Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

YUJI SHINANO, DANIEL REHFELDT, TRISTAN GALLY

An Easy Way to Build Parallel State-of-the-art Combinatorial Optimization Problem Solvers: A Computational Study on Solving Steiner Tree Problems and Mixed Integer Semidefinite Programs by using ug[SCIP-*,*]-libraries

The work for this article has been conducted within the Research Campus Modal funded by the German Federal Ministry of Education and Research (fund number 05M14ZAM).This work was also supported by the North-German Supercomputing Alliance (HLRN). Supported by BMWi project BEAM-ME (fund number 03ET4023DE). Further funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Projektnummer 57157498 – SFB 805.

ZIB-Report 19-14 (March 2019)

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

An Easy Way to Build Parallel State-of-the-art Combinatorial Optimization Problem Solvers: A Computational Study on Solving Steiner Tree Problems and Mixed Integer Semidefinite Programs by using ug[SCIP-*,*]-libraries

Yuji Shinano*, Daniel Rehfeldt†, Tristan Gally‡

*Zuse Institute Berlin, †TU Berlin, ‡TU Darmstadt

Takustr. 7, 14195 Berlin, Germany

*shinano@zib.de, †rehfeldt@zib.de,

‡gally@mathematik.tu-darmstadt.de

March 20, 2019

Abstract

Branch-and-bound (B&B) is an algorithmic framework for solving NP-hard combinatorial optimization problems. Although several well-designed software frameworks for parallel B&B have been developed over the last two decades, there is very few literature about successfully solving previously intractable combinatorial optimization problem instances to optimality by using such frameworks. The main reason for this limited impact of parallel solvers is that the algorithmic improvements for specific problem types are significantly greater than performance gains obtained by parallelization in general. Therefore, in order to solve hard problem instances for the first time, one needs to accelerate state-of-the-art algorithm implementations. In this paper, we present a computational study for solving Steiner tree problems and mixed integer semidefinite programs in parallel. These state-of-the-art algorithm implementations are based on SCIP and were parallelized via the ug [SCIP-*,*]-libraries—by adding less than 200 lines of glue code. Despite the ease of their parallelization, these solvers have the potential to solve previously intractable instances. In this paper, we demonstrate the convenience of such a parallelization and present results for previously unsolvable instances from the well-known PUC benchmark set, widely regarded as the most difficult Steiner tree test set in the literature.

1 Introduction

Branch-and-bound (B&B) is an algorithmic framework for solving NP-hard combinatorial optimization problems. Several software frameworks for parallel B&B have been described in the literature [1–9], but when it comes to solving hard combinatorial optimization problem, one finds few success stories among those publications [2, 9–12]. The main reason for this limited impact of parallel solvers is that algorithmic improvements for specific problem types are significantly greater than performance gains obtained by parallelization in general. Therefore, in order to solve previously intractable problem instances, one needs to focus on accelerating state-of-the-art algorithm implementations.

SCIP, introduced in section 2, is a solver framework that is developed intensively by a group of researchers centered at Zuse Institute Berlin (ZIB), TU Darmstadt, and FAU Erlangen-Nürnberg. UG, also introduced in section 2, is a parallelization framework for existing B&B-based solvers on any kind of parallel computing environments. A particular instantiated parallel solver is referred to as *ug [solver name, parallelization library name]*. ug [SCIP,*], which is referred to as either FiberSCIP (for * = Pthreads/C++11) or ParaSCIP (for * = MPI), can be considered the parallel version of SCIP. The ug [SCIP-*,*]-libraries have been developed to allow SCIP users to parallelize a customized SCIP solver with minimal effort. This paper presents a computational study of solving Steiner tree problems and mixed integer semidefinite programs in parallel by using the ug [SCIP-*,*]-libraries.

The *Steiner tree problem in graphs* (SPG) is one of the fundamental NP-hard optimization problems [13]. 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. Practical applications of the SPG include for example the design of fiber-optic networks [14]. Furthermore, the relevance of the SPG in computer science and mathematics is highlighted by the existence of hundreds of research articles on both theoretical and practical aspects of the SPG.

The 2014 DIMACS Challenge, dedicated to Steiner tree problems, marked a revival of research on the SPG and related problems: Both at and in the wake of the challenge several new Steiner problem solvers were introduced and many articles were published. One of these new solvers is SCIP-Jack, which was by far the most versatile solver participating in the DIMACS Challenge, being able to solve the SPG and 10 related problems (in the current version two additional problem classes can be handled). Moreover, SCIP-Jack was able to win two parallel and two sequential categories of the DIMACS Challenge. Other solvers that successfully participated in the DIMACS Challenge are described in [15] and [16]. SCIP-Jack is described in detail in the article [17], but already in an updated version that vastly outperforms its predecessor participating in the DIMACS Challenge. The current version of SCIP-Jack, used in this article, again drastically improves on the state reported in [17]. These improvements were demonstrated at the *Parameterized Algorithms and Computational Experiments* (PACE) Challenge 2018 [18], dedicated to the SPG. The SPG allows for fixed-parameter tractable (FPT) algorithms in the number of terminals, and in the treewidth, and such algorithms were the focus of PACE 2018. Although SCIP-Jack does not include any FPT algorithms, it finished 3rd place in Track A (exact solution of problems with few terminals), 1st place in Track B (exact solution of problems with bounded treewidth), and 2nd in Track C (heuristic solution of problems with different structures). The high competitiveness with specialized FPT solvers was achieved despite the fact that the non-commercial, but considerably slower, LP solver SoPlex [19] was used by SCIP-Jack at PACE 2018 instead of the default, but commercial, Cplex.

Mixed integer semidefinite programming (MISDP) is the problem of optimizing a linear function under the constraint that some matrix $C - \sum_{i=1}^{m} A_i\, y_i$, depending affinely on the variables y_i, should be positive semidefinite, with some or all of the variables being integer. Many combinatorial problems can be brought in this form by incorporating stronger semidefinite relaxations, e. g., the stable set problem [20], the graph partitioning problem [21], the linear ordering problem [22], the quadratic assignment problem [23], the traveling salesperson problem [24], the bandwidth problem in graphs [25] or the single row-facility problem [26]. Furthermore, semidefinite relaxations also form the basis of some of the most successful solvers for the max-cut problem like Biq Mac [27] and BiqCrunch [28]. Outside of combinatorial optimization, there are also many other applications, e. g., in robust truss topology design [29,30], optimal power flow [31,32], regression models avoiding multicollinearity [33] and consensus-based communication systems [34, 35].

SCIP-SDP [36] supports two different solution approaches for mixed integer semidefinite programming, namely a nonlinear branch-and-bound approach with a penalty formulation for ill-posed relaxations and an LP-based cutting-plane approach using eigenvector cuts. A comparison with other MISDP solvers on the conic benchmark library (CBLIB) [37] is given in [38], showing that in particular the nonlinear branch-and-bound solver in SCIP-SDP currently seems to outperform its competitors on this test set. For specific applications, however, the LP-based approach can be preferable, which, as we will later explain, can be exploited in the parallelization.

This paper is organized as follows. In section 2, we introduce the solver framework SCIP, the parallelization framework UG and the ug [SCIP-*,*]-libraries to show the general concept of building a parallel combinatorial optimization solver by using the SCIP and ug [SCIP-*,*]-libraries. This is followed by an introduction of the two customized SCIP solvers SCIP-Jack and SCIP-SDP. Afterwards, we show some results for their parallel versions, before providing concluding remarks.

2 SCIP based Software libraries for general purpose parallel B&B

In this section, we introduce SCIP and the ug [SCIP-*,*]-libraries as general purpose parallelization libraries for state-of-the-art problem-specific implementations. In order to show their generality, we first introduce SCIP, before introducing UG and its capability to handle large-scale parallelism. Finally, we show how to combine SCIP and UG to introduce the ug [SCIP-*,*]-libraries.

2.1 SCIP: Solving Constraint Integer Programs

SCIP implements the idea of Constraint Integer Programming (CIP). CIP is formally defined as follows:

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

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

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

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

where $C := [n] \setminus I$ and $k \in \mathbb{N}$.

Condition (1) states that the problem becomes a linear program if 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. Many combinatorial optimization problems can be formulated as CIPs.

A CIP where 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

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

Extending the definition of MIP towards nonlinear objective and constraint functions leads to the class of mixed-integer nonlinear programs (MINLPs):

Definition 3 (mixed-integer nonlinear program)**.** A *mixed-integer nonlinear program* (MINLP) is given by a tuple (c, g, I) with vector $c \in \mathbb{R}^n$, a function $g : \mathbb{R}^n \to \mathbb{R}^m$, and a subset $I \subseteq [n]$. The task is to solve

$$\min\{c^\top x : g(x) \leq 0,\ x_I \in \mathbb{Z}^{|I|}\}.$$

Unlike MIP, MINLP is in general not a special case of CIP, since the nonlinear constraint $g(x) \leq 0$ may preclude a linear representation of the MINLP after fixing the integer variables, i.e., (1) would be violated (unless $I = [n]$). However, the main purpose of condition (1) is to ensure that the problem that remains after fixing all integer variables in the CIP can be solved efficiently. For practical applications, a spatial branch-and-bound algorithm that can solve the remaining nonlinear program within a finite number of steps up to a given precision is sufficient.

SCIP solves MIPs and MINLPs by a B&B algorithm that utilizes an LP relaxation for bounding. For a MIP, the LP relaxation is readily given by dropping the integrality restrictions from (2). For a MINLP, the LP relaxation is constructed by computing for each function $g_j(x)$, $j \in \{1, \ldots, m\}$, linear functions that underestimate $g_j(x)$ for all x within the current variable bounds, see also [39] for details. The "convexification gap" between the underestimator of $g_j(x)$ and $g_j(x)$ itself depends on the width of the domain of the variables involved in $g_j(x)$. Thus, to close this gap, branching on any variable that is involved in $g_j(x)$ may be applied.

The important features of SCIP for this paper are that SCIP

- is a branch-cut-and-price framework, and
- has a modular structure via plugins.

That is, SCIP is an extensible plugin-based software framework to develop discrete optimization solvers. Even the MIP and MINLP solvers have been developed using the plugin structure. The performance of SCIP has kept improving over the last decade and the composed plugins made it a full-scale state-of-the-art MIP and MINLP solver. The SCIP-Jack and SCIP-SDP extensions presented in this paper have also been developed by adding user plugins for the specific problems as SCIP user applications. A notable advantage of this approach is that SCIP users can enjoy all benefits of state-of-the-art MIP or MINLP solving techniques immediately in their customized applications.

2.2 UG: Ubiquity Generator framework

UG is a generic framework to parallelize any existing state-of-the-art B&B-based solver, subsequently referred to as the *base solver*. UG is composed of a collection of base C++ classes, which define interfaces that can be customized for any base solver and allow descriptions of subproblems and solutions to be translated into a solver independent form. Additionally, there are base classes that define interfaces for different message-passing protocols. Implementations of a *ramp-up*, which is the process until all solvers become active, dynamic load balancing, and check-pointing and restarting mechanisms are available as a generic functionality (for more details see [40]).

The base solver is wrapped to UG's `ParaSolver` and a specified number of `ParaSolvers` are generated as threads or processes depending on the run-time system. The LoadCoordinator thread or process coordinates the workload among the ParaSolvers. The B&B tree is maintained in the base solvers, while UG only extracts and manages a small number of subproblems from the base solvers for load balancing.

According to the term defined in [41], UG employs a *Supervisor-Worker coordination mechanism* with subtree-level parallelism (the unit of work is a subtree). In UG, the LoadCoordinator corresponds to the Supervisor and the ParaSolver corresponds to the Worker. Algorithm 1 and Algorithm 2 show a parallel algorithm with a simplified Supervisor-Worker coordination mechanism, see also [42].

One of the most important characteristics of UG is that it makes "algorithmic changes" to the base solver, such as *layered presolving*, and performs highly adaptive algorithms, such as *racing ramp-up* [40], or distributed domain propagation [43]. Here, "algorithmic changes" means that the base solver and the instantiated parallel solver ug [base solver,*] perform algorithmically differently. For example, SCIP and FiberSCIP solve a MIP instance quite differently. Current state-of-the-art MIP solvers perform strong preprocessing (presolving) on the *original instance* to be solved before they start solving the instance. We refer to the adjusted instance after the preprocessing as the *presolved instance*. In a solver instantiated by UG, in general, the preprocessing is performed once in the LoadCoordinator and then all ParaSolvers solve the presolved instance. Inside of each ParaSolver, a received subproblem instance is presolved again. We refer to this presolving mechanism as *layered presolving*.

A natural way to ramp-up is for all active ParaSolvers, i. e., all ParaSolvers that have already received a subproblem, to send one of their branched nodes to another ParaSolver via the LoadCoordinator. We refer to this procedure as *normal ramp-up*. In racing ramp-up, after initialization the LoadCoordinator sends the root node of the branch-and-bound tree to all ParaSolvers simultaneously and each ParaSolver starts solving the root node of the presolved instance immediately. In order to generate different search trees, even though they work on the same problem, each ParaSolver uses different parameter settings and permutations of variables and constraints. As shown in [44], the latter can have a considerable impact on the performance of a solver due to imperfect tie breaking. Due to these variations, one can expect that the ParaSolvers will generate different search trees. After a specified amount of time or after the number of open nodes in the most promising ParaSolver has reached a specified limit, one ParaSolver is chosen

Algorithm 1 Supervisor

Input: Single base solver, set of N processors $i \in S = \{1, \ldots, N\}$ and an instance to be solved
Output: An optimal solution

```
Spawn N Workers with the base solver on processors 1 to N
collectMode ← false
x* ← NULL
I ← N \ {1}
A ← {1}
Q ← ∅
R ← {(1,0)} // Subproblems currently being processed, 0 is the index of the root
            // problem
Send the root problem to processor 1
while Q ≠ ∅ and R ≠ ∅ do
    (i, tag) ← Wait for message // Returns processor identifier and message tag
    if tag = solutionFound then
        Receive solution x̂ from processor i
        if x* = NULL or c⊤x̂ < c⊤x* then
            x* ← x̂
        end if
    else if tag = subproblem then
        Receive a subproblem indexed by k from processor i
        Q ← Q ∪ {k}
    else if tag = terminated then
        R ← R \ {(i,j)} // j is the index of the terminated subproblem
        A ← A \ {i}, I ← I ∪ {i}
    else if tag = status then
        if collectMode = true then
            if there are enough heavy subproblems in Q then
                // heavy subproblem is a subproblem which is expected to generate
                //  a large subtree
                Send message with tag = stopCollecting to processors in collecting mode.
                collectMode ← false
            end if
        else
            // collectMode = false
            if there are not enough heavy subproblems in Q then
                Select processors which have heavy subproblems
                Send message with tag = startCollecting to the selected processors
                collectMode ← true
            end if
        end if
    end if
    while I ≠ ∅ and Q ≠ ∅ do
        choose processor i ∈ I, I ← I \ {i}, A ← A ∪ {i}
        choose subproblem j ∈ Q, Q ← Q \ {j}, R ← R ∪ {(i,j)}
        Send subproblem j and x* to processor i
    end while
end while
∀i ∈ S : Send message with tag = termination to processor i
Output x*
```

Algorithm 2 Worker

Input: A base solver and an instance to be solved

```
collectMode ← false
terminate ← false
while terminate = false do
    (i, tag) ← Wait for message from Supervisor // Returns Supervisor identifier 0
                                                // and message tag
    if tag = subproblem then
        Receive subproblem and solution from Supervisor
        Solve the subproblem, periodically communicating with supervisor as follows
          - Send message with tag solutionFound anytime a new solution is discovered.
          - Periodically send message with tag status to report current lower bound for this subproblem.
          - When messages with tag startCollecting or stopCollecting are received, toggle collectMode.
          - When collectMode = true, periodically send message with tag subproblem containing best
                candidate subproblem.
        Send a message with tag = terminated
    else if tag = termination then
        terminate ← true
    end if
end while
```

as the "winner" of this *racing stage*. The winning criterion is a combination of the lower bound and the number of open nodes of the ParaSolver. All open nodes of the "winner" are then collected by the LoadCoordinator and a termination message is sent to all other ParaSolvers. The search trees of the other ParaSolvers are discarded and the solvers become idle. Only the feasible solutions found during their solving process are kept. The collected nodes are then redistributed to the now idle ParaSolvers. The current UG version includes *customized racing*, which allows users to give a set of problem-specific parameters for the racing stage.

In UG, checkpointing saves only *primitive* nodes, which are nodes that have no ancestor nodes in the LoadCoordinator. This strategy requires much less effort for the I/O system than saving all open nodes to a disk, in particular in large-scale parallel computing environments, but potentially creates a computational overhead after the restart. However, the effort to regenerate the search tree is often outweighed by the benefits of re-applying a global presolving procedure during the restart (see [45]).

The concept of UG is thus to abstract from a base solver and parallelization library and to provide a framework that can be used, in principle, to parallelize any powerful state-of-the-art base solver on any computational environment. For a particular base solver, only the interface to UG in the form of specializations of base classes needs to be implemented. Similarly, for a particular parallelization library, a specialization of an abstract UG class is necessary.

As we already mentioned, a particular instantiated parallel solver is referred to as ug [base solver name, parallelization library name]. Here, the specific parallelization library is used to realize the message-passing based communications. The following solvers are parallelized by UG as the base solvers:

- Single thread academic solver SCIP[1]
- Multi-threaded commercial solver FICO Xpress[2]
- Distributed memory parallel solver for two-stage stochastic programming problems PIPS-SBB [42]

This means that UG can be used to parallelize multi-threaded and distributed memory solvers. The following parallelization libraries can be used currently:

- Message Passing Interface libraries, referred to as "MPI"
- pthreads library, referred to in the instantiated solver as "Pthreads"
- C++11 threads, abbreviated in the instantiated solver as "C++11"

In the following we introduce the parallel solvers instantiated by UG. ParaSCIP (= ug [SCIP, MPI]) [46] and FiberSCIP (= ug [SCIP, Pthreads/C++11]) [40] are algorithmically identical, since they are parallelized by the same software framework UG. The run-time behavior has been investigated in detail for the MIPLIB2010 benchmark instances by using FiberSCIP. ParaSCIP successfully solved 14 previously unsolved instances from MIPLIB2003 and MIPLIB2010 as of writing this document [45, 47]. The longest and the biggest scale computation conducted to solve an open instance by ParaSCIP is presented in [47, 48]. The `rmine10` instance from MIPLIB2010 was solved for the first time with 48 restarted runs from checkpoint files that were generated by previous runs using between 6144 and 80,000 cores of the HLRN III supercomputer at Zuse Institute Berlin and the TITAN supercomputer at

[1] http://scip.zib.de/

[2] https://www.fico.com/en/products/fico-xpress-optimization

Oak Ridge National Laboratory. In total, it took about 75 days or 6,405 years of CPU core hours.

ParaXpress (= ug [Xpress, MPI]) and FiberXpress (= ug [Xpress, Pthreads/C++11]) are instantiated versions of the shared memory parallel MIP solver Xpress. Therefore, FiberXpress can be viewed as a multi-level threaded parallel shared memory MIP solver. When there is more than one core, it is necessary to decide how many cores are assigned to UG threads and how many to the Xpress threads. The assignment also changes the solving behavior of the algorithm. ParaXpress has the same assignment issue between UG processes and FICO Xpress internal threads. The difference in assignments was investigated in [49].

ug [PIPS-SBB,MPI]) [42] is an instantiated version of PIPS-SBB, which can solve large-scale LPs on distributed memory computing environments. Therefore, this parallel solver instantiation shows that UG is capable of parallelizing parallel distributed memory base solvers.

2.3 ug[SCIP-*,*]-libraries: Parallelization libraries for customized SCIP solvers

Since SCIP has a plugin-based software architecture, problem-specific *customized SCIP solvers* like SCIP-Jack and SCIP-SDP presented in this paper can be developed by adding *user-plugins* which are algorithm implementations for the specific problem. Conceptually, the user-plugins can be installed into FiberSCIP (= ug [SCIP, Pthreads/C++11]) and ParaSCIP (= ug [SCIP, MPI]), too, though it was difficult to make it work as we expected. We added a feature to ug [SCIP,*] so that they optionally can install also the user-plugins and built the ug [SCIP-*,Pthreads/C++11]- and ug [SCIP-*, MPI]-libraries. These libraries are referred as *ug [SCIP-*,*]-libraries*.

Users of ug [SCIP-*,*]-libraries need to write glue code to install their user-plugins to FiberSCIP or ParaSCIP. However, the glue code is basically just a list of user-plugin declarations for the specific problem. In order to parallelize a customized SCIP solver, the user needs to add a file which contains the declarations in an extended class of `ScipUserPlugins`. Actually, the latest release of UG, which is included in the SCIP Optimization Suite 6.0.1[3], contains the Steiner tree problem and mixed integer semidefinite programming applications as `ug_scip_applications/STP` and `ug_scip_applications/MISDP`, respec-

[3] `https://scip.zib.de/index.php#scipoptsuite`

tively. Each of them contains only a single source file: `stp_plugins.cpp` within `STP/src` and `misdp_plugins.cpp` within `MISDP/src`. The number of lines counted by *cloc (Count Lines of Code)*[4] is 173 for `stp_plugins.cpp` and 106 for `misdp_plugins.cpp` without blank and comment lines. All remaining code is included in the sequential distributions which are available in the SCIP and SCIP-SDP package. Therefore, the additional effort needed to parallelize their sequential versions is less than 200 lines of code.

A big advantage of the ug [SCIP-*, *]-libraries is not only the minimal effort to parallelize a customized SCIP solver, but also that performance improvements of both SCIP and the customized solver are directly applicable to the parallelized version as long as they are included in the interface in case of new plugins. Only the rare cases of fundamental algorithmic changes, like constraint branching, require adjustments directly within the ug [SCIP-*,*]-libraries.

3 State-of-the-art solvers to be parallelized

In this section, we briefly describe the customized SCIP solvers SCIP-Jack and SCIP-SDP, which are SCIP-based state-of-the-art algorithm implementations for the Steiner tree problem and for mixed integer semidefinite programming, respectively—their code is included in the SCIP Optimization Suite and the SCIP-SDP package.

3.1 SCIP-Jack: A Steiner tree problem solver

SCIP-Jack includes a wide range of generic and problem-specific algorithmic components, most of them falling into one of the following three categories.

First, reduction techniques are extremely important (both in presolving and domain propagation). Apart from some instances either specifically constructed or insightfully handpicked to defy reduction techniques, such as the PUC [50] and I640 [51] test sets, preprocessing is usually able to significantly reduce instances. Often more than 90 % of the edges of a given problem can be deleted by reduction techniques.

Second, heuristics are essential to find good or even optimal solutions and help find strong upper and lower bounds quickly. Having a strong primal

[4] `https://github.com/AlDanial/cloc`

bound available is a prerequisite for the reduced cost based domain propagation routines in SCIP-Jack. Furthermore, heuristics can be especially important for hard instances, for which the dual bound often stays substantially below the optimum for a long time. Most heuristics implemented in SCIP-Jack can be used for several problem classes, but there are also problem-specific ones, e.g., for the maximum-weight connected subgraph problem [52].

Finally, the core of SCIP-Jack is constituted by graph-transformations and a branch-and-cut procedure used to compute lower bounds and prove optimality. SCIP-Jack transforms all problem classes to the Steiner arborescence problem (sometimes with additional constraints), which is defined as follows. Given a directed graph $D = (V, A)$, costs $c : A \to \mathbb{Q}_+$, a set $T \subseteq V$ of terminals, and a root $r \in T$, a directed tree $S = (V(S), A(S)) \subseteq D$ is required that first, for all $t \in T$ contains exactly one directed path from r to t and second, minimizes

$$\sum_{a \in A(S)} c(a).$$

Thereupon, one can use the following formulation:

Formulation 1. Flow Balance Directed Cut Formulation

$$\begin{aligned}
&\min c^\top y && && (3)\\
y(\delta^+(W)) &\geq 1, && \forall W \subset V : r \in W, (V \setminus W) \cap T \neq \emptyset && (4)\\
y(\delta^-(v)) &\leq y(\delta^+(v)), && \forall v \in V \setminus T && (5)\\
y(\delta^-(v)) &\geq y(a), && \forall a \in \delta^+(v), \forall v \in V \setminus T && (6)\\
y(a) &\in \{0,1\}, && \forall a \in A, && (7)
\end{aligned}$$

where the notation $y(A') := \sum_{a \in A'} y(a)$ for a set $A' \subseteq A$ is used. Only constraints (4) and (7) are necessary for the validity of the IP formulation, but (5) can improve the LP-relaxation [53] and (6) (while not changing the optimal value of the LP-relaxation [54]) can often speed up the solving process when a branch-and-cut approach is used. Both theoretically [53] and practically [54] the LP-relaxation of Formulation 1 has been shown to be superior to other (in particular undirected) MIP formulations.

After presolving, SCIP-Jack runs a dual-ascent heuristic [55] to select a set of constraints from (4) to be included into the initial LP (and to find a feasible solution [17]). Subsequently, the LP is solved and a separator routine based on a maximum-flow algorithm is used to find violated constraints. The violated constraints are added to the LP and the procedure is reiterated as long as the dual-bound can be sufficiently improved. Otherwise branching is initiated.

During branch-and-cut, domain propagation and several (constructive and local) primal heuristics are applied to speed up the solution process.

3.2 SCIP-SDP: A solver for mixed integer semidefinite problem

SCIP-SDP [36] is a solver for mixed integer semidefinite programs of the form

$$\begin{aligned} \sup \quad & b^\top y \\ \text{s.t.} \quad & C - \sum_{i=1}^{m} A_i\, y_i \succeq 0, \\ & \ell_i \leq y_i \leq u_i \qquad\qquad \forall\, i \in [m]\,, \\ & y_i \in \mathbb{Z} \qquad\qquad\qquad \forall\, i \in I \end{aligned} \tag{8}$$

with a symmetric matrix $C \in \mathcal{S}^n$, $b \in \mathbb{R}^m$, $A_i \in \mathcal{S}^n$, $\ell_i \in \mathbb{R} \cup \{-\infty\}$, $u_i \in \mathbb{R} \cup \{+\infty\}$ for all $i \in [m]$ and index set of integer variables $I \subseteq [m]$.

For solving MISDPs, SCIP-SDP supports two different solution approaches. On the one hand, the MISDPs can be solved similarly to general MINLPs in SCIP by combining LP relaxations and polyhedral approximations of the nonlinear constraints. For SDP constraints, this can be done through the eigenvector cuts introduced by Sherali and Fraticelli [56]. Since $C - \sum_{i=1}^{m} A_i\, y_i$ is positive semidefinite if and only if

$$v^\top \Big(C - \sum_{i=1}^{m} A_i\, y_i \Big) v \geq 0 \tag{9}$$

holds for all $v \in \mathbb{R}^n$, Inequality (9) is a valid inequality for the convex hull of the feasible set of (8) for any $v \in \mathbb{R}^n$. To enforce the positive semidefiniteness, one only needs to find a $v \in \mathbb{R}^n$ such that (9) is violated for a given solution y^* of the polyhedral approximation that is not feasible for the SDP constraint. One possible choice for v is an eigenvector to the smallest eigenvalue of $Z^* \coloneqq C - \sum_{i=1}^{m} A_i\, y_i^*$. Since y^* is not feasible for the SDP constraint, the smallest eigenvalue $\lambda_{\min}(Z^*)$ is negative and we get that

$$v^\top \Big(C - \sum_{i=1}^{m} A_i\, y_i^* \Big) v = \lambda_{\min}\left(Z^*\right) \|v\|_2^2 < 0,$$

thus Inequality (9) with this choice of v can be used to enforce positive semidefiniteness.

The second solution approach implemented in SCIP-SDP is nonlinear branch-and-bound. In this case in each node of the branch-and-bound tree a continuous semidefinite program is solved by interfacing interior-point SDP solvers like Mosek [5]. One difficulty in this case is ensuring the necessary assumptions to guarantee convergence of the interior-point solvers, namely the existence of primal and dual strictly feasible solutions, usually referred to as the *Slater condition*, which may be harmed by branching. Within SCIP-SDP, a penalty approach is used to ensure in particular the dual Slater condition in this case, as explained in [36].

Furthermore, SCIP-SDP includes additional branching rules, heuristics, presolving and propagation techniques like dual fixing and randomized rounding, for more details see [36] and [38].

In addition to allowing a parallelization of the branch-and-bound tree in either the LP- or the SDP-based approach within SCIP-SDP, ug [SCIP-SDP,*] exploits the racing ramp-up to create a hybrid solver utilizing both solution approaches. More precisely, the solution process in ug [SCIP-SDP,*] starts by creating a number of SCIP-SDP solver instances with half of them using LP-based settings and the rest using SDP-settings, with other parameter settings also being changed within the different LP- or SDP-based solvers. In this way, racing ramp-up allows to dynamically choose between linear and semidefinite relaxations for solving MISDPs, depending on whichever approach works best for a particular instance.

4 Computational experiments

Computational experiments were conducted to show the effectiveness and potential of our parallelization approach using SCIP and the ug [SCIP-*,*]-libraries.

4.1 Results of ug [SCIP-Jack,*]

Before looking at the massive parallelization provided by ug [SCIP-Jack, MPI], we present results of ug [SCIP-Jack, C++threads] on selected instances to provide some insight into the behavior and difficulties of a (shared-memory)

[5] `https://www.mosek.com/`

B&B parallelization of a state-of-the-art Steiner tree problem solver. The experiments were performed on a machine with 88 cores equipped with Intel(R) Xeon(R) E7-8880 v4 CPUs with 2.20GHz, and 2 TB RAM. Normal ramp-up was used in ug and CPLEX 12.7.1 was used as underlying LP solver for SCIP-Jack. Table 1 provides results for five instances from the PUC test set. *root time* gives the time that was spent at the root of the B&B tree, *max # solvers* states the maximum number of ParaSolvers that were used during the computation, and *first max active time* signifies the first point of time when this maximum number of solvers was used.

The worst scaling behavior can be observed for the first instance *cc3-4p*, the best for the last one *hc7u*. A look at the statistics explains this behavior: Instance *cc3-4p* shows the relative highest root time (at which no parallelization can be performed), and, more importantly, the maximum number of active solvers during the computation is 13 (so one cannot expect any speed-up when using more than 14 threads). On the other hand, the instance *hc7u* spends relatively less time at the root node, can utilize all 64 threads, and has the (relatively) shortest ramp-up phase of all instances—with just 42 seconds.

Table 1: Shared memory results for selected Steiner tree instances. All times in seconds.

# Threads	cc3-4p	cc3-5u	cc5-3p	hc7p	hc7u
1	105	3,979	8,234	3,970	5,106
8	56	2,614	2,349	1,683	1,738
16	58	1,801	2,500	1,060	1,106
32	55	2,249	1,793	671	761
64	53	1,899	1,433	479	498
root time	6	10	109	8	16
max # solvers	13	52	64	64	64
first max active time	11	152	466	41	42

ug [SCIP-Jack, MPI] was the only solver that could run on a distributed environment at the 11th DIMACS Challenge. Moreover, it solved three open instances and updated 14 best known solutions to instances [57] of the notoriously hard PUC test set from the SteinLib [58]. After that, no open instance was solved by ug [SCIP-Jack, MPI] until new features were added to the ug [SCIP-*,*]-libraries, even though SCIP-Jack had continuously been improving. However, most of these improvements could not be exploited in the parallel version due to missing support for constraint branching in the ug [SCIP-*, *]-libraries and the lack of a user routine to communicate previous branching decisions to each ParaSolver. After the support had been added in version ug-0.8.6, ug [SCIP-Jack, MPI] caught up with the improvements of SCIP-Jack: ug [SCIP-Jack, MPI] solved hp9p and updated the best known solution of hc11p [59].

Recently, SCIP-Jack has again been improved: The most important enhancement is a (still rather limited) implementation of *extended reduction techniques* [54]. These techniques try to prove that a subgraph G' (usually a single edge or vertex) is not part of at least one optimal Steiner tree by considering a (sufficient) set of supergraphs of G' and showing that all of them are not contained in at least one optimal Steiner tree. The realization of these techniques is highly intricate, but already the initial, and rather restricted, implementation in SCIP-Jack allowed us to delete about 8% more edges in general—which still falls far short of the results reported in [54]. It should be noted that for the PUC instances the effect of presolving is usually very limited. Nevertheless, the above-mention initial implementation of extended reduction techniques has proven useful if combined with a massive B&B search, as provided by UG. Since each branching either deletes a vertex or adds a terminal, the underlying graph can take a very different shape deep in the B&B tree, as compared to the original problem. On these modified graphs the extended reduction method often can lead to considerable further reductions of the problem (which can be translated into variable fixings in the IP formulation). With this improved version of ug [SCIP-Jack, MPI] we could solve the previously unsolved PUC instance bip52u to optimality and moreover updated the best known solution to hc10p.

For solving open instances of the PUC test set we used two supercomputers. One, based at ISM (Institute of Statistical Mathematics), is a HPE SGI 8600 with 384 compute nodes, with each node consisting of two Intel Xeon Gold 6154 3.0GHz CPUs (18 cores×2) sharing 384GB of memory, and an Infiniband (Enhanced Hypercube) interconnect. The other (HLRN III) is a Cray XC40 with 1872 compute nodes, each node consisting of two 12-core Intel Xeon IvyBridge/Haswell CPUs sharing 64 GiB of RAM, and with an Aries interconnect.

Table 2 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 the ParaSolvers, primal and dual bounds, the 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 are shown in the lower row for each run. The run number 1.* means that they are a series of runs from the previous checkpoint files.

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 subtree roots. For example, run 1 ends with 271,781 open B&B nodes, but run 2 starts with only 18 open ones. This means that only 18 B&B subtree roots existed at the end of run 1.1 and the other subtree roots were descendants of one of these 18 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. Naturally, at larger scale one would expect more layered presolving.

Table 2: Statistics for solving `bip52u` 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	604,790 (335)	< 0.1	2,021	233.0000	229.1728	1.67	0	0
						233.0000	230.9019	0.91	79,002,896	271,781
1.2	ISM	72	604,798	< 0.1	2,311	233.0000	230.9018	0.91	0	18
						233.0000	230.9137	0.90	80,790,403	225,548
1.3	HLRN III	12,288	431,992	< 0.3	3,451,630	233.0000	230.9137	0.90	0	11
						233.0000	230.9575	0.88	5,712,626,116	465,910
1.4	HLRN III	12,288	561,590	< 1.5	9,518,991	233.0000	230.9575	0.88	0	24
						233.0000	231.2956	0.74	7,173,350,123	47,488
1.5	HLRN III	12,288	43,180	< 4.7	2,236,869	233.0000	231.2956	0.74	0	5
						233.0000	231.2956	0.74	54,2635,223	237,489
1.6	ISM	2,304	302,900	0.2	3,797,932	233.0000	231.2956	0.74	0	1,113
						233.0000	233.0000	0.00	1,465,480,096	0

The best known solution to the `hc10p` instance could be updated (to an objective value 59,733, as compared to 59,797 at the DIMACS Challenge). Table 3 shows the statistics. The first additional run (1) on the ISM supercomputer generated five new incumbent solutions, with the best objective value being 59,776. Afterwards we just reran from scratch with the best solution from run 1 with racing ramp-up (run 2)—since the best solution can be used for presolving, propagation, and heuristics. The second run with the new solution again generated an improved solution. The job was killed to restart with this updated solution. The third run with the new solution once more generated an improved solution, after 76,405 seconds.

Table 3: Statistics for solving `hc10p` on supercomputers

Run	Computer	Cores	Time (sec.)	Idle (%)	Trans.	Primal bound (Upper bound)	Dual bound (Lower bound)	Gap (%)	Nodes	Open nodes
1	ISM	72	604,796	< 0.1	118	59,797.0000	59213.4370	0.99	0	0
			(926)			59,776.0000	59,330.3673	0.75	19,811,438	1,030,317
2	ISM	72	5,857	< 5.2	91	59,776.0000	59,213.8774	0.94	0	0
			(973)			59,772.0000	59,237.1542	0.90	160,594	11,365
3	ISM	72	604,805	< 0.1	86,152	59,772.0000	59,213.4370	0.94	0	0
			(1021)			59,733.0000	59,331.5374	0.68	18,458,047	887,762

4.2 Results of ug [SCIP-SDP,*]

For measuring the speedup of the parallelization of SCIP-SDP via the ug [SCIP-*,*]-libraries, we ran SCIP-SDP and ug [SCIP-SDP,C++11] with a different number of threads on a shared memory environment of Intel Xeon E5-4650 CPUs running at 2.70 GHz with 512 GB of shared RAM. The tests used current developer versions of SCIP-SDP 3.1.1, SCIP 6.0.0 and ug 0.8.6 together with Mosek 8.1.0.54. Table 4, which first appeared in [38], shows an overview of the solution times as a shifted geometric mean with shift $s = 10$ as well as the number of solved instances for SCIP-SDP and ug [SCIP-SDP,C++11] with 1 to 32 threads over the complete CBLIB [37] and the different application-specific test sets.

Table 4: Results for ug [SCIP-SDP,C++11] over all 194 CBLIB instances

solver	TTD		CLS		Mk-P		Total	
	solved	time	solved	time	solved	time	solved	time
SCIP-SDP	55	84.01	62	142.19	**67**	**54.44**	**184**	86.59
ug [SCIP-SDP,C++11] 1 thr.	54	107.49	62	156.70	58	107.81	174	122.23
ug [SCIP-SDP,C++11] 2 thr.	56	64.93	64	23.31	56	92.25	176	53.79
ug [SCIP-SDP,C++11] 4 thr.	58	39.76	**65**	18.48	60	85.61	183	42.07
ug [SCIP-SDP,C++11] 8 thr.	58	32.07	**65**	**14.51**	60	72.35	183	34.57
ug [SCIP-SDP,C++11] 16 thr.	**59**	**21.03**	**65**	16.37	59	78.46	183	**32.65**
ug [SCIP-SDP,C++11] 32 thr.	**59**	21.27	**65**	18.38	56	92.14	180	36.11

The first observation is that we get a slowdown when running ug [SCIP-SDP,C++11] single-threaded compared to SCIP-SDP between 10 % on the cardinality-constrained least squares instances and 98 % over the minimum k-partitioning test set, with an average of 41 % over the whole CBLIB. When comparing ug [SCIP-SDP,C++11] with different thread numbers on the truss topology test set, we get a relatively constant speedup of 20 to 40 % when doubling the number of threads. The maximum speedup is already reached for 16 threads, however, due to the size of these instances, which have been designed as a test set for sequential solvers. Nevertheless, the parallelization leads to a total speedup of 75 % for 16 threads compared to regular SCIP-SDP on this test set and allows to solve four of the five instances that could not be solved by regular SCIP-SDP on this machine.

On the cardinality-constrained least squares instances, the results are significantly different. For these instances, we get a very significant speedup of 85 % from one to two threads, i. e., when first including the LP-based cutting plane approach within the racing settings, showing that these instances are much more suited for an LP-based approach. When increasing the number of threads further, the speedup is smaller with a relatively constant 20 %, and the best performance already occurs on eight threads with a slowdown of around 12 % when increasing the number of threads to 16 and 32. The main reason for tailing off earlier is the much smaller number of branch-and-bound nodes compared to the truss topology instances, which does not allow a large number of SCIP-SDP ParaSolvers to be active at the same time.

The parallelization performs worst on the minimum k-partitioning instances. Not only is the slowdown on one thread larger than for any other test set, we also neither get a large speedup when adding the LP settings nor do we get a significant speedup when increasing the number of threads beyond that. The speedup is in the range of 7 to 15 % when doubling the number of threads up to eight threads, but we already get slowdowns for 16 and 32 threads. This causes the minimum k-partitioning test set to be the only one where ug [SCIP-SDP,C++11] never reaches the performance of SCIP-SDP. The main reason for the bad performance on these combinatorial instances seems to be that the additional local presolving performed by the UG framework leads to different search paths being taking in the branch-and-bound tree, which for some reason are worse and lead to longer solving times than in the sequential case.

Over the whole CBLIB, we see a combination of the effect of adding LP settings and the general speedup of the parallelization. Together this leads to a speedup of 56 % for two threads, with further speedups of 22 % and 18 % when increasing to four and eight threads, respectively. The optimal performance on this architecture and test set is obtained for 16 threads with a total speedup of 73 % compared to running ug [SCIP-SDP,C++11] single-threaded and 62 % compared to SCIP-SDP. Note, however, that the UG parallelization is designed for larger instances, thus on different test sets one should expect additional speedups for 32 and more threads.

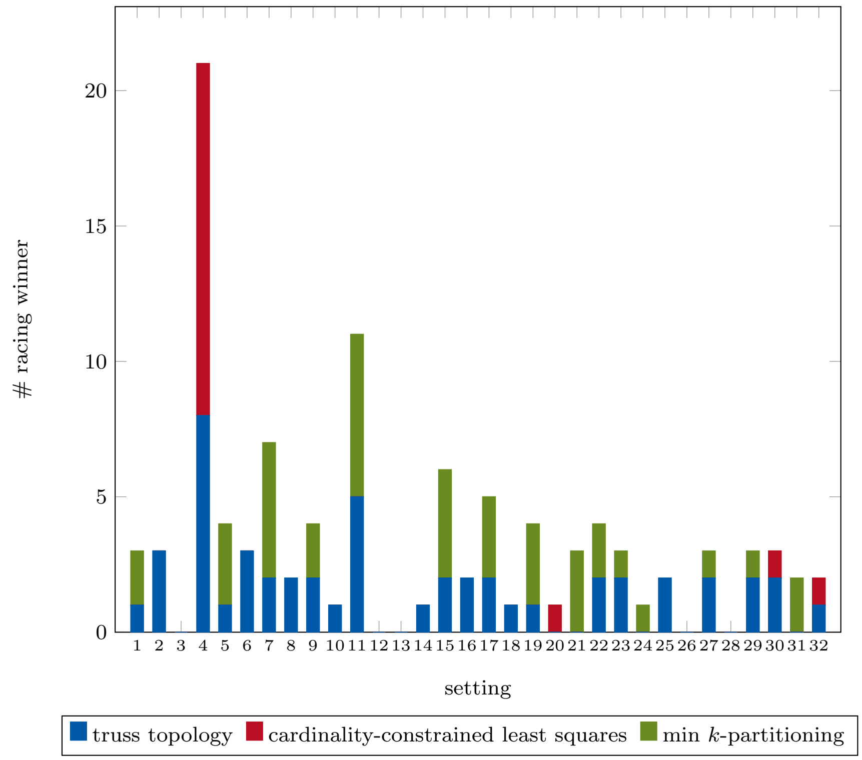

Figure 1: Racing ramp-up statistics for the different settings over CBLIB

Figure 1 shows for how many instances each of the different settings has been declared the winner after racing, with the color indicating the test set the instance belongs to. In this case, each odd number refers to an SDP-based setting while all even numbers belong to LP-based settings, the exact changes for each setting can be found in the `settings` subfolder or in [38]. Note that the statistics only include instances that continued after the racing ramp-up phase, while instances that were solved to optimality by one of the SCIP-SDP instances during racing have been excluded. First of all, it can be observed that most settings turn out to be optimal for at least some subset of the instances. For truss topology design, many different settings are chosen with the most successful one being the LP approach with `easycip` emphasis. In total, LP-based settings are chosen for 52 % of all truss topology instances. The results for the cardinality-constrained least squares instances are much more lopsided. For these instances, only LP settings are chosen and in all but three cases the `easycip` emphasis. Note, however, that many of these instances are already solved during racing. The minimum k-partitioning instances show a completely opposite behavior, with almost exclusively SDP-based settings being chosen and no single setting being chosen significantly more often than the others.

Unfortunately, we did not have enough supercomputer resources to conduct computational experiments for ug [SCIP-SDP,MPI]. However, the results presented in this section, arguably, show its potential to tackle previously unsolvable large-scale instances.

5 Concluding remarks

The two customized SCIP solvers SCIP-Jack and SCIP-SDP were parallelized through the ug [SCIP-*,*]-libraries. Currently SCIP has over 800,000 lines of C code, which includes many MIP solving algorithm implementations, and SCIP-Jack and SCIP-SDP have been developed on top of that, benefiting from the SCIP features. In general, it would be an extremely hard task to parallelize such a huge code on a large scale distributed memory computing environment. Through integration with the continuous development and testing of Fiber-SCIP and ParaSCIP, the ug [SCIP-*,*]-libraries allow SCIP users to parallelize their customized SCIP solvers to run on supercomputers with at least up to 80,000 cores [47] by adding less than 200 lines of glue code. Due to a lack of supercomputer resources, we only provided restricted computational results, but potentially, ug [SCIP-Jack, MPI] could solve more previously unsolvable instances of the Steiner tree problem in graphs and related problems and also ug [SCIP-SDP,MPI] should be able to run on a supercomputer immediately.

CIP covers almost all classes of combinatorial optimization problems. Therefore, SCIP can be used as a solver for almost all of them. Users of SCIP can get the benefits of the MIP solving technology within a customized SCIP solver, which can be developed without any consideration for its parallelization. As long as the user utilizes SCIP's plug-in based software architecture, the customized solver can be parallelized easily by using the ug [SCIP-*,*]-libraries. In this way, the authors hope that the results presented in this article will encourage additional scientists and practitioners who wish to solve a particular combinatorial optimization problem on supercomputers, to implement their algorithms in SCIP and immediately profit from the parallelization capabilities provided by the ug [SCIP-*,*]-libraries.

Acknowledgment

The authors would like to thank Chuen Teck See for first experimenting with a parallel version of SCIP-SDP, and Marc E. Pfetsch for many helpful suggestions regarding the parallelization of SCIP-SDP.

References

[1] Y. Shinano, M. Higaki, and R. Hirabayashi, "A generalized utility for parallel branch and bound algorithms," in *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, 1995, pp. 392–401.

[2] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt, "The parallel search bench zram and its applications," *Annals of Operations Research*, vol. 90, no. 0, pp. 45–63, 1999.

[3] S. Tschöke and T. Polzer, "Portable Parallel Branch-and-Bound Library PPBB-Lib," University of Paderborn, User Manual Version 2.0, 1996.

[4] M. Bénichou, V.-D. Cung, S. Dowaji, B. L. Cun, T. Mautor, and C. Roucairol, "Building a parallel branch and bound library," in *Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science* **1054**. Berlin: Springer, 1996, pp. 201–231.

[5] A. Djerrah, B. L. Cun, V. D. Cung, and C. Roucairol, "Bob++: Framework for solving optimization problems with branch-and-bound methods," in *2006 15th IEEE International Conference on High Performance Distributed Computing*, 2006, pp. 369–370.

[6] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth, "Master–worker: An enabling framework for applications on the computational grid," *Cluster Computing*, vol. 4, no. 1, pp. 63–70, 2001.

[7] T. K. Ralphs, L. Ladányi, and M. J. Saltzman, "A library hierarchy for implementing scalable parallel search algorithms," *The Journal of Supercomputing*, vol. 28, pp. 215–234, 2004.

[8] Y. Sun, G. Zheng, P. Jetley, and L. Kale, "Parssse: An adaptive parallel state space search engine," *Parallel Processing Letters*, vol. 21, no. 3, pp. 319–338, 2011.

[9] J. Eckstein, W. E. Hart, and C. A. Phillips, "Pebbl: an object-oriented framework for scalable parallel branch and bound," *Mathematical Programming Computation*, vol. 7, no. 4, pp. 429–469, 2015.

[10] Y. Shinano, T. Fujie, Y. Ikebe, and R. Hirabayashi, "Solving the maximum clique problem using pubb," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998, pp. 326–332.

[11] K. Anstreicher, N. Brixius, J.-P. Goux, and J. Linderoth, "Solving large quadratic assignment problems on computational grids," *Mathematical Programming*, vol. 91, no. 3, pp. 563–588, 2002.

[12] M. R. Bussieck, M. C. Ferris, and A. Meeraus, "Grid-enabled optimization with GAMS," *INFORMS Journal on Computing*, vol. 21, no. 3, pp. 349–362, 2009.

[13] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. Plenum Press, 1972, pp. 85–103.

[14] M. Leitner, I. Ljubic, M. Luipersbeck, M. Prossegger, and M. Resch, "New Real-world Instances for the Steiner Tree Problem in Graphs," ISOR, Uni Wien, Tech. Rep., 2014.

[15] M. Fischetti, M. Leitner, I. Ljubić, M. Luipersbeck, M. Monaci, M. Resch, D. Salvagnin, and M. Sinnl, "Thinning out steiner trees: a node-based model for uniform edge costs," *Mathematical Programming Computation*, vol. 9, no. 2, pp. 203–229, 2017.

[16] T. Pajor, E. Uchoa, and R. F. Werneck, "A robust and scalable algorithm for the steiner problem in graphs," *Mathematical Programming Computation*, vol. 10, no. 1, pp. 69–118, 2018.

[17] G. Gamrath, T. Koch, S. Maher, D. Rehfeldt, and Y. Shinano, “SCIP-Jack—a solver for STP and variants with parallelization extensions,” *Mathematical Programming Computation*, vol. 9, no. 2, pp. 231 – 296, 2017.

[18] “PACE 2018: Parameterized algorithms and computational experiments challenge.” https://pacechallenge.wordpress.com/pace-2018/.

[19] A. Gleixner et. al., “The SCIP Optimization Suite 6.0,” Zuse Institute Berlin, ZIB-Report 18-26, 2018.

[20] L. Lovász and A. Schrijver, “Cones of matrices and set-functions and 0-1 optimization,” *SIAM Journal on Optimization*, vol. 1, no. 2, pp. 166–190, 1991.

[21] H. Wolkowicz and Q. Zhao, “Semidefinite programming relaxations for the graph partitioning problem,” *Discrete Applied Mathematics*, vol. 76-77, pp. 461–479, 1999.

[22] P. Hungerländer and F. Rendl, “Semidefinite relaxations of ordering problems,” *Mathematical Programming*, vol. 140, no. 1, pp. 77–97, 2013.

[23] Q. Zhao, S. E. Karisch, F. Rendl, and H. Wolkowicz, “Semidefinite programming relaxations for the quadratic assignment problem,” *Journal of Combinatorial Optimization*, vol. 2, no. 1, pp. 71–109, 1998.

[24] E. de Klerk, D. V. Pasechnik, and R. Sotirov, “On semidefinite programming relaxations of the traveling salesman problem,” *SIAM Journal on Optimization*, vol. 19, no. 4, pp. 1559–1573, 2008.

[25] C. Helmberg, F. Rendl, B. Mohar, and S. Poljak, “A spectral approach to bandwidth and separator problems in graphs,” *Linear and Multilinear Algebra*, vol. 39, no. 1–2, pp. 73–90, 1995.

[26] P. Hungerländer and F. Rendl, “A computational study and survey of methods for the single-row facility layout problem,” *Computational Optimization and Applications*, vol. 55, no. 1, pp. 1–20, 2013.

[27] F. Rendl, G. Rinaldi, and A. Wiegele, “Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations,” *Mathematical Programming*, vol. 121, no. 2, pp. 307–335, 2010.

[28] N. Krislock, J. Malick, and F. Roupin, "BiqCrunch: A semidefinite branch-and-bound method for solving binary quadratic problems," *ACM Transactions on Mathematical Software*, vol. 43, no. 4, pp. 32:1–32:23, 2017.

[29] M. Kočvara, "Truss topology design with integer variables made easy," Optimization Online, Tech. Rep., 2010.

[30] S. Mars, "Mixed-integer semidefinite programming with an application to truss topology design," Ph.D. dissertation, FAU Erlangen-Nürnberg, 2013.

[31] T. Wollenberg, "Two-stage stochastic semidefinite programming: Theory, algorithms, and application to AC power flow under uncertainty," Ph.D. dissertation, Universität Duisburg-Essen, 2016.

[32] B. Ghaddar and R. A. Jabr, "AC transmission network expansion planning: A semidefinite programming branch-and-cut approach," arXiv, Tech. Rep., 2017.

[33] R. Tamura, K. Kobayashi, Y. Takano, R. Miyashiro, K. Nakata, and T. Matsui, "Best subset selection for eliminating multicollinearity," *Journal of the Operations Research Society of Japan*, vol. 60, no. 3, pp. 321–336, 2017.

[34] R. Dai and M. Mesbahi, "Optimal topology design for dynamic networks," in *50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)*, 2011.

[35] M. Rafiee and A. M. Bayen, "Optimal network topology design in multi-agent systems for efficient average consensus," in *49th IEEE Conference on Decision and Control*, 2010.

[36] T. Gally, M. E. Pfetsch, and S. Ulbrich, "A framework for solving mixed-integer semidefinite programs," *Optimization Methods and Software*, vol. 33, no. 3, pp. 594–632, 2018.

[37] H. A. Friberg, "CBLIB 2014: a benchmark library for conic mixed-integer and continuous optimization," *Mathematical Programming Computation*, vol. 8, no. 2, pp. 191–214, 2016.

[38] T. Gally, "Computational mixed-integer semidefinite programming," Ph.D. dissertation, TU Darmstadt, 2019.

[39] S. Vigerske, "Decomposition of multistage stochastic programs and a constraint integer programming approach to mixed-integer nonlinear programming," Ph.D. dissertation, Humboldt-Universität zu Berlin, 2013.

[40] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler, "FiberSCIP – a shared memory parallelization of SCIP," *INFORMS Journal on Computing*, vol. 30, no. 1, pp. 11–30, 2018.

[41] T. Ralphs, Y. Shinano, T. Berthold, and T. Koch, "Parallel solvers for mixed integer linear programming," Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Tech. Rep. 16-74, 2016.

[42] L.-M. Munguía, G. Oxberry, D. Rajan, and Y. Shinano, "Parallel PIPS-SBB: multi-level parallelism for stochastic mixed-integer programs," *Computational Optimization and Applications*, 2019.

[43] R. L. Gottwald, S. J. Maher, and Y. Shinano, "Distributed domain propagation," in *16th International Symposium on Experimental Algorithms (SEA 2017)*, vol. 75, 2017, pp. 6:1 – 6:11.

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

[45] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch, and M. Winkler, "Solving hard MIPLIB2003 problems with ParaSCIP on supercomputers: An update," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, 2014, pp. 1552–1561.

[46] 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.

[47] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch, and M. Winkler, "Solving open MIP instances with ParaSCIP on supercomputers

using up to 80,000 cores," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2016, pp. 770–779.

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

[49] Y. Shinano, T. Berthold, and S. Heinz, *A First Implementation of ParaXpress: Combining Internal and External Parallelization to Solve MIPs on Supercomputers*. Cham: Springer International Publishing, 2016, pp. 308–316.

[50] I. Rosseti, M. de Aragão, C. Ribeiro, E. Uchoa, and R. Werneck, "New benchmark instances for the Steiner problem in graphs," in *Extended Abstracts of the 4th Metaheuristics International Conference (MIC'2001)*, Porto, 2001, pp. 557–561.

[51] C. Duin, "Steiner problems in graphs," Ph.D. dissertation, University of Amsterdam, 1993.

[52] D. Rehfeldt and T. Koch, "Combining NP-Hard Reduction Techniques and Strong Heuristics in an Exact Algorithm for the Maximum-Weight Connected Subgraph Problem," *SIAM Journal on Optimization*, vol. 29, no. 1, pp. 369–398, 2019.

[53] T. Polzin and S. V. Daneshmand, "A comparison of Steiner tree relaxations," *Discrete Applied Mathematics*, vol. 112, no. 1, pp. 241 – 261, 2001, combinatorial Optimization Symposium, Selected Papers.

[54] T. Polzin, "Algorithms for the Steiner problem in networks," Ph.D. dissertation, Saarland University, 2004.

[55] R. Wong, "A dual ascent approach for Steiner tree problems on a directed graph," *Mathematical Programming*, vol. 28, p. 271287, 1984.

[56] H. D. Sherali and B. M. Fraticelli, "Enhancing RLT relaxations via a new class of semidefinite cuts," *Journal of Global Optimization*, vol. 22, pp. 233–261, 2002.

[57] G. Gamrath, T. Koch, S. J. Maher, D. Rehfeldt, and Y. Shinano, "SCIP-Jack – A solver for STP and variants with parallelization extensions," 11th DIMACS Competition workshop paper, 2014.

[58] T. Koch, A. Martin, and S. Voß, "SteinLib: An updated library on Steiner tree problems in graphs," in *Steiner Trees in Industries*, D.-Z. Du and X. Cheng, Eds. Kluwer, 2001, pp. 285–325.

[59] Y. Shinano, D. Rehfeldt, and T. Koch, "Building optimal Steiner trees on supercomputers by using up to 43,000 cores," Zuse Institute Berlin, ZIB-Report 18-58, 2018.