Mallob in the SAT Competition 2022

Dominik Schreiber
Institute of Theoretical Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
dominik.schreiber@kit.edu

***Abstract*—We describe our submissions to the parallel and cloud tracks of the SAT Competition 2022. Notable differences over last year's submission include a reworked clause sharing mechanism with a new approach to distributed clause filtering; further solvers and updated solver configurations; and an additional kind of memory awareness.**

***Index Terms*—Parallel SAT solving, distributed SAT solving**

I. Introduction

In this report we describe the configurations of our system **Mallob** which we submit to this year's International SAT Competition. Mallob (**Mal**leable **Lo**ad **B**alancer / **M**ultitasking **A**gi**l**e **Lo**gic **B**lackbox) is a decentralized job scheduling platform capable of prioritizing, balancing, and processing many SAT instances at once [1]. However, due to the rules of the competition, we configure our system to immediately schedule a single instance (i.e., the problem input) with full demand of resources and to quit after its processing.

II. System and Solver Setup

As in last years [2], [3], we subdivide each physical compute node into groups of four hardware threads each and run one MPI process on each such group. Each MPI process then deploys four core solvers. Contrary to last years where we run solvers as separate threads within each MPI process, this year each MPI process spawns a separate subprocess which runs four solver threads. This has two advantages: First, from a fault-tolerance perspective, individual solvers crashing (e.g., due to pathological inputs or internal errors) do not break the entire system but trigger a clean restart of the concerned subprocess. Secondly, we can deliberately restart individual solver processes for the purpose of memory awareness (see IV.). Despite these benefits, we acknowledge that this approach incurs some overhead for Inter-Process Communication, especially for transferring the formula and for periodic clause sharing.

In its current state, Mallob features four full-featured solver interfaces, namely for Lingeling [4], Glucose [5], CaDiCaL [6], and Kissat [6]. Most recently, we modified Kissat's codebase to support import and export of redundant clauses (only triggered at decision level 0 and every 500 conflicts) as well as setting initial phases for individual variables.

For the Parallel Track we submit a version with a portfolio purely consisting of Kissat configurations. We refer to this version as *Mallob-Ki*. In addition, we submit the most diverse portfolio Mallob can currently employ to the Cloud Track: We mix Kissat, CaDiCaL, Lingeling, and Glucose solvers roughly weighted according to their relative base performance and memory efficiency (eight parts Kissat, six parts CaDiCaL, four parts Lingeling, and two parts Glucose). We refer to this version as *Mallob-Kicaliglu*.

For both of these versions, we have identified strong solver configurations by running each SAT solver in various different configurations on the benchmarks of the International SAT Competition 2020.

III. Clause Exchange

We have reimplemented and overhauled large portions of Mallob's clause sharing strategy. Most significantly, we introduce a new approach to clause filtering, i.e., the problem of deciding for a shared clause c and a solver S whether S has received or produced c before and should therefore not receive c (again). The previous clause filtering mechanism of Mallob (inherited from HordeSat [7]) featured multiple large Bloom Filters at each solver process which occasionally result in erroneous rejection of unseen clauses. The probability for such *false positives* grows with the number of clauses registered in the filters, which may become noticeable in large distributed systems with millions of clauses being shared.

Our new clause filtering mechanism is exact and requires memory proportional to the set of "potentially good" clauses produced by a given solver process. We use two local datastructures: First, a hash table H of clauses maps each produced clause to a small bundle (32 bits) of meta data, including its LBD score, which local solver(s) produced it, and whether it was shared before. Secondly, a buffer structure B maintains a space-limited selection of the best clauses ready for export, discarding some of the worst clauses if better clauses arrive. Clause quality is determined by clause length and (secondarily) by LBD score. Our approach functions as follows:

- A clause c learnt by a solver which meets a basic quality criterium (length ≤ 20) is checked against H. If $c \notin H$ and if c fits into B, then c is inserted into B and H.
- At clause exchange time, each process flushes the highest priority clauses from B up to a certain total length.
- A buffer b of globally best clauses is aggregated and then shared among all processes as described in [1].
- Each process iterates over each clause $c_i \in b$ and checks whether $q_i := [c_i \in H \text{ and } c_i \text{ is marked as shared}] = 1$. A bit vector $\tilde{v}$ is constructed: $\tilde{v}[i] := q_i$ for each c_i. If $c \in H$ and c was not shared before, c is marked as shared.

- All created bit vectors $\tilde{v}$ are reduced to a single *filter vector* v via bitwise OR operations. v is aggregated and then shared among all processes just like b.
- Each process iterates over b and v simultaneously and only considers clauses c_i for which $v[i] = 0$. Each such clause c is forwarded to all local solvers which have not produced c yet according to $H[c]$.

The described approach ensures that a clause c shared in a given epoch $e \in \mathbb{N}$ will not be re-shared in a later epoch $e' > e$, since there is at least one process where c was produced and where, consequently, it was marked as shared in epoch e. At epoch e', this status is propagated to all processes via v, hence c is filtered. We can still allow for clauses to be reshared after a certain period of time elapsed: We can store in $H[c]$ the epoch e where c was last shared, and we re-admit c for sharing if e is sufficiently old. Likewise, we can allow re-sharing a clause if its LBD score improved since the last sharing. However, we did not find re-sharing upon improved LBD to be promising, and we set the minimum period until a clause is re-shared to a conservative 500 s.

We implemented B as an array of buckets, one bucket for each clause length $1 \leq l \leq 20$. Each bucket features a list of clauses which can be added to or removed from. A global *budget* integer represents the remaining number of literals which can still be inserted until B is full. If this budget is insufficient for inserting a given clause c of length l, an attempt is made to discard clauses from a bucket $l' > l$ in order to "steal" space for c. If this is unsuccessful, c is discarded. With this flexible buffering structure, we account for the observation that different solvers export differently sized clauses at different points in time during the solving procedure: For this reason, the available space is balanced dynamically among the different clause quality levels.

We employ the same data structure as B for the *import buffer* B_S of each solver S. This way, the buffering of incoming clauses is robust towards solvers which may not import clauses for a long period of time and therefore necessitate dropping some buffered clauses.

IV. Memory Awareness

Last year we introduced a rudimentary kind of memory awareness to Mallob: When starting to solve a formula, the number of contained literals is used to decide on how many threads to spawn in each MPI process. While this measure can be effective for some inputs, it does not address all issues. Memory usage which is initially acceptable but then grows to unsustainable levels is not accounted for. Furthermore, for extreme inputs even spawning a single solver thread for each MPI process may require too much memory.

This year we introduce an additional measure to counteract excessive memory usage. At program start, we create one communicator for the MPI processes at each physical machine. In other words, we identify groups of MPI processes with a shared RAM budget. Each group periodically checks the current memory usage of its machine and exchanges certain diagnostics for each MPI process. If a certain memory limit is exceeded ($> 90\%$ of RAM used), one or multiple MPI processes are chosen to trigger a memory panic. The heuristic which decides on the particular process(es) considers the memory used by each process as well as the importance of its role in the portfolio. A memory panic at a process which currently runs t solver threads triggers an immediate restart of the SAT solving process with $t-1$ solver threads. For extreme cases this can go as low as $t = 0$, i.e., no more solvers are executed on this MPI process. The decision heuristic ensures that at least one active solver thread remains on each machine.

Acknowledgment

The author expresses his heartfelt thanks to Laurent Simon and Armin Biere for allowing the use of Glucose, Kissat, CaDiCaL, and Lingeling in the competition. Furthermore, the author thanks Maximilian Schick for implementing initial CaDiCaL bindings for Mallob which we built upon. The author gratefully acknowledges the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de). Moreover, some preparation for this work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

References

[1] D. Schreiber and P. Sanders, "Scalable SAT solving in the cloud," in *International Conference on Theory and Applications of Satisfiability Testing*, 2021. In review.

[2] D. Schreiber, "Engineering HordeSat towards malleability: mallob-mono in the SAT 2020 cloud track," in *Proc. of SAT Competition*, pp. 45–46, 2020.

[3] D. Schreiber, "Mallob in the SAT competition 2021," *SAT COMPETITION 2021*, p. 38.

[4] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT competition 2018," *Proc. of SAT Competition*, pp. 13–14, 2018.

[5] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

[6] A. B. K. F. M. Fleury and M. Heisinger, "CaDiCaL, kissat, paracooba, plingeling and treengeling entering the sat competition 2020," *SAT COMPETITION*, vol. 2020, p. 50, 2020.

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