Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

YUJI SHINANO, DANIEL REHFELDT, THORSTEN KOCH

Building Optimal Steiner Trees on Supercomputers by using up to 43,000 Cores

This work has also been supported by the Research Campus Modal *Mathematical Optimization and Data Analysis Laboratories* funded by the Federal Ministry of Education and Research (BMBF Grant 05M14ZAM), and partially supported by the BMWi project Realisierung von Beschleunigungsstrategien der anwendungsorientierten Mathematik und Informatik für optimierende Energiesystemmodelle - BEAM-ME (fund number 03ET4023DE). All responsibility for the content of this publication is assumed by the authors.

ZIB Report 18-58 (December 2018)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

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

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

Building Optimal Steiner Trees on Supercomputers by using up to 43,000 Cores

Yuji Shinano, Daniel Rehfeldt, Thorsten Koch*†

February 11, 2019

Abstract

SCIP-JACK is a customized, branch-and-cut based solver for Steiner tree and related problems. ug [SCIP-JACK, MPI] extends SCIP-JACK to a massively parallel solver by using the Ubiquity Generator (UG) framework. ug [SCIP-JACK, MPI] was the only solver that could run on a distributed environment at the (latest) 11th DIMACS Challenge in 2014. Furthermore, it could solve three well-known open instances and updated 14 best known solutions to instances from the benchmark libary STEINLIB. After the DIMACS Challenge, SCIP-JACK has been considerably improved. However, the improvements were not reflected on ug [SCIP-JACK, MPI]. This paper describes an updated version of ug [SCIP-JACK, MPI], especially branching on constrains and a customized racing ramp-up. Furthermore, the different stages of the solution process on a supercomputer are described in detail. We also show the latest results on open instances from the STEINLIB.

1 Introduction

The *Steiner tree problem in graphs* (SPG) is one of the fundamental $\mathcal{NP}$-hard optimization problems [5]. Given an undirected connected graph $G = (V,E)$, costs $c : E \to \mathbb{Q}_{\geq 0}$ and a set $T \subseteq V$ of *terminals*, the problem is to find a tree $S \subseteq G$ of minimum cost that includes T. The 2014 DIMACS Challenge, dedicated to Steiner tree problems, marked a revival of research on the SPG and related problems. SCIP-JACK [2], which is a customized SCIP solver for SPG and related problems, was initially developed to attend the DIMACS Challenge. SCIP-JACK was by far the most versatile solver participating in the Challenge, being able to solve the SPG and 10 related problems. After the DIMACS Challenge, the performance of SCIP-JACK has continuously improved, both for SPG [13] and related problems [11, 12, 14]. The improvements were for instance marked by SCIP-JACK being the most successful solver at the PACE 2018 Challenge [1] dedicated to fixed-parameter tractable (FPT) SPG instances (although SCIP-JACK does not include any FPT specific algorithms).

ug [SCIP-JACK, MPI] is an extension of SCIP-JACK to a massively parallelized solver by using the *Ubiquity Generator (UG) framework* [16], a software package to parallelize branch-and-bound (B&B) based solvers. ug [SCIP-JACK, MPI] was the only solver which could run on a distributed environment at the 11th DIMACS Challenge. Moreover, it solved three open instances and updated 14 best known solutions

*Zuse Institute Berlin, Takustraße 7, 14195 Berlin, Germany, {shinano, rehfeldt, koch}@zib.de

†TU Berlin, Str. des 17. Juni 135, 10623 Berlin, Germany

to instances from the STEINLIB [7]. However, no detailed statistics on the solving process have been published yet. After the DIMACS Challenge, solving new open instances from the STEINLIB by ug [SCIP-JACK, MPI] looked hopeless for all open instances—judging from their run-time log files—and there have been no new result published prior to this paper. For the results presented throughout this paper, we used the ug [SCIP-JACK, MPI] code included in the SCIP Optimization Suite 6.0 [3].

ug [SCIP-JACK, MPI] was not implemented from scratch by using UG, but it was parallelized by using the *ug[*SCIP-*,MPI]-library*, which is a software library to parallelize SCIP applications. SCIP is a plugin based software framework [3] and by adding new user-plugins it can be extended to create a customized solver like SCIP-JACK. The ug [SCIP-*,MPI]-library allows its users to include these user-plugins to PARASCIP by adding a small amount of glue-code (typically $100-200$ lines). Usually, if a solver performance parallelized by UG is improved, this is directly reflected in the performance of its parallel extension. Since the SCIP-JACK performance has improved tremendously after the DIMACS Challenge, see Section 3, one would expect the same of ug [SCIP-JACK, MPI]. However, several idiosyncrasies of SCIP-JACK required to develop new features of the ug [SCIP-*,MPI]-library, in order to also obtain the performance improvements in its massively parallel extension.

In the following, we briefly describe UG, and go on to introduce the newly added features of the ug [SCIP-*, MPI]-library that aim to improve the performance of ug [SCIP-JACK, MPI]. Finally, first results obtained with the new features will be presented.

2 Key features of UG and ug [SCIP-*,MPI]-library

A uniqueness of UG is that it is a software framework to parallelize an existing state-of-the-art B&B based solvers. We call the B&B based solver parallelized by UG *base solver*. In UG, the base solver is encapsulated in an abstracted *ParaSolver*. The ParaSolver accesses the base solver via its API. At run-time on a supercomputer, there are ParaSolver processes, which solve subproblems, and there is a special process called *LoadCoordinator (LC)* , which makes all decisions about load balancing among the ParaSolvers. To realize the load balancing, message passing based protocols are defined between the LC and the ParaSolvers. The LC also has a base solver environment, which does presolving (all ParaSolvers solve the presolved instance internally) and converts the solution to the presolved problem to a solution to the original one. Key features of UG are:

Ramp-up Ramp-up is the phase until all solvers have become active. In *Normal ramp-up*, only one ParaSolver receives the root node, and it distributes one of the branched nodes to the other solvers via the LC. All the ParaSolvers do the same when they receive a node. The transferred node (subproblem) data contains only the difference between the subproblem and the (presolved) instance data. *Racing ramp-up* exploits the performance variability commonly observed in MIP solving [6]. An instance is solved multiple times by ParaSolvers in parallel, each time with a different parameter setting. If the instance has not been solved to optimality once a predefined termination criterion, e.g., a time limit, is reached, the most promising branch-and-bound tree is distributed among the ParaSolvers and the default solving procedure is initiated. The effectiveness of racing ramp-up is described in [17, 20].

Dynamic load balancing UG provides a Supervisor-Worker load coordination scheme [10]. In the Master-Worker paradigm, all B&B search tree data is managed by the

Master. In contrast to the Master-Worker paradigm, the idea of Supervisor-Worker is that the Supervisor functions only to make decisions about the load balancing, but does not actually store the data associated with the B&B search tree. In UG, the Supervisor is the `LC` and the Workers are the `ParaSolver`s. The B&B search tree data is managed by the `ParaSolver`s. The terminal nodes (subproblems) of the B&B search tree in the `ParaSolver`s are sent on demand to the `LC`; a set of subproblems in the `LC` works as a buffer to ensure subproblems are available to idle `ParaSolver`s as needed.

Load balancing is accomplished mainly by switching the collection mode in the `ParaSolver`. Turning collecting mode on results in additional "high quality" subproblems being sent to the `LC`, which can then be distributed to the `ParaSolver`s. The method of selecting which `ParaSolver` to collect from is crucial and is controlled very carefully. Some additional keys to avoid having the Supervisor becoming a communication bottleneck are:

- Frequency of status updates can be controlled depending on the number of `ParaSolver`s.
- The maximum number of `ParaSolver`s in collection mode is capped and the `ParaSolver`s are chosen dynamically.

A detailed description of the dynamic load balancing is presented in [18, 20].

Checkpointing and restrating mechanism By the dynamic load balancing of UG, B&B nodes in a sub-tree can be transferred recursively to the other solvers. Therefore, at each checkpoint, only essential B&B nodes, i.e., sub-tree roots whose ancestor node is not available on a run-time system, are saved. The number of such nodes is extremely small compared to the number of open nodes; thus the checkpointing is very light weight. However, a huge search tree has to be regenerated at restart. This regeneration might look redundant and inefficient. However, for MIP solvers, this procedure has been shown to be notably efficient [17], since dual bounds of the checkpoint nodes are calculated more precisely and the B&B tree is regenerated based on these values at restart—the regenerated B&B tree can thus be different than that of the previous run.

Deterministic mode for debugging One of the most difficult parts of software development is debugging. Before running a parallel solver instantiated by UG, extensive debugging for a set of instances with different number of solvers is usually needed. Without having a deterministic mode, this would be extremely inefficient.

PARASCIP (=`ug` [SCIP,MPI]) is an instantiated parallel solver that uses UG, in which SCIP is the base solver. Since SCIP is plugin-based, it is natural to make a `ug` [SCIP-*, MPI]-library in which user plugins are installed automatically by providing a small amount of glue code. `ug` [SCIP-JACK, MPI] is realized by using such a library and is distributed as a UG application. The Steiner tree application directory of SCIP Optimization Suite 6.0 contains only one source file stp_plugins.cpp and it has only 173 lines of glue code without empty and comment lines.

3 Improvements of SCIP-JACK after the DIMACS Challenge

SCIP-JACK has seen a large number of improvements after the 11th DIMACS Challenge, both for SPG and related problems. These developments include new primal and dual heuristics [2, 14], reduction techniques [15], and various technical improvements such as a fast maximum-flow implementation [8] (used for separation). Of particular relevance for massive parallelization is the subsequently described improvement for domain propagation: During the solving process it is usually possible to fix many (binary) edge variables of the IP formulation to 0 or 1—for instance by using reduced cost arguments [2] or branching information. These fixings can be directly translated into edge deletions and contractions in the underlying graph, which can allow for further eliminations by the powerful graph reduction techniques of SCIP-JACK. However, as already observed by other authors [9], such graph reductions can change the graph in a complex way, which cannot be easily translated into variable fixings in the IP formulation. However, we have devised a simple mapping that given an original instance $P = (V,E,T,c)$ and reduced instance $P' = (V',E',T',c')$ allows to map P' to a problem P'' such that P'' can be obtained from P' by deletion of edges only. First, note that the reduction techniques of SCIP-JACK provide a mapping $p : E' \to \mathscr{P}(E)$ such that for each (optimal) solution $S' \subseteq E'$ to P', set $\bigcup_{e \in S'} p(e)$ is an (optimal) solution to P. With this information one obtains:

Proposition 1 *Let $P = (V,E,T,c)$ be an SPG and (V',E',T',c') be an instance obtained by using the reduction techniques of* SCIP-JACK. *Each solution S'' to the SPG $P'' = (V'',E'',T'',c'')$ defined by*

$$\begin{aligned} E'' &:= \bigcup_{e \in E'} p(e), \\ V'' &:= \{v \in V \mid \exists (v,w) \in E'', w \in V\}, \\ T'' &:= \{t \in T \mid \exists (t,w) \in E'', w \in V\}, \\ c'' &:= c|_{E''}, \end{aligned}$$

is a solution to P. Furthermore, if S'' is an optimal solution to P'', it is an optimal solution to P.

One readily acknowledges, that $V'' \subseteq V$ and $E'' \subseteq E$. Note, however, that usually $|V''| > |V'|$ and $|E''| > |E'|$, so we first apply only techniques that can be directly translated into variable fixings (such as deletion of edges) and apply the corresponding fixings to the IP; only afterward we perform more complex reductions (and use Proposition 1 to apply further fixings).

4 New features of ug [SCIP-JACK, MPI]

In this section, we describe new general features added to `ug` [SCIP-*,MPI]-library, and also specialized new features added to `ug` [SCIP-JACK,MPI].

4.1 Branching on constraints

After the DIMACS Challenge, instead of branching on variables, which in the case of Steiner tree problem correspond to edges, default SCIP-JACK uses vertex branching [4]. During the B&B process, SCIP-JACK selects a non-terminal vertex of the Steiner tree problem graph to be rendered a terminal in one B&B child node and to be excluded in the other child. These two operations are modeled in the underlying IP

formulation by including one additional constraint. This procedure could not be used in previous versions of ug [SCIP-JACK,MPI], since branching on constrains was only possible in SCIP, but not in the ug [SCIP-*, MPI]-library. Therefore, a new feature for transferring branching constrains has been added to the ug [SCIP-*, MPI]-library. The new feature allows ug [SCIP-JACK,MPI] to use the vertex branching.

4.2 Callback to initialize a transferred subproblem

A distinguishing feature of UG solvers is that it can naturally realize *layered presolving*, in which B&B tree nodes are transferred to the other ParaSolvers recursively and additional presolving is performed on the subproblems. The effectiveness of the layered presolving is documented in [19, 20]. When using ug [SCIP-*, MPI]-library, MIP presolving realized by SCIP can work without any additional coding. However, SCIP-JACK performs presolving on the underlying graph before it formulates the subproblem as an IP. In order to realize the graph based presolving, a callback to initialize the transferred subproblem has been added to the ug [SCIP-*, MPI]-library. To retain previous graph based branching decisions, ug [SCIP-JACK,MIP] transfers the branching history together with a subproblem, enabling SCIP-JACK to change the underlying graph (by adding terminals and deleting vertices). Additionally, whenever a subproblem has been transfered, SCIP-JACK performs aggressive reduction routines to reduce the (modified) problem further and translates the reductions into variable fixings by means of Proposition 1.

4.3 Customized racing

The latest ug [SCIP-*, MPI]-library includes *customized racing*, which allows the user to specify their own parameter settings for racing. If the number of UG solvers exceeds the number of provided parameter sets, then the customized parameter settings are combined with random number seeds. While the latest release version of PARASCIP does not use customized racing by default, it is applied in ug [SCIP-JACK,MPI]. For this article we used 30 parameter settings, where we varied: the aggressiveness of the primal heuristics, the aggressiveness of domain propagation, the branching rule (LP-based [2] or based on primal solution [9]), and various parameter for the cut selection.

5 Updated computational results for open instances

For solving open instances of the PUC test set from STEINLIB as of 1st of November 2018, we used two supercomputers. One is an ISM (Institute of Statistical Mathematics) supercomputer which is a HPE SGI 8600 with 384 compute nodes, each node has two Intel Xeon Gold 6154 3.0GHz CPUs (18 cores$\times$2) sharing 384GB of memory, and an Infiniband (Enhanced Hypercube) interconnect. The other is HLRN III which is a Cray XC40 with 1872 compute nodes, each node with two 12-core Intel Xeon Ivy-Bridge/Haswell CPUs sharing 64 GiB of RAM, and with an Aries interconnect. The interval time of checkpointing was set to 1,800 seconds. The maximum number of ParaSolvers in collection mode was capped at 500.

5.1 hc9p (solved)

This instance was solved by five restarted runs and by using up to 24,576 cores. The initial primal solution was found by ug [SCIP-JACK,MPI] at the DIMACS Challenge. All computations were used to prove its optimality. The racing termination criteria was a node limit of 50, that is: once the number of open B&B nodes in a ParaSolver

with the largest dual bound surpasses 50, racing is terminated. Table 1 shows the supercomputer used, the computing time in seconds (racing time is shown within parentheses), the idle time ratio for all ParaSolvers, the number of transferred B&B nodes to ParaSolvers, primal and dual bounds, gap, the number of B&B nodes generated, and the number of open B&B nodes for each run. The initial values are shown in the upper row and the final values of those are shown in the lower row for each run.

The final dual bound in the previous run is sometimes slightly different from that of the initial one in the following run. This means that the dual bound in the previous run was updated after the final checkpoint. The number of open B&B nodes decreases a lot at restart, since the checkpointing mechanism only saves essential sub-tree roots. For example, run 1.1 ends up with 1,257,112 open B&B nodes, but run 1.2 starts with 15 open ones. This means that only 15 B&B sub-tree roots existed at the end of run 1.1 and the other sub-tree roots were descendants of one of the 15 B&B nodes.

The number of transferred B&B nodes can be considered as an indicator of how frequently ParaSolvers became idle and also how frequently layered presolving was applied. It is natural that at larger scale we can expect more layered presolving. Actually, the number of transferred B&B nodes of run 1.1 with 72 cores was only 738 nodes in a one week long execution. It was increased by using 2,304 cores to 979,695 in another one week execution. In the following bigger jobs it was drastically increased.

Figure 1 shows the evolution of the computation for the final run 1.5. The number of B&B open nodes continuously increases and decreases during the computation and it looks sometimes difficult to make all ParaSolvers active. However, dynamic load balancing recovered the situation well and all the ParaSolvers were active during almost the entire computing time. The idle time ratio was only 1.5%. The number of checkpoint nodes also changed a lot during the computation.

We can obtain the idle time ratio for all ParaSolvers only if ug [SCIP-JACK,MPI] finishes its computation and cannot get it if the program is canceled by the system in case the time-limit is hit. After racing ramp-up, all ParaSolver statistics are collected. Therefore, by using its partial data, an upper bound on the idle time ratio is calculated. The lack of data is complemented by the maximum idle time ratio in the case of racing ramp-up, and complemented by the idle time ratio of run 1.5. Table 1 also shows the upper bounds of the idle time ratio. The idle time ratios for all runs are notably small, which indicates that the supercomputers are used efficiently.

Table 1: Statistics for solving hc9p on supercomputers

Run	Computer	Cores	Time (sec.)	Idle (%)	Trans.	Primal bound (Upper bound)	Dual bound (Lower bound)	Gap (%)	Nodes	Open nodes
1.1	ISM	72	604,796 (317)	< 0.3	738	30,242.0000	29,879.3721	1.21	0	0
						30,242.0000	30,058.9366	0.61	110,012,624	1,257,112
1.2	ISM	2,304	604,794	< 1.5	979,695	30,242.0000	30,058.7930	0.61	0	15
						30,242.0000	30,102.7556	0.46	3,758,532,600	723,167
1.3	HLRN III	24,576	86,336	< 1.7	8,811,512	30,242.0000	30,102.6645	0.46	0	35
						30,242.0000	30,116.3592	0.42	2,402,406,311	575,678
1.4	HLRN III	12,288	43,199	< 1.5	1,709,027	30,242.0000	30,115.3331	0.42	0	3,709
						30,242.0000	30,120.4801	0.40	664,909,985	602,323
1.5	HLRN III	12,288	118,259	1.5	9,158,920	30,242.0000	30,120.4801	0.40	0	285
						30,242.0000	30,242.0000	0.00	1,677,724,126	0

5.2 hc11p (updated the best known solution)

During the new developments in ug [SCIP-JACK, MPI], the best known solution to the hc11p instance could be updated (with objective value 119,492 compared to 119,689

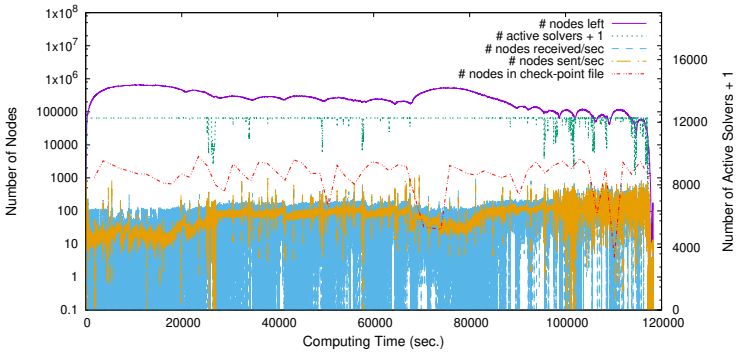

Figure 1: Evolution of computation for solving `hc9p` by using 12,288 cores (Run 1.5)

at the DIMACS Challenge). The first additional run 1 on the ISM supercomputer generated 11 new incumbent solutions, with the best objective value being 119,297. Afterwards we just solved it from scratch with the best solution in racing ramp-up (run 2.1) again, since it can be used for presolving, propagation, and heuristics. The racing termination criteria for run 1 was the same as that for `hc9p`, but the node limit 100 was used for run 2.1. The restarted job was conducted from the checkpoint file of run 2.1, since run 2.1 could not improve the incumbent solution. Run 1 consumed 12,095 cores-hours ($= (72 \times 604799)/3600$) and it reached a 1.53(%) gap. Runs 2.1 and 2.2 consumed 118,582 cores-hours ($= ((1288 \times 43149) + (4300 \times 86354))/3600$) reached a 1.56(%) gap. To improve the gap with the same amount of computing resources, initial longer run at small scale look more promising than large scale runs with short computing time.

Table 2: Statistics for solving `hc11p` on supercomputers

Run	Computer	Cores	Time (sec.)	Idle (%)	Trans.	Primal bound (Upper bound)	Dual bound (Lower bound)	Gap (%)	Nodes	Open nodes
1	ISM	72	604,799 (2,558)	< 0.3	71	119,492.0000	117,388.8528	1.79	0	0
						119,297.0000	117,496.5470	1.53	4,314,198	1,109,629
2.1	HLRN III	12,288	43,149 (7,164)	< 0.5	31,304	119,297.0000	117,388.7971	1.63	0	0
						119,297.0000	117,426.2226	1.59	28,491,470	5,433,482
2.2	HLRN III	43,000	86,354	< 4.9	86,152	119,297.0000	117,426.2226	1.59	0	103
						119,297.0000	117,468.8459	1.56	267,513,609	40,499,188

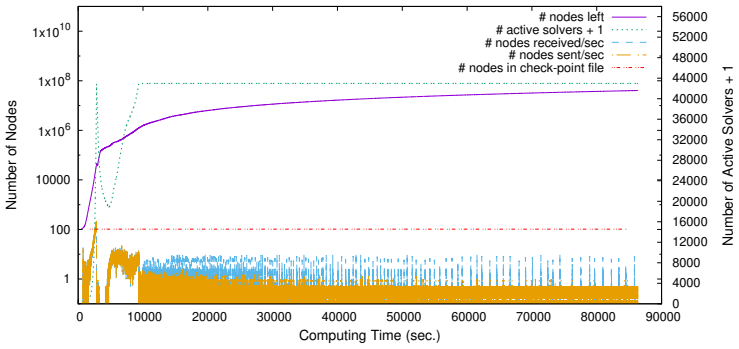

Figure 2: Evolution of computation for solving `hc11p` by using 43,000 cores (Run 2.2)

The numbers of transferred B&B nodes were very small compared to those for `hc9p`. This shows a fundamental hardness of `hc11p` compared to that of `hc9p`. Figure 2 shows the evolution of computation for run 2.2—the largest scale used with `ug`

[SCIP-JACK, MPI] so far. The restart is always normal ramp-up from the nodes in checkpoint file. In the normal ramp-up, all ParaSolvers send one of two branched nodes to the other ParaSolvers via LC. This lasts until all ParaSolvers have become active. SCIP-JACK does presolving and adds cutting planes aggressively at its root node, making ramp-up difficult. Additionally, once in ramp-up the LC's internal mode changes to collection mode. In this mode, only a restricted number of ParaSolvers can be in collection mode. Therefore, the number of active ParaSolvers decreases after the first peak. However, once the LC has collected enough nodes again, the quality of the nodes in the LC is very good and less and less dynamic load balancing is needed. Figure 2 shows this behavior. Taking into account this difficulty of ramp-up, the idle time ratio of run 2.2 is less than 4.9%. The number of checkpoint nodes stays the same and the open B&B nodes keep increasing. Thus further improvements of SCIP-JACK or much larger runs are needed to solve hc11p.

6 Concluding remarks

We have extended ug [SCIP-JACK, MPI] to immediately obtain the benefits of any SCIP-JACK improvements, allowing us to solve one previously unsolved benchmark instance to optimality. We also showed that ug [SCIP-JACK, MPI] can run on up to 43,000 cores efficiently in terms of computing resources usage. Therefore, when SCIP-JACK has been further improved (as planned for the near future) we expect to solve additional open instances. Also, the techniques presented in this paper work on other problems related to the SPG that can be handled by SCIP-JACK.

7 Acknowledgements

The authors would like to thank Utz-Uwe Haus for his help in tracking down a particularly insistent bug. This work has been supported by the Research Campus MODAL *Mathematical Optimization and Data Analysis Laboratories* funded by the Federal Ministry of Education and Research (BMBF Grant 05M14ZAM). This work was also supported by the North-German Supercomputing Alliance (HLRN). Supported by BMWi project BEAM-ME (fund number 03ET4023DE).

References

[1] PACE Challenge 2018. https://pacechallenge.wordpress.com/pace-2018/, accessed: November 10. 2018

[2] Gamrath, G., Koch, T., Maher, S., Rehfeldt, D., Shinano, Y.: SCIP-Jack—a solver for STP and variants with parallelization extensions. Mathematical Programming Computation **9**(2), 231 – 296 (2017). https://doi.org/10.1007/s12532-016-0114-x

[3] Gleixner, A., Bastubbe, M., Eifler, L., Gally, T., Gamrath, G., Gottwald, R.L., Hendel, G., Hojny, C., Koch, T., Lübbecke, M.E., Maher, S.J., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schlösser, F., Schubert, C., Serrano, F., Shinano, Y., Viernickel, J.M., Walter, M., Wegscheider, F., Witt, J.T., Witzig, J.: The scip optimization suite 6.0. Tech. Rep. 18-26, ZIB, Takustr. 7, 14195 Berlin (2018)

[4] Hwang, F., Richards, D., Winter, P.: The Steiner tree problem. Annals of Discrete Mathematics **53** (1992)

[5] Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press (1972)

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

[7] Koch, T., Martin, A., Voß, S.: SteinLib: An updated library on Steiner tree problems in graphs. In: Du, D.Z., Cheng, X. (eds.) Steiner Trees in Industries, pp. 285–325. Kluwer (2001)

[8] Maher, S.J., Fischer, T., Gally, T., Gamrath, G., Gleixner, A., Gottwald, R.L., Hendel, G., Koch, T., Lübbecke, M.E., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schenker, S., Schwarz, R., Serrano, F., Shinano, Y., Weninger, D., Witt, J.T., Witzig, J.: The SCIP Optimization Suite 4.0. Tech. Rep. 17-12, ZIB, Takustr.7, 14195 Berlin (2017)

[9] Polzin, T.: Algorithms for the Steiner problem in networks. Ph.D. thesis, Saarland University (2004), http://scidok.sulb.uni-saarland.de/volltexte/2004/218/index.html

[10] Ralphs, T., Shinano, Y., Berthold, T., Koch, T.: Parallel Solvers for Mixed Integer Linear Optimization, pp. 283–336. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-63516-3_8, https://doi.org/10.1007/978-3-319-63516-3_8

[11] Rehfeldt, D., Koch, T.: Combining NP-Hard Reduction Techniques and Strong Heuristics in an Exact Algorithm for the Maximum-Weight Connected Subgraph Problem. SIAM Journal on Optimization **29**(1), 369–398 (2019). https://doi.org/10.1137/17M1145963, https://doi.org/10.1137/17M1145963

[12] Rehfeldt, D., Koch, T.: Reduction-based exact solution of prize-collecting Steiner tree problems. Tech. Rep. 18-55, ZIB, Takustr. 7, 14195 Berlin (2018)

[13] Rehfeldt, D., Koch, T.: SCIP-Jack—a solver for STP and variants with parallelization extensions: An update. In: Operations Research Proceedings 2017. pp. 191 – 196 (2018)

[14] Rehfeldt, D., Koch, T.: Transformations for the Prize-Collecting Steiner Tree Problem and the Maximum-Weight Connected Subgraph Problem to SAP. Journal of Computational Mathematics **36**(3), 459 – 468 (2018)

[15] Rehfeldt, D., Koch, T., Maher, S.: Reduction Techniques for the Prize Collecting Steiner Tree Problem and the Maximum-Weight Connected Subgraph Problem. Networks (in press). https://doi.org/10.1002/net.21857, https://onlinelibrary.wiley.com/doi/abs/10.1002/net.21857

[16] Shinano, Y.: The ubiquity generator framework: 7 years of progress in parallelizing branch-and-bound. In: Kliewer, N., Ehmke, J.F., Borndörfer, R. (eds.) Operations Research Proceedings 2017. pp. 143–149. Springer International Publishing, Cham (2018)

[17] Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T., Winkler, M.: Solving hard miplip2003 problems with parascip on supercomputers: An update. In: IEEE (ed.) IPDPSW'14 Proceedings of the 2014 IEEE, International Parallel & Distributed Processing Symposium Workshops. pp. 1552 – 1561 (2014). https://doi.org/10.1109/IPDPSW.2014.174

[18] Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T., Winkler, M.: Solving open mip instances with parascip on supercomputers using up to 80,000 cores. In: Proc. of 30th IEEE International Parallel & Distributed Processing Symposium (2016). https://doi.org/10.1109/IPDPS.2016.56

[19] Shinano, Y., Berthold, T., Heinz, S.: A first implementation of paraxpress: Combining internal and external parallelization to solve mips on supercomputers. In: Greuel, G.M., Koch, T., Paule, P., Sommese, A. (eds.) Mathematical Software – ICMS 2016. pp. 308–316. Springer International Publishing, Cham (2016)

[20] Shinano, Y., Heinz, S., Vigerske, S., Winkler, M.: FiberSCIP – a shared memory parallelization of SCIP. INFORMS Journal on Computing **30**(1), 11–30 (2018). https://doi.org/10.1287/ijoc.2017.0762, https://doi.org/10.1287/ijoc.2017.0762