Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

YUJI SHINANO
TOBIAS ACHTERBERG*
TIMO BERTHOLD**
STEFAN HEINZ**
THORSTEN KOCH

ParaSCIP – a parallel extension of SCIP

* ILOG, on IBM Deutschland GmbH, Ober-Eschbacher Str. 109, 61352 Bad Homburg v.d.H., Germany
** Supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin.

ZIB-Report 10-27 (December 2010)

ParaSCIP – a parallel extension of SCIP*

Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch

Abstract *Mixed integer programming (MIP)* has become one of the most important techniques in Operations Research and Discrete Optimization. SCIP (Solving Constraint Integer Programs) is currently one of the fastest non-commercial MIP solvers. It is based on the *branch-and-bound* procedure in which the problem is recursively split into smaller subproblems, thereby creating a so-called *branching tree*. We present ParaSCIP, an extension of SCIP, which realizes a parallelization on a distributed memory computing environment. ParaSCIP uses SCIP solvers as independently running processes to solve subproblems (nodes of the branching tree) locally. This makes the parallelization development independent of the SCIP development. Thus, ParaSCIP directly profits from any algorithmic progress in future versions of SCIP. Using a first implementation of ParaSCIP, we were able to solve two previously unsolved instances from MIPLIB2003, a standard test set library for MIP solvers. For these computations, we used up to 2048 cores of the HLRN II supercomputer.

1 Introduction

Branch-and-bound is a very general and widely used method to solve discrete optimization problems. An important class of problems which can be solved using this method are *mixed integer programs (MIP)*. The challenge of these problems is to find a feasible assignment to a set of decision variables which yields a mini-

Yuji Shinano · Timo Berthold · Stefan Heinz · Thorsten Koch
Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
e-mail: {shinano, berthold, heinz, koch}@zib.de

Tobias Achterberg
ILOG, on IBM Deutschland GmbH, Ober-Eschbacher Str. 109, 61352 Bad Homburg v.d.H., Germany, e-mail: achterberg@de.ibm.com

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

mum/maximum value with respect to a given linear objective function. The feasible region for these problems is described by linear inequalities. In addition a subset of the variables are only allowed to take integer values. These problems are *NP*-hard in general [12].

The well-known idea of branching is to successively subdivide the given problem instance into smaller problems until the individual subproblems (or sub-MIPs) are easy to solve. During the course of the algorithm, a branching tree is generated in which each node represents one of the subproblems. To be able to prune the vast majority of nodes at an early stage, sophisticated mathematical techniques are used. This allows a dramatic reduction of the size of the branching tree. Typically, problems with ten thousand variables and constraints (i.e., approximately 2^{10000} potential solutions) can be solved by investigating a few hundred thousand branch-and-bound nodes.

State-of-the-art MIP solvers such as CPLEX [3], Gurobi [1], or SCIP [8] are based on a branch-and-cut [15] procedure, a mathematically involved variant of branch-and-bound. Parallelizing branch-and-cut algorithms has been proven to be difficult, due to fact that the decisions involved depend on each other [16]. State-of-the-art codes *learn* from the decisions already taken, assuming a sequential ordering. 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 the parallelizing branch-and-cut algorithms.

Since its introduction in 1992, the MIPLIB [11] has become a standard test set library used to compare the performance of MIP solvers. The MIPLIB contains a collection of difficult real-world instances mostly from industrial applications. Its availability has provided an important stimulus for researchers in this active area. The current version, MIPLIB2003 [9, 4], contains more than thirty unsolved instances when it was originally released. This number could be reduced to six; stalling at this level since 2007. These six instances resisted all attempts of the commercial vendors and the research community to solve them to proven optimality.

Algorithmic improvements for state-of-the-art sequential MIP solvers have been tremendous during the last two decades [10]. For an overview on large scale parallelization of MIP solvers, see [18]. Most of these approaches struggled, however, to catch up with the performance of state-of-the-art commercial and non-commercial sequential MIP solvers when it comes to solving really hard MIP instances of general nature. Many unsolved instances of MIPLIB2003 were first solved using sequential solvers [13].

In the following we describe how we developed a massive parallel distributed memory version of the MIP solver SCIP [5] to harness the power of the HLRN II supercomputer [2] in order to solve two of the remaining open instances of the MIPLIB 2003.

2 SCIP– Solving Constraint Integer Programs

SCIP (Solving Constraint Integer Programs) is a framework for constraint integer programming. Constraint integer programming is an extension of MIP and a special case of the general idea of *constraint programming (CP)*. The goal of SCIP is to combine the advantages and compensate the weaknesses of CP and MIP.

An important point for the efficiency of MIP and CP solving algorithms is the interaction between constraints. SCIP provides two main communication interfaces:

(i) propagation of the variables' domains as in CP and
(ii) the linear programming relaxation as in MIP.

SCIP uses a branch-and-bound scheme to solve constraint integer programs (see Section 2.2). The framework is currently one of the fastest MIP solvers [14], even so it is suitable for a much richer class of problems. For more details about SCIP we refer to [7, 8, 5].

2.1 Mixed integer programs

In this paper, we only focus on mixed integer programs (MIPs), which can be defined as follows:

Definition 1 (mixed integer program). Let $\hat{\mathbb{R}} := \mathbb{R} \cup \{\pm\infty\}$. Given a matrix $A \in \mathbb{R}^{m\times n}$, a right-hand-side vector $b \in \mathbb{R}^m$, an objective function vector $c \in \mathbb{R}^n$, a lower and an upper bound vector $l, u \in \hat{\mathbb{R}}^n$ and a subset $I \subseteq N = \{1,\ldots,n\}$, the corresponding *mixed integer program* $\text{MIP} = (A, b, c, l, u, I)$ is to solve

$$\begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Ax \leq b & \\ & l \leq x \leq u & \\ & x_j \in \mathbb{R} & \text{for all } j \in N \setminus I \\ & x_j \in \mathbb{Z} & \text{for all } j \in I. \end{array}$$

The goal is to find an assignment to the (decision) variables x such that all linear constraints are satisfied and the objective function $c^T x$ is minimized. Note that, the above format is quite general. First, maximization problems can be transformed to minimization problems by multiplying the objective function coefficients by -1. Similarly, "$\geq$" constraints can be multiplied by -1 to obtain "$\leq$" constraints. Equations can be replaced by two opposite inequalities.

The *linear programming relaxation* is achieved by removing the integrality conditions. The solution of the relaxation provides a *lower bound* on the optimal solution value.

2.2 Branch-and-bound

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.

The intention of *bounding* is to avoid a complete enumeration of all potential solutions of the initial problem, which usually are exponentially many. For a minimization problem, the main observation is that if a subproblem's lower (dual) bound is greater than 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, found, e.g., if the solution of the relaxation is also feasible for the corresponding subproblem.

3 ParaSCIP

In this section, we introduce ParaSCIP, a parallel extension of SCIP. The design goals of ParaSCIP are to exploit SCIP's complete functionality, to keep the interface simple, and to scale to at least 10 000 cores in parallel.

We will focus on two important features, the dynamic load balancing and the checkpointing mechanism.

3.1 A dynamic load balancing mechanism

In this section we illustrate the workflow of the dynamic load balancing mechanism for ParaSCIP. Workload of a sub-MIP computation strongly depends on two factors. One is the number of branching nodes per solver, which may vary from one to several millions. The other is the computing time of a single branch-and-bound node, which may vary from less than one millisecond to several hours. Therefore, the dynamic load balancing mechanism is a key factor for the parallelization of branch-and-bound algorithms.

Initialization phase

In the beginning, the LOADCOORDINATOR, which acts as a master process, reads the instance data for a MIP model which we refer to as the *original instance*. This instance is presolved (see Section 4.2) directly inside the LOADCOORDINATOR.

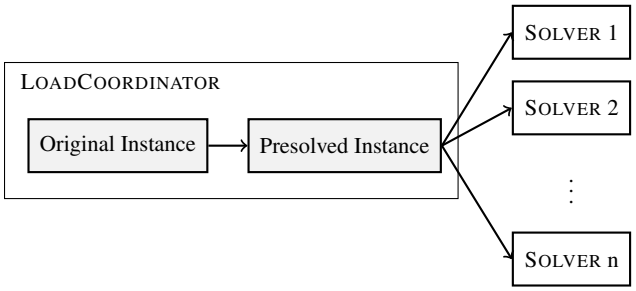

Fig. 1 Initialization phase

MIP presolving tries to fix variables and to detect redundancy of certain constraints, for details see [7]. The resulting, typically quite smaller, instance will be called the *presolved instance*. The presolved instance is extracted from the SCIP environment, broadcasted to all available Solver processes, and embedded into the (local) SCIP environment of each Solver process. This is the only time when the complete instance is transferred. Later, only the differences between a subproblem and the presolved problem will be sent. Figure 1 illustrates this initialization procedure. At the end of this phase all Solvers are instantiated with the presolved instance.

Transferring branch-and-bound nodes

After the initialization step, the LoadCoordinator creates the root node of the branch-and-bound tree. Each node transferred through the system acts as the root of a subtree. The information that has to be sent consists only of bound changes for variables between the presolved instance and the subproblem, which gets transferred. For the initial root node there is no difference between the presolved instance and the subproblem.

All nodes, which are transferred to Solvers, are kept in the LoadCoordinator with their solver statuses until the corresponding solving process terminates. The Solver which receives a new branch-and-bound node instantiates that subproblem using the presolved instance (which was distributed in the initialization phase) and the received bound changes. After that, the Solver starts working on the subproblem.

Load balancing

Load balancing for MIP solving highly depends on the *primal* and *dual bounds*, which are updated during the solving process. The primal bound is given by the value of the best solution that has been found so far during the solving process. If one of the Solvers finds an improved solution, this solution is sent to the Load-

COORDINATOR, which distributes the updated primal bound to all other SOLVERs. If a SOLVER receives an improved primal bound, it will immediately apply bounding, hence, prune all nodes in its search tree that cannot contain any better solution anymore.

Periodically, each SOLVER notifies the LOADCOORDINATOR about the number of unexplored nodes in its SCIP environment and the dual bound of its subtree which define the *solver status*. The dual bound is a proven lower bound on the value of the best solution in that subtree. It is derived from the linear programming relaxation (see Section 2.1) at the individual nodes. At the same time the SOLVER is notified about the best dual bound value of all nodes in the node pool of the LOADCOORDINATOR, which we will refer to as BESTDUALBOUND.Note that this does not include nodes that are currently processed by any SOLVER.

If a SOLVER is idle and the LOADCOORDINATOR has unprocessed nodes available in the node pool, then the LOADCOORDINATOR sends one of these nodes to the idle SOLVER.

To handle the situation that several solvers become idle at the same time, the LOADCOORDINATOR should always have a sufficient amount of unprocessed nodes left in its node pool. This ensures that the SOLVERs are kept busy throughout the computation.

In order to keep at least p "good" nodes in the LOADCOORDINATOR, we introduce the *collecting mode*, similar to the one introduced in [17]. 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).

Consider the case that the LOADCOORDINATOR is not in collecting mode, and it detects that less than p good nodes with

$$\frac{\text{NODEDUALBOUND} - \text{GLOBALDUALBOUND}}{\max\{|\text{GLOBALDUALBOUND}|, 1.0\}} < \text{THRESHOLD} \qquad (1)$$

are available in the node pool, the LOADCOORDINATOR switches to collecting mode and requests selected SOLVERs that have nodes which satisfy (1) to switch also into collecting mode. If the LOADCOORDINATOR is in collecting mode and the number of nodes in its pool that satisfy (1) is larger than $m_p \cdot p$, it requests all collecting mode SOLVERs to stop the collecting mode.

If a SOLVER receives the message to switch into collecting mode, it changes the search strategy to either "best estimate value order" or "best bound order" (see [7]). It will then alternately solve nodes and transfer them to the LOADCOORDINATOR. This is done until the SOLVER receives the message to switch out of the collecting mode. If a node of the branch-and-bound tree is selected to be sent to the LOADCOORDINATOR, the corresponding SOLVER collects the bound changes of that node w.r.t. the presolved instance, transfers the differing bounds to the LOADCOORDINATOR, and prunes the node from the subproblem's branch-and-bound tree.

In the context of parallel branch-and-bound, the process until all SOLVERs are busy is called *ramp-up phase* [18]. In the ramp-up phase, all SOLVERs run in col-

lecting mode. The ramp-up phase continues until the number of nodes in the node pool of the LOADCOORDINATOR is greater than the value p.

The most crucial issue for the load balancing mechanism is to avoid solving useless subproblems. Consider the situation that a SOLVER is solving a subproblem for which the dual bound is already quite large. Then, an improvement in the primal bound will cause all nodes of the subproblem to be pruned.

The SOLVER can detect this situation locally using the best dual bound value of all nodes in the node pool of the LOADCOORDINATOR (BESTDUALBOUND). In this situation, the SOLVER requests another node from the LOADCOORDINATOR while still continuing to solve the current node. After the LOADCOORDINATOR sent a new node to the SOLVER and restored the old solving node in its node pool, the SOLVER stops the solution process and restarts with the new node. The solution of the old node is "delayed". Note that in case that there is no node available in the LOADCOORDINATOR, the SOLVER keeps continuing to solve the old node.

The termination phase is started after the LOADCOORDINATOR detects that the node pool is empty and all SOLVERs are idle.

3.2 Checkpointing and restarting

Checkpointing mechanisms are a common concept in parallel computing to protect a code against hardware and software failures. The chance that a compute node crashes within a given time frame increases with the number of compute nodes in the parallel system. Further, it is not possible to estimate the computing time for solving a given MIP instance. If the usage of a parallel computing environment is restricted by a certain time limit, we cannot predict reliably whether the computation will be finished within that time window. These two issues make checkpointing and restarting prerequisite functions of a parallel MIP solver.

A natural way for checkpointing would be to save all open nodes of all branch-and-bound trees and the best primal solution found so far. The number of open nodes, however, typically grows very fast for hard problem instances. If checkpointing is performed frequently, this will lead to a huge amount of I/O, slowing down the computation.

Therefore, we decided to save only *primitive* nodes, that is, nodes for which no ancestor nodes are in the LOADCOORDINATOR. 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 restore them into the node pool of the LOADCOORDINATOR. After that, the LOADCOORDINATOR distributes these nodes to the SOLVERs ordered by their dual bounds.

In the ramp-up phase, the distributed subproblems are aggressively broken down, because always one of two branched nodes is transferred to the LOADCOORDINATOR. Further, this node will be presolved as it will become the root node of a subproblem. Therefore, the checkpointing and restarting mechanism can be understood

as an implicit load balancing mechanism. It detects the hardest part of the branch-and-bound tree and automatically breaks it down to easier subproblems.

4 Solving open instances from MIPLIB2003 on HLRN II

In this section, we present computational results for solving two open problem instances, `ds` and `stp3d`, from MIPLIB2003 which were conducted on the HLRN II supercomputer [2]. The computations were performed using SCIP 1.2.1.2 with CPLEX 12.1 as underlying linear programming solver.

The best known upper bound for the instance `ds` and `stp3d` were 116.59 and 500.736, respectively [13]. The optimal values that we proved are 93.52 for the `ds` instance (decreased by about 25%) and 493.71965 (decreased by 1%) for `stp3d`.

4.1 `ds` and `stp3d` instances

The instance `ds` models a real-world duty scheduling problem of a German public transportation company. In this context, duty scheduling means the assignment of daily shifts of work to bus or tram drivers by means of a schedule. For this particular model, the number of duties, represented by 0-1-variables is 67 732, the number of tasks, represented by linear constraints, is 656. The number of variables set to one in an optimal solution is equal to the cost minimal number of duties to cover all tasks.

`stp3d` is a Steiner tree packing problem in a three dimensional grid graph. The instance is a "switchbox routing problem" where connections (wires) between various endpoints (terminals) have to been routed in the graph. Each set of endpoints defines a Steiner tree problem which is already *NP*-hard. In `stp3d` there are several Steiner trees, which have to be placed at the same time into the graph in a node disjoint way. The objective is to minimize the total length of all networks. Here, already showing feasibility is *NP*-hard. Consisting of 204 880 variables and 159 488 constraints, `stp3d` is the largest instance in MIPLIB2003.

4.2 Extended presolving

Instances that are to be solved on a supercomputer are usually expected to have an enormous running time. Hence, all possibilities to reduce the overall running time in advance should be exploited.

Presolving is an important feature of state-of-the-art MIP solvers that often reduces the overall computation time considerably. The task of presolving is twofold: first, it reduces the size of the model by proving that certain constraints or variables are redundant and can be removed from the problem formulation or fixed to

a certain value, respectively, without changing the optimal solution value. Second, it strengthens the LP relaxation of the model by exploiting integrality information, e.g., to tighten the bounds of the variables or to improve coefficients in the constraints. SCIP provides several presolving techniques, some of which are deactivated by default, due to their computational complexity.

Before starting the parallel computation, we applied an *extended preprocessing* on a single machine. Therefore, we used SCIP's "aggressive presolving" settings and afterwards performed strong branching on all problem variables. For 0-1 variables, *strong branching* tentatively fixes a variable to zero (and subsequently to one), and solves the corresponding LP relaxation. If this LP turns out to be infeasible, the variable can be fixed to the opposite value in the original problem. If we could fix at least one of the problem variables, we iterate the process, starting again with aggressive presolving.

In particular for the instance `stp3d`, extended presolving helped to reduce the problem size and thereby the expected computation time. For this instance, the default presolving of SCIP reduces the problem size to 136 500 variables and 97 144 constraints, whereas nine iterations of extended presolving reduced the problem size to 123 637 integer variables and 88 388 constraints.

4.3 HLRN II

HLRN II is a massive parallel supercomputing system which is one of the most powerful computers in Germany and number 64 in the TOP500 list as of November 2010 [6]. From the global system view, HLRN II consists of two identical complexes located at RRZN in Hannover and ZIB in Berlin. Both complexes are coupled by the HLRN link, a dedicated fiber connection for HLRN. In the current stage each complex consists of three parts, the so-called MPP1, MPP2, and the SMP part. We used MPP2 part whose specification is as follows:

- 960 eight-core compute nodes (2 quad-core sockets each for Intel Xeon Gainestown processors (Nehalem EP, X5570) running at 2.93 GHz) with 48 GB memory
- Total peak performance 90 TFlop/s
- Total memory 45 TByte
- 4x DDR Infiniband Dual Rail network for MPI
- 4x DDR Infiniband network for I/O to the global Lustre filesystems

4.4 Computational results

The instances `ds` and `stp3d` were solved by ParaSCIP on HLRN II using up to 2048 cores for each run and needed to be warm started 16 and 10 times, respectively. Tables 1 and 2 show the status of each job for `ds` and `stp3d`, respectively. HLRN II is not able to swap, hence real memory size is a strict limit. If a memory shortage

occurs, all SOLVER processes will terminate immediately and solving has to be restarted.

Tables 1 and 2 show the number of runs needed to solve the instance, the ramp-up times and the elapsed computation times until the final checkpoint of each single run. When a SOLVER finishes a sub-MIP computation, it sends its statistical data to the LOADCOORDINATOR. In case a SOLVER did not finish the first sub-MIP computation by the final checkpoint, its performance will not be taken into consideration for the statistics. The tables show the number of SOLVERs that solved at least one sub-MIP to optimality. Note that the SOLVERs which have been idle until the final checkpoint, e.g. because ramp-up did not finish, have no statistical data. The "# of nodes to restart" column shows the number of nodes used for restarting the computation. This corresponds to the number of nodes that had been saved at the final checkpoint of the previous run. Thus, it is always zero for the first job and greater zero for all subsequent jobs. The final column shows the number of nodes solved at the job. Note that the number of nodes at the checkpoints is very small compared to the number of nodes solved for the computations.

The ramp-up time varies among different runs for the same instance. This is due to the fact, that we changed some parameter settings to better adopt to the individual behavior of an instance. These changes only influence the path the solver takes, not the overall result. It took approximately 86 hours to solve `ds` and approximately 114 hours to solve `stp3d` to proven optimality. Due to the checkpoint system described above, some parts of the tree might be resolved several times and only the final solve will be counted in the statistics.

The summary of statistical data therefore gives an underestimation of the number of branch-and-bound nodes. The number of nodes was 1 174 818 123 for `ds` and 14 788 888 for `stp3d`. All SOLVERs bookkeep the idle time, that is not used for solving any sub-MIP. The idle time ratio of all SOLVERs was about 2.2% for solving `ds` and 3.9% for solving `stp3d`. We plan to conduct a single job computation to solve these instances, in order to provide precise values for the number of branch-and-bound nodes and the idle time ratios.

Figures 2 and 4 show how the primal and dual bound evolved during the course of the solution process. The behavior is typical for MIP instances. The primal bound moves stepwise and reaches the optimal values significantly faster than the dual bound. The dual bound moves smoothly, stays nearly constant for a long time and collapses towards the end of the solution process.

Figures 3 and 5 show how the workloads (that is, the number of branch-and-bound nodes left in all SOLVERs) and the number of nodes in the LOADCOORDINATOR node pool change during the computations. During the whole solution process, nearly all nodes are "good" w.r.t. our definition from Section 3.1. ParaSCIP manages very well to keep the node pool of the LOADCOORDINATOR filled. As long as the node pool is not empty, no SOLVER will become idle. Thus, the small idle time ratio is due to the management of the node pool.

Table 1 Each job status for solving `ds` instance

Job No.	# of cores used	Comp. Time (sec.)	Ramp-up Time(sec.)	# of solvers solved at least one sub-MIP	# of nodes to restart	# of nodes solved
1	512	14400.5	249.2	185	0	1328147
2	512	18000.6	247.4	105	1	1766849
3	1024	32401.2	333.1	1019	1	71108615
4	2048	23401.1	535.1	2045	8	137270553
5	2048	14400.9	1335.0	2046	5	88130034
6	2048	18001.1	366.5	2020	89	127816887
7	2048	12600.8	324.8	1997	157	77649716
8	2048	16201.2	333.0	1976	248	102950355
9	2048	14401.0	282.0	1927	311	87778722
10	2048	12600.9	309.0	1930	301	71857862
11	2048	12600.8	299.0	1937	292	74279899
12	2048	14400.9	277.5	1911	219	90106215
13	2048	30673.3	–	57	243	477
14	2048	10800.7	294.3	1917	191	57494144
15	2048	14401.0	300.5	1935	196	87130452
16	2048	41402.7	363.4	2047	256	97785724
17	1024	8820.1	353.2	1023	196	363472

– : stopped before ramp-up

Table 2 Each job status for solving `stp3d` instance

Job No.	# of cores used	Comp. Time (sec.)	Ramp-up Time(sec.)	# of solvers solved at least one sub-MIP	# of nodes to restart	# of nodes solved
1	512	41467.4	6165.8	239	0	64545
2	1024	41407.4	2927.4	320	146	199719
3	2048	41403.9	2362.3	1185	592	766133
4	2048	1800.3	–	83	527	100
5	2048	1800.4	–	2	446	2
6	2048	1800.3	–	1	444	1
7	2048	1800.9	–	1	443	1
8	2048	43203.9	2474.6	822	442	806159
9	2048	41403.9	2914.2	1817	626	1555160
10	2048	41405.6	4406.7	1778	229	1538151
11	2048	152912.0	3841.8	2047	429	9858917

– : stopped before ramp-up

5 Concluding remarks

We have shown that using our approach, ParaSCIP is able to effectively use the computing power scales of several thousand cores to solve mixed integer programs. Furthermore, this could be done without changing the inner workings of the sophisticated sequential algorithm.

Still, many open questions remain. Using a shared memory version of ParaSCIP, we are planning to estimate how much speed we loose by not using a more

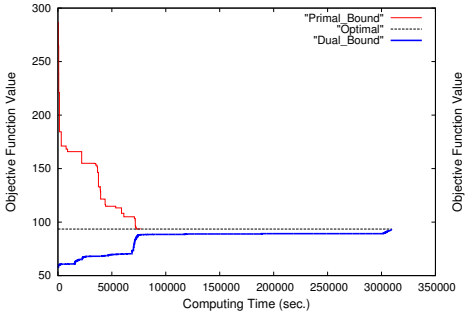

Fig. 2 Bounds evolution for `ds`

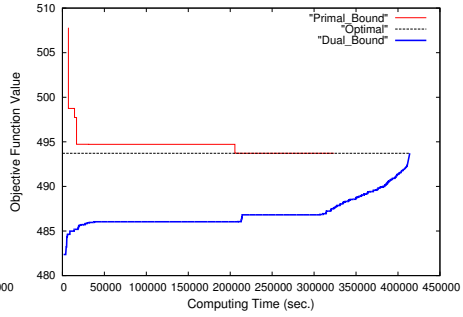

Fig. 4 Bounds evolution for `stp3d`

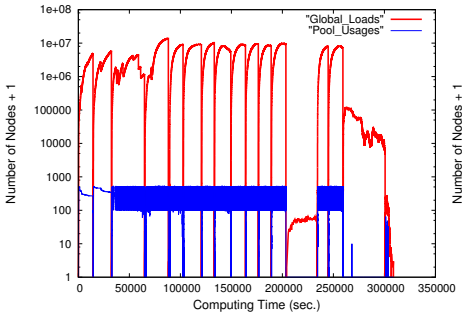

Fig. 3 Workload evolution for `ds`

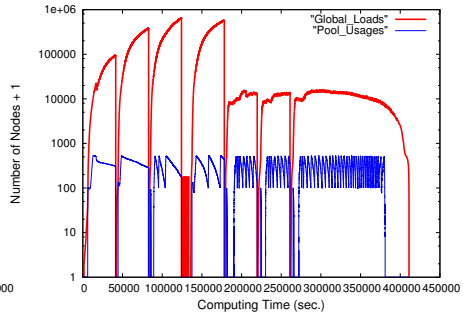

Fig. 5 Workload evolution for `stp3d`

fine grained parallelism. When increasing the number of cores the time spent in the ramp-up and ramp-down phases also increases, hampering scalability. We are currently investigating ways to improve the effectiveness of load balancing during these phases. While `ds` and `stp3d` could be successfully solved, others instances remain, for which it is unclear which amount of computing time is needed to produce an optimal solution with today's MIP solver technology. Trying to judge in advance, whether an instances is suitable for massive parallel solving and predicting remaining running times is an important topic to be investigated in the future.

References

1. Gurobi Optimizer. http://www.gurobi.com/
2. HLRN – Norddeutscher Verbund zur Förderung des Hoch- und Höchstleistungsrechnens. http:/www.hlrn.de/
3. IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/
4. Mixed Integer Problem Library (MIPLIB) 2003. http://miplib.zib.de/
5. SCIP: Solving Constraint Integer Programs. http://scip.zib.de/
6. TOP500 Supercomputer Sites. http://www.top500.org/list/2010/11/100

7. Achterberg, T.: Constraint integer programming. Ph.D. thesis, Technische Universität Berlin (2007)
8. Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation **1**(1), 1–41 (2009)
9. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Operations Research Letters **34**(4), 1–12 (2006)
10. Bixby, R., Rothberg, E.: Progress in computational mixed integer programming – A look back from the other side of the tipping point. Annals of Operations Research **149**(1), 37–41 (2007)
11. Bixby, R.E., Boyd, E.A., Indovina, R.R.: MIPLIB: A test set of mixed integer programming problems. SIAM News **25**, 16 (1992)
12. Karp, R.M.: Reducibility among combinatorial problems. In: R.E. Miller, J.W. Thatcher (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press, New York, USA (1972)
13. Laundy, R., Perregaard, M., Tavares, G., Tipi, H., Vazacopoulos, A.: Solving hard mixed-integer programming problems with Xpress-MP: A miplib 2003 case study. INFORMS Journal on Computing **21**(2), 304–313 (2009)
14. Mittelmann, H.: Mixed integer linear programming benchmark (serial codes). http://plato.asu.edu/ftp/milpf.html
15. Padberg, M., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Review **33**, 60–100 (1991)
16. Ralphs, T.K., Ladányi, L., Saltzman, M.J.: Parallel branch, cut and price for large-scale discrete optimization. Mathematical Programming Series B **98**(1–3), 253–280 (2003)
17. Shinano, Y., Achterberg, T., t: Fujie: A dynamic load balancing mechanism for new paralex. In: In: Proceedings of ICPADS 2008, pp. 455–462 (2008)
18. Xu, Y., Ralphs, T.K., Ladányi, L., Saltzmann, M.J.: Computational experience with a software framework for parallel integer programming. INFORMS Journal on Computing **21**(3), 383–397 (2009)