painless-mcomsps and painless-mcomsps-sym

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

***Abstract*—This paper describes the solvers `painless-mcomsps`, and `painless-mcomsps-sym` submitted to the parallel track of the SAT Competition in 2018. They are parallel solvers instantiated with PArallel INstantiabLE Sat Solver (`PaInleSS`) framework and using `MapleCOMSPS` as core sequential solver.**

I. Introduction

painless-mcomsps and painless-mcomsps-sym are parallel SAT solvers built by instantiating components of the PaInleSS parallel framework [1]. They are Portfolio based solvers implementing a diversification strategy, fine control of learnt clause exchanges, and using MapleCOMSPS [2] as a core sequential solver. Moreover, painless-mcomsps-sym included dynamic symmetry breaking [3] by using the Cosy library.

Section II gives an overview on PaInleSS framework. Section III details the implementation of painless-mcomsps using PaInleSS and MapleCOMSPS. Section IV explains how dynamic symmetry breaking has been incorporated in painless-mcomsps to give the solver painless-mcomsps-sym.

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 required 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 lockfree queues [4]. The sharing of these learnt clauses is dedicated to particular components called Sharers. Each Sharer 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. painless-mcomsps

This section describes the overall behaviour of our competing instantiation named painless-mcomsps. Its architecture is highlighted in Fig. 1.

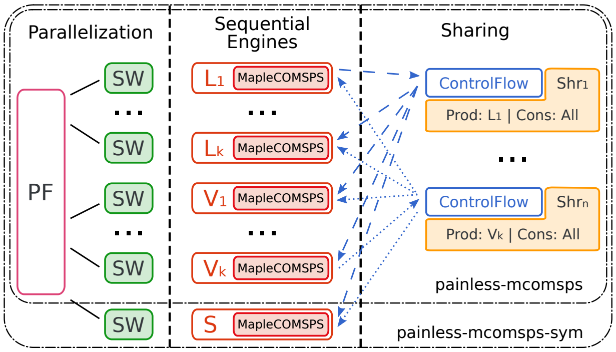

Fig. 1. Architecture of `painless-mcomsps`.

A. Sequential Engines: `MapleCOMSPS`

`MapleCOMSPS` is a sequential solver that finished second of the main track of the SAT Competition 2017. It is based on `MiniSat` [5], and uses as decision heuristics the classical Variable State Independent Decaying Sum (VSIDS) [6], and newly defined Learning Rate Branching (LRB) [7]. These heuristics are used in one-shot phases: first LRB, then VSIDS. Moreover, it uses Gaussian Elimination (GE) at preprocessing time.

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.

B. Parallelization: Portfolio and Diversification

`painless-mcomsps` is a solver implementing a basic Portfolio strategy (`PF`), where the underlying core engines are either `L` or `V` instances.

For each type of instances, we apply a sparse random diversification similar to the one introduced in [8]. 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.

C. Sharing: Controlling the Flow of Shared Clauses

In `painless-mcomsps`, the sharing strategy `ControlFlow` is inspired from the one used by [8]. We instantiate a `Sharer` per solver (the producer). 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 0.5 seconds, 1500 literals (the sum of the size of the shared clauses) are selected by the `Sharer` and dispatched to consumers. The LBD threshold of the concerned solver is increased if an insufficient number of literals (less than 1200) are dispatched.

IV. `PAINLESS-MCOMSPS-SYM`

This section describes the overall behaviour of our competing instantiation named `painless-mcomsps-sym`. Its architecture is highlighted in Fig. 1.

A. Dynamic Symmetry Breaking

The idea we bring is to break symmetries *on the fly*: when the current partial assignment can not be a prefix of a *lex-leader* (of an orbit), a constraint called *esbp* is generated. This constraint prunes this forbidden assignment and all its extensions.

B. Integration to `painless-mcomsps`

`Cosy`, a C++ library, provides dynamic symmetry breaking primitives. We integrated the library into `MapleCOMSPS`, and we added a parameter to activate or not dynamic symmetry breaking mode.

In `painless-mcomsps-sym`, there is only one solver that used dynamic symmetry breaking, we call it `S` in the Fig. 1. This solver uses the VSIDS heuristics.

The solver `S`, receives clauses from all the others, but it does not export clauses.

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 *International Conference on Theory and Applications of Satisfiability Testing*, pp. 233–250, Springer, 2017.

[2] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, “Maplecomsps lrb vsids, and maplecomsps chb vsids,” *SAT COMPETITION 2017*, pp. 20–21.

[3] M. Hakan, B. Souheib, C. Maximilien, and K. Fabrice, “Cdclsym: Introducing effective symmetry breaking in sat solving,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2018. To appear.

[4] M. M. Michael and M. L. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pp. 267–275, ACM, 1996.

[5] N. Eén and N. Sörensson, “An extensible sat-solver,” in *Theory and applications of satisfiability testing*, pp. 502–518, Springer, 2003.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient sat solver,” in *38th annual Design Automation Conference*, pp. 530–535, ACM, 2001.

[7] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, “Learning rate based branching heuristic for sat solvers,” in *Theory and Applications of Satisfiability Testing*, pp. 123–140, Springer, 2016.

[8] T. Balyo, P. Sanders, and C. Sinz, “Hordesat: A massively parallel portfolio sat solver,” in *int. conf. on Theory and Applications of Satisfiability Testing*, pp. 156–172, Springer, 2015.