P-MCOMSPS-STR: a Painless-based Portfolio of MapleCOMSPS with Clause Strengthening.

Vincent Vallade*, Ludovic Le Frioux†, Souheib Baarir*‡, Julien Sopena*§, Fabrice Kordon*
*Sorbonne Université, LIP6, CNRS, UMR 7606, Paris, France
†LRDE, EPITA, Le Kremlin-Bicêctre, France
‡Université Paris Nanterre, France
§INRIA, Delys Team, Paris, France

***Abstract*—This paper describes the solver P-MCOMSPS-STR submitted to the parallel track of the 2020's SAT Competition. It is a concurrent portfolio solver instantiated with the Painless (PArallel INstantiabLE Sat Solver) framework and using MapleCOMSPS as core sequential solver.**

I. Introduction

P-MCOMSPS-STR is a parallel SAT solvers built by instantiating components of the Painless parallel framework [1]. It is a portfolio-based [2] solver implementing a diversification strategy [3], fine control of learnt clause exchanges [4], using MapleCOMSPS [5] as a core sequential solver, and where learnt clause strengthening [6] has been integrated.

Section II gives an overview on Painless framework. Section III details the implementation of P-MCOMSPS-STR using Painless and MapleCOMSPS.

II. Description of Painless

Painless is a framework that aims at simplifying the implementation and evaluation of parallel SAT solvers for many-core environments. Thanks to its genericity and modularity, the components of Painless can be instantiated independently to produce new complete solvers.

The main idea of the framework is to separate the technical components (e.g., those dedicated to the management of concurrent programming aspects) from those implementing heuristics and optimizations embedded in a parallel SAT solver. Hence, the developer of a (new) parallel solver concentrates his efforts on the functional aspects, namely parallelization and sharing strategies, thus delegating implementation issues (e.g., data concurrent access protection mechanisms) to the framework.

Three main components arise when treating parallel SAT solvers: *sequential engines*, *parallelization*, and *sharing*. These form the global architecture of Painless.

A. Sequential Engines

The core element that we consider in our framework is a sequential SAT solver. This can be any CDCL state-of-the art solver. Technically, these engines are operated through a generic interface providing basics of sequential solvers: *solve, interrupt, add clauses*, etc.

Thus, to instantiate Painless with a particular solver, one needs to implement the interface according this engine.

B. Parallelization

To built a parallel solver using the aforementioned engines, one needs to define and implement a parallelization strategy. Portfolio and Divide-and-Conquer are the basic known ones. Also, they can be arbitrary composed to form new strategies.

In Painless, a strategy is represented by a tree-structure of arbitrary depth. The internal nodes of the tree represent parallelization strategies, and leaves are core engines. Technically, the internal nodes are implemented using WorkingStrategy component and the leaves are instances of SequentialWorker component.

Hence, to develop its own parallelization strategy, the user should create one or more strategies, and build the associated tree-structure.

C. Sharing

In parallel SAT solving, the exchange of learnt clauses warrants a particular focus. Indeed, beside the theoretical aspects, a bad implementation of a good sharing strategy may dramatically impact the solver's efficiency.

In Painless, solvers can export (import) clauses to (from) the others during the resolution process. Technically, this is done by using lock-free queues [7]. The sharing of these learnt clauses is dedicated to particular components called Sharers. Each Sharer is in charge of sets of producers and consumers and its behaviour reduces to a loop of sleeping and exchange phases.

Hence, the only part requiring a particular implementation is the exchange phase, that is user defined.

III. P-MCOMSPS-STR

This section describes the overall behaviour of our competing instantiation named P-MCOMSPS-STR. Its architecture is highlighted in Fig. 1. It implements the Painless strengthening described in [8]. In the following, we highlight the outline.

A. MapleCOMSPS

MapleCOMSPS [5] is based on MiniSat [9], and relies on the classical VSIDS [10], and the more recently defined LRB [11] for its decision heuristics. These two are used in one-shot phases: first LRB, then VSIDS. Moreover, it uses Gaussian Elimination (GE) at preprocessing time.

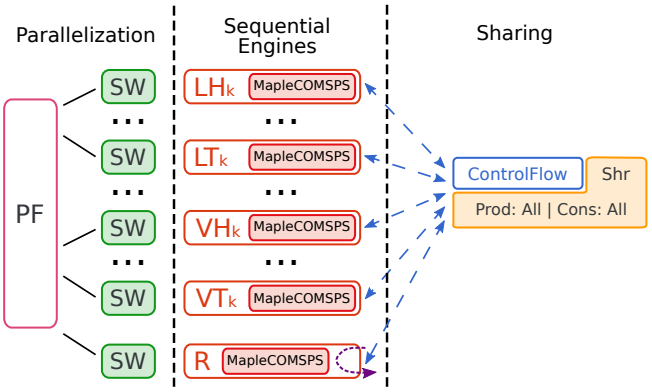

Fig. 1. Architecture of `P-MCOMSPS-STR`.

We adapt this solver for the parallel context as follows: (1) we parametrized the solver to select either LRB, or VSIDS for all solving process (noted respectively, `L` and `V`); (2) we added callbacks to export and import clauses; (3) we added an option to use or not the GE preprocessing; (4) we parametrized the solver to use as variable score comparator either $<$ or $<=$ (noted respectively head: `H` and tail: `T`).

B. Strengthener

A *reducer* engine (`R` in Fig. 1) implements the algorithm introduced in [6].

We implemented the strengthening operation as a decorator of *SolverInterface*. This decorator is a *SolverInterface* itself that uses, by delegation, another *SolverInterface* to apply the strengthening, in the present case a `MapleCOMSPS` solver.

C. Portfolio and Diversification

`P-MCOMSPS-STR` is a solver implementing a basic portfolio strategy (`PF`), where one solver is used as a `reducer`, and the other underlying core engines are either `LH`, `LT`, `VH` or `VT` instances (i.e., combination of VSIDS or LRB, and head or tail).

For each type of instances, we apply a sparse random diversification similar to the one introduced in [3]. That is for each group of k solvers, the initial phase of a solver is randomly set according the following settings: every variable gets a probability $1/2k$ to be set to false, $1/2k$ to true, and $1 - 1/k$ not to be set.

Moreover, only one of the solvers performs the GE preprocessing.

D. Controlling the Flow of Shared Clauses

In `P-MCOMSPS-STR`, the sharing strategy `ControlFlow` is inspired from the one used by [3], [4]. We instantiate one `Sharer` for which all solvers are producers. It gets clauses from this producer and exports some of them to all others (the consumers).

The exchange strategy is defined as follows: each solver exports clauses having a LBD value under a given threshold (2 at the beginning). Every 1.5 seconds, 1500 literals (the sum of the size of the shared clauses) are selected from each producers by the `Sharer` and dispatched to consumers. The LBD threshold of the concerned solver is increased (resp. decreased) if an insufficient (resp. a too big) number of literals are dispatched: respectively, less than 75% (1125 literals) and more than 98% (1470 literals).

E. Online Strengthening

The `reducer` engine is both a consumer and a producer of the sharer (`Shr`). It receives clauses from the different cores, strengthened them, in case of success it then exports them back. The sharing mechanism will then share this strengthened clauses to all the other solvers.

Since, a strengthened clause subsumes the original one, it is likely that cores will forget the original clause over time.

Acknowledgment

We would like to thank Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart, the authors of `MapleCOMSPS`.

References

[1] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "Painless: a framework for parallel sat solving," in *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 233–250, Springer, 2017.

[2] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2009.

[3] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio sat solver," in *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 156–172, Springer, 2015.

[4] Y. Hamadi, S. Jabbour, and J. Sais, "Control-based clause sharing in parallel sat solving," in *Autonomous Search*, pp. 245–267, Springer, 2011.

[5] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "Maplecomsps lrb vsids, and maplecomsps chb vsids," pp. 20–21, 2017.

[6] S. Wieringa and K. Heljanko, "Concurrent clause strengthening," in *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 116–132, Springer, 2013.

[7] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 267–275, ACM, 1996.

[8] V. Vallade, L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "On the usefulness of clause strengthening in parallel sat solving," in *Proceedings of the 12th NASA Formal Methods Symposium (NFM)*, Springer, 2020.

[9] N. Eén and N. Sörensson, "An extensible sat-solver," in *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 502–518, Springer, 2003.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th Design Automation Conference (DAC)*, pp. 530–535, ACM, 2001.

[11] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for sat solvers," in *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 123–140, Springer, 2016.