Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

ROBERT GOTTWALD, STEPHEN J. MAHER, YUJI SHINANO

Distributed domain propagation

ZIB Report 16-71 (November 2016)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

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

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

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

Distributed Domain Propagation*

Robert Gottwald† · Stephen J. Maher · Yuji Shinano

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 it's 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 a variables domain has been reduced. This paper investigates the impact of distributed domain propagation 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.

1 Introduction

A mixed-integer program (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 $b \in \mathbb{R}^m$ and $c, l, u \in \mathbb{R}^n$, and a subset $I \subseteq \{1, \ldots, n\}$. This paper deals with algorithmic approaches that aim to reduce the size of a variables domain—methods to increase or decrease l or u respectively. An algorithmic approach of particular interest is domain propagation.

MIP and boolean satisfiability problem (SAT) solvers employ domain propagation after a variables domain has been reduced to find further reductions for variables occurring in the same constraints or clauses. In modern branch-and-bound based MIP solvers domain propagation has a major positive impact on performance [3]. 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 less 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 [9] 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.

*The work done for this article was supported by the BMBF Research Campus Modal SynLab.

†Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, {robert.gottwald, maher, shinano}@zib.de

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 variables domain might be reduced. For instance MIP solvers employ a technique called *reduced cost strengthening* that exploits dual information. Particularly, if a variable has non-zero reduced costs in a node's LP relaxation, a bound can be inferred given the objective value $\hat{z}$ of a feasible primal solution. Because the variable has non-zero reduced costs, the LP solution value must be at the variable's bound. Furthermore, the reduced costs of this variable tell us how much the objective function changes if the variable moves away from it's bound. Thereby a bound can be obtained for the variable, which must be satisfied by any solution with objective value $\hat{z}$ or better. If this technique is employed using the root node's LP relaxation, the obtained bound is globally valid.

In modern MIP solvers parallelization can be employed in a variety of ways. A common approach is to parallelize the branch-and-bound algorithm by processing the subproblems concurrently. Another method is *portfolio parallelization*. In this form of parallelization multiple solvers with different configurations solve the same problem instance in parallel. It can be extended by the communication of global information like feasible solutions, cutting planes or conflicts. of the approaches that parallelize a MIP solver are difficult to implement efficiently due to the complexity that arises from the synchronization of global information.

Although portfolio parallelization multiplies 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 smaller numbers of processors [6, 11]. One reason for this is a phenomenon called *performance variability* [12]. It refers to the large differences in a solver's performance that are observed after changes that are expected to have a neutral performance impact; e.g. setting a different random seed or permuting the problem instance.

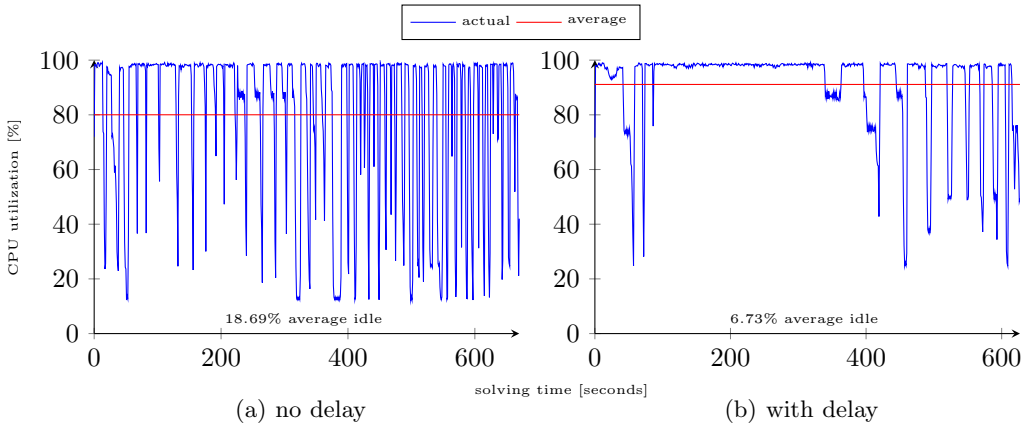
2 Parallelization in SCIP

In SCIP [10], 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. Besides, there exists a shared memory parallelization of SCIP called FiberSCIP [18], and a distributed memory parallelization called ParaSCIP [17]. 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 ramp-up only. This paper only compares the shared memory parallelizations of concurrent SCIP and FiberSCIP.

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. Concurrent SCIP allows to run multiple solver instances in parallel on separate threads. It is implemented as a new plugin type of SCIP, therefore not only SCIP solvers using any custom parameter settings but also other algorithms and solvers can be included into a parallel portfolio. The parallelization of concurrent SCIP uses either tinycthread [2], a thin wrapper around the platform specific threads, or OpenMP [7]. In this paper only the tinycthread version is used, because on Linux it will rely on Pthreads, which is also used by FiberSCIP.

In concurrent SCIP feasible solutions and global variable bounds are shared throughout the solving process. Of particular importance is the sharing of global variable bounds, which is the focus of this paper. Communication starts after the root node's LP relaxation has been



(a) no delay (b) with delay

Figure 1: The CPU utilization of concurrent SCIP using 8 threads on the instance biella1, once without a delay and once using a delay.

solved. The frequency of communication is then adjusted dynamically based on the amount of the gap—difference between upper and lower bounds—that was closed between communication points.

An important requirement for concurrent SCIP was a deterministic solving process. This requirement was to ensure the reproducibility of solver behavior. However, to satisfy this requirement care must be taken when implementing the methods of communication. The reproducible behavior of concurrent SCIP is achieved by ensuring that shared information only becomes available at deterministic points during the solving process for each participating solver. A deterministic clock is necessary to identify communication points and postpone all communication until each solver reaches these points. The deterministic clock in SCIP consists of a statistically tuned linear combination of solution statistics that are updated frequently throughout the solving process.

A deterministic communication scheme can suffer from high idle times since solvers might need to wait to read information shared by other solvers. Particularly a solver must wait if it wants to read information from a communication point that was not yet reached by all solvers, otherwise this would incur non-determinism. If solvers are able to access shared information immediately, a barrier is required at each communication point, which causes waiting. This is further amplified when using a deterministic clock, due to it's inability to perfectly resemble the wall clock. We addressed this issue by introducing a delay before the solvers read data from a communication point. Using 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. Even though the solvers thereby receive information that is slightly outdated, the performance is better because the solvers are waiting significantly less. This is reflected in the CPU utilization, as can be seen in Figure 1.

2.2 FiberSCIP

The Ubiquity Generator (UG) Framework 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 [8] and PIPS-SBB [14]. To provide the capability to employ the UG 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]) [18] and ParaSCIP (ug[SCIP, MPI]) [16], 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 [15], Xu et al. [19] and Shinano et al. [18]. In the current work, the focus will be 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 [18]. 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 and commences concurrent 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 main difference between concurrent SCIP and FiberSCIP is the method of communication and the timing of sending and receiving messages. FiberSCIP has a controller thread called the LoadCoodinator which sends the root node to all available solver threads and terminates all of them when one of the racing solver threads is terminated. All communications between solver threads are done via the LoadCoodinator fully asynchronously.

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 it's 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 solve 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 and cuts not generated in any other solver. Also the reduced costs in the root nodes's LP relaxation 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 the synchronization 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 is in the method of communication. When either a new incumbent solution is found or the domain of a variable is reduced in a solver, this information is sent to the LoadCoodinator immediately. The LoadCoodinator keeps the best incumbent solution and the tightest lower and upper bounds for each variable. When the LoadCoodinator receives an updated solution or bound, it is broadcast 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 are allowed to cut of 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. This ensures that the variable domains remain valid for all solvers.

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 using a release candidate of SCIP 3.3. The time limit was set to 1 hour on a cluster where each node has 128GB memory and two Intel Xeon E5-2690 v4 2.60GHz processors. A subset of instances collected from the test sets of MIPLIB 3.0 [5], MIPLIB 2003 [1], and the benchmark set of MIPLIB 2010 [13] have been 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 instance mspp16 was excluded because of memory issues and the instances bley_xl1, rocII-4-11, and vpphard have been excluded due to errors in one of the solvers. The resulting test set contains 123 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 comparison of 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 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.

Additionally, the results show a huge difference in the number of domain reductions found by DDP between concurrent SCIP and FiberSCIP. This can be explained by the different

	Concurrent SCIP		FiberSCIP	
Settings	All Bounds	Int. Bounds	All Bounds	Int. Bounds
4 threads	15.4	3.4	111.6	53.1
8 threads	22.4	4.2	157.6	68.2
12 threads	25.3	6.9	185.5	72.3

Table 1: Number of bounds on all variables and on integral variables that where tightened via DDP in FiberSCIP and concurrent SCIP

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 this 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 instances where at least one bound was tightened by DDP. Due to the non-deterministic behavior of FiberSCIP, all instance where no bound was tightened by DDP would introduce random noise to the comparison. In Table 2, the number of nodes were aggregated with a geometric mean shifted by 100 and the time as well as the primal integral [4] were aggregated with a geometric mean shifted by 10.

In FiberSCIP the number of nodes clearly decreased in all cases by using DDP. In concurrent SCIP it only decreased in the 4 thread setting whereas the number of nodes increased with 8 and 12 threads. A possible explanation is, that a node might already be created when DDP renders the node infeasible. Thus concurrent SCIP will count such a node even though it will be pruned during domain propagation and no LP will be solved. This also explains, why despite the increased number of nodes, the solving time and the primal integral of concurrent SCIP improved with DDP. In FiberSCIP this happens less frequently as the delay of DDP is smaller than in concurrent SCIP.

Because of the overhead introduced by the deterministic synchronization, FiberSCIP is expected to outperform concurrent SCIP. However, the large difference indicates that the parameters which control the communication have some tuning potential in concurrent SCIP. Also it can be observed on both implementations that DDP performs better with fewer threads. This is caused by an increased communication effort when more solvers are used.

Since some of the global domain reductions are based on a solution, DDP can help to direct the search into parts of the tree where different solutions are located. Therefore a positive effect on the primal integral can be observed.

5 Conclusions

This paper 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.

The computational experiments show that DDP improves the overall performance of a

		with DDP			without DDP		
Solver	Settings	Time	Nodes	Prim. Int.	Time	Nodes	Prim. Int.
SCIP	default				128.0	7574.6	2.580
Concur. SCIP	4 threads	130.1	4687.6	2.233	132.8	4985.1	2.342
	8 threads	127.4	3995.4	2.182	130.8	3720.3	2.313
	12 threads	130.3	4102.3	2.248	131.9	3825.2	2.257
FiberSCIP	4 threads	104.6	4346.9	1.946	114.4	4718.8	2.076
	8 threads	95.2	3288.9	1.766	101.9	3808.5	1.830
	12 threads	98.8	3327.7	1.776	97.6	3519.7	1.604

Table 2: Comparison of default SCIP, concurrent SCIP, and FiberSCIP on the 101 instances where DDP was able to find at least one domain reduction in any setting. The time and the number of nodes are with respect to the 68 instances solved by all settings. The primal integral is with respect to all the instances.

portfolio solver significantly. Notably, the wall clock time and the primal integral improved in both implementations. The communication strategies that were used in concurrent SCIP and FiberSCIP have different weaknesses. While concurrent SCIP looses some of the positive effects of DDP due to a high communication delay, FiberSCIP suffers from the high communication costs if more than 8 threads are used. Hence, for optimal performance of this feature one has to strike a balance between these two communication strategies so that the communication overhead is minimized while domain reductions are still applied within a short time frame.

Acknowledgements

The work for this article has been conducted within the *Research Campus MODAL* funded by the German Federal Ministry of Education and Research (find number 05M14ZAM).

References

[1] Mixed Integer Problem Library (MIPLIB) 2003, http://miplib.zib.de/

[2] TinyCThread, http://tinycthread.github.io/

[3] Achterberg, T.: Scip: Solving constraint integer programs. Math. Prog. Comp. 1(1), 1–41 (2009)

[4] Berthold, T.: Measuring the impact of primal heuristics. Operations Research Letters 41(6), 611 – 614 (2013)

[5] Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.P.: An updated mixed integer programming library: MIPLIB 3.0. Optima 58, 12–15 (1998)

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

[7] Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: portable shared memory parallel programming, vol. 10. MIT press (2008)

[8] FICO Xpress-Optimizer, http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Optimizer.aspx

[9] Gamrath, G.: Improving strong branching by domain propagation. EURO Journal on Computational Optimization 2(3), 99–122 (2014), http://dx.doi.org/10.1007/s13675-014-0021-8

[10] Gamrath, G., Fischer, T., Gally, T., Gleixner, A.M., Hendel, G., Koch, T., Maher, S.J., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schenker, S., Schwarz, R., Serrano, F., Shinano, Y., Vigerske, S., Weninger, D., Winkler, M., Witt, J.T., Witzig, J.: The SCIP Optimization Suite 3.2. Tech. Rep. 15-60, ZIB, Takustr.7, 14195 Berlin (2016)

[11] Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel SAT solver. JSAT 6(4), 245–262 (2009), http://jsat.ewi.tudelft.nl/content/volume6/JSAT6_12_Hamadi.pdf

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

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

[14] Munguia, L.M., Oxberry, G., Rajan, D.: Pips-sbb: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs. Tech. rep., Optimization Online (2015)

[15] Ralphs, T.K.: Parallel branch and cut. In: PARALLEL COMBINATORIAL OPTIMIZATION. pp. 53–101. Wiley (2006)

[16] Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T.: ParaSCIP – a parallel extension of SCIP. In: Bischof, C., Hegering, H.G., Nagel, W.E., Wittum, G. (eds.) Competence in High Performance Computing 2010. pp. 135–148. Springer (2012)

[17] Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T., Winkler, M.: Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores. In: Proc. of 30th IEEE International Parallel & Distributed Processing Symposium (2016), to appear

[18] Shinano, Y., Heinz, S., Vigerske, S., Winkler, M.: FiberSCIP – a shared memory parallelization of SCIP. Tech. Rep. ZR 13-55, Zuse Institute Berlin (2013)

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