painless-maplecomsps

Ludovic Le Frioux[*][†], Souheib Baarir[*][†][‡], Julien Sopena[†], Fabrice Kordon[†]
[*]LRDE, EPITA, Kremlin-Bicêctre, France
[†]Sorbonne Universités, UMPC Univ Paris 06, UMR 7606, LIP6, Paris, France
CNRS, UMR 7606, LIP6, Paris, France
[‡]Université Paris Nanterre, France

***Abstract*—This paper describes the `painless-maplecomsps` solver submitted to the parallel track of the SAT Competition in 2017. It is a parallel solver instantiated with PArallel INstantiabLE Sat Solver (`PaInleSS`) framework and using `MapleCOMSPS` as core sequential solver.**

I. Introduction

`painless-maplecomsps` is a parallel SAT solver built by instantiating components of the `PaInleSS` parallel framework. It is a Portfolio based solver implementing a diversification strategy, fine control of learnt clause exchanges, and using `MapleCOMSPS` [1] as a core sequential solver.

Section II gives an overview on `PaInleSS` framework. Section III details the implementation of `painless-maplecomsps` 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 are depicted in Fig. 1, and form the global architecture of `PaInleSS`.

A. Sequential Engines

The core element that we consider in our framework is a sequential SAT solver (called *sequential engine*). 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, bump activities, interrupt, add classes*, etc.

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

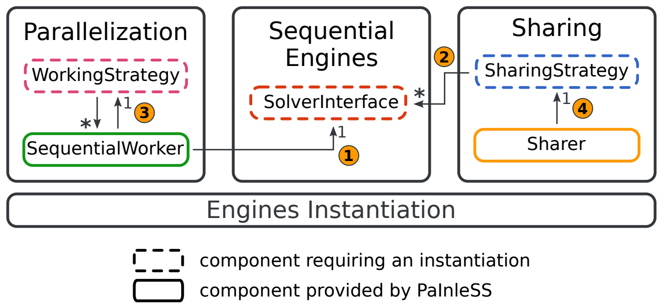

Fig. 1. Architecture of `PaInleSS`.

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 [2]. 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.

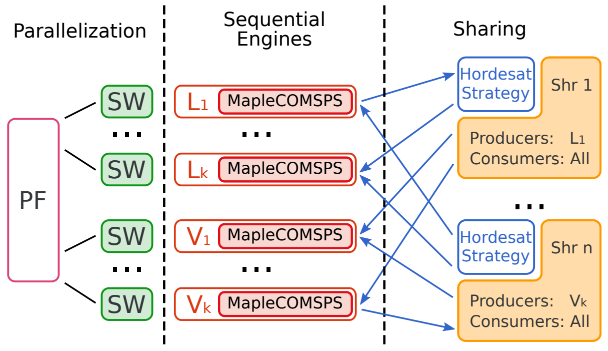

Fig. 2. Architecture of `painless-maplecomsps`.

III. PAINLESS-MAPLECOMSPS

This section describes the overall behaviour of our competing instantiation, namely `painless-maplecomsps`. Its architecture is highlighted in Fig. 2.

A. *Sequential Engines:* `MapleCOMSPS`

`MapleCOMSPS` is the winner sequential solver of the main track of the SAT Competition 2016. It is based on `MiniSat` [3], and uses as decision heuristics the classical Variable State Independent Decaying Sum (VSIDS) [4], and newly defined Learning Rate Branching (LRB) one [5]. These heuristics are used in one-shot phases: first LRB, then VSIDS.

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. The export is parametrized according to a Literal Block Distance (LBD) [6] threshold.

B. *Parallelization: Portfolio and Diversification*

`painless-maplecomsps` 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 [7]. 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.

C. *Sharing: Controlling the Flow of Shared Clauses*

In `painless-maplecomsps`, the sharing strategy is inspired from the one used by [7]. 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 1.5 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 (¡ 1200) are dispatched.

Acknowledgment

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

References

[1] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, “Maplecomsps, maplecomsps lrb, maplecomsps chb,” *SAT COMPETITION 2016*, p. 52.

[2] 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.

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

[4] 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.

[5] 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.

[6] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern sat solvers.,” in *IJCAI*, vol. 9, pp. 399–404, 2009.

[7] 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.