Mallob{32,64,1600} in the SAT Competition 2023

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

***Abstract*—We describe our submissions of *Mallob* to the parallel and cloud tracks of the SAT Competition 2023. Our changes mostly aim at reducing (computational and memory) overhead and at reducing turnaround times of shared clauses in order to reduce redundant work performed.**

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

I. Introduction

In this report we describe the configurations of our scheduling and SAT solving system **Mallob** [1], [2] which we submit to this year's International SAT Competition. As in prior years [3]–[5] 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. While this mode of operation now supports producing UNSAT proofs [6], we are not using this functionality since it adds overhead and limits the set of usable solvers.

II. Overview and Setup

In contrast to previous submissions [3]–[5] where we spawn one MPI process for each group of four hardware threads — a heritage from Mallob's precursor HordeSat [7] — we now spawn only one MPI process for each physical machine. This change reduces overhead in terms of run time and memory usage.[1] It also allows running Mallob as a shared-memory parallel solver without MPI on a single large machine. However, the increased degree of concurrency within each process also uncovered issues in some of our concurrent data structures that were previously "good enough" when only using four threads. For this reason we rewrote large portions of Mallob's data structures for handling produced clauses (see Section III-C). As last year [5], we run solver threads within a separate sub-process that is forked from the respective MPI process. Since restarting a solver process that orchestrates dozens of solvers leads to significant loss of progress, we also adjusted our *memory panic* mechanism [5] to gracefully terminate and clean up individual solvers in a solver process. For cases where an out-of-memory situation occurs despite our precautions, each subprocess now adjusts its out-of-memory score (`oom_score_adj`) in such a way that it is the first process to be killed by the operating system. Killing a SAT subprocess is always preferrable to killing an MPI process, since the latter crashes the distributed program.

[1]In particular, each MPI process keeps a copy of the problem input and additionally writes a copy to a shared-memory segment.

We submit two parallel versions and one distributed ("cloud") version of Mallob. We employ 32 (64) Kissat instances[2] in the parallel configuration **Mallob32** (**Mallob64**) and employ a mix of 800 Kissats, 533 CaDiCaLs, and 267 Lingelings in the distributed configuration **Mallob1600**. Compared to last year we omit Glucose, which might be slightly detrimental to overall performance but simplifies our setup and also renders all parts of our submission Free Software.

Inspired by recent work that involved modifications to CaDiCaL [6], we enhanced the clause export implementation of our Kissat backend: Instead of exporting clauses at conflict analysis, we now export any new redundant clause that is created (and any unit that is fixed). As such, solvers can now also share insights gained from inprocessing techniques such as probing, vivification, or hyper-ternary resolution. In addition, we now allow Kissat to import incoming clauses whenever at decision level zero without waiting for a certain number of conflicts to occur in between (500 conflicts in our previous submissions), which can reduce turnaround times of shared clauses (see Section III-B). We also added some minor improvements to Kissat's clause import code and extended its portfolio to a total of 15 distinct configurations.

III. Clause Sharing

Regarding Mallob's clause sharing, we introduce a change in handling LBD scores; an increase of the frequency at which all-to-all clause sharing is performed; and improvements to Mallob's clause filtering and buffering data structures.

A. Handling LBD Scores

We have integrated a technique that was already featured in TopoSAT 2 [8]: If a solver imports a clause, the clause's LBD value is reset to the clause's length, contrary to our (HordeSat's) earlier approach of importing each clause with its original LBD. The TopoSAT 2 approach takes into account that LBD is a local metric that depends on the solver state and therefore may not be meaningful for all solvers globally. Note that the HordeSat approach may force solvers to keep an unsustainable number of low-LBD clauses over time while the TopoSAT 2 approach rather results in solvers discarding many incoming clauses after a few conflicts.

[2]In last year's competition, a misconfiguration lead to our submission "Mallob-Ki" to use Lingeling instead of Kissat as a solver backend. See http://algo2.iti.kit.edu/schreiber/downloads/mallob-ki-mallob-li.pdf

B. Sharing Frequency

Since clause sharing may be considered a kind of distributed pruning of search space, we suspect that it is beneficial to minimize the latency between a clause's production by solver S and its import by a solver S'. Intuitively, lowering this "turnaround time" of a clause c may reduce the chance that S' enters a sub-space which S already reported as pruned via c. Therefore, more frequent clause sharing may decrease the amount of redundant work performed. We increased the frequency at which all-to-all clause sharing is performed from 1/s to 0.33/s. Accordingly, we scaled down the buffer limit for each sharing by a factor of three.

C. Clause Filtering and Buffering

We added some improvements and bugfixes to Mallob's exact distributed clause filtering mechanism [5]. For instance, we now use the clause metadata in hash table H to keep track of the last epoch where a local solver produced a clause and successfully wrote it into clause buffer B, and we added a periodic garbage collection which erases clauses from H whose last sharing and production both range back beyond the user-defined resharing interval. We also use this additional information to more reliably block clauses from being imported by a solver which recently produced them.

In our 2022 implementation of adaptive clause buffers B [5], each *slot* l for clauses of length l is guarded by a mutex. Inserting a clause c of length l in B requires locking slot l as well as potentially all slots $l' > l$ in succession in order to erase "worse" clauses and then use the freed budget to insert c to l. This is acceptable with just four solver threads but may not scale to our new setup. Rather than actually erasing clauses from a worse slot l', we now just *mark* a deletion by manipulating an atomic clause counter of slot l', hence we only need to lock slot l. The actual deletion takes place the next time a lock for slot l' is held. If B is close to full before flushing, then we also determine the minimum $\hat{l}$ such that $\geq 95\%$ of all clauses in B had length $\hat{l}$ or below. If a produced clause c is larger than $\hat{l}$, then it is highly unlikely that c is ever exported from B before it is deleted[3] and we discard c without attempting its insertion.

Solvers may occasionally produce large bursts (hundreds of thousands) of unit clauses, which overburdens B and results in discarding most produced clauses. For this reason, we now allow the buffers to store an unlimited number of unit clauses while keeping the shared budget for all other slots.

Lastly, we have noticed a shortcoming in the merge of clause buffers during our distributed aggregation [1]. If the set of available clauses exceeds the current aggregated buffer limit, then the buffer is truncated, returning excess clauses to the local solver process. Since clauses are sorted alphanumerically, this may introduce a slight bias to our sharing. We now randomly select the clauses from the buffer's "worst" bucket which make the cut and return the remaining clauses.

[3]The budget of B is set to $10\times$ the export limit per flush.

IV. Input Permutation

Permuting the input before handing it to a solver can be used as an additional source of diversification. We experimented with this kind of diversification in 2021 [4] but did not include it in 2022 since its implementation incurred too much overhead to be worthwhile. The formula is present as a chunk of shared memory that is parsed by many solvers concurrently, so direct manipulations of the formula should be avoided.

This year we reintroduce input permutation for all but the first ten solvers. In our new implementation, we select up to $k = 128$ input clauses to which we store a pointer. The first clause in the input is always selected while the remaining $k - 1$ clauses are selected at random. Each of the k pointers represents a chunk of the input beginning at the referenced clause. These k pointers are then permuted and the input chunks are read in the corresponding order. This procedure is cache-friendly and features a non-zero probability for any pair of clauses (c_1, c_2) to be read in reverse order. In addition, the order of literals in each clause is shuffled using a single clause buffer for each solver thread.

Acknowledgment

The author thanks Armin Biere for providing the solvers Kissat, CaDiCaL, and Lingeling which our solving system is 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). 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 *Proc. SAT*, pp. 518–534, Springer, 2021.

[2] P. Sanders and D. Schreiber, "Decentralized online scheduling of malleable NP-hard jobs," in *Proc. Euro-Par*, pp. 119–135, Springer, 2022.

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

[4] D. Schreiber, "Mallob in the SAT competition 2021," in *Proc. of SAT Competition 2021*, p. 38.

[5] D. Schreiber, "Mallob in the SAT competition 2022," in *Proc. of SAT Competition 2022*, pp. 46–47.

[6] D. Michaelson, D. Schreiber, M. J. Heule, B. Kiesl-Reiter, and M. W. Whalen, "Unsatisfiability proofs for distributed clause-sharing SAT solvers," in *Proc. TACAS*, pp. 348–366, Springer, 2023.

[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.

[8] T. Ehlers and D. Nowotka, "Tuning parallel SAT solvers," *Proceedings of Pragmatics of SAT*, vol. 59, pp. 127–143, 2019.