Distributed Domain Propagation[*]

Robert Lion Gottwald[1], Stephen J. Maher[2], and Yuji Shinano[3]

1 **Zuse Institute Berlin, Berlin, Germany**
robert.gottwald@zib.de
2 **Zuse Institute Berlin, Berlin, Germany**
maher@zib.de
3 **Zuse Institute Berlin, Berlin, Germany**
shinano@zib.de

Abstract

Portfolio parallelization is an approach that runs several solver instances in parallel and terminates when one of them succeeds in solving the problem. Despite its simplicity, portfolio parallelization has been shown to perform well for modern mixed-integer programming (MIP) and boolean satisfiability problem (SAT) solvers. Domain propagation has also been shown to be a simple technique in modern MIP and SAT solvers that effectively finds additional domain reductions after the domain of a variable has been reduced. In this paper we introduce distributed domain propagation, a technique that shares bound tightenings across solvers to trigger further domain propagations. We investigate its impact in modern MIP solvers that employ portfolio parallelization. Computational experiments were conducted for two implementations of this parallelization approach. While both share global variable bounds and solutions, they communicate differently. In one implementation the communication is performed only at designated points in the solving process and in the other it is performed completely asynchronously. Computational experiments show a positive performance impact of communicating global variable bounds and provide valuable insights in communication strategies for parallel solvers.

1998 ACM Subject Classification G.1.6 Optimization, D.1.3 Concurrent Programming

Keywords and phrases mixed integer programming, parallelization, domain propagation, portfolio solvers

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.6

1 Introduction

A MIP is a problem with the general form:

$$\min\{c^\top x : Ax \leq b, l \leq x \leq u, x_j \in \mathbb{Z}, \text{ for all } j \in I\},$$

with matrix $A \in \mathbb{R}^{m\times n}$, vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $l, u \in (\mathbb{R} \cup \{-\infty, +\infty\})^n$, as well as a set $I \subseteq \{1, \dots, n\}$, which identifies the subset of variables that are integer. This paper deals with algorithmic approaches that aim to reduce the domain of a variable—methods to increase or decrease components of l or u respectively—within a parallel solver. An algorithmic approach of particular interest is domain propagation.

MIP and SAT solvers employ domain propagation after the domain of a variable has been reduced to find further reductions for variables occurring in the same constraints or clauses.

[*] 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). All responsibility for the content of this publication is assumed by the authors.

In modern branch-and-bound based MIP solvers domain propagation has a major positive impact on performance [2, 3, 4, 17]. It is usually performed at every node of the branch-and-bound tree to exploit the possible additional domain reductions that result from applying branching decisions. Regularly performing domain propagation is advantageous since it is able to achieve domain reductions and detect infeasible nodes with a small computational effort compared to solving the respective linear programming (LP) relaxation.

Beyond the traditional application, domain propagation has been incorporated into many different parts of a MIP solver. Gamrath [11] applied domain propagation during strong branching. This use of domain propagation has been shown to significantly improve the solver performance and reduce the branch-and-bound tree size. The average number of LP iterations for strong branching decreased and better dual bounds were obtained while no more time was spent in strong branching.

Modern solvers only propagate constraints if there is the potential of finding domain reductions; i.e. for general linear constraints at least one variable must have a tighter bound than in the last propagation. In branch-and-bound based solvers this generally occurs after each branching decision. However, there are other reasons why a the domain of a variable might be reduced. For instance, some MIP solvers employ a technique called *reduced cost strengthening* [19] that exploits dual information. Particularly, if a variable has non-zero reduced cost in the LP relaxation of a node, a bound can be inferred given the objective value $\hat{z}$ of a feasible primal solution. Because the variable has non-zero reduced cost, its LP solution value must be at the variable's bound. Furthermore, the reduced cost of this variable tells us how much the objective function changes if the variable moves away from its bound. Thereby a bound can be obtained for the variable, that must be satisfied by any solution with objective value $\hat{z}$ or better. If this technique is employed by using the LP relaxation at the root node, the obtained bound is globally valid.

In MIP solvers parallelization can be employed in a variety of ways [21]. A common approach is to parallelize the branch-and-bound algorithm by processing the subproblems concurrently. Another method is *portfolio parallelization*, sometimes also called *(parallel) racing*. In this form of parallelization multiple solvers with different configurations solve the same problem instance in parallel. The approach can be extended by the communication of global information such as feasible solutions and cutting planes. An efficient implementation of these approaches can be difficult due to the complexity that arises from the synchronization of global information.

Although portfolio parallelization does not distribute the work required to solve an instance, it has been shown to be competitive with the parallelization of the branch-and-bound tree search for a small number of processors [7, 14, 10, 5]. One reason for this is a phenomenon called *performance variability* [15]. It refers to the large differences in the performance of a solver that are observed after alterations expected to entail a neutral performance impact; e.g. setting a different random seed or permuting the problem instance.

2 Parallelization in SCIP

In SCIP [12], one of the fastest non-commercial MIP solvers, different forms of parallelization have been implemented. A deterministic shared memory portfolio parallelization of SCIP, referred to as concurrent SCIP, will be presented. Also, there exists a shared memory parallelization of SCIP called FiberSCIP [25], and a distributed memory parallelization called ParaSCIP [24]. The latter two only differ in the framework used for communication and both aim at parallelizing the tree search, but can also be configured to perform racing only. This paper compares both shared memory parallelizations, concurrent SCIP, and FiberSCIP.

The main difference between concurrent SCIP and FiberSCIP is the method of communication and the timing for sending and receiving information. In FiberSCIP all communications between solver threads are done via a controller thread, the LoadCoordinator, fully asynchronously. In concurrent SCIP the solvers instead gather the information and then communicate when they reach certain points in the solving process using a shared data structure.

2.1 Concurrent SCIP

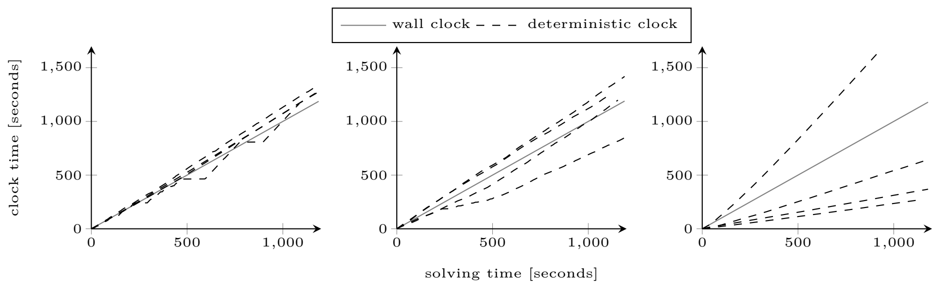
The development of concurrent SCIP was motivated by an effort to exploit performance variability and aid the fast discovery of feasible solutions. The experiments and parts of the description of concurrent SCIP in this section are also contained in the first author's master thesis [13], while the distributed domain propagation is a new feature presented in this paper.

Concurrent SCIP allows to run multiple solver instances in parallel on separate threads. For the purpose of diversification the solvers can be configured to use different settings and random seeds. Additionally, they can share feasible solutions and global variable bounds throughout the solving process. Of particular importance is the sharing of global variable bounds, which is the focus of this paper. The frequency of communication is adjusted dynamically based on the amount of the gap—difference between upper and lower bounds—that was closed between communication points.

Each type of solver used in concurrent SCIP is implemented as a new plugin type of SCIP. Therefore, in addition to SCIP solvers with different parameter settings, other algorithms and solvers could be included into a parallel portfolio. Concurrent SCIP can be compiled either with tinycthread [1], a thin wrapper around the platform specific threads, or with OpenMP [8]. In this paper the tinycthread version is used to compare with FiberSCIP, because they both use Pthreads on Linux.

An important requirement for concurrent SCIP is a deterministic solving process, so that the behavior of the solver can be reproduced. To satisfy this requirement care must be taken when implementing communication between the solvers. In a single-threaded program the sequence of instructions is the same between multiple runs; or at least it appears to be from the programmer's viewpoint even though modern microprocessors reorder instructions internally. In contrast, there are usually millions of different interleavings of instructions that can occur for a single multi-threaded program. One execution of such a program depends on the scheduling decisions of the operating system and other factors that are beyond the control of the programmer. Not only does this make it hard to develop correct multi-threaded applications, it also makes such applications non-deterministic by default. Therefore, in concurrent SCIP the solvers only share information at *communication points*, which are determined by using a deterministic clock [4]. However, a solver must wait if it wants to read information from a communication point that has not yet been reached by all solvers, otherwise it would incur non-determinism. For this reason a deterministic communication scheme can suffer from high idle times. To measure the impact of such behavior, concurrent SCIP can also be configured to use the wall clock instead of the deterministic clock for determining the communication points.

[1] `http://tinycthread.github.io/`



(a) instance biella1 **(b)** instance net12 **(c)** instance aflow40b

Figure 1 Deterministic clock with different settings.

An ideal deterministic clock closely resembles the CPU time of each thread. The deterministic clock in SCIP was chosen to be a linear combination of solving statistics that are already available in SCIP, such as the number of LP iterations. For the purpose of choosing good coefficients for this linear combination, data was collected from runs on several instances of MIPLIB 2010 [15] with different predefined emphasis settings, e.g. for finding feasible solutions or proving optimality. The data consists of observations of the solving time and the values of the statistics at that time. All the observations collected from the same run are dependent, because the statistics and the solving time are counted from the beginning of the solving process. Hence, the data was transformed by subtracting from each observation the values of the previous observation for each run. The resulting observations pertain to a roughly equal time interval in the solving process, and depend only on solving statistics counted within that interval. Still, predicting the solving time from the statistics across different instances requires to also take the size of a problem into account. Therefore, the observations were scaled by the number of non-zeros in the presolved problem.

To compute the coefficients for the scaled statistics we used a linear regression method called *Lasso* [26], which applies a parametrized ℓ_1-regularization. Penalizing the ℓ_1-norm of the solution vector causes the regression algorithm to bias towards a sparse solution with small coefficients, which aids the prediction accuracy. Other linear regression methods that use a different regularization where also considered, e.g. ℓ_2-regularization or a combination of those. However, the solutions obtained using *Lasso* gave the best predictions.

For the regression an implementation of scikit-learn [20] was used and the amount of regularization was chosen by cross-validation. To avoid overfitting to instance specific traits, the data of a single instance at a time was left out during cross-validation. The resulting linear combination has non-zero coefficients for the number of primal and dual LP iterations with warm-start information, the number of bound changes in probing mode, and the number of calls to an internal function that checks whether the solving has been stopped.

In Figure 1 the deterministic clock is visualized for different instances. The dashed lines show the progression of the deterministic clock with different parameter settings, the solid line shows the wall clock. For the deterministic clock, the emphasis settings for optimality, feasibility and easy instances, that are provided by SCIP, where used in addition to the default settings. The emphasis setting for optimality separates more aggressively, the setting for feasibility applies heuristics more aggressively, and the setting for easy instances avoids expensive techniques for presolving, separation and heuristics to focus on the tree search. The deterministic clock is usually close to the wall clock (Figure 1a and 1b), but on some instances it does not run at the same speed for different settings (Figure 1c). Such a behavior cannot be avoided in general as the deterministic clock is tuned for good results only on average

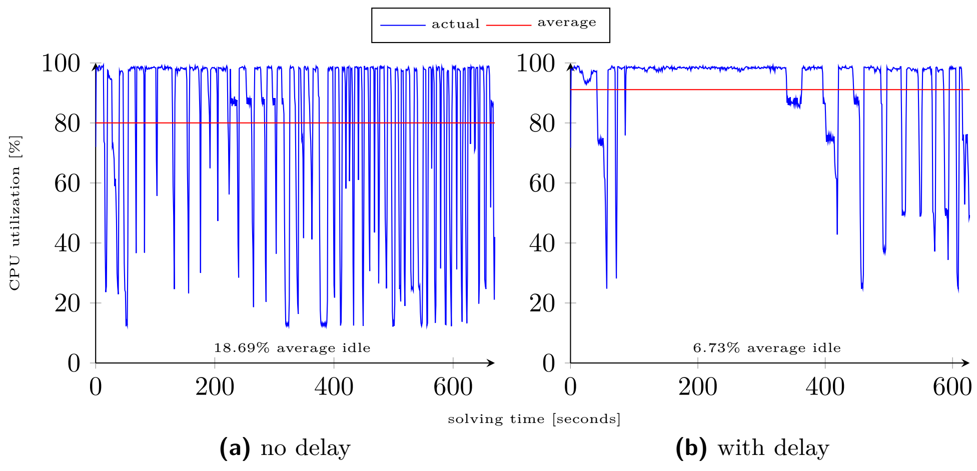


Figure 2 The CPU utilization of concurrent SCIP using 8 threads on the instance biella1, once without a delay and once using a delay.

and the same setting can produce very different results on different instances. Therefore, a setting on which the deterministic clock runs slower can be detrimental to the performance of concurrent SCIP, even though this setting would be beneficial if the wall clock was used instead.

If solvers are able to access shared information immediately, a *barrier* is required at each communication point, i.e. a point in the program at which a thread can only continue if all other threads have reached the barrier. Because such a barrier-based synchronization scheme would cause a large amount of idle time—especially when the deterministic clock deviates between threads—we introduced a delay before the solvers read data from a communication point. With a delay d, the solvers only read information from communication points that occurred at time $t - d$ or earlier, if their own deterministic clock is at time t. The solvers thereby receive information that is slightly outdated, still, the performance is better because they are waiting to a lesser extent. Even though the solvers still have to wait if the drift of the deterministic clock becomes too large, the CPU utilization improved significantly, as can be seen in Figure 2. Therein the CPU utilization was measured for concurrent SCIP running with different emphasis settings on eight threads. In Figure 2a no delay was used and in Figure 2b the delay was chosen as the deterministic equivalent of one second. Choosing a shorter delay did not make much of a difference here, because the delay effectively amounts at least to the time since the last communication point. With this delay the idle time reduced from 18.69% to 6.73%; put in other words, by using a delay concurrent SCIP was able to utilize roughly one CPU core more.

2.2 FiberSCIP

The Ubiquity Generator (UG) Framework[2] is a framework for the parallelization of branch-and-bound based solvers on distributed or shared memory computing environments. The aim of the UG framework is to parallelize branch-and-bound based solvers from the "outside". In this regard, the UG framework has been used to provide external parallelization for the base solvers SCIP, XPRESS [9] and PIPS-SBB [18]. To provide the capability to employ the UG

[2] `http://ug.zib.de/`

framework on shared and distributed memory environments, two different parallelization implementations are available. The distributed memory implementation of the UG framework uses the standardized Message Passing Interface (MPI). Alternatively, the shared memory implementation makes use of the Pthreads library.

The application of UG to parallelize SCIP has resulted in the solvers FiberSCIP (ug[SCIP, Pthreads]) [25] and ParaSCIP (ug[SCIP, MPI]) [23], for shared and distributed memory respectively. Since FiberSCIP was designed as a development environment for ParaSCIP, it serves as an ideal platform to evaluate the performance of distributed domain propagation and potentially leading to the adoption of this algorithmic feature into a large-scale parallel branch-and-bound solver.

There are three main phases of parallel branch-and-bound based solvers: ramp-up, primary, and ramp-down phases. For details regarding each of these phases, the reader is referred to Ralphs [22], Xu et al. [27], and Shinano et al. [25]. In the current work, the focus will be on the ramp-up phase, which is defined as the time period at the beginning of the solving process until sufficiently many branch-and-bound nodes are available to keep all processing units busy the first time. In the ramp-up phase FiberSCIP provides an implementation of racing ramp-up [25]. At the start of computation this form of ramp-up immediately sends a copy of the root branch-and-bound node to all available threads via the LoadCoordinator and commences parallel solving. To diversify the resulting branch-and-bound trees that are found across the set of all threads, different parameter settings are provided. This form of ramp-up is similar to a portfolio solving approach for MIP.

The different SCIP parameter settings used during racing ramp-up are compiled into FiberSCIP. They are a combination of the emphasis settings provided by SCIP labeled as *off*, *fast*, *default*, and *aggressive* for the different components in SCIP such as primal heuristics, presolvers, and separators. Exactly one solver uses the default settings of SCIP.

3 Distributed domain propagation

Our goal is to exploit variable bound information in a parallel portfolio solver to identify additional domain propagations. We let each solver in a parallel portfolio share new global variable bounds with the other solvers. A solver receiving these bounds propagates them against its local information and again shares the resulting domain reductions with the other solvers. We call this technique *distributed domain propagation* (DDP) and expect it to help solving problems within fewer branch-and-bound nodes, as a result of tighter variable bounds reducing the search space.

Portfolio parallelization involves having different settings in each solver, which results in different solution processes. Notably, each solver may generate conflicts [1] and cuts not generated in any other solver. Also the reduced costs in the LP relaxation of the root node may not be the same due to degeneracy. Since all of this information is used for domain propagation, a bound reduction that can be found in one solver may not be found in the other solvers. As such, DDP is able to perform additional domain reductions in each individual solver by sharing global variable information.

The DDP is implemented on top of the plugin structure of SCIP. It uses an event handler that reacts on global domain reductions for each variable and a propagator that applies the domain reductions received from other solvers. The implementations for concurrent SCIP and FiberSCIP differ in how they transfer the bound from the event handler in one solver to the propagator in another solver.

In concurrent SCIP the event handler stores the best bound for a variable whenever it reacts on a global bound change event. Once a communication point is reached, the bounds stored in the event handler are passed to a shared data structure where they are merged with bound changes from other solvers. If a solver reads this data structure, all bounds that are tighter than the current ones are passed to the propagator. The next time SCIP does domain propagation it will call the propagator, which will then apply the domain reductions.

In FiberSCIP a different implementation of DDP is provided. The major difference lies in the method of communication. When either a new incumbent solution is found or the domain of a variable has been reduced in a solver, this information is sent to the LoadCoordinator immediately. The LoadCoordinator stores the best incumbent solution and the tightest lower and upper bounds for each variable. When the LoadCoordinator receives an updated solution or bound, it is broadcasted to all solvers immediately. This results in asynchronous communication between all solvers. After receiving the bound, the procedure is the same as that of concurrent SCIP.

SCIP applies so-called dual reductions, which may cut off feasible solutions. Some of these reductions can cut off optimal solutions, but guarantee to keep at least one optimal solution. However, if such reductions from different solvers are applied together, it may happen that all optimal solutions are cut off. Accordingly, these dual reductions are disabled in all but one of the solvers. Thus it is ensured that the variable domains remain valid for all solvers. Note that reduced cost strengthening can be applied in all solvers, since it does not cut off optimal solutions.

Another difficulty for sharing variable bounds is the different formulations that arise from using different presolving techniques in each solver. When transferring a variable bound from one solver to another, it must first be transformed back into the original problem formulation and then re-transformed into the formulation of the solver that receives the bound.

4 Computational results

Computational experiments have been performed with SCIP 4.0.0 using SoPlex 3.0.0 [16] as an LP solver. The time limit was set to one hour on a cluster with 128GB memory and two Intel Xeon E5-2690 v4 2.60GHz processors per node. A subset of instances collected from the test sets of MIPLIB 3.0 [6], MIPLIB 2003 [3], and the benchmark set of MIPLIB 2010 [15] were used for the experiments. The subset was selected by excluding instances that default SCIP solved in less than a second or within the root node. Furthermore, the instances mspp16 and bley_xl1 were excluded due to memory issues and errors in one of the solvers, respectively. The resulting test set contains 125 instances.

The settings used for the different SCIP solvers in concurrent SCIP and in FiberSCIP were the same settings that FiberSCIP uses for racing ramp-up. The default behavior of presolving a problem instance before distributing it to the solvers was disabled. This makes the solving behavior of concurrent SCIP and FiberSCIP closer to that of default SCIP—aiding the comparability of their results.

Table 1 shows a comparison of the number of bounds that where tightened via DDP. For both implementations the number of such domain reductions were counted on all variables and also the subset applied to integer variables. The results are given for the winning solver and were aggregated by using a shifted geometric mean with a shift of 10. An interesting observation is that a larger number of threads leads to more domain reductions being found by DDP. This stems from the effect explained in the previous section, since more solvers with different configurations result in more diverse information being used for domain propagation.

[3] `http://miplib.zib.de/`

Table 1 Comparison of the number of domain reductions that were found via DDP in concurrent SCIP and FiberSCIP. The domain reductions on the subset of integer variables are given additionally in the second column.

Solver	Settings	#Dom. red.	#Int. dom. red.
Concurrent SCIP	4 threads	15.3	7.6
	8 threads	17.6	7.9
	12 threads	21.9	8.8
Concurrent SCIP (wall clock)	4 threads	16.0	8.7
	8 threads	18.8	9.8
	12 threads	27.9	14.3
FiberSCIP	4 threads	89.9	42.8
	8 threads	130.7	55.9
	12 threads	147.9	60.5

Additionally, the results show a huge difference in the number of domain reductions found by DDP between concurrent SCIP and FiberSCIP which is caused by the different communication schemes; in FiberSCIP a new bound reduction is communicated immediately and will therefore be received with a much smaller delay than in concurrent SCIP. Thus DDP could find a domain reduction that the solvers may have found by themselves shortly after. In concurrent SCIP that is more unlikely since it will communicate less frequently and the other solvers will read the shared domain reductions later, due to the delay used in this implementation. Also, concurrent SCIP will only communicate the best bound of a variable for which SCIP finds subsequent domain reductions between two communication points.

The performance of each portfolio solver with and without DDP is presented in Table 2. In preparing these results, only a subset of the original test set was used that contained 75 instances where at least one bound was tightened by DDP. The reduction of the test set is justified, because DDP does no additional computations and only communicates if domain reductions are found. Hence, it has no measurable impact if there are no domain reductions and including all the instances would merely introduce random noise to the comparison due to the non-deterministic behavior of FiberSCIP. In Table 2 the number of nodes were aggregated with a geometric mean shifted by 100 and the time was aggregated with a geometric mean shifted by 10.

In most settings a positive impact of DDP on the running time and the number of nodes is visible. However, on the 4 thread setting for FiberSCIP and concurrent SCIP using the deterministic clock DDP did not seem to help. Because DDP performed very well with the same settings in concurrent SCIP using the wall clock, we attribute the outlier to performance variability. Also it is expected that the performance variability is larger with a smaller number of threads.

Due to the overhead introduced by the deterministic synchronization, FiberSCIP is expected to outperform concurrent SCIP. Nevertheless, the large difference that also occurs when using the wall clock in concurrent SCIP indicates that the parameters which control the communication need to be adjusted. Notably, the delay and the synchronization frequency seem to be suboptimal. Also it can be observed on both implementations that DDP performs better with fewer than 12 threads. This is only partially caused by an increased communication effort when more solvers are used. More importantly, the architecture of SCIP is not yet exploiting shared memory parallelism and requires to duplicate an unnecessary large amount

Table 2 Comparison of default SCIP, concurrent SCIP using the deterministic clock or the wall clock, and FiberSCIP on the 75 instances that were solved with all settings and where DDP was able to find at least one domain reduction in any setting.

		with DDP		without DDP	
Solver	Settings	Time	Nodes	Time	Nodes
FiberSCIP	4 threads	119.9	5129.5	118.8	5217.5
	8 threads	112.9	4087.6	120.2	4406.2
	12 threads	121.5	4294.3	123.7	4286.3
Concurrent SCIP	4 threads	172.2	5354.9	172.9	5512.8
	8 threads	179.4	4971.4	182.1	4821.3
	12 threads	202.8	4976.6	205.8	4543.8
Concurrent SCIP (wall clock)	4 threads	136.2	5243.8	143.9	5631.0
	8 threads	140.7	4527.8	145.0	4660.2
	12 threads	152.6	4557.7	155.8	4799.4
SCIP	default			148.2	8556.1

of data and SCIP's performance is already memory bandwidth bound in many parts of the code. An increased number of threads slows down the computations in each thread even without any communication. The reason for this slow down are various effects caused by the increased load on the systems resources, e.g. more context switches, page faults and cache misses. For a detailed discussion we refer to Shinano et al. [25]. In due consideration of these effects we conclude that an increased performance for a larger number of threads is not an issue of the algorithmic approach of DDP, but of an efficient implementation that better exploits a shared memory architecture.

5 Concluding Remarks

This paper has introduced distributed domain propagation (DDP), a technique for finding global variable domain reductions in a parallel portfolio solver. Computational experiments were conducted to compare a deterministic synchronized implementation in concurrent SCIP and an asynchronous implementation in FiberSCIP on standard MIP instances.

In order to reproduce the results presented in this paper the readers can find the source code of SCIP and FiberSCIP in the release version 4.0.0 of the SCIP Optimization Suite (`http://scip.zib.de/#scipoptsuite`) and all instance data that was used can be retrieved from the MIPLIB homepage (`http://miplib.zib.de`).

The computational experiments show that DDP improves the overall performance of a portfolio solver significantly. The communication strategy of FiberSCIP gives better results in the experiments presented here. However, the communication settings and the parameter settings used in the individual solvers are not optimal for concurrent SCIP. Additionally, only concurrent SCIP provides deterministic communication. Both implementations suffer from a general slowdown due to a limited memory bandwidth if more than eight threads are used. For optimal performance one has to strike a balance between the two communication strategies to minimize the communication overhead while domain reductions are still applied within a short time frame. The algorithmic approach of DDP, however, has been shown to significantly improve the performance in a parallel portfolio of MIP solvers.

References

1 Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007. Mixed Integer ProgrammingIMA Special Workshop on Mixed-Integer Programming. `doi:10.1016/j.disopt.2006.10.006`.

2 Tobias Achterberg. Scip: Solving constraint integer programs. *Math. Prog. Comp.*, 1(1):1–41, 2009.

3 Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. Technical Report 16-44, ZIB, Takustr.7, 14195 Berlin, 2016.

4 Tobias Achterberg and Roland Wunderling. *Mixed Integer Programming: Analyzing 12 Years of Progress*, pages 449–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. `doi:10.1007/978-3-642-38189-8_18`.

5 Tomáš Balyo, Peter Sanders, and Carsten Sinz. *HordeSat: A Massively Parallel Portfolio SAT Solver*, pages 156–172. Springer International Publishing, Cham, 2015. `doi:10.1007/978-3-319-24318-4_12`.

6 R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 58:12–15, 1998.

7 R. Carvajal, S. Ahmed, G. Nemhauser, K. Furman, V. Goel, and Y. Shao. Using diversification, communication and parallelism to solve mixed-integer linear programs. *Oper. Res. Lett.*, 42(2):186–189, March 2014. `doi:10.1016/j.orl.2013.12.012`.

8 Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

9 FICO Xpress-Optimizer. `http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Optimizer.aspx`.

10 Matteo Fischetti, Andrea Lodi, Michele Monaci, Domenico Salvagnin, and Andrea Tramontani. Improving branch-and-cut performance by random sampling. *Mathematical Programming Computation*, 8(1):113–132, 2016. `doi:10.1007/s12532-015-0096-0`.

11 Gerald Gamrath. Improving strong branching by domain propagation. *EURO Journal on Computational Optimization*, 2(3):99–122, 2014. `doi:10.1007/s13675-014-0021-8`.

12 Gerald Gamrath, Tobias Fischer, Tristan Gally, Ambros M. Gleixner, Gregor Hendel, Thorsten Koch, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Sebastian Schenker, Robert Schwarz, Felipe Serrano, Yuji Shinano, Stefan Vigerske, Dieter Weninger, Michael Winkler, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 3.2. Technical Report 15-60, ZIB, Takustr.7, 14195 Berlin, 2016.

13 Robert Lion Gottwald. Experiments with a Parallel Portfolio of SCIP Solvers. Master's thesis, Freie Universität Berlin, 2016.

14 Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009. URL: `http://jsat.ewi.tudelft.nl/content/volume6/JSAT6_12_Hamadi.pdf`.

15 Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010. *Math. Prog. Comp.*, 3:103–163, 2011.

16 Stephen J. Maher, Tobias Fischer, Tristan Gally, Gerald Gamrath, Ambros Gleixner, Robert Lion Gottwald, Gregor Hendel, Thorsten Koch, Marco E. Lübbecke, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Sebastian Schenker, Robert Schwarz, Felipe Serrano, Yuji Shinano, Dieter Weninger, Jonas T. Witt, and Jakob Witzig. The scip optimization suite 4.0. Technical Report 17-12, ZIB, Takustr.7, 14195 Berlin, 2017.

17 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC'01, pages 530–535, New York, NY, USA, 2001. ACM. `doi:10.1145/378239.379017`.

18 Lluis-Miquel Munguia, Geoffrey Oxberry, and Deepak Rajan. Pips-sbb: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs. Technical report, Optimization Online, 2015.

19 George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, New York, NY, USA, 1988.

20 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

21 Ted Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear programming. Technical Report 16-74, ZIB, Takustr.7, 14195 Berlin, 2016.

22 T. K. Ralphs. Parallel branch and cut. In *Parallel Combinatorial Optimization*, pages 53–101. Wiley, 2006.

23 Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. ParaSCIP – a parallel extension of SCIP. In Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010*, pages 135–148. Springer, 2012. `doi:10.1007/978-3-642-24025-6_12`.

24 Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, Thorsten Koch, and Michael Winkler. 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. to appear.

25 Yuji Shinano, Stefan Heinz, Stefan Vigerske, and Michael Winkler. Fiberscip – a shared memory parallelization of scip. Technical Report 13-55, ZIB, Takustr.7, 14195 Berlin, 2013.

26 Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

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