, \end{aligned} \tag{1}$$

where $k \in \mathbb{N}$.

Condition (1) states that the problem becomes a *linear program* (LP) when all integer variables are fixed. Thus, if the discrete variables are bounded, a CIP can be solved, in principle, by enumerating all values of the integral variables and solving the corresponding LPs.

A CIP for which all constraints are linear is a mixed-integer linear program (MIP):

Definition 2 (mixed-integer linear program)**.** A *mixed-integer linear program* (MIP) is given by a tuple (A, b, c, I) with matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, and a subset $I \subseteq [n]$. The task is to solve

$$\min\{\langle c, x\rangle : Ax \leq b,\ x_I \in \mathbb{Z}^{|I|}\}. \tag{2}$$

The *linear programming relaxation* is achieved by removing the integrality conditions. The solution of the relaxation provides a *lower (dual) bound* on the optimal solution value, whereas the objective function value of any feasible solution provides an *upper (primal) bound*.

At first glance, integer programming looks like a perfect candidate for parallelized algorithms. One main technique to solve MIPs is the branch-and-bound procedure. The idea of *branching* is to successively subdivide the given problem instance into smaller subproblems until the individual subproblems are easy to solve. The best of all solutions found in the subproblems yields the global optimum. During the course of the algorithm, a *branching tree* is generated in which each node represents one of the subproblems (sub-MIPs).

The intention of *bounding* is to avoid a complete enumeration of all potential solutions of the initial problem, which usually grow exponentially. For a minimization problem, the main observation is that if a subproblem's lower (dual) bound is greater than or equal to the global upper (primal) bound, the subproblem can be pruned. Lower bounds are calculated with the help of the linear programming relaxation, which typically is easy to solve. Upper bounds are obtained by feasible solutions, e.g., if the solution of the relaxation is also feasible for the corresponding subproblem and hence for the original problem.

Note that the above is a highly simplified presentation. State-of-the-art MIP solvers, such as SCIP 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 dual value obtained by the LP relaxation. It has to be emphasized that the most important ingredient to a parallel

MIP solver is a state-of-the-art sequential MIP solver. As shown in [3], simply increasing the computational power of an inferior implementation is not going to make much of a difference.

A second glance, parallelizing branch-and-bound algorithms for MIP has shown to be surprisingly difficult [4]. This is due to the fact that the decisions taken in different subtrees depend on each other, e.g., for history-based branching rules [5], and significant effort is spent before the tree search starts, e.g., in presolving and cut generation. Furthermore, basically all algorithmic improvements presented in the literature aim at reducing the size of the branching tree, thereby making a parallelization less effective and even more difficult. The latter is due to the observation, that they typically increase the need for communication and make the algorithm less predictable. Therefore, a well-designed dynamic load balancing mechanism is an essential part of parallelizing branch-and-bound algorithms.

The MIPLIB2003 [6], a standard test set for benchmarking MIP solvers, contained more than thirty unsolved instances when it was originally released. By the mutual effort of the research community, this number could be reduced to six; stalling at this level from 2007 to 2010. Since the release of MIPLIB2003, to the best of our knowledge, there had been only two successful attempts to solve open instances of MIPLIB2003 by large scale parallelization: GAMS/CPLEX/Condor by Bussieck et al. [7] who solved three instances of MIPLIB2003 by a grid computing approach and ParaSCIP. A predecessor work which was competitive to commercial sequential MIP solver was presented by Bixby et al. [8] in 1999, who solved two previously unsolved real-world MIP instances by running a branch-and-bound algorithm on eight parallel processors. In [9], we presented for the first time optimal solutions for two of the six extremely hard instances of MIPLIB2003: `ds` and `stp3d`. In order to solve these instances, runs needed to be restarted from a checkpoint more than 10 times. In this paper, we present computational results to solve these instances by single runs without restarting from checkpoints, and we present improved bounds for all of the remaining four unsolved instances.

The following sections are organized as follows. First, we briefly explain how ParaSCIP works (see more details in [9] and [10]). Next, three stages of ParaSCIP's development are presented to clarify what was changed for each stage. Then, the main computational results of this paper will be presented. After that, the current progress of ParaSCIP and an update of the bounds for the four unsolved instances of MIPLIB2003 will be given. We finish with some concluding remarks.

2 Ubiquity Generator Framework

ParaSCIP has been developed by using a software framework: the *Ubiquity Generator (UG) framework* [10]. UG is written in C++. Generally, parallel tree search based solvers have three running phases, see also [11] and [12]. The *ramp-up phase* is the time period from the beginning until sufficiently many branch-and-bound nodes are available to keep all processing units busy the first time. The *primary phase* is the period between the first and last time when all processing units were busy. The *ramp-down phase* is the time period

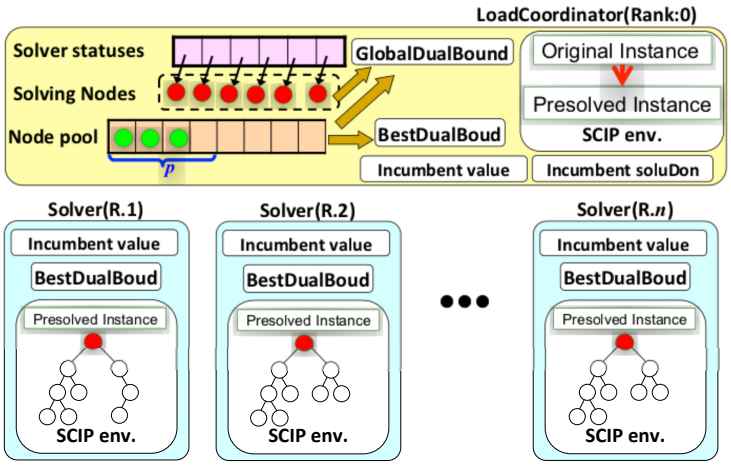

Figure 1: Process composition and data arrangement of ParaSCIP.

from the last time all processing units were busy until the problem is solved. UG provides several ramp-up mechanisms to shorten the ramp-up phase, a dynamic load balancing mechanism for all three phases, and a checkpointing and restarting mechanism as a generic functionality.

ParaSCIP uses the MPI library for communications. Two types of processes exist when running ParaSCIP on a supercomputer. One is the LoadCoordinator which makes all decisions concerning the dynamic load balancing, the other is the Solver which solves subproblems.

2.1 Initialization

The LoadCoordinator reads the instance data for a MIP model which we refer to as the *original instance*. This instance is presolved directly inside the LoadCoordinator. The resulting instance will be called the *presolved instance*. The presolved instance is broadcasted to all available Solver processes, and is embedded into the (local) SCIP environment of each Solver. This is the only time when the complete instance is transferred. Later, only the differences between a subproblem and the presolved problem will be communicated. At the end of this initialization, all Solvers are instantiated with the presolved instance (see Figure 1).

2.2 Ramp-up

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 subproblem using the presolved instance (which was distributed in the initialization step)

and the received bound changes.

UG has two kinds of ramp-up mechanisms.

Normal ramp-up Active SOLVERs, which are the ones that already received a branch-and-bound node, are solving this subproblem by alternately solving nodes and transferring half of the child nodes back to the LOADCOORDINATOR. The LOADCOORDINATOR has a *node pool* to keep unassigned nodes, from which it assigns nodes to idle SOLVERs as long as an idle SOLVER exists. Even if none exists, the LOADCOORDINATOR still 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 of UG, by default set to half of the number of SOLVERs. When the LOADCOORDINATORs node pool accumulated p "good" nodes, it sends a message to quit sending nodes to all SOLVERs.

Racing ramp-up In this mechanism, the LOADCOORDINATOR sends the root branch-and-bound node to all SOLVERs and all SOLVERs start solving the root node of the presolved instance immediately. In order to generate different search trees, each SOLVER uses a different variation of parameter settings, branching variable selections, and permutations of variables and constraints. As shown in [13], the latter can have a considerable impact on the performance of a solver. Due to these variations, we can expect that many SOLVERs independently generate different search trees. However, an incumbent solution found by one of the SOLVERs is broadcasted to all other SOLVERs and is used to prune parts of the SOLVERs' search trees. Under certain criteria, involving the duality gap and the number of open nodes of each SOLVER, a particular SOLVER is chosen as the "winner" of the *racing stage*. All open nodes of the "winner" are then collected by the LOADCOORDINATOR and a termination message is sent to all other SOLVERs. Next, the collected nodes are redistributed to all idle SOLVERs. If the "winner" did not provide enough nodes, UG changes its strategy to normal ramp-up.

We note, that racing ramp-up has already been considered in [14] and [15]. In order to use it with state-of-the-art MIP solvers on large-scale distributed computing environments, our extended implementation can switch to the second stage seamlessly, without waiting for all SOLVERs to terminate the first stage. Further, it can switch to normal ramp-up adaptively.

2.3 Dynamic load balancing

Periodically, each SOLVER notifies the LOADCOORDINATOR about the number of unexplored nodes in its SCIP environment and the dual bound of its subtree; we call this information the *solver status*. At the same time, the SOLVER is notified about the lowest dual bound value of all nodes in the node pool of the LOADCOORDINATOR, which we will refer to as BESTDUALBOUND.

If a SOLVER is idle and the LOADCOORDINATOR has unprocessed nodes available in the node pool, the LOADCOORDINATOR sends one of these nodes 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 [16]. We call a node *good*, if the dual bound value of its subtree (NODEDUALBOUND) is close to the dual bound value of the complete search tree (GLOBALDUALBOUND).

If a SOLVER receives the message to switch into the collecting mode, it changes the search strategy to either "best estimated value order" or "best bound order" (see [1]) depending on the specification of UG run-time parameters. It will then alternately solve nodes and transfer them to the LOADCOORDINATOR. This is done until the SOLVER receives the message to stop the collecting mode.

If the LOADCOORDINATOR is not in the collecting mode and detects that less than p good nodes with

$$\frac{\text{NODEDUALBOUND} - \text{GLOBALDUALBOUND}}{\max\{|\text{GLOBALDUALBOUND}|, 1.0\}} < \text{THRESHOLD} \tag{3}$$

are available in the node pool, the LOADCOORDINATOR switches to collecting mode and requests selected SOLVERs that have nodes that satisfy (3) to switch also to the collecting mode. The selection is done in ascending order of the minimum lower bound of open nodes in the SOLVERs. The number of selected SOLVERs increases dynamically depending on the time that the node pool in the LOADCOORDINATOR was empty. If the LOADCOORDINATOR is in collecting mode and the number of nodes in its pool that satisfy (3) is larger than $m_p \cdot p$, it requests all collecting mode SOLVERs to stop the collecting mode. Note that m_p is a runtime parameter and is set to 1.5 as a default value.

2.4 Termination

The termination phase starts when the LOADCOORDINATOR detects that 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 the statistics.

2.5 Checkpointing and restarting

ParaSCIP saves only *primitive* nodes, that is, nodes for which no ancestor nodes are in the LOADCOORDINATOR at each checkpoint. This strategy requires much less effort for the I/O system, even in large scale parallel computing environments. For restarting, however, it will take longer to recover the situation from the previous run. 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 dual bounds.

Note, that the number of nodes saved at checkpointing is not only depending on the node pool size in the LOADCOORDINATOR, but also on how the load balancing is performed and on the difficulties to solve each distributed sub-MIP. For example, if the LOADCOORDINATOR distributes sub-MIPs to all SOLVERs from successor sub-trees of the

root node and all sub-MIPs including one for the root node cannot be solved within the timelimit, the number of nodes saved is only one, that is, only the root node. In this case, in general, we cannot expect any progress of computation after the next restart and the following runs will not help to solve the instance.

3 Stages of ParaSCIP development

We had three stages of ParaSCIP development so far, and this paper presents computational experiments conducted over all three stages. In the first stage, ParaSCIP only featured normal ramp-up, checkpointing and restarting mechanism. Already that simple version of ParaSCIP solved two open instances from MIPLIB2003 by restarting more than ten times [9]. In the second stage, we tried to solve the two instances with a single execution without restarting in order to confirm the optimal solutions. To achieve this, we implemented the racing ramp-up. We will present computational results from a large-scale racing ramp-up in the following subsection. Note that we also tried to verify our results by giving the instances plus the optimal solutions to CPLEX 12.2. However, the run aborted without proof of optimality after one month of execution on a 32-core Sun Galaxy 4600 equipped with eight Quad-Core AMD Opteron 8384 processors at 2.7 GHz and 512 GB RAM. We conclude that these instances are still very hard to solve on a single computer, even for state-of-the-art commercial solvers with shared-memory parallelization and additional information (optimal solution given).

The computational results obtained by the first two stage systems motivated us to refine the *communication point*. A communication point is a function in a Solver, in which a series of information is exchanged between the LoadCoordinator and the Solver which is performed by message passing. Originally, the communication point was limited to the time when a branch-and-bound node was selected for processing. Therefore, there was no communication between the LoadCoordinator and a Solver during the processing of a branch-and-bound node. To allow for more frequent communication, we added additional communication points. These take place after every global and local variable bound change, every solution of the LP relaxation, every modification of the LP, or after finding a new incumbent. The third version development was done by using a shared memory parallel extension of SCIP. See [10] for more details about the modification in the third version.

This paper mainly presents new computational results for ParaSCIP in the second and the third version. Computational results of the first version can be found in [9].

4 Computational results

In this section, we present computational results for large-scale computations to solve the following hard instances from MIPLIB2003: `ds` and `stp3d`, that were solved by ParaSCIP for the first time in [9], and `dano3mip`, `t1717`, `liu` and `momentum3` are still open. For the computations, we used the following supercomputers.

HLRN II:[1] SGI Altix ICE 8200EX, Xeon QC E5472 3.0 GHz/X5570 2.93 GHz (up to 7,168 cores)
HLRN III:[2] Cray XC30, Xeon E5-2670 8C 2.6GHz, Aries interconnect (up to 17,088 cores)
Titan:[3] Cray XK7, Opteron 6274 16C 2.2GHz, Cray Gemini interconnect, NVIDIA K20x (up to 35,200 cores)

In this paper, we refer to the second version of ParaSCIP and SCIP 1.2.1.2 with CPLEX 12.1 as underlying linear programming solver as the "*old version of ParaSCIP*" and refer to the third version of ParaSCIP and SCIP 3.0.1 with CPLEX 12.5 as underlying linear programming solver as the "*new version of ParaSCIP*".

4.0.1 Single run computations for `ds` and `stp3d`

In this subsection, we present computational results for large-scale single job computations to solve `ds` and `stp3d`. For these two instances, we conducted the following computations:

`ds` Original instance data was used. This instance was solved from scratch on HLRN II with 4,096 cores.

`stp3d` Extended presolved instance data was used (see [9]). This instance was solved from a given optimal solution. This instance was solved on HLRN II with 4,096 cores using racing and normal ramp-ups. This instance was also solved on HLRN II with 7,168 cores using racing ramp-up.

At first, we summarize the results of the racing stage computations. After the racing finished, we determined a winner, the Solver whose dual bound is the largest in all Solvers which have above a certain threshold number of open nodes. The threshold numbers of open nodes were 4,000 and 500 for solving `ds` and `stp3d`, respectively.

To generate thousands of settings that can be run concurrently, we applied three different diversification strategies:

Meta parameter settings Presolving, primal heuristics, and cutting plane separation are crucial components of MIP solving. In SCIP, there are certain meta parameter settings for each of these components. For each racing parameter setting, one combination of "aggressive(0)", "default(1)", "fast(2)" and "off(3)" meta-parameter settings related to "Heuristics(h)", "Presolving(p)" and "Separation(s)" was set. The numbers and characters within parentheses indicate the settings for a single solver, e.g., "h0p1s2" shows that aggressive settings for heuristics, default settings for presolving and fast settings for separation have been used.

Branching variables The choice of the branching variables in the top levels of the branch-and-bound tree heavily influences the performance of a solver. To diversify, we select the branching variables at the first two levels of the branch-and-bound tree randomly.

[1] http://www.hlrn.de
[2] http://www.hlrn.de
[3] http://www.olcf.ornl.gov/titan/

Permutations of columns The columns and rows of instance data are permuted to exploit performance variability, see e.g. [17].

We applied all of these strategies simultaneously. For each parameter setting, the pictures in Figure 2 show the "root node computing time at racing stage", the "elapsed time to terminate racing computation", the "bound evolution after the racing stage" and the "number of open nodes after the racing stage" for computations of `ds` with 4,096 cores. In all figures, the results are grouped by identical and similar meta parameter settings. The order on the x-axis is the same for all figures. We only present the picture of the "elapsed time to terminate racing computation" for the `stp3d` instance in Figure 3, because its characteristics are similar to those for the `ds` instance.

The top picture of Figure 2 shows how the computing time for the root node varies depending on parameter settings and shows that the behavior within each class of meta-parameter settings is quite homogeneous. The results still vary due to the permutation of columns.

The second picture in Figure 2 shows the number of open nodes that each SOLVER has at termination time of the racing stage. The third picture in Figure 2 shows the dual bound value that each SOLVER has at the termination time of the racing stage. The number of open nodes could be weakly classified depending on meta-parameter settings, but the values do not have recognizable patterns.

The bottom picture in Figure 2 shows the termination time for each racing stage SOLVER. After the winner is determined, the LOADCOORDINATOR sends a message to all SOLVERs, except the winner, to request termination of the racing stage. The picture shows that most of the SOLVERs terminated immediately, but some of them took a long time to terminate. An explanation might be that for this version of ParaSCIP the messages were only checked when the SOLVER starts solving a branching node. If a branching node computation takes extremely long time, the SOLVER has no chance to communicate with the LOADCOORDINATOR. This means that these might be SOLVERs which take long to terminate. Figure 3 shows the results for `stp3d`. `stp3d` is a very large instance, therefore, the node LPs often take a long time to solve. In fact, therefore, more SOLVERs take long to terminate. These results motivated us to introduce further communication points.

Note, that the winner's data are missing for all pictures, because the winner keeps running after the racing stage. The winner's meta-parameter settings were as follows:

- `ds` (4,096) h3p2s3: no heuristics, fast presolving, no separator.
- `stp3d` (4,096) h2p3s2: fast heuristics, no presolving, fast separation.
- `stp3d` (7,168) h2p3s2: fast heuristics, no presolving, fast separation.

Next, we present computational results for the large-scale single jobs. All computation times for the jobs are shown in the top four rows of Table 1. The numbers of nodes solved and transferred are presented in the top four rows of Table 2. For the racing stage, the number of nodes counts only the nodes of the winner. Table 2 shows that the number of transferred nodes between SOLVERs via the LOADCOORDINATOR is very small compared to

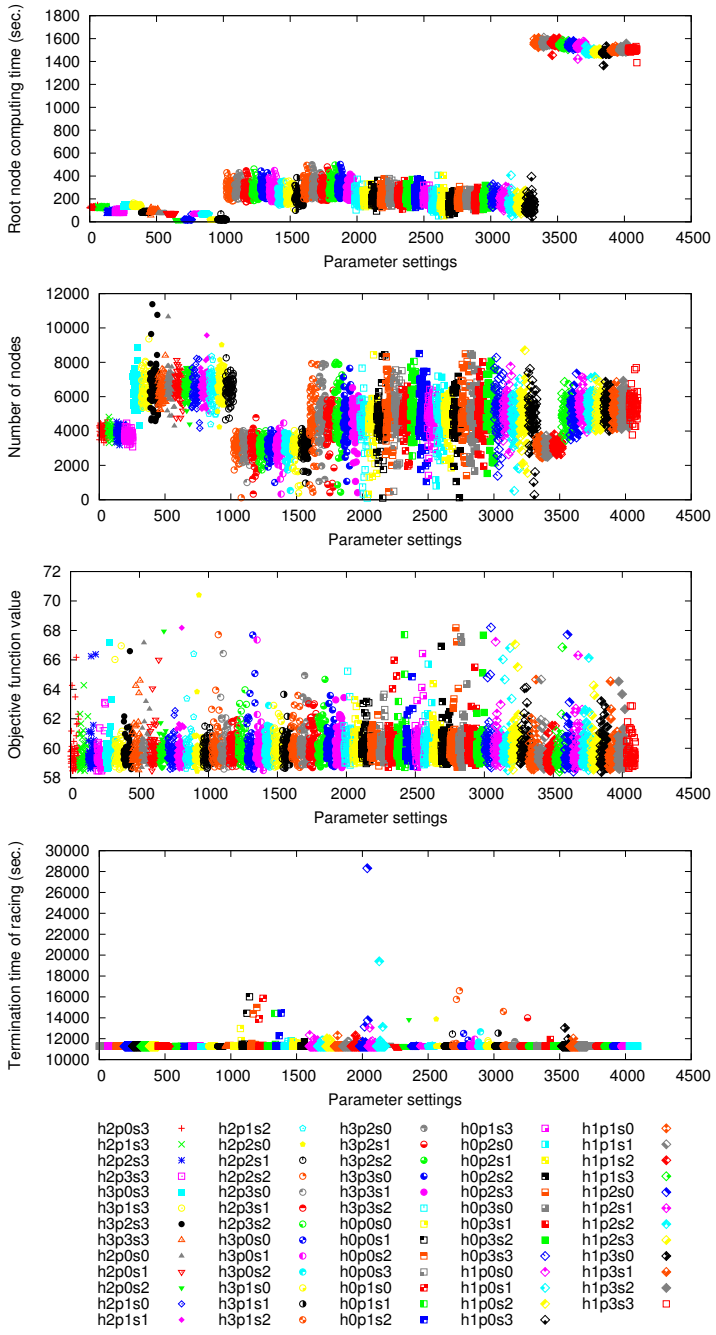

Figure 2: Results after racing stage for `ds`. From the top to bottom as follows: Root node computation time, the number of open nodes, bounds evolution and elapsed time to terminate.

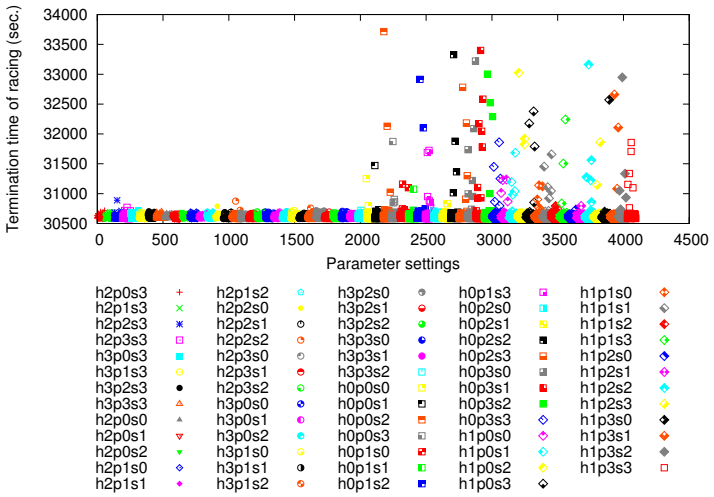

Figure 3: Elapsed time to terminate racing computation for `stp3d`

the total number of nodes solved in our load balancing mechanism. The transferred nodes ratio is less than 10% for all computations. Figures from 4 to 6 show how upper and lower bounds evolve, the solver usage and workloads during computation, and the computation time and idle time ratios of SOLVERs, for each of the large-scale jobs. These figures show that `ds` was solved quite well in terms of CPU cores usage. Figure 5 shows all SOLVERs were busy during most of the computing time. Actually, the maximum idle time ratio in all SOLVERs was at most 10% from that of Figure 6.

For `stp3d` starting with racing ramp-up, ParaSCIP achieved 1.34 times speedups by 1.75 scale ups, see Table 1. Although the ramp-up time in racing ramp-up was quite long, the computation time of the run with racing ramp-up was at least 18 hours shorter than that of normal ramp-up in the end. The number of nodes solved is similar for all `stp3d` runs. Therefore, the difference between normal and racing ramp-ups with 4,096 cores comes from the idle time of SOLVERs. Though, it is not clear whether a good load balancing in the beginning of computation with racing ramp-up leads to a good situation in the ramp-down phase or not, the behavior of the result of 7,168 cores with racing ramp-up is very similar to that of 4,096 cores.

4.0.2 Restarted runs vs. single run computation for solving `ds`

We calculated cores×hours for solving `ds` with 16 times restarted runs (17 jobs) obtained in [9] and the result in this paper.

Restarted: 181,248 [cores×hours]

Single: 310,791 [cores×hours]

Note, that cores×hours for restarted runs above is bigger than that calculated from the results presented in [9], because the above number includes the computing time after the

Table 1: Over all computation time of the large scale single jobs. (Top four rows: old version, bottom two rows: new version)

Instance Name	Ramp-up Process	# of cores	Comp. (sec.)	Racing (sec.)	Ramp-up (sec.)	Ramp-down (sec.)
ds	Racing	4,096	273,157	11,278	28,318	5,437
stp3d	Racing	4,096	158,595	30,580	33,716	5,193
stp3d	Normal	4,096	225,577	-	5,486	77,490
stp3d	Racing	7,168	118,386	30,075	33,146	6,628
ds	Racing	4,096	29,095	3,600	3,847	27,110
ds	Normal	4,096	31,798	-	477	28,360

Table 2: Number of nodes solved and transferred of the large scale single jobs. (Top four rows: old version, bottom two rows: new version)

Instance Name	Ramp-up Process	# of cores	# of nodes solved	# of nodes transferred	Transferred nodes (%)
ds	Racing	4,096	3,010,465,526	43,334,558	1.4
stp3d	Racing	4,096	9,779,864	434,344	4.4
stp3d	Normal	4,096	10,573,696	880,850	8.3
stp3d	Racing	7,168	10,328,112	549,226	5.3
ds	Racing	4,096	466,021,554	825,373	0.2
ds	Normal	4,096	554,331,872	926,212	0.2

final checkpoint for each run. Despite this fact, the restarted runs were more efficient in solving ds from the computing resource usage point of view (58% of the resources), although the restarted runs threw away a huge branch-and-bound tree for each run. A single run needs to keep running until the instance is solved. Therefore, parameter settings of each Solver were conservative in terms of memory usage, that is more frequently the search strategy was changed to depth-first search.

4.0.3 Old version vs. new version for solving ds

We conducted computational experiments for solving ds every time when a new SCIP version was released. They always brought us new insights, because the run time behavior of SCIP changed a lot. New versions of SCIP make much more effort to keep the search tree small compared to old versions. As a consequence, the parameter settings for ParaSCIP had to be changed. For example, the threshold number of open nodes to terminate racing stage had been reduced to 500.

Computation times for racing and normal ramp-up for single runs are shown in the bottom two rows of Table 1. The new version solved 9.4 times faster for racing ramp-up and 8.6 times faster for normal ramp-up compared to the old version for racing ramp-up. The number of nodes solved and transferred are presented in the bottom two rows of Table 2. Table 2 shows that the number of transferred nodes between Solvers via LoadCoordinator is rather small compared to that of the old version. The transferred nodes ratio is less than 0.2% in both cases. Figures from 7 to 9 show how the upper and lower bounds evolve, the solver usage and the workloads during computation, and the computation time and the idle time ratio of Solvers, for the large-scale job of racing ramp-up. Again, ds was solved quite well in terms of CPU cores usage. The average idle time ratio of all solvers for the old version is 4.3% and that of the new version of racing ramp-up is 6.8% and that of normal ramp-up is 11.1%. The reason why the ratios increased is that ds became easier to solve for the new version of SCIP. As a consequence, a more dynamic load balancing was needed. Figure 8 shows that ramp-up finished soon after the racing stage finished, which is the effect of the additional communication points.

5 Current progress of ParaSCIP

In the previous subsection 4.0.3, we showed the performance difference between an old version and a new version of ParaSCIP for solving ds. A main source for the improvement is the performance improvement of SCIP. However, there have been several enhancements of ParaSCIP as well. Since SCIP tends to produce a small search tree, we need to assume that after the racing stage the number of open nodes is less than the number of Solvers in large-scale runs. This means that the dynamic load balancing becomes more important.

For dynamic load balancing, the biggest improvement is the more frequent call of communication points in the Solvers. This enables ParaSCIP to realize fast and flexible control of the dynamic load balancing. The new version of ParaSCIP uses as few Solvers

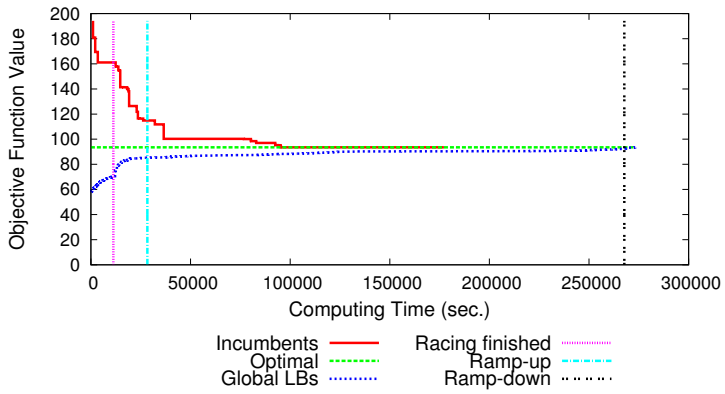

Figure 4: Lower and upper bounds evolution of old version: `ds`

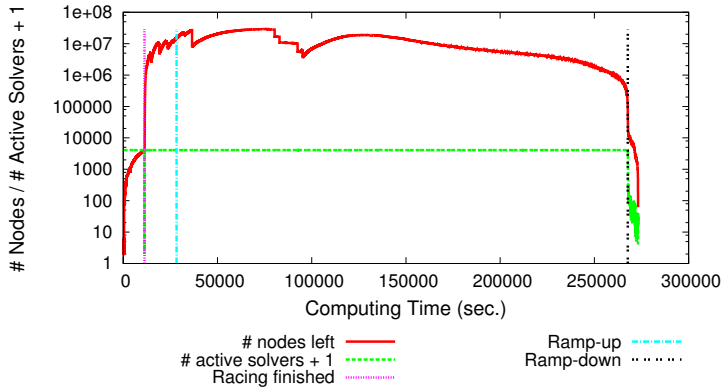

Figure 5: Active solvers and the number of nodes left of old version: `ds`

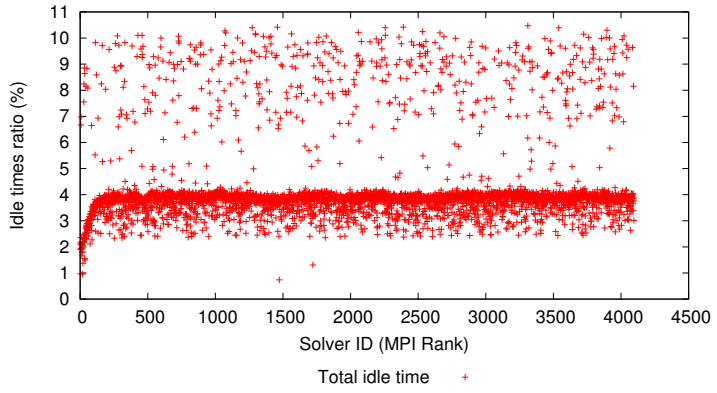

Figure 6: Idle time ratio of old version: `ds`

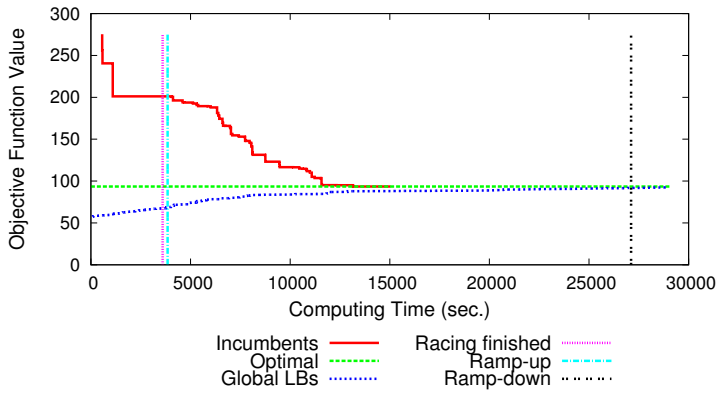

Figure 7: Lower and upper bounds evolution of new version: `ds`

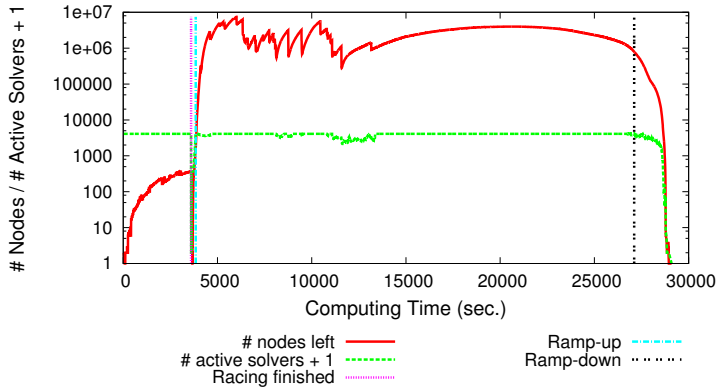

Figure 8: Active solvers and the number of nodes left of new version: `ds`

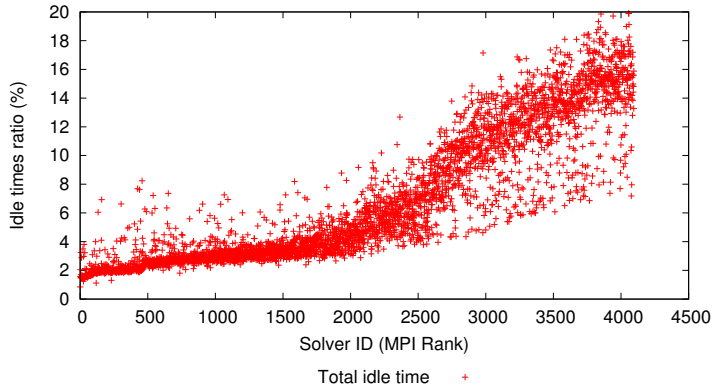

Figure 9: Idle time ratio of new version: `ds`

as possible in the collecting mode and switches the collecting mode SOLVERs frequently. Actually, the number of collecting mode SOLVERs is less than 50 in most of the computing time for solving ds. Figure 9 shows that it was controlled well, because small-rank SOLVERs assigned a new branch-and-bound node first, and we can observe reasonable idle time ratio of SOLVERs for the assignment order.

ParaSCIP can solve ds with racing and normal ramp-up without too much difference in computing time, because of the improvement of the dynamic load balancing. Racing ramp-up itself seems to be good for load balancing according to our computational experiments. The reason might be that the run with racing ramp-up collects all nodes from the winner once and redistributes them to all SOLVERs. At this moment, it uses the whole branch-and-bound tree for load balancing. Table 1 shows that racing ramp-up leads to shorter ramp-down times.

We have conducted experiments for the remaining open instances. Currently our strategy is to generate a checkpointing file which contains good quality sub-MIPs. Here, we exploited the possibility to generate a first checkpointing file of promising sub-MIPs on one supercomputer and move it to another supercomputer for the final run. In order to generate the checkpointing file, we used best dual bound first search, but for the large-scale computation on supercomputers, we have changed the search strategy to default to obtain a smaller memory footprint for each SOLVER. For the computation on supercomputers, the best known primal solution of each instance was given. We present two results for solving dano3mip on HLRN III by using 17,088 cores starting from 33,332 branch-and-bound nodes in the checkpointing file and on Titan by using 35,200 cores starting from 33,481 nodes. The primal and the dual bounds have not been improved in either case. Figures 10 and 11 show the results of SOLVER usage, workloads and how many number of nodes were transferred for the two runs. In Figure 10, 17,088 cores are not enough to solve dano3mip by using current ParaSCIP, because LOADCOORDINATOR did not receive any branch-and-bound node from SOLVER and the checkpointing file size stayed the same after the computation. However, Figure 11 shows that ParaSCIP went into the collecting mode and the number of nodes in the checkpointing file was reduced to 32,023 for the run with 35,200 cores. Although ParaSCIP only has one LOADCOORDINATOR, it could handle 35,200 cores and it might solve dano3mip by using such large-scale distributed memory computing environments, or even a bigger one, in the future.

Finally, we present an update of primal solutions from [18] and summarize the best known lower bounds of the remaining open instances in Table 3 and Table 4 ('*' indicates our update), respectively[4].

6 Concluding remarks

This paper presented an update of the results on MIPLIB2003 and the progress of ParaSCIP. As we have shown, it is very difficult to evaluate the performance of a parallel MIP solver

[4]The update of momentum3 was mainly done by ParaCPLEX which is a parallel MIP solver realized by Ubiquity Generator Framework replacing SCIP by CPLEX.

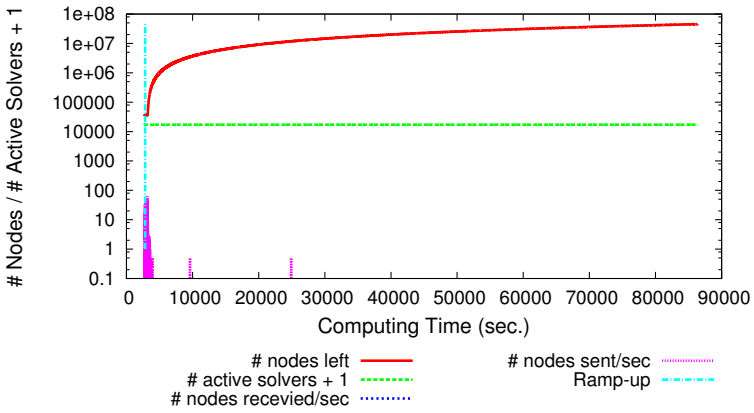

Figure 10: Active solvers, the number of nodes left, nodes transferred per seconds on HLRN III (17,088 cores) : `dano3mip`

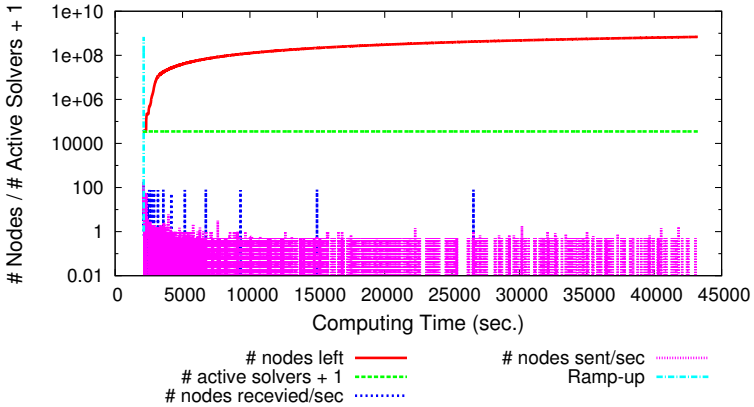

Figure 11: Active solvers, the number of nodes left, nodes transferred per seconds on Titan (35,200 cores) : `dano3mip`

Table 3: Improved objective value of all remaining MIPLIB2003 unsolved problems

Instance Name	Old best-known obj. value	New improved obj. value	Gain (%)
dano3mip	687.733333	665.571428571428	3.2
liu	1,102	1,088	1.3
momentum3	236,426.335	185,509.733224132	21.5
t1717	170,195	161,330	5.2

Table 4: Best-Known Lower and Upper Bounds of all remaining MIPLIB2003 unsolved problems. Gap is calculated by $|1 - (\text{upper bound}/\text{lower bound})|$

Instance Name	Lower Bound	Upper Bound	Gap (%)
dano3mip	581.9224*	665.571428571428*	14.4
liu	613	1,088*	77.5
momentum3	95,217.2220*	185,509.733224132*	94.8
t1717	139,790.5882*	161,330*	15.4

regarding hard instances on a large-scale distributed memory computing environment. One way is to show that well-known hard instances like ds can be solved. As can be observed from the results on solving ds with different versions of SCIP, it is crucial to exploit a powerful MIP solver in parallelization in order to use computing resources efficiently. Furthermore, we have shown that for hard MIP instances, ParaSCIP could handle more than 35,000 cores. Some questions remain open: Up to which number of cores can ParaSCIP reasonably scale? How many SOLVERs will be necessary to solve the remaining open instances? And which of these two numbers is bigger?

Acknowledgment

We are thankful to the HRLN II and HRLN III supercomputer staff, especially Bernd Kallies, Hinnerk Stüben and Matthias Läuter who gave us support at any time we needed it. We would like to thank Michael R. Hilliard and Rebecca J. Hartman-Baker for their help to access Titan.

References

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

[2] ——, "Scip: Solving constraint integer programs," *Math. Prog. Comp.*, vol. 1, no. 1, pp. 1–41, 2009.

[3] T. Koch, A. Martin, and M. E. Pfetsch, "Progress in academic computational integer programming," in *Facets of Combinatorial Optimization*, M. Jünger and G. Reinelt, Eds. Springer, 2013, pp. 483–506.

[4] T. Ralphs, L. Ladnyi, and M. Saltzman, "Parallel branch, cut, and price for large-scale discrete optimization," *Mathematical Programming*, vol. 98, no. 1-3, pp. 253–280, 2003.

[5] T. Achterberg and T. Berthold, "Hybrid branching," in *Proceedings of the CPAIOR 2009*, ser. LNCS, W. J. van Hoeve and J. N. Hooker, Eds., vol. 5547. Springer, May 2009, pp. 309–311.

[6] T. Achterberg, T. Koch, and A. Martin, "MIPLIB 2003," *ORL*, vol. 34, no. 4, pp. 1–12, 2006.

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

[8] R. E. Bixby, W. Cook, A. Cox, and E. K. Lee, "Computational experience with parallel mixed integer programming in a distributed environment," *AoOR*, no. 90, pp. 19–43, 1999.

[9] 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.

[10] 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.

[11] T. K. Ralphs, "Parallel branch and cut," in *PARALLEL COMBINATORIAL OPTIMIZATION*. Wiley, 2006, pp. 53–101.

[12] Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzman, "Computational experience with a software framework for parallel integer programming," *The INFORMS Journal on Computing*, vol. 21, pp. 383–397, 2009.

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

[14] G. Mitra, I. Hai, and M. Hajian, "A distributed processing algorithm for solving integer programs using a cluster of workstations," *Parallel Computing*, vol. 23, no. 6, pp. 733 – 753, 1997.

[15] V. Nwana, K. Darby-Dowman, and G. Mitra, "A two-stage parallel branch and bound algorithm for mixed integer programs," *IMA JoMM*, vol. 15, no. 3, pp. 227–242, 2004.

[16] 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.

[17] M. Fischetti and M. Monaci, "Exploiting erraticism in search," Technical report of University of Padova, Tech. Rep., 2012.

[18] R. Laundy, M. Perregaard, G. Tavares, H. Tipi, and A. Vazacopoulos, "Solving hard mixed-integer programming problems with XPress-MP: A MIPLIB 2003 case study," *IJoC*, vol. 21, no. 2, pp. 304–313, 2009.