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 (Malleable Load Balancer / Multitasking Agile Logic Blackbox) 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-Kicaligu.

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 \textit{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 data-structures: 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 \( \leq 20 \)) 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 \( q_i := |c_i| \in H \) and \( c_i \) is marked as shared \( |1 = 1 \).
- 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.
for a long period of time and therefore necessitate ing clauses is robust towards solvers which may not import diagnostics for each MPI process. If a certain memory limit is reached, we might consider halting the solver. However, for extreme inputs even spawning a single solver thread for each process may require too much memory.

In other words, we identify groups of MPI processes with excessive memory usage. At program start, we create one MPI process may require too much memory. To address this issue, we introduce an additional measure to counteract unsustainable levels is not accounted for. Furthermore, 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