An Empirical Study on Learned Clause Overlaps In Distributed SAT Solving[⋆]

Jannick Borowitz[1,**], Dominik Schreiber[1] and Peter Sanders[1]

[1]*Karlsruhe Institute of Technology, Germany*

Abstract

Parallel and distributed clause-sharing solving is useful to conquer hard instances of the widely applicable Boolean satisfiability (SAT) problem. Based on the observation that similar learned clause sets across solver threads can indicate high amounts of redundant work, we conduct an empirical study on learned clause overlaps with the distributed solver system MallobSat. Our findings include that the ratio of clauses produced redundantly increases modestly with the number of solvers (5% to 34% going from one to 16 nodes) and is highest in the first seconds of solving. The gathered results provide deeper insights into the distributed solver's behavior and the effectiveness of certain techniques such as clause filtering. We also present evidence that simplification and inprocessing, which is performed redundantly by all solvers, can cause a substantial share of overlaps. We expect our results and the underlying methodology to be useful for parallel solver design in the future.

Keywords

SAT solving, distributed algorithms, HPC

1. Introduction

In recent years, parallel and distributed approaches to solve instances of the widely applicable Boolean satisfiability (SAT) problem [1] have gained traction [2] and increasingly lead to appealing results even on thousands of cores [3, 4]. Today's most successful parallelization paradigm for tackling diverse application instances is to run many solvers at once on the given formula (*portfolio*) and to let them exchange insights in the shape of *learned conflict clauses* (*clause sharing*). Nevertheless, today's clause-sharing solvers are still performing highly redundant work, as indicated by strongly sublinear mean and median speedups, e.g., a mean speedup of 7.2 at 24 cores and 44 at 3072 cores [5].

In this work, we want to shed light on the degree of redundant work performed across individual solver threads in modern parallel and distributed clause-sharing solving. We note that a clause produced by a solver can be considered a witness for some specific *work* this solver performed. In turn, many solvers producing the same clause indicates that the corresponding work was performed redundantly, which is undesirable for scalability. The *overlap* between sets of produced clauses may therefore indicate the amount of redundant work performed across the respective solvers. We present an empirical study where we investigate the nature of such produced clause duplicates and overlaps in the distributed clause-sharing solver MallobSat [4, 5], both on a global scale as well as in terms of pairwise solver overlaps, with up to 768 solver threads at once. Our gathered insights include that the ratio of redundantly produced clauses only increases modestly when increasing the scale of solving; even at 768 cores, around two thirds of produced clauses are unique. Most duplicates are produced in the first few seconds of solving, and most redundant productions of the same clause co-occur within a few seconds, which

[⋆]The authors gratefully acknowledge 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 of 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 program (grant agreement No. 882500).

[**]Corresponding author.

uzoes@student.kit.edu (J. Borowitz); dominik.schreiber@kit.edu (D. Schreiber); sanders@kit.edu (P. Sanders)

0000-0002-8419-6324 (J. Borowitz); 0000-0002-4185-1851 (D. Schreiber); 0000-0003-3330-9349 (P. Sanders)

can explain the benefit of short-horizon clause filtering [5]. We are also able to trace back high overlaps, especially for unit and ternary clauses, to particular solver backends and/or configurations and propose some first mitigations. Our results support that the current way of performing pre–/inprocessing tasks fully redundantly is highly suboptimal. We expect our findings and the underlying methodology to prove useful for designing and improving parallel clause-sharing solvers.

2. Preliminaries

In SAT solving, we are concerned with finding a variable assignment that satisfies a given Boolean expression, usually given in *conjunctive normal form* (CNF), i.e., as a set of *clauses*. Most modern SAT solvers perform *conflict-driven clause learning* (CDCL): They search the space of partial assignments while deriving redundant *conflict clauses*, which mark unsatisfiable sub-spaces, and maintain a database of the most helpful such clauses [6].

Roughly speaking, there are two approaches to parallelizing SAT solving. The first one is to partition the search space of assignments and search the partitions in parallel [7, 8]. The second strategy follows a *portfolio-based approach*: Different sequential solvers all aim to solve the original formula in parallel and can exchange information, in particular learned clauses. All winning solvers in the recent parallel and cloud tracks of the International SAT Competition follow this second approach [9, 10, 11]. Therefore, we do not cover the first approach.

Today's portfolio-based SAT solvers rely on *clause sharing* and *diversification*. We distinguish *native diversification*, which encompasses running different configurations of a particular sequential solver [12]; and *generic diversification*. The latter encompasses variations via solver-independent interfaces, such as setting random seeds [12] or choosing random *phases* [3] which indicate the polarity to first assign to a variable. In terms of clause sharing, different solver systems use different approaches and metrics on which clauses to share and when [12, 13, 14, 15, 16, 17]. Second, parallel solvers differ regarding the sharing volume and how processes communicate, ranging from sparse *communication graphs* [18] over rigid all-to-all exchanges [3] to MallobSat's hierarchical merging of the globally best (i.e., shortest) available distinct clauses [4, 5]. Some approaches employ *clause filtering*, where recently shared clauses are detected and prevented from being shared again [3, 5, 19]. We refer to the 2$^{\text{nd}}$ author's dissertation [20] for a broader overview of parallel SAT solving.

The authors of MallobSat as well as Vallade et al. have recently analyzed the effectiveness of different clause filtering approaches [5, 21], also including analyses of the amount of observed clause duplicates. Our study, by contrast, does not put a particular focus on shared clauses but rather aims to shed light on the overlaps between *all* produced clauses (or a representative sample thereof) across hundreds of solvers in order to gain deeper insights.

3. Methodology

We now describe our methodology for the presented study.

When a solver thread in MallobSat produces a clause, it executes a callback which decides whether the clause is written to an export buffer of the respective process. Moreover, a filtering mechanism blocks clauses which have already (recently) been produced. We modified this callback to log clauses of sufficiently small size and *literal block distance* (LBD, a metric generalizing clause length [13]), independently of whether they are considered for sharing later. We neglect very large clauses (length > 60) not only because they are irrelevant for sharing but also because a statistical argument can be made that finding the same clause redundantly becomes less likely for increasing clause lengths. Our results confirm this conjecture.

Since distributed solvers can produce millions of clauses per second ($\approx 10^4$ clauses per second per solver), we found exact logging of *all* produced clauses with their respective metadata to be infeasible with our setup. As such, we follow an approximate logging scheme: Each produced clause c is sorted, hashed with a high-quality 64-bit hash function h, and is then logged only if the hash value $h(c)$ is divisible by 16. Our logging scheme then writes a report $r = (\frac{h(c)}{16}, t_c, p_c, s_c, d_c, g_c)$, where t_c is the timestamp of logging, p_c is the index of the producing process, s_c is the producer's local solver ID on the process, d_c is the clause size, and g_c is the clause's LBD. Notably, we do not store a clause's literals but identify clauses solely based on the hashes, which may result in overestimated overlaps in case of hash collisions. Given uniformly distributed 60-bit hashes, at most one collision is to be expected even at a billion hashed objects (which our runs never reached). As such, we can safely neglect collisions.

A completed run of the distributed solver yields a set $\mathcal{R}$ of clause productions, each formed as r above. We can extract and count the *unique clause hashes* of $\mathcal{R}$, $\mathcal{H}(\mathcal{R})$, and define the *duplicate clause production ratio* of $\mathcal{R}$ as

$$\mathit{DCPR}(\mathcal{R}) = \begin{cases} \frac{|\mathcal{R}| - |\mathcal{H}(\mathcal{R})|}{|\mathcal{R}|}, & \text{if } |\mathcal{R}| \geq 1 \\ 0, & \text{otherwise.} \end{cases}$$

Note that this metric also accounts for clauses produced more than twice. Additionally, we can restrict $\mathcal{R}$ to the clause productions of a solver $x = (p, s)$, denoted $\mathcal{R}_x \subseteq \mathcal{R}$. This allows us to examine the *pairwise produced clause overlap* of solvers x and y,

$$\mathit{PPCO}(\mathcal{R}_x, \mathcal{R}_y) = \begin{cases} J(\mathcal{R}_x, \mathcal{R}_y) = \frac{|\mathcal{H}(\mathcal{R}_x) \cap \mathcal{H}(\mathcal{R}_y)|}{|\mathcal{H}(\mathcal{R}_x) \cup \mathcal{H}(\mathcal{R}_y)|}, & \text{if } |\mathcal{H}(\mathcal{R}_x)| + |\mathcal{H}(\mathcal{R}_y)| \geq 1 \\ 0, & \text{otherwise,} \end{cases}$$

where J is the well-known *Jaccard index* for measuring similarities between two sets.

4. Implementation and Tooling

In order to enable our experimental study and facilitate similar analyses in the future, we have developed a novel tool, called ClauseLab, for analysing produced clauses for arbitrary clause-sharing SAT solvers. It covers most of the analyses used for this study. Moreover, ClauseLab features a plugin architecture in order to easily extend the analyses performed.

The integration of this tool is simple: For each solver thread, a logger instance must be created to write a solver-specific log file to a process-specific directory. Each solver thread's clause export interface must then be extended by a call to the respective logger instance with the produced clause and optional metadata. We provide a plug-and-play C++ header that offers such a logger interface, implementing the approximate logging scheme outlined in Section 3.

After the experiments are finished, ClauseLab's analyses can be run on the gathered raw data in order to obtain insights into the produced clauses regarding duplicates and overlaps. To this end, ClauseLab outputs numerous statistics, such as the *DCPR* and *PPCO* metrics given above, and produces corresponding plots. The full list of statistics and plots can be found in the repository.

As a first example for a solver integration, we connected ClauseLab to MallobSat. The source code of the modified MallobSat version and of ClauseLab itself can be found on GitHub.[1]

5. Results

We now proceed with presenting our obtained results.

5.1. Experimental Setup

We conducted our experiments on two different clusters. First, we used up to 16 nodes of **SuperMUC-NG** interconnected by Intel OmniPath, where each node features two Intel Skylake Xeon Platinum 8174 sockets, with 24 cores each, and 96 GB of RAM. Secondly, we used 10 nodes of **HoreKa** interconnected by InfiniBand, where each node features 256 GB of RAM and two Intel Xeon Platinum 8368 sockets with 38 cores each.

[1] ClauseLab: https://github.com/jabo17/clause-lab; MallobSat with clause logging: https://github.com/jabo17/mallob/tree/clause-overlap-baseline; MallobSat-base+: https://github.com/jabo17/mallob/tree/clause-overlap-baseline-impr.

Table 1
Native diversification options of the SAT solver backends in MallobSat-base. "+" and "-" indicate setting an option on and off, respectively, "±" indicates toggling the option each time the configuration is used.

config	Kissat 2020	Kissat 2023	CaDiCaL	Lingeling
0	-elimination	-elimination	-phase	gluescale=5
1	delay=10	restartint=10	sat config	+plain, +decompose
2	restartint=100	walkinitially=1	-elim	locs=-1, locsrtc=1, locswait=0, locslim=2^{24}, ±plain
3	+walkinitially	restartint=100	unsat config	restartint=100
4	restartint=1000	-sweep	+condition	+sweeprtc
5	-sweep	unsat config	-walk	restartint=10^3
6	unsat config	sat config	restartint=100	scincinc=50
7	sat config	-probe	+cover	restartint=4
8	-probe	minimizedepth=10^4	+shuffle, +shufflerandom	phase=pos
9	failedcont=50, failedrounds=10	reducefactrion=90	-inprocessing	phase=neg
10	minimizedepth=10^4	vivifyeffort=1000		-block, -cce
11	modeconflicts=10^5, modeconflicts=10^9			
12	reducefraction=90			
13	vivifyeffort=1000			
14	xorclslim=8			

Table 2
Median duplicate clause production metrics for MallobSat-base on 1–16 nodes.

nodes	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{R}\rvert - \lvert\mathcal{H}(\mathcal{R})\rvert$	$DCPR(\mathcal{R})$
1	201 764	13 607	0.05
4	607 782	92 898	0.19
16	1 559 177	326 275	0.34

We use the benchmark instances of the International SAT Competition 2022. For faster turnaround times and less resource spending, we limit some runs to 349 out of 400 instances which have been solved by at least one (sequential, parallel, or cloud) solver in SAT Competition 2022, as in prior work [5]. We allow up to 300 s of wallclock time per instance.

5.2. Global Clause Duplicates

We begin with the state-of-the-art configuration of MallobSat which we denote MallobSat-base in the following. It features Kissat [22], CaDiCaL [23], and Lingeling [24] threads initialized alternatingly in this order. For each backend, native diversification is achieved by cycling through a set of specific configurations—specifics are provided in Tab. 1. In addition, MallobSat-base applies generic diversification via random variable phases, random seeds, and disabling clause import for some threads.

First of all, we investigate the scaling of the clause overlap when increasing the number of solvers. Therefore, we compare MallobSat-base on one (48 solvers), four (192 solvers), and 16 nodes (768 solvers). Tab. 2 and Fig. 1 show results. The number of reports grows by a factor of 2.5–3 when quadrupling the number of solvers—lower running times partially compensate for the increase in clause producers. The median number of duplicates grows superlinearly in the number of reports ($6.8\times$ from 1 to 4 nodes, $3.5\times$ from 4 to 16 nodes) and yet by smaller margins than we anticipated. Likewise, the median *DCPR* increases moderately, not even doubling when going from 4 to 16 nodes. As such, at 768 solvers (16 nodes), still **around two thirds of all produced clauses are unique**. That being said, the *DCPR* varies significantly (Fig. 1) and heavily depends on the particular input. Interestingly, we noticed

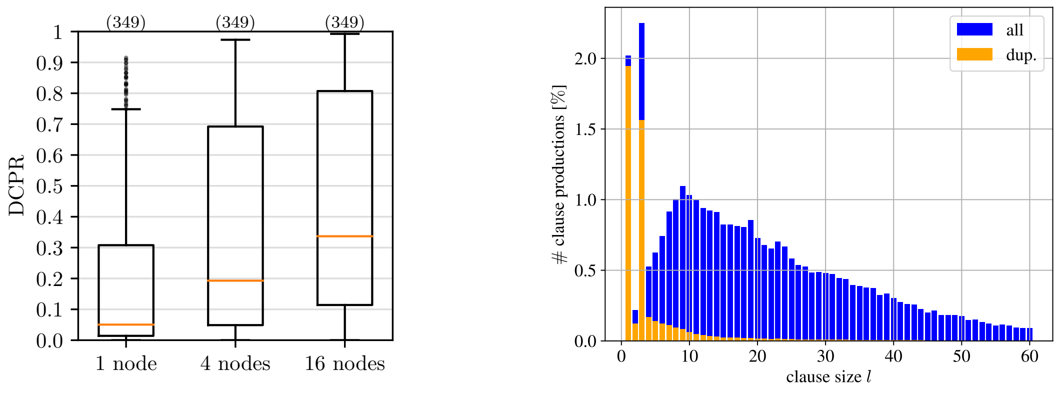

Figure 1: MallobSat-base results. Left: DCPR distribution at 1, 4, and 16 nodes. Right: Geometric mean ratios of *all* productions by clause length with according *duplicates*, for MallobSat-base (left) and the improved configuration discussed later (right).

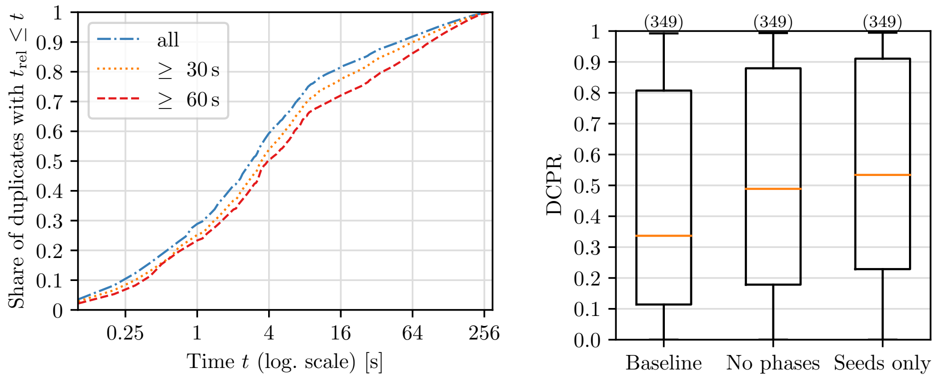

Figure 2: MallobSat-base results. Left: Cumulative distribution of the time period between the production of a duplicate clause and that clause's *first* production, overall and for instances which took at least {30,60} s to solve. Right: *DCPR* at 16 nodes, without phase diversification, and without all "generic" diversification save seeds.

some matches with recent scaling results of MallobSat [5]: Certain model checking tasks [25], where MallobSat hardly scales, result in an exceptionally high *DCPR* of 0.98, whereas some combinatorial problems allowing for good scaling, such as relativized pigeon holes [26], result in a $DCPR \leq 0.05$. Details are provided in Tab. 3.

Next, we investigate the produced clauses at 16 nodes more closely. Fig. 1 (right) shows the geometric mean of the ratios of (duplicate) clause productions restricted to a given clause size. Most notably, **unary and ternary clauses are by far the most frequent, with most of them being duplicates**. Although only 2% (2.2%) of the produced clauses are duplicate unary (ternary) clauses, these are approximately 90% (70%) of all produced unary (ternary) clauses. A possible cause can be simplification and inprocessing techniques, which we later confirm to be the case for the ternary clauses. We found most units to originate from Lingeling and most ternary clauses to originate from Kissat. Productions of clauses larger than 10 become decreasingly likely and their duplicate production ratios fall below 5%. We observed the distribution over the clauses' LBD scores to be very similar.

We now turn to the distribution of duplicates along the *temporal* dimension. For 16 nodes on all 400 problem instances, we observed a median duplicate clause production ratio of 28% after 3 s of running time, which amounts to $\approx$ 80% of the full run's duplicates. In other words, **the vast majority of duplicates are produced in the first few seconds of solving** which, again, hints at high amounts of redundant work at the beginning, possibly because all threads simplify the formula redundantly. In addition, Fig. 2 (left) shows the cumulative distribution over duplicates produced in a certain time frame from a clause's first production. **80% (60%) of duplicates are produced within 16 s (4 s) after**

Table 3
Duplicate clause production ratios of MallobSat-base by instance family and result, considering all known families with ≥ 2 solved instances, in terms of minimum, median, geometric mean, and maximum *DCPR* computed for each instance. Rows are sorted by geometric mean.

		DCPR			
Family, result	#	min	median	mean ▲	max
`pigeon-hole-unsat`	2	0.004	0.082	0.019	0.082
`minimum-disagreement-parity-unsat`	3	0.018	0.046	0.034	0.047
`grid-coloring-sat`	10	0.007	0.043	0.042	0.311
`algorithm-equivalence-checking-unsat`	13	0.018	0.049	0.050	0.161
`graph-isomorphism-unsat`	8	0.024	0.063	0.052	0.091
`scheduling-unsat`	14	0.006	0.070	0.054	0.280
`algebra-sat`	4	0.031	0.063	0.061	0.223
`minimum-disagreement-parity-sat`	7	0.018	0.071	0.066	0.273
`maxsat-optimal-unsat`	4	0.062	0.072	0.070	0.084
`hypertree-decomposition-sat`	2	0.065	0.161	0.102	0.161
`algebra-unsat`	4	0.064	0.109	0.103	0.207
`circuit-equivalence-checking-unsat`	10	0.056	0.130	0.131	0.415
`hardware-verification-unsat`	3	0.030	0.173	0.143	0.555
`maxsat-optimum-sat`	3	0.108	0.123	0.145	0.232
`scheduling-sat`	23	0.010	0.160	0.147	0.978
`gray_codes-unsat`	2	0.126	0.208	0.162	0.208
`circuit-equialence-checking-unsat`	4	0.059	0.404	0.167	0.461
`independent-set-reconfiguration-unsat`	7	0.034	0.231	0.194	0.762
`set-covering-unsat`	13	0.119	0.219	0.203	0.266
`maxsat-optimal-sat`	7	0.141	0.215	0.219	0.345
`prime-factoring-sat`	2	0.193	0.252	0.220	0.252
`cryptography-unsat`	2	0.096	0.518	0.223	0.518
`graph-isomorphism-sat`	5	0.149	0.256	0.235	0.384
`miter-unsat`	4	0.090	0.673	0.329	0.712
`petrinet-concurrency-sat`	3	0.279	0.377	0.356	0.430
`independent-set-reconfiguration-sat`	6	0.096	0.478	0.359	0.600
`cryptography-sat`	14	0.137	0.450	0.371	0.712
`software-verification-sat`	2	0.263	0.557	0.383	0.557
`hardware-model-checking-unsat`	15	0.140	0.516	0.433	0.893
`cardinality-constraints-sat`	3	0.231	0.601	0.450	0.657
`tseitin-formulas-unsat`	5	0.643	0.681	0.683	0.718
`summle-sat`	13	0.483	0.769	0.702	0.848
`cellular-automata-unsat`	3	0.568	0.775	0.717	0.839
`sudoku-unsat`	11	0.480	0.754	0.726	0.814
`multiplier-circuits-sat`	11	0.696	0.837	0.818	0.885
`hardware-model-checking-sat`	9	0.609	0.878	0.819	0.988
`planning-unsat`	2	0.817	0.992	0.900	0.992
`planning-sat`	3	0.943	0.966	0.964	0.982
`graceful-production-sat`	13	0.969	0.981	0.979	0.984
`sat-x-sat`	2	0.974	0.986	0.980	0.986
`software-verification-unsat`	14	0.971	0.980	0.981	0.990

the clause's first production. Even if we only consider instances with running times beyond 60 s, the same still holds for 70% (50%) of duplicates—indicating that the small time frames between the productions of a clause are not merely due to low running times. This harmonizes well with an earlier observation [5] that preventing shared clauses from being re-shared (*clause filtering*) is beneficial for around 5–15 s after its initial sharing but levels off for longer periods.

We now take a brief look at some generic diversification. Fig. 2 (right) shows the respective *DCPR* distributions; corresponding performance results are provided in Tab. 4. Most notably, **disabling diversification via sparse random variable phases results in substantially increased duplicate ratios** (median 0.35 to 0.49). This confirms that the randomly chosen phases clearly influence the solvers' observed behavior and performed work (cf. [3]).

5.3. Pairwise Solver Overlaps

We now examine pairwise produced clause overlaps (*PPCO*) between solver threads. We run each of MallobSat-base's solver backends alone to get a cleaner picture. Fig. 3 shows results for MallobSat-base. For the following discussion, Fig. 4 illustrates the geometric mean pairwise overlap ratios for the first two processes for each solver backend.

Lingeling [24]. We observe pairwise overlaps of at least 5% in a rather smooth structure. The choice of the solver configuration appears to have a small influence on the observed overlap. We do observe particularly small overlaps for configuration 1, which sets the *plain* option and hence disables a range of pre- and inprocessing techniques—indicating that they are responsible for many redundant clause productions. Further, we can observe side diagonals with higher overlaps, which represent pairs of solver threads of the same configuration.

CaDiCaL [23]. Apart from side diagonals with a few pronounced overlaps, most overlaps are below 5%. Configurations 0, 1, 6 incur very few overlaps—C0 flips the default phase to `false`, C1 is a preset for satisfiable instances, and C6 modifies the restart interval.

Kissat [22] (Fig. 4 bottom left). Average pairwise overlaps are much higher than for CaDiCaL, ranging up to 11%. This may explain why Kissat, somewhat surprisingly, performed worse as a single solver backend than CaDiCaL in recent MallobSat experiments [5]. Configurations 2–4, 6–8, and 11 seldom have an overlap larger than 2% with different configurations. C2, C4, and C7 have in common that they alter the restart interval, and C8 disables a plethora of simplification techniques such as *hyper-ternary resolution* (HTR), vivification, and sweeping. In particular, we suspected Kissat's HTR to contribute significantly to the exceptionally high number of ternary clause duplicates (see Fig. 1). In a follow-up experiment, we disabled HTR for all solvers. This not only reduced the median *DCPR* from 0.47 down to 0.33 but also improved performance (Tab. 4). Fig. 5 illustrates the impact on clause length distribution; pairwise overlaps are shown in Fig. 6.

The latest version of Kissat dropped many simplification techniques, including HTR [27]. We integrated this **2023 version of Kissat** (Fig. 4 bottom right) into MallobSat, reducing its diversification to the still supported options. Overlaps dropped drastically and are now typically below 1.5% except for pairs of solvers from process one, which range up to 6% (see below). Again, we note that **configurations adjusting the restart intervals have particularly small overlaps with other configurations**.

Across all backends, we observed that pairs of solvers within the same process show increased overlaps. To investigate this effect, we examined pairwise overlaps across six processes for the Kissat 2023 portfolio. Each process is at a different level of MallobSat's binary tree communication topology. For each distance Δ, we computed the pairwise overlaps of solvers with $|l_1 - l_2| = \Delta$ where l_1 and l_2

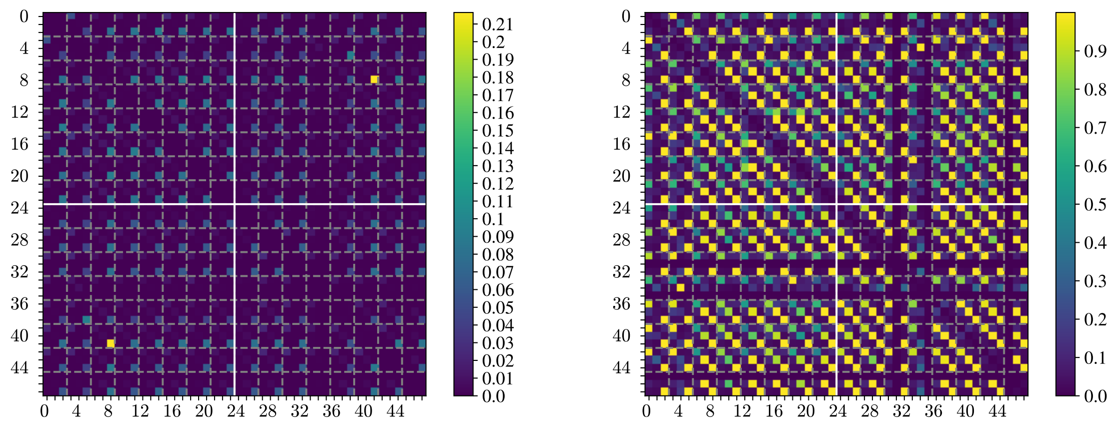

Figure 3: PPCO matrix (mean left, max. right) of the solvers of the first two processes (2×24 solvers) of MallobSat-base. The system cycles through the solvers Kissat, CaDiCaL, Lingeling in that order given the global solver IDs $0, 1, 2 \ldots$.

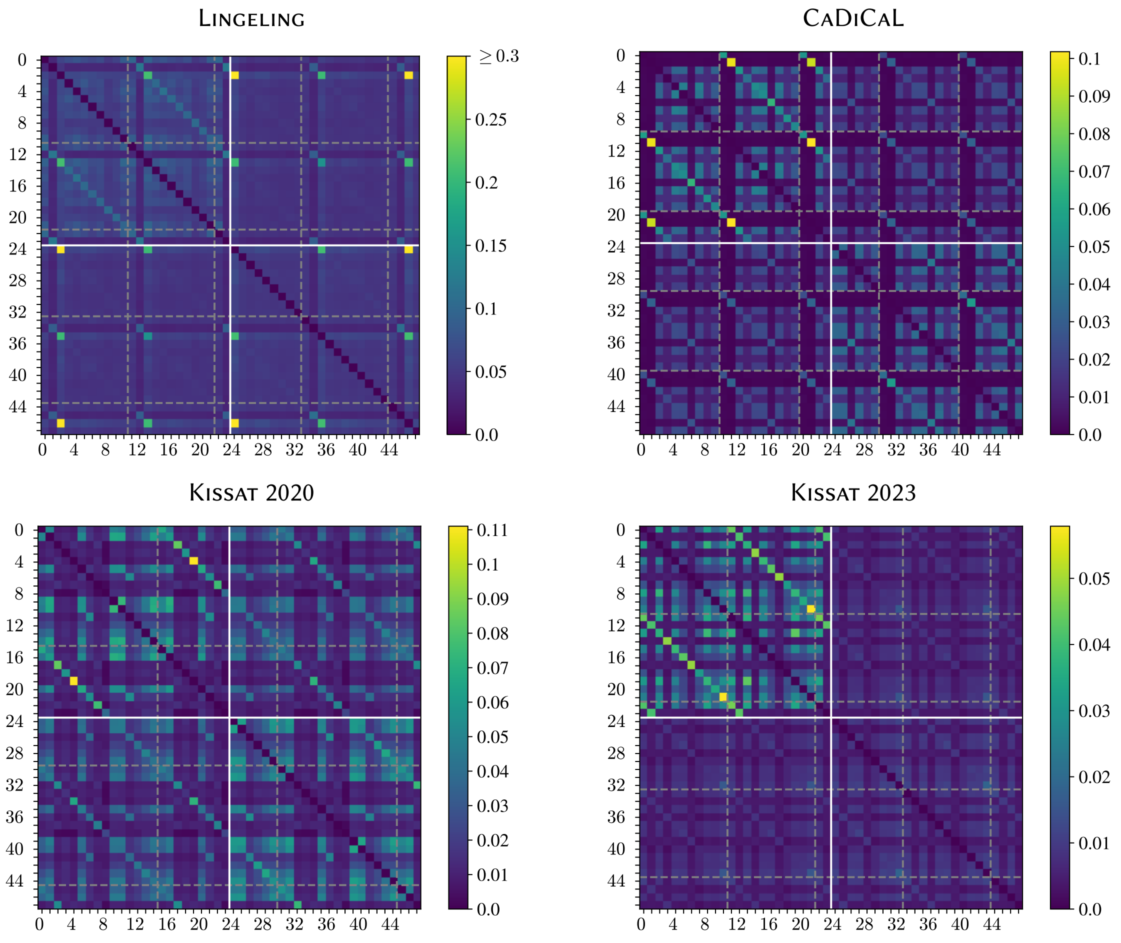

Figure 4: Geometric mean *PPCO* over all problem instances for the first two processes (48 solvers). White lines separate the processes; gray dashed lines separate the periods of solver configurations. In each period, configurations are assigned in the order as in Tab. 1. The solvers can be identified by their global solver IDs on the axes. Note the different color scales.

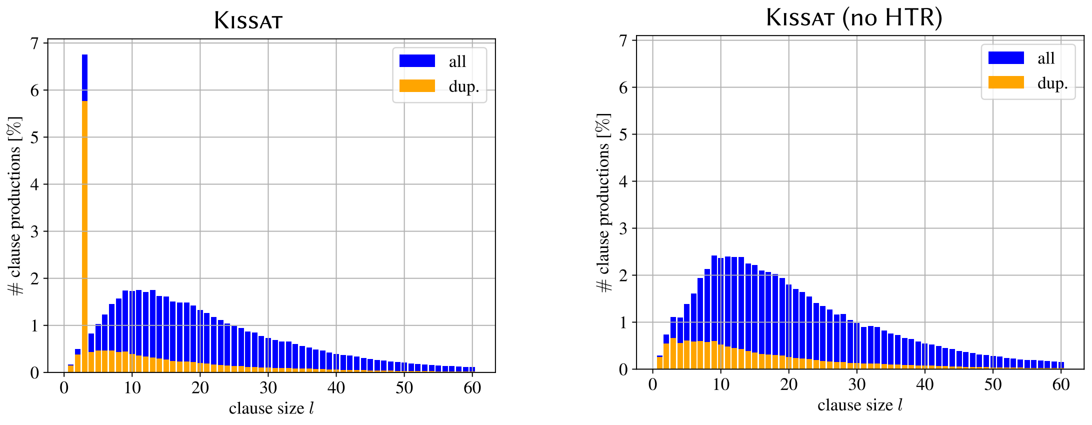

Figure 5: Effects of HTR in Kissat in terms of mean clause productions (with their fractions of duplicate productions) by clause length.

are the tree levels of their processes. We found that the pairwise overlaps drop for larger Δ as the median and the Q_3 percentile decrease almost monotonically from 0.007 to 0.0045 and from 0.025 to 0.015, respectively (see Fig. 7). We explain this trend as follows: Solver threads which import clauses at differing points in their program noticeably diversify their internal behavior. This is also supported by recent findings that MallobSat can perform well even when running *identical solver threads* [5]. In

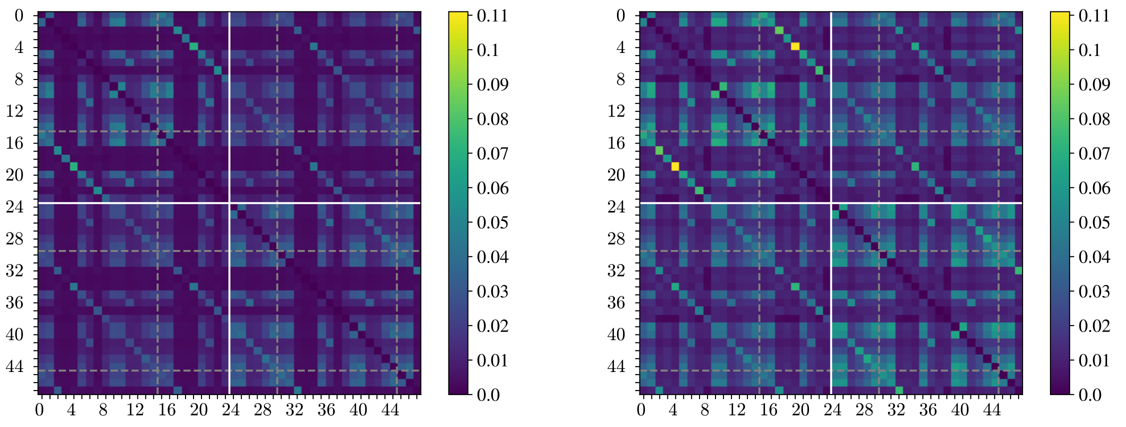

Figure 6: Effects of HTR Kissat: On the left the mean PPCO matrix of the first two processes of a Kissat portfolio is shown that disables *ternary resolution*. For comparison, the mean PPCO matrix of the Kissat protfolio (with *ternary resolution*) is shown on the right.

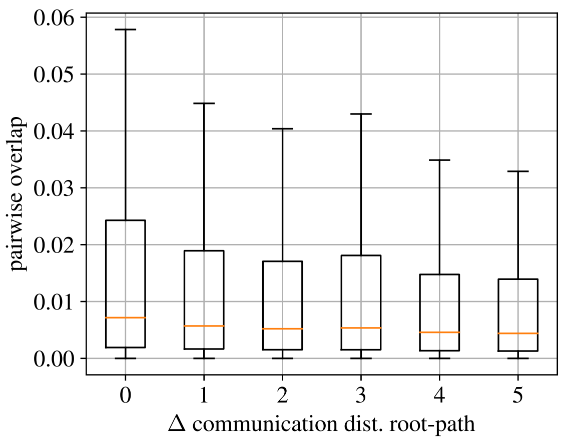

Figure 7: Boxplot i shows the distribution of the pairwise overlaps for a pairs of solvers (s_1, p_1) and (s_2, p_2) where the difference of the root-paths in MallobSat's communication topology (binary tree) for p_1 and p_2 is i. We considered 286 of 349 formulas in Kissat'23 where all 24 solvers per process were run.

addition to this effect, we acknowledge that our data may suffer from a slight selection bias concerning the initial process in particular, since this process always runs the longest and may sometimes solve a formula before other processes are initialized.

5.4. Mitigations and Improvements

To conclude our study, we want to gain an impression on how our findings can translate to solver improvements. We thus make small changes to MallobSat-base which we expect to mitigate some of the uncovered shortcomings. First, we turn off Kissat's HTR to combat the excessive ternary duplicates. Secondly, we address the peak in duplicate unit clauses, which we were able to trace back to Lingeling but not to any particular configuration thereof. Knowing that the vast majority of units is produced redundantly, we limit the Lingeling thread i out of n to only export units with $h \bmod n \in \{i, i+1\}$, where h is the unit's hash value; i.e., each unit clause can only be exported by two (adjacent) Lingeling threads. While this does not avoid the redundant production of these units, we prevent them from jamming MallobSat's clause sharing (since processes always prefer exporting them over other non-units). Thirdly, we test diversification via clause import times. Specifically, we delay the i-th solver thread's first successful clause import by $(i \bmod 11)$ import queries.

Table 4

Performance and statistics of configurations in two setups: with 349 "solvable" instances from SAT competition 2022 on 768 SuperMUC-NG cores ("SMNG"), and with 400 instances from the 2023 competition on 760 HoreKa cores ("HK"). PAR-2 averages running times, penalizing each timeout with twice the time limit. $\mathcal{B}$ denotes MallobSat-base, $\mathcal{B}*$ denotes $\mathcal{B}$ without phase diversification and without some solvers skipping clause sharing, and $\mathcal{B}+$ denotes our modifications.

		PAR-2	# solved	median $\lvert\mathcal{R}\rvert$	median $\lvert\mathcal{R}\rvert - \lvert\mathcal{H}(\mathcal{R})\rvert$	median $DCPR(\mathcal{R})$
SMNG	$\mathcal{B}$	**71.4**	**325**	1 559 177	326 275	0.34
SMNG	$\mathcal{B}$ (only `div-seeds`)	73.7	323	1 842 675	697 560	0.53
SMNG	$\mathcal{B} \setminus$ `div-phases`	74.5	323	1 468 320	485 933	0.49
SMNG	$\mathcal{B}*$ Kissat	90.6	315	1 485 261	684 272	0.47
SMNG	$\mathcal{B}*$ Kissat (w.o. `ternary`)	88.4	318	911 983	257 491	0.33
SMNG	$\mathcal{B}*$ Kissat 2023	79.9	322	785 487	201 848	0.24
SMNG	$\mathcal{B}*$ CaDiCaL	87.1	321	703 893	213 659	0.27
SMNG	$\mathcal{B}*$ Lingeling	126.3	297	923 702	217 103	0.43
HK	$\mathcal{B}$	160.2	308	1 966 634	556 837	0.37
HK	$\mathcal{B}+$	**158.9**	308	1 386 429	307 948	0.22

We tested our changes on a validation set, namely the SAT competition 2023 instances, and used 10 nodes (760 cores) of HoreKa. Tab. 4 shows results. Indeed, the median *DCPR* drops drastically from 0.37 to 0.22. Performance improvements are mild (geom. mean speedup 2.7%) and yet distinct from noise: 58% of instances were solved faster while 41% were solved more slowly. Overall, while our modifications were effective in terms of the metric they were based on, they do not translate to performance improvements of a similar magnitude. A possible explanation is that bursts of redundant clause productions may often involve low computational cost per production (e.g., a single simplification producing many clauses at once), rendering them a weak indicator for redundant work. That being said, our experiments do confirm that analyzing a clause-sharing solver through the lens of produced clause overlaps can help understand its behavior, uncover issues, and identify directions for improvements.

6. Conclusion

We have presented an empirical study on produced clause overlaps across solver threads in clause-sharing solving. Among other results, we found possible explanations for the relatively poor performance of MallobSat's Kissat portfolio, for the benefit of short-horizon clause filtering, and for MallobSat's scalability when running identical solver threads. Results also indicate that clause-sharing solvers are being set back by redundant inprocessing efforts.

Limitations. Our study only considers simple *syntactical equality* of clauses (when seen as sets of literals). By contrast, in many cases syntactically different clauses can be logically equivalent under the input formula (e.g., when variables are constrained to be equivalent), which we neglect in this study. Furthermore, we have consciously limited our study to a *black box* approach with unmodified solver backends, whereas future studies may consider a deeper look into the provenance of individual clauses. Lastly, considering different parallel solver systems was out of scope for this work but may result in new and/or different insights.

Future Work. We believe that our study indicates clear directions for improving parallel solvers going forward. In particular, fostering cooperation between solver threads in terms of pre– and inprocessing is an important open task in order to reduce the redundant work performed (cf. [28, 29]). On a pragmatic level, we prepare to publish our data acquisition and analysis code as a framework which developers and researchers can use with little effort.

References

[1] A. Biere, M. Heule, H. van Maaren, Handbook of satisfiability, volume 185, IOS press, 2009.

[2] T. Balyo, C. Sinz, Parallel Satisfiability, in: Y. Hamadi, L. Sais (Eds.), Handbook of Parallel Constraint Reasoning, Springer International Publishing, Cham, 2018, pp. 3–29. URL: http://link.springer.com/10.1007/978-3-319-63516-3_1. doi:10.1007/978-3-319-63516-3_1.

[3] T. Balyo, P. Sanders, C. Sinz, HordeSat: A Massively Parallel Portfolio SAT Solver, in: M. Heule, S. Weaver (Eds.), Theory and Applications of Satisfiability Testing – SAT 2015, volume 9340, Springer International Publishing, Cham, 2015, pp. 156–172. URL: http://link.springer.com/10.1007/978-3-319-24318-4_12. doi:10.1007/978-3-319-24318-4_12, series Title: Lecture Notes in Computer Science.

[4] D. Schreiber, P. Sanders, Scalable SAT Solving in the Cloud, in: C.-M. Li, F. Manyà (Eds.), Theory and Applications of Satisfiability Testing – SAT 2021, volume 12831, Springer International Publishing, Cham, 2021, pp. 518–534. URL: https://link.springer.com/10.1007/978-3-030-80223-3_35. doi:10.1007/978-3-030-80223-3_35, series Title: Lecture Notes in Computer Science.

[5] D. Schreiber, P. Sanders, MallobSat: Scalable SAT solving by clause sharing, Journal of Artificial Intelligence Research (JAIR) (2024). URL: https://dominikschreiber.de/papers/2024-jair-mallobsat-pre.pdf, in press.

[6] J. Marques-Silva, I. Lynce, S. Malik, Chapter 4. Conflict-Driven Clause Learning SAT Solvers, in: A. Biere, M. Heule, H. Van Maaren, T. Walsh (Eds.), Frontiers in Artificial Intelligence and Applications, IOS Press, 2021. URL: http://ebooks.iospress.nl/doi/10.3233/FAIA200987. doi:10.3233/FAIA200987.

[7] M. Böhm, E. Speckenmeyer, A fast parallel SAT-solver – efficient workload balancing, Annals of Mathematics and Artificial Intelligence 17 (1996) 381–400. doi:10.1007/bf02127976.

[8] M. J. H. Heule, O. Kullmann, S. Wieringa, A. Biere, Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads, in: D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, K. Eder, J. Lourenço, O. Shehory (Eds.), Hardware and Software: Verification and Testing, volume 7261, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 50–65. URL: http://link.springer.com/10.1007/978-3-642-34188-5_8. doi:10.1007/978-3-642-34188-5_8, series Title: Lecture Notes in Computer Science.

[9] T. Balyo, M. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions, Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland, 2023.

[10] T. Balyo, M. J. Heule, M. Iser, M. Järvisalo, M. Suda (Eds.), Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions, Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland, 2022.

[11] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, M. Suda, SAT Competition 2020, Artificial Intelligence 301 (2021) 103572. URL: https://linkinghub.elsevier.com/retrieve/pii/S0004370221001235. doi:10.1016/j.artint.2021.103572.

[12] Y. Hamadi, S. Jabbour, L. Sais, ManySAT: a Parallel SAT Solver, Journal on Satisfiability, Boolean Modeling and Computation 6 (2009) 245–262. URL: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SAT190070. doi:10.3233/SAT190070.

[13] G. Audemard, L. Simon, Predicting learnt clauses quality in modern SAT solvers, in: Twenty-first international joint conference on artificial intelligence, Citeseer, 2009.

[14] T. Ehlers, D. Nowotka, Tuning parallel sat solvers, Proceedings of Pragmatics of SAT 59 (2019) 127–143.

[15] Y. Hamadi, S. Jabbour, J. Sais, Control-Based Clause Sharing in Parallel SAT Solving, in: Y. Hamadi, E. Monfroy, F. Saubion (Eds.), Autonomous Search, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 245–267. URL: https://link.springer.com/10.1007/978-3-642-21434-9_10. doi:10.1007/978-3-642-21434-9_10.

[16] C. Sinz, W. Blochinger, W. Küchlin, PaSAT — Parallel SAT-Checking with Lemma Exchange:

Implementation and Applications, Electronic Notes in Discrete Mathematics 9 (2001) 205–216. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571065304003233. doi:10.1016/S1571-0653(04)00323-3.

[17] V. Vallade, L. Le Frioux, S. Baarir, J. Sopena, V. Ganesh, F. Kordon, Community and LBD-Based Clause Sharing Policy for Parallel SAT Solving, in: L. Pulina, M. Seidl (Eds.), Theory and Applications of Satisfiability Testing – SAT 2020, volume 12178, Springer International Publishing, Cham, 2020, pp. 11–27. URL: http://link.springer.com/10.1007/978-3-030-51825-7_2. doi:10.1007/978-3-030-51825-7_2, series Title: Lecture Notes in Computer Science.

[18] T. Ehlers, D. Nowotka, P. Sieweck, Communication in Massively-Parallel SAT Solving, in: 2014 IEEE 26th International Conference on Tools with Artificial Intelligence, IEEE, Limassol, Cyprus, 2014, pp. 709–716. URL: http://ieeexplore.ieee.org/document/6984547/. doi:10.1109/ICTAI.2014.111.

[19] L. Le Frioux, S. Baarir, J. Sopena, F. Kordon, PaInleSS: A Framework for Parallel SAT Solving, in: S. Gaspers, T. Walsh (Eds.), Theory and Applications of Satisfiability Testing – SAT 2017, volume 10491, Springer International Publishing, Cham, 2017, pp. 233–250. URL: http://link.springer.com/10.1007/978-3-319-66263-3_15. doi:10.1007/978-3-319-66263-3_15, series Title: Lecture Notes in Computer Science.

[20] D. Schreiber, Scalable SAT Solving and its Application, Ph.D. thesis, Karlsruhe Institute of Technology, 2023.

[21] V. Vallade, J. Sopena, S. Baarir, Enhancing state-of-the-art parallel SAT solvers through optimized sharing policies, Pragmatics of SAT (2023).

[22] A. Biere, K. Fazekas, M. Fleury, M. Heisinger, CaDiCaL, kissat, paracooba, plingeling and treengeling entering the SAT competition 2020, in: Proc. SAT Competition, 2020, p. 50.

[23] A. Biere, M. Fleury, M. Heisinger, CaDiCaL, kissat, paracooba entering the SAT competition 2021, in: Proc. SAT Competition, 2021, pp. 10–12.

[24] A. Biere, Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018, Proceedings of SAT Competition (2018) 14–15.

[25] M. Osama, A. Wijs, Verifying string safety properties in AWS C99 package with CBMC, in: Proc. SAT Competition, 2021, p. 64.

[26] J. Elffers, J. Nordström, Documentation of some combinatorial benchmarks, in: Proc. SAT Competition, 2016.

[27] A. Biere, M. Fleury, F. Pollitt, CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT entering the SAT competition 2023, in: Proc. SAT Competition, 2023, p. 14.

[28] M. Iser, T. Balyo, C. Sinz, Memory efficient parallel SAT solving with inprocessing, in: Proc. ICTAI, IEEE, 2019, pp. 64–70. doi:10.1109/ictai.2019.00018.

[29] M. Osama, A. Wijs, A. Biere, Certified SAT solving with GPU accelerated inprocessing, Formal Methods in System Design (2023) 1–40. doi:10.1007/s10703-023-00432-z.