Streamlining Distributed SAT Solver Design

Dominik Schreiber ✉ 🏠
Karlsruhe Institute of Technology, Germany

Niccolò Rigi-Luperti ✉ 🏠
Karlsruhe Institute of Technology, Germany

Armin Biere ✉ 🏠
University of Freiburg, Germany

Abstract

Distributed clause-sharing SAT solvers have recently been established as powerful automated reasoning tools that can conquer previously infeasible instances. A common design of distributed SAT solvers is to run many off-the-shelf SAT solvers with world-leading sequential performance, employ some best-effort diversification (e.g. restart intervals, decision orders, different implementations), and share conflict clauses among the solver threads. This approach, naïvely, adopts all best practices of sequential solver design for distributed solving, where these practices may be less useful or even actively detrimental. In this work we diagnose such shortcomings in the state-of-the-art system MallobSat and propose first effective mitigations. In particular, we replace the redundant pre– and inprocessing at all threads with single-core preprocessing that runs next to the parallel search, remove LBD values from the clause-sharing operation, and slim down solver diversification to very few lightweight and uniform methods. Experimental evaluations on up to 3072 cores (64 nodes) confirm that our measures improve performance while also drastically simplifying the SAT solving program that is run in parallel.

2012 ACM Subject Classification Hardware → Theorem proving and SAT solving; Theory of computation → Distributed algorithms

Keywords and phrases Satisfiability, parallel SAT solving, distributed systems, preprocessing

1 Introduction

Over the last years, parallel and distributed approaches to propositional satisfiability (SAT) solving have gained significant traction and established themselves as powerful automated reasoning tools to conquer previously infeasible problems [22, 55]. In large parts, this success is due to careful exchange of *conflict clauses* that are learned by individual sequential SAT solver threads during their search [41]. Recent findings [55] indicate that such clause sharing can serve as its own powerful kind of parallelization, even if all solver threads are initially completely identical, challenging the predominant view that solver threads need to either explicitly operate on different sub-spaces (*search space splitting*) or be explicitly *diversified* in terms of their approaches (*portfolio*) [55]. In light of these new insights, we explore ways to improve both simplicity *and* performance of distributed SAT solving.

State-of-the-art parallel and distributed solvers commonly employ cutting-edge SAT solvers as sequential building blocks. These sequential solvers, most recently CaDiCaL [11] and Kissat [12] in particular, are highly engineered pieces of software optimized for sequential, general-purpose SAT solving performance [26]. To this end, they feature a large number of simplification or *preprocessing* techniques, some of which are heuristically scheduled into their solver program (*inprocessing*) [17]. In most established parallel solvers, these tasks are performed by all solver threads in a fully redundant fashion, with the exception of some coordinated preprocessing [21, 49] and disabling individual techniques for individual threads in the name of diversification [4, 54]. While inprocessing significantly improves sequential SAT solving performance [17], its impact for parallel and distributed clause-sharing solving is largely unexplored (cf. [20]). One notable caveat is that pre– and inprocessing tasks, performed redundantly at all solver threads, are not parallelized and can therefore present a scalability bottleneck. This holds especially if the effort spent on a pre–/inprocessing task is not a fixed, small *share* of each thread's CPU time (e.g., 10%) but rather takes up large portions of the *total, parallel* CPU time – which we do confirm to be the case for some techniques. Another aspect is that simplification techniques often heavily modify the original formula. When two solvers operate on different representations of the problem, efficient and effective information exchange can be obstructed. For instance, Schreiber and Sanders noticed that bounded variable elimination (BVE) can prevent solver threads from importing many incoming clauses because the clauses feature variables that are locally already eliminated [55].

In this work, we diagnose and mend some of the mentioned shortcomings in modern distributed SAT solver design. A large portion of our contributions are of an analytical and observational nature. In particular, we profile the cutting-edge distributed SAT solver MallobSat [54, 55] in terms of the amount of time individual solver threads spend in different tasks. We observe that pre–/inprocessing provides little to no net benefit for distributed solving and that disabling it does in fact improve performance. Similarly, we complement recent findings [35, 50] questioning the significance of the Literal Block Distance (LBD) [2] metric for clause-sharing solving. We subsequently propose first improvements as a reaction to these insights: We implement a single-core preprocessing approach that runs while "pure" CDCL solving is already ongoing, remove LBDs from distributed clause sharing, and drastically cut diversification of solver threads down to two trivial methods (random seeds and variable phases). Our enhanced setup of MallobSat outperforms the prior state of the art, especially on application-oriented benchmarks, while only executing pure, uniform, clause-sharing-powered CDCL search at the parallel cores – further departing from a "portfolio solver". Going forward, our results call for designing distributed SAT solvers in a more holistic and unified manner, whereas our proposed solving approach opens up new opportunities for understanding, analyzing, and advancing distributed clause-sharing SAT solving.

The paper at hand is structured as follows. Section 2 introduces relevant preliminaries and related work. Two empirical studies follow, first on examining pre–/inprocessing in distributed SAT solving (Section 3) and secondly on the role of LBDs (Section 4). In Section 5, we propose an alternative approach to integrating simplification techniques into distributed SAT solving in a more scalable manner. Section 6 features a thorough experimental evaluation. We discuss results in Section 7 and conclude our work in Section 8.

2 Background

In the following, we provide some necessary background for the work at hand.

Given a propositional formula $F = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{k_i} l_{ij}$, where each l_{ij} is a Boolean variable or its negation, the propositional satisfiability (SAT) problem is to find a variable assignment that satisfies F or, if impossible, to report *unsatisfiability* [16]. Most sequential state-of-the-art SAT solvers are based on the *Conflict-Driven Clause Learning* paradigm [41], which encompasses a heuristic search over the space of partial variable assignments. When a CDCL solver encounters a logical conflict under its current assignment, it can derive a so-called *conflict clause* and add it to its knowledge base. Intuitively, such a clause represents a *pruned sub-space* of search space that has been deemed unsatisfiable.

In *distributed computing*, multiple machines with no shared memory perform a joint computation. They usually coordinate by exchanging messages over a network. Parallel and distributed approaches to SAT solving have mostly been separated into two major classes of approaches: search space splitting and portfolio solving [5]. *Search space splitting* (a.k.a. search space partitioning, divide&conquer, later cube&conquer [34]) was the earliest parallelization technique, forcing workers to search pairwise disjoint sub-spaces of search space. While this is situationally powerful [31, 33], load balancing problems can occur if the problem is split in an uneven or ineffective manner [56]. *Portfolio solving* has been proposed to circumvent these problems, instead aiming to achieve speedups by running many *different* sequential SAT solvers on the same, original, formula [30]. Both parallelization paradigms have been enhanced by *clause sharing*, i.e., mechanisms to exchange learned conflict clauses across workers [30, 57]. The result is a form of *parallel search space pruning*: If solver thread A learns clause c and transmits it to thread B, then B will not re-explore the sub-space prohibited by clause c. This pruning is imperfect in nature since solver threads periodically *forget* (i.e., discard) clauses, losing pruning information in the process [2, 43]. Still, earlier works showed that clause sharing is beneficial for performance of portfolios [4, 30] (resulting in *clause-sharing portfolio solvers*) and search space splitting solvers [57].

Modern distributed SAT solving systems include PRS [61], Painless [48], and MallobSat [54, 55], which generally fit the clause-sharing portfolio paradigm. MallobSat in particular features a highly engineered form of periodic *all-to-all* clause sharing. Its authors demonstrated that this clause sharing is the central driver of the system's scalability. Notably, MallobSat performs rather competitively even if *all solver threads are initially identical* – down to the exact same seed – the reason being that individual threads import slightly different clause sets at slightly different points in time due to the non-determinism of the distributed program. This alone suffices for clause sharing to act as effective search space pruning [55]. While this finding does not immediately translate to a new distributed solving approach – usual diversification of solvers still benefits performance – it raises the question of whether the notion of *clause-sharing portfolios* still applies to a system that does not depend on any actual portfolio. Schreiber & Sanders suggested the term *clause-sharing solver* [55] to emphasize that clause sharing is, demonstrably, its very own kind of parallelization and should not be considered a mere supplement to portfolio solving or search space splitting.

We refer to the Handbook of Satisfiability [17] for an overview of pre– and inprocessing techniques. Regarding *parallel* pre- and inprocessing, Gebhardt et al. [27] presented a shared-memory parallelization of certain techniques including *subsumption*, *clause strengthening*, and *bounded variable elimination* (BVE). The scalability of this approach is rather limited due to the employed memory lock schemes. Wieringa et al. [59] showed that vivification-like inprocessing could be elegantly achieved in a parallel fashion by adding a second pure clause strengthening thread to each CDCL solver. The ParKissat system [60] applies some single-core preprocessing prior to parallel solving, including techniques to handle XOR and cardinality constraints, which was adopted by PRS [21] and subsequently by the state-of-the-art shared-memory solver PL-PRS-BVA-KISSAT [48]. The latter system also runs a number of *bounded variable addition* (BVA) [29, 40] threads next to CDCL [49]. The results from all but one BVA thread are discarded; other threads can seamlessly integrate the "winning" result in their ongoing CDCL search. To our knowledge, the solver threads in the mentioned systems still perform inprocessing during search, on top of the coordinated preprocessing.

Focusing on shared clauses, Vallade et al. [58] proposed to boost sharing efficiency by concurrently strengthening *shared* clauses since, in a parallel setting, these might be the most rewarding clauses to invest work in. They observed improved performance for unsatisfiable but negative results for satisfiable instances. Iida et al. [35, 36] recently tested several metrics for selecting clauses to share. In contrast to the older LBD metric [2], which depends on the solver state [3], their proposed metrics are based on the formula's (static) graph structure.

Osama et al. [44] introduced utilizing GPUs for pre-/inprocessing, achieving speedups by more than an order of magnitude for garbage collection and gate detection. For variable elimination however, the speedups were limited to around 1.5, highlighting the difficulty of uniting the rather irregular program flow of SAT solving with monolithic GPU architecture.

3 Examining Pre–/Inprocessing in Distributed SAT

In the following, we present and analyze running time profiles of a state-of-the-art distributed clause-sharing solver in order to gain insights in its behavior and directions for improvements.

We use MallobSat, essentially configured as in the International SAT Competition 2024 [52] but using only 2024 Kissat [12] as its solver backend. In the scope of this work, we decided to focus on a single cutting-edge solver backend (Kissat) in order to obtain clean results and keep the experimental design space more manageable. We use the HPC cluster SuperMUC-NG, where each node features a two-socket Intel Skylake Xeon Platinum 8174 processor clocked at 2.7 GHz with 48 physical cores (96 hardware threads) and 96 GB of RAM. For the following experiments, we use 16 nodes (768 cores), which amounts to a decently large scale and still allows us to run more experiments than we could at larger scales. In order to obtain profiling results, we make use of Kissat's integrated profiling capabilities: each solver thread bookkeeps the accumulated time it spends in relevant sub-procedures. Given that used CPU time is challenging to obtain asynchronously for a solver thread in MallobSat's multi-threaded processes, we changed Kissat's profiling to use *wallclock* rather than CPU time.

In terms of problem inputs, we need to balance responsible use of computational resources with instance diversity and robustness of results [52]. To ensure a great variety of benchmark families, we consider all 796 unique instances from the last *two* SAT Competitions, 2023 and 2024. We then reduce the resulting benchmark set to a random selection of 396 instances – reserving the remaining 400 instances for scaling experiments (Section 6).

Fig. 1 shows the ratio of time the solver threads spend in selected tasks for single-core Kissat (left) and for 768-core MallobSat (right) for each of the 396 instances. Fig. 2 (left) shows according statistics; more details are provided in the appendix (Tab. 4). First, it is worth mentioning that Kissat, as other sequential solvers participating in the SAT Competition, is tuned for running times up to 5000 s. At distributed scales we usually aim for much lower running times on the order of 300 s [55]. For such time scales, the inprocessing scheduling may turn out suboptimal. We thus ran Kissat for up to 5000 s (vs. 300 s for MallobSat) to aggregate more representative statistics for it.

The label "`search`" in Fig. 1 denotes actual CDCL search, which is the only parallelized

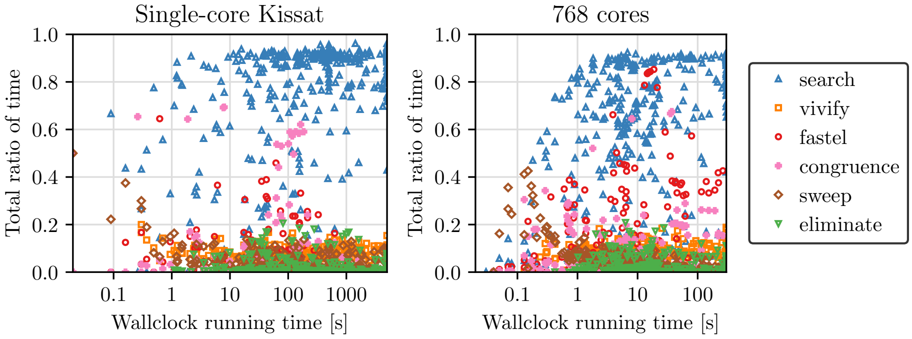


Figure 1 By-input distribution over the total solver time ratio spent in actual (CDCL) search and in selected pre–/inprocessing tasks, for single-threaded Kissat (left) and at 768 cores (right), relative to overall (wallclock) running time.

	Seq. Kissat		768 cores	
Category	mean	total	mean	total
search	66.74	86.32	63.29	79.15
vivify	6.30	7.03	8.32	8.35
fastel	0.30	0.30	0.76	2.96
congruence	0.24	0.46	0.55	1.85
sweep	2.08	2.18	2.88	1.54
eliminate	0.31	1.01	0.75	1.33

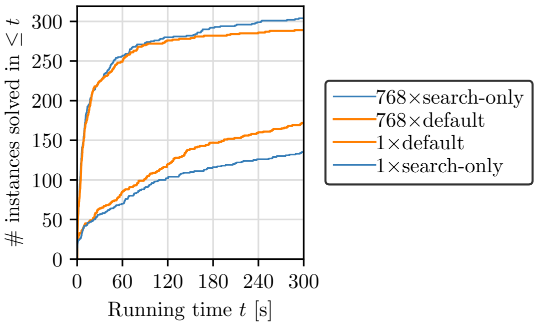


Figure 2 Left: Percentages of time spent in selected tasks, for sequential Kissat and 768-core MallobSat, in terms of the geometric mean over all non-zero time ratios reported across all solver threads and inputs ("mean") and in terms of adding up the total time all solver threads spent on a technique and then dividing by the sum of total running times across all solver threads and inputs ("total"). Right: Performance with and without pre–/inprocessing, at one and 768 cores each.

task (via clause sharing). Visually, the markers for sequential Kissat corresponding to search are strongly clustered at about 0.9, while they appear lowered and more scattered for distributed MallobSat. The total ratio of solver time spent in search is 86% for single-core Kissat but only 79% for 768-core MallobSat. By contrast, most pre–/inprocessing techniques [12] take proportionally more time in our distributed setting. Some of the most costly tasks are vivification [39], initial fast variable elimination ("fastel") and, to a lesser extent, congruence closure, SAT sweeping, and variable elimination during inprocessing. For example, the relative cost of vivification increases from 7% at one core to 8.4% at 768 cores, possibly because of the increased number of vivification candidates due to clause sharing.

Other techniques however, such as initial variable elimination, do not follow this logic and are expected to incur a *fixed* amount of work. At a distributed scale, where CDCL search profits from speedups, the redundant and non parallelized preprocessing on all cores accounts for increasing portions of the total work performed (from 0.3% to 2.96% for "fastel"). SAT sweeping is an exception to this effect (relative cost *decrease* from 2.18% to 1.54%), the probable reason being that it only incurs significant relative cost at very low running times (< 1 s), where MallobSat performs similar or even worse than sequential Kissat. Also note how some tasks such as vivification use relatively consistent shares of time since Kissat can control their computational budget in an accurate manner. Other tasks that are highly dependent on problem structure, such as initial variable elimination, show high variances in the time spent and occasionally even account for the majority of compute time.

The impact on solving performance is shown in Fig. 2 (right). We ran MallobSat as before and also in a search-only configuration where pre– and inprocessing is disabled for all threads. At a sequential scale, Kissat with pre–/inprocessing significantly outperforms its search-only variant, as one would expect. At 768 cores, however, many instances are slowed down by the cost of preprocessing and simplification. As such, our "search-only" setup achieves a geometric mean speedup of 3.7% over the default setup (computed over commonly solved instances) and solved 12 more instances. Let us discuss some potential causes for this effect:

- As presented above, certain pre–/inprocessing techniques can constitute a large share of the *total* (parallel) work performed, whereas their merit remains constant regardless of the number of cores computing it. Due to this scalability bottleneck, the techniques incur increasing opportunity cost that eventually outweighs their benefits: Our search-only setup raises the share of time spent on search from 79.15% to 95.4% (appendix Tab. 4).

- Clause sharing may functionally "simulate" certain pre–/inprocessing techniques to some degree. This is plausible for techniques related to identifying short clauses that are implied by the input formula, such as subsumption or vivification: Since MallobSat's clause sharing aggregates the globally shortest distinct clauses [55], it may become unlikely that vivifying a clause results in a shorter clause that *no other solver thread* has found so far. That being said, we are yet to find specific evidence for (or against) this suspicion.
- Deviating preprocessing levels across solver threads may result in "language barriers" across solvers, hindering cooperation and rendering clause sharing less effective. This may especially apply to techniques eliminating variables, which restrict the clauses that a solver thread can subsequently import [55]: In our 768-core experiment with pre–/inprocessing, the geometric mean ratio of incoming clauses that were imported successfully by a solver thread is 62%. The median ratio is higher (75%) since every tenth solver thread is configured to skip variable elimination completely and can therefore import all clauses. Still, about 22% of solver threads could import *less than half* of their incoming clauses.

In terms of benchmark families [37], we observed that pre–/inprocessing is most beneficial for unsatisfiable miter instances (i.e., combinational circuit equivalence checking; nine instances with mean speedup 3.4 and two more instances not solved by "search-only"), which is unsurprising since Kissat's SAT sweeping and congruence closure are designed for this exact application [13, 14]. Notable benchmark families where the search-only setup performed better include satisfiable instances on scheduling (9 instances, speedup 1.59) and Folkman graphs (5 instances, speedup 1.83). For the latter the mean time ratio spent on CDCL search increases from 89.1% to 99.2% when disabling pre–/inprocessing, mostly due to vivification (7.7%). Additional family-specific results are given in the appendix (Table 2).

All in all, the presented results paint a clear picture: Pre– and inprocessing techniques are currently being employed in distributed SAT solving in a suboptimal manner. They are neither designed nor tuned for clause-sharing solving and incur significant cost which limits the attainable speedup and, in many cases, outweighs their benefits.

4 Revisiting LBDs in Distributed Solving

In sequential solving, the Literal Block Distance (LBD) of a clause indicates the number of decisions that were made to arrive at the conflict [2]. LBDs have proven useful as a quality metric, i.e., to decide whether to keep or to discard a clause [43]. LBDs have also been used in many parallel SAT solvers [1, 4, 25, 38] as a metric to decide which learned clauses a solver unit should contribute to clause sharing. However, the usefulness of LBDs does not necessarily translate to parallel clause sharing: A clause's LBD depends on the solver state and may thus not be meaningful to a different solver (thread) [3]. As recently evidenced, using LBD as a clause selection metric is not beneficial compared to clause length [35, 54]. MallobSat currently uses LBDs only as a tie-breaker, after clause length, when prioritizing clauses to share. Its authors also reported very similar performance for different ways of manipulating the LBD values of *incoming* shared clauses, just before a solver thread imports them [55]. However, some of the underlying tests have the potential flaw that they do not only change individual LBD values but also their overall *distribution*. This distribution can be relevant since it influences the volume of kept clauses and, therefore, the propagation speed and search behavior. Changing it, e.g., by assigning random LBD values to incoming clauses, may thus result in side effects which improve or degrade performance.

In the following, we examine two aspects of LBDs in distributed solving separately: The significance of a particular incoming clause carrying a particular LBD, and subsequently the

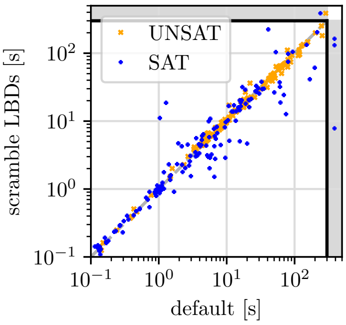


Figure 3 By-instance solving time comparison between a default run of MallobSat and a run where LBD scores of shared clauses are shuffled within each clause length, both at 768 cores.

distribution over the LBDs of incoming clauses.

To reexamine the significance of individual LBDs, we suggest the following experiment: After globally aggregating all clauses to share, we *shuffle* the mapping of clauses to LBD values *within* each clause length uniformly at random. As such, we can cleanly observe the effect of disconnecting LBDs from their meaning while the distribution over LBD values in solver threads remains fixed. As Fig. 3 shows, we were unable to discern any meaningful differences in solving performance with vs. without shuffling of LBD scores: Shuffling LBD scores resulted in one additional solved instance (290 vs. 289) and a speedup of 1.27%. This complements recent evidence [35, 55] that communicating and forwarding the LBDs of individual shared clauses does not carry any useful information in parallel clause-sharing.

Given that *individual* LBD values do not appear to be of any significance, let us now separately evaluate the impact of different *distributions* over incoming LBD values. Specifically, we explore different strategies to fabricate LBDs synthetically. For a synthetic distribution to be sensible, our intuition is that high-LBD clauses should be more frequent, low-LBD clauses should be rare, and min-LBD clauses (LBD 1 or 2) should be avoided, reflecting the typical three-tier clause management in modern solvers [15, 43]. We refrain from assigning min-LBD clauses because solvers commonly keep them *indefinitely*, which may lead to overcrowding if too many such clauses arrive from external sources [55]. We compare four strategies:

1. **Original LBDs**: The original LBD values are kept, used as a secondary clause sharing selection metric (after clause length), and incremented upon import [55].
2. **Deactivated LBDs**: When a clause is produced, its LBD value is set to its maximum value possible [24], the size of the clause. This functionally deactivates any logic related to LBDs in the distributed solver.
3. **Uniform LBDs**: When a clause is produced, its LBD value is set to its maximum. When a clause is imported and its size is $|c| \geq 3$, the importing thread randomly draws a new LBD $\in [3, \ldots, |c|]$ with uniform probability for each value. Initially setting LBDs to their maximum ensures that they play as little a role as possible during clause sharing.
4. **Triangular LBDs**: When a clause is produced, its LBD value is set to its maximum. When a clause is imported and its size is $|c| \geq 3$ the importing thread randomly draws a new LBD $\in [3, \ldots, |c|]$ whereas each LBD value s has a probability weight of $s - 2$. Visually, the probability density resembles a triangle; low LBDs receive a lower probablity to be sampled and high LBDs a higher probablity. This distribution produces fewer low-LBD clauses than the uniform distribution, which could affect the clause databases.

We ran each of these four strategies on 396 instances with a timeout of 300 s, distributed

LBD Strategy	#	+	−	PAR2
Deactivated	322	153	169	137.2
Original	322	153	169	137.3
Triangular	319	150	169	139.9
Uniform	318	148	170	141.2

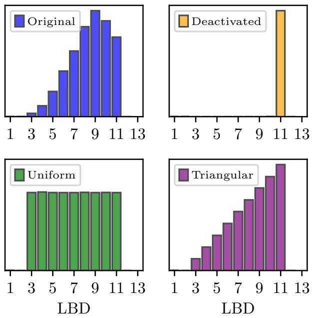


Figure 4 Left: Performance of different strategies for fabricated LBDs, showing number of solved instances (#), of which satisfiable (+) and unsatisfiable (−), and PAR2 scores. Right: Measured LBD distributions, observed directly at import in each thread. Here shown are statistics for clauses of size 11 during solving of a single instance. Each histogram represents ≈ 800 000 clauses.

on 768 cores. We also deactivate all other sources of diversification (random seeds and sparse random variable phases [4]), as in ref. [55], such that any diversification introduced by just the LBD settings would be most observable. Results are shown in Fig. 4. Overall, all strategies performed similarly; best performance is achieved by using and incrementing original LBD scores or by removing them from the equation entirely. We were unable to observe any positive diversification effect due to randomized LBD scores in the triangular and uniform strategies. We suspect that these two strategies may still result in too many low-LBD clauses, at least at the tested scale and with MallobSat's default sharing configuration.

Put together, we found that removing LBD scores from clause sharing alltogether (and assigning the maximum LBD score to each incoming clause during import) preserves performance. Together with earlier findings [35, 55], this paints the clear picture that LBDs are not a useful metric for modern clause-sharing solvers. Going forward, we consider to either replace LBD with a different metric, as proposed by Iida et al. [35], or to remove them entirely from MallobSat's solver interfaces and sharing logic.

5 Streamlined Solver Design

Our profiling results (Fig. 1–2) have shown that distributed clause-sharing solving performs pre– and inprocessing in a highly suboptimal manner. We now suggest some first improvements to this situation, incorporating insights from the above discussion and measurements. We can discern several general approaches for any single pre–/inprocessing technique:

1. **Keep** the technique as is, i.e., computed redundantly by all threads. This (non-)approach can be sensible if the technique consistently incurs little cost.
2. **Disable** the technique. This is reasonable especially if the technique is fairly expensive and is difficult or even infeasible to parallelize, or if it causes negative side effects for the parallelization. An example is variable elimination during inprocessing, which prevents the import of many clauses if not performed in a synchronized fashion [55].
3. **Sequentially** apply the technique with a single solver thread, then distribute the result. This can be a practical compromise for techniques that are challenging to parallelize, feasible to perform sequentially in little time, but still costly if performed by all threads.
4. **Parallelize** the technique. This can be an option if the technique is beneficial in a distributed setting and if it is expensive sequentially but feasible to parallelize. While

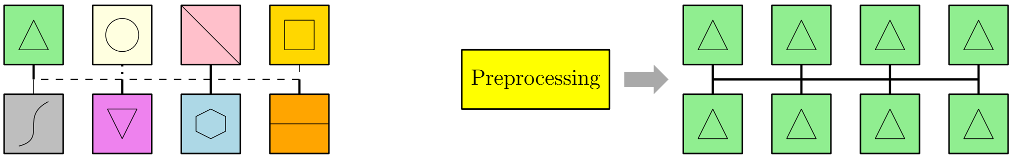


Figure 5 Left: Common distributed solver design, where a portfolio of solvers (colored squares) with diverse strategies (symbols in the squares) periodically exchange insights (connections between squares) with some "language barriers". Right: Streamlined design, where parallel solver threads are *uniform* CDCL searchers and operate on the results of preprocessing that is logically *separate*.

there have been some works in this regard (see Section 2), so far next to no techniques have been explored in the context of state-of-the-art *distributed* SAT solving.

MallobSat's default configuration follows option 1 (for most threads) and 2 (for some threads, for the sake of diversification). In the long term, we are interested in finding scalable parallel algorithms for promising inprocessing tasks (option 4). In the scope of this mostly empirical work, we consider options 2 and 3 to obtain a more scalable distributed solver.

Specifically, we propose the following changes to MallobSat's setup: We drop all pre– and inprocessing, which, for the case of Kissat, amounts to disabling the options `preprocess`, `simplify`, `probe`, and `lucky`. Since many of MallobSat's "native" diversification options [55] concern individual pre–/inprocessing techniques, we disable this kind of diversification as well. This results in a "pure" clause-sharing solver with uniform worker threads (Fig. 5 right) and next to no characteristics of a *portfolio* (Fig. 5 left). We then configure a single thread to perform a single run of a number of crucial simplification techniques – similar to the preprocessing performed by PRS and PL-PRS-BVA-KISSAT (see Section 2) and also how the early portfolio solver Plingeling used to handle preprocessing [6].[1] Our approach differs in that all distributed resources are tasked *immediately* with search-only SAT solving as long as preprocessing is still ongoing. In terms of employed preprocessing techniques, we use the out-of-the-box preprocessing capabilities of Kissat, which includes not only bounded variable addition but also congruence closure and SAT sweeping, fast initial variable elimination, backbones, and lucky phases [12].

Once completed, the preprocessor reports the entire preprocessed formula. Since our execution environment Mallob natively supports concurrent execution and flexible load balancing of SAT solving tasks [47], we can simply introduce the preprocessed formula as an additional task that runs next to the original solving task. This raises the question of how to distribute the available computational resources among these two tasks. Assume that the original task j_o has been running for time t_o whereas the preprocessed task j_p has been running for time t_p. Given equal time and resources, we assume that the preprocessed task is likely to be easier to solve. Therefore, as $\frac{t_p}{t_o}$ increases and eventually approaches 1, the relative worth of j_p over j_o increases. Following up on this intuition, we suggest to gradually shift computational resources from j_o to j_p until only a single process for j_o is remaining. Eventually, j_o is interrupted completely. A result is reported if the preprocessor manages to solve the formula directly or when either j_o or j_p report a result. A satisfying assignment reported by j_p must be converted to a satisfying assignment to the original formula, using the preprocessor's data structures for solution reconstruction.

[1] In a sense, Plingeling's "preprocess-then-fork" model has gotten buried due to HordeSat's [4] slim and modular solver interface, prohibiting this mode of usage when integrating Plingeling's solver diversification, and thus never found its way into HordeSat and later MallobSat.

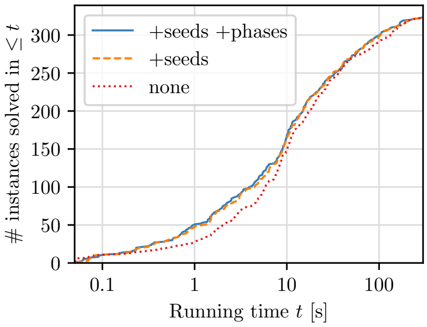
6 Evaluation

In the following, we provide an evaluation of our improvements.

First of all, we examine the impact of different sources of solver thread diversification. Our search-only setup with prior single-core preprocessing only features a single solver backend (Kissat) and no longer features a hand-crafted sequence of solver configurations that are being cycled over (cf. [54]). This drastically cuts diversity and leaves what can be characterized rather as a uniform array of CDCL searchers than a classical "portfolio" (Fig. 5). In our study on LBDs (Section 4), we found that the diversification introduced by deviating LBD distributions across solver threads does not influence performance even when all other sources of diversification are disabled. We thus follow the "Deactivated" LBD strategy in our new setup. Two other computationally cheap, basic, and uniform diversification techniques we consider are setting different random seeds and variable phases per solver [4, 55]. We thus ran our baseline without any diversification, with seeds only, and with seeds and phases.

Results are reported in Fig. 6. While MallobSat still performs well *without* any explicit diversification – an effect examined in detail in ref. [55] and attributed to deviating clause imports across solver threads – we see that different random seeds are beneficial for short running times, especially below 10 s. Random seeds essentially provide a warm-start for deviating search behavior rather than having to wait for diversity induced by the first few clause imports. This contrasts randomized LBDs, which can only have any impact after the first sharing(s). Randomized phases only have a minimal effect, but still lead to a net positive and incur no opportunity cost. We also explored some additional diversification, such as randomizing VSIDS score decays or clause database reduction ratios, without observing significant improvements; according results are given in the appendix (Fig. 12–13).

Next, we evaluate different strategies surrounding our single-core preprocessing. An overview is provided in Fig. 7. First, adding sequential preprocessing to an otherwise search-only run and launching a task j_p that operates on the preprocessed formula is highly beneficial to performance. The run with preprocessing solves 19 additional instances and improves performance for several unsatisfiable instances (which incur much less running time variance than satisfiable instances). The preprocessor solved 18 instances directly and mostly finishes in few seconds. Secondly, rather than allotting equal resources to j_0 and j_p (as in the prior run), we gradually shift resources from j_0 to j_p, over a time span corresponding to twice the preprocessor's running time, and then remove j_0 entirely. For long running times, this approach is preferable for unsatisfiable instances since it eventually commits all available



Diversification	#	+	−	PAR2
+seeds +phases	324	155	169	131.9
+seeds	323	154	169	133.2
none	322	153	169	137.2

Figure 6 Left: Effect of basic diversification techniques on performance in our 768-core search-only setup with single-core preprocessing. Right: Corresponding number of solved instances (#), of which satisfiable (+) and unsatisfiable (−) instances, and PAR2 scores.

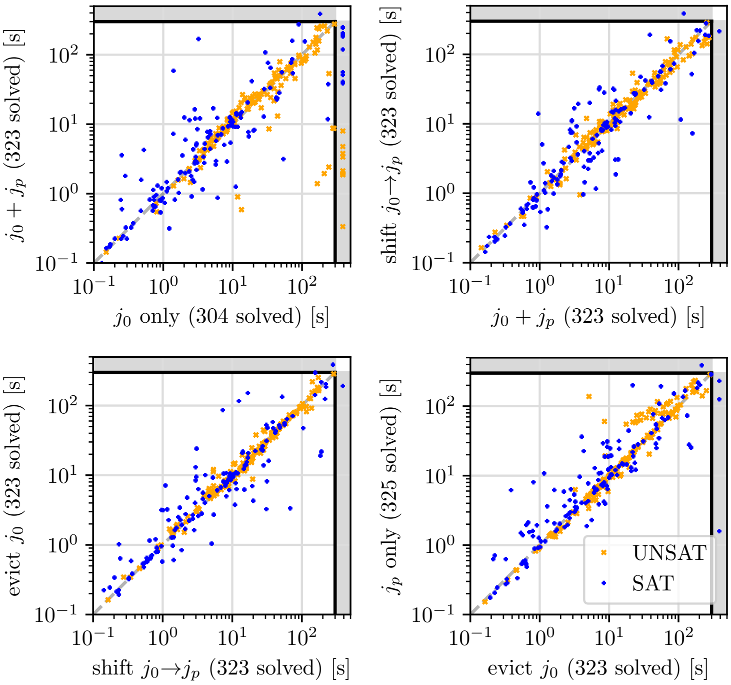


Figure 7 768-core performance comparisons. Top left: Only running a search-only task j_0 vs. adding a preprocessed task j_p via sequential preprocessing ("$j_0 + j_p$"). Top right: "$j_0 + j_p$" vs. shifting resources gradually from j_0 to j_p ("shift $j_0 \to j_p$"). Bottom left: "shift $j_0 \to j_p$" vs. evicting j_0 immediately once j_p enters. Bottom right: evicting j_0 vs. never launching j_0 in the first place.

resources to a single task. Next, we evict j_0 immediately as soon as j_p enters, which results in mildly worse performance (2.4% mean slowdown), indicating that j_o occasionally solves an instance after the arrival of j_p. Lastly, omitting j_o entirely, thus deferring distributed solving until a preprocessing result is available, clearly performs worse (14.7% slowdown).

We continue our experiments with the "shift" strategy since it assumedly results in the best scalability in the long term (i.e., eventually allotting all resources to a single task) while not immediately discarding the progress made by the plain searcher task.

We now assess the scaling of our streamlined setup. For these measurements, we use the remaining 400 instances from the set of 2023–2024 benchmark instances that we did not use for the earlier experiments. We use 1–64 nodes (48–3072 cores), doubling the computational resources at each step. As points of reference, we use two state-of-the-art configurations of MallobSat: Only running Kissat ("K"), and running Kissat, CaDiCaL, and Lingeling alternatingly ("KCL"). The first configuration is the (prior) single-solver configuration with assumedly best general-purpose performance whereas KCL corresponds to the version of MallobSat submitted to the 2024 SAT Competition Cloud Track (which it won). Notably, the inclusion of Lingeling [7, 9] is known to noticeably boost performance on the 2023–2024 benchmark sets due to some peculiar instances which can be solved immediately by Lingeling's cardinality constraint reasoning [10] (see also [53]). This circumstance provides KCL with a distinct advantage over Kissat-only configurations in terms of solving performance metrics. Due to native diversification, roughly 13% of solver threads in the K and KCL setups run without or with significantly reduced pre– and inprocessing.

Fig. 8 shows the scaling behavior of the three considered configurations, with precise statistics given in Tab. 1. In terms of baselines, we included the performance of sequential

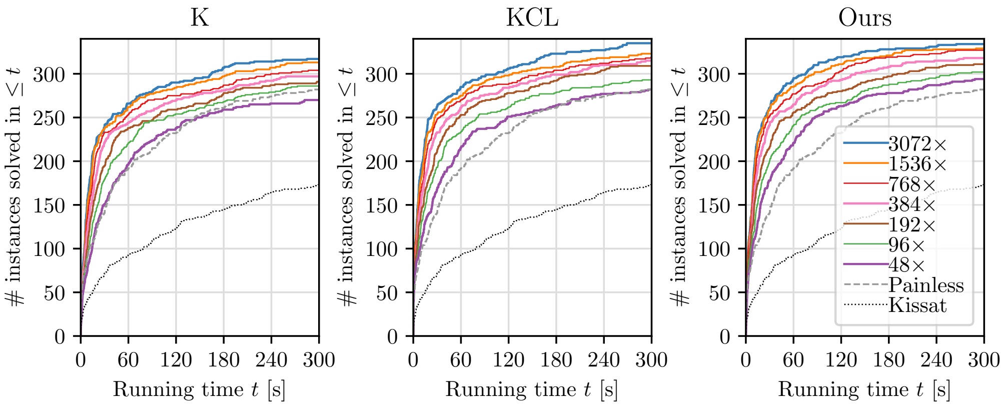


Figure 8 Strong scaling of MallobSat's baseline K and KCL configurations (left, center) and our new setup (right) in terms of absolute performance from 1 to 64 nodes (48–3072 cores). We also show running times of sequential Kissat and 48-core PL-PRS-BVA-KISSAT ("Painless").

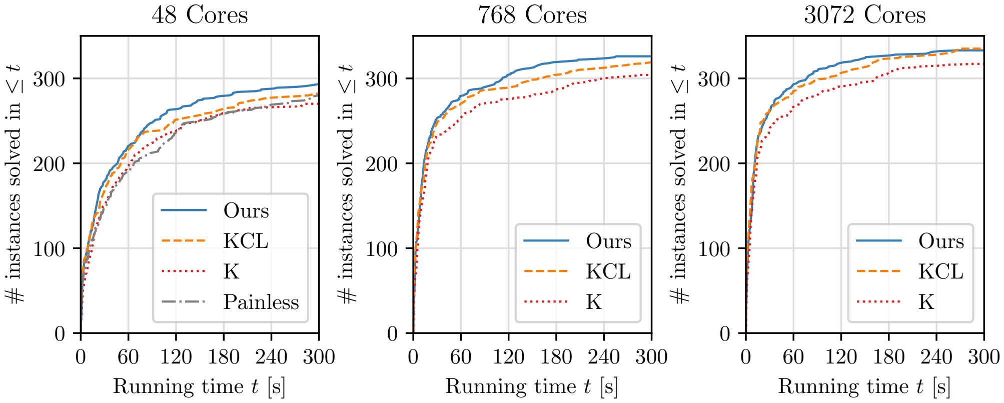


Figure 9 Direct comparison of the three tested MallobSat configurations at 1, 8, and 64 nodes. At one node (48 cores), the run of PL-PRS-BVA-KISSAT ("Painless") is shown as well.

Kissat (winner of the SAT Competition 2024 Main track) and of PL-PRS-BVA-KISSAT [49] ("painless-2" in SAT Competition 2024, winner of the Parallel track) executed at 48 cores. All configurations of MallobSat show consistent performance improvements up to the largest scale considered (3072 cores). The out-of-the-box Kissat-only configuration results in the worst performance. Notably, it features neither cardinality constraint reasoning (as in Lingeling) nor any reasoning beyond general resolution (as in bounded variable addition, which is part of our preprocessing). The diverse Kissat–CaDiCaL–Lingeling portfolio improves performance substantially (25.8 PAR-2 score points at 3072 cores). Our approach, in turn, outperforms all other approaches, achieving the lowest PAR-2 scores at all scales.

The performance margin to KCL is the smallest at 3072 cores, and mean speedups are consistently the highest for KCL. This raises the question whether we are observing a distinct scalability limit of our new approach. A direct comparison of the runs at 3072 cores (Fig. 10) suggests that this is not the case: Compared to both K and KCL, our approach clearly tends to perform better as running times increase. A distinct advantage of the KCL setup is that it solves a number of unsatisfiable instances in the sub-second range which are challenging or even infeasible to solve with Kissat only. At 64 cores, we identified 17 such instances; all of them are in fact solved by a Lingeling solver thread. The concerned instances are eleven

	K						KCL						Ours					
cores	#	+	−	PAR	S_g	S_t	#	+	−	PAR	S_g	S_t	#	+	−	PAR	S_g	S_t
48	270	132	138	226.4	7.8	17.9	282	129	153	208.6	10.8	18.2	294	135	159	191.2	8.4	20.0
96	286	140	146	202.2	11.1	29.3	293	132	161	188.7	14.3	24.4	302	137	165	177.0	11.5	30.4
192	292	143	149	191.3	13.4	36.9	309	140	169	165.3	17.7	35.7	311	142	169	161.0	14.7	40.2
384	297	144	153	180.7	17.1	52.7	315	140	175	155.1	21.6	50.7	318	145	173	147.6	17.2	63.5
768	305	147	158	169.4	19.0	57.3	319	143	176	146.9	26.0	61.4	327	149	178	133.6	21.7	82.7
1536	314	151	163	155.8	21.1	67.0	323	144	179	139.0	30.2	72.1	329	149	180	127.4	24.6	98.3
3072	318	152	166	148.5	23.3	75.6	335	154	181	122.7	34.4	88.3	334	151	183	119.4	26.7	111.5

Table 1 Performance of three MallobSat configurations at 48–3072 cores, showing the number of solved instances (#), thereof satisfiable (+) and unsatisfiable (−) instances, Penalized Average Runtime which rates each timeout as solved in twice the time limit, i.e., 600 s (PAR-2), geometric mean speedup vs. sequential Kissat over all commonly solved instances (S_g), and corresponding total speedup, i.e., summing up the sequential solver's running times and then dividing by the sum of the parallel solver's running times (S_t). At each scale, the best values of each metric are highlighted.

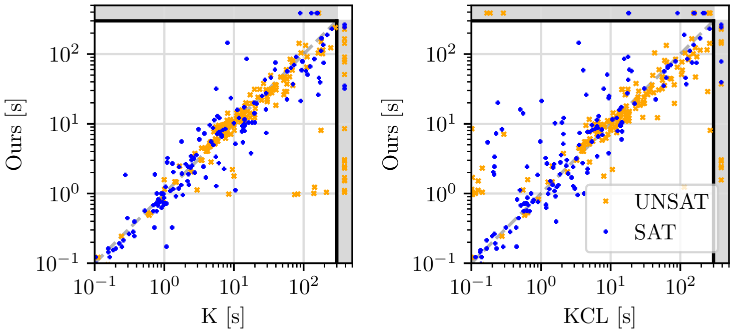


Figure 10 Direct performance comparison of our approach vs. the two baseline configurations K (left) and KCL (right) at 3072 cores.

register allocation graph coloring problems [28], four Tseitin formulas [19], and two Pidgeon hole problems [8]. Our approach compensates for this disadvantage with generally better scalability, solving other instances which KCL fails to solve; however, at the largest scale, our approach runs out of such instances while KCL manages to close the gap.

A second advantage of KCL is the dedicated Stochastic Local Search solver YalSAT [9] in Lingeling's portfolio: KCL outperforms our new setup by a factor of ≥ 5 for 21 satisfiable instances. 18 of them were solved by a Lingeling instance, and 14 of them specifically by YalSAT. This includes seven instances surrounding Folkman graphs [32] and six set covering problems [62]. On the flipside, our new approach performs remarkably well on many benchmarks adjacent to real-world applications (see also appendix Tab. 3), such as bitvectors (4 instances, 67% mean speedup over KCL), verifying floating-point commutativity (6×, 66%), hashtable safety (7×, 48%), software (5×, 38%) and hardware verification (7×, 38%), school timetabling (9×, 26%), and hash function properties (18×, 23%). We attribute this success to our streamlined solver design (cf. Fig. 2); while KCL is missing BVA, this technique is highly unlikely to cause such scaling improvements on such kinds of instances [29].

Fig. 11 provides insights into MallobSat's *weak scaling* for the "K" baseline and our new setup (cf. [55]). Intuitively, the further right a curve goes, the more difficult are the instances that we consider for computing the mean speedup. We measure "difficulty" in terms of the sequential baseline's running time. For example, our approach at 3072 cores reaches a speedup of 315.1 for instances that took sequential Kissat at least 1 h to solve, which corresponds to a resource efficiency of 315.1/3072 = 10.3%. Under the same circumstances,

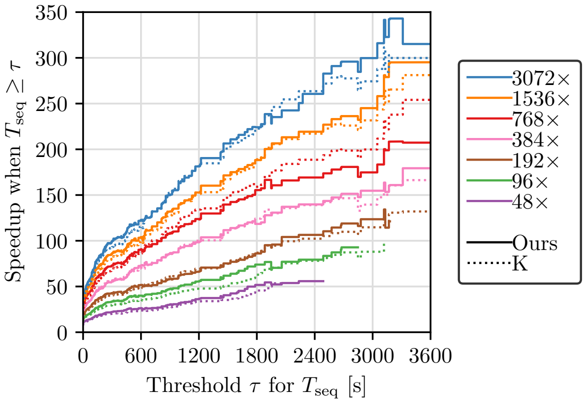


Figure 11 *Weak Scaling* of MallobSat. The color denotes the scale of solving (1–64 nodes); continuous lines denote our approach whereas dotted lines denote the "K" baseline configuration. Point (x, y) on a curve indicates that the corresponding run achieved a geometric mean speedup of y on the instances which took sequential Kissat at least time x to solve. A curve ends prematurely if this amounts to less than 20 considered instances.

the baseline configuration "K" achieves a speedup of 299.8 (resource efficiency 9.8%). The KCL portfolio (not shown[2]) results in the best weak scaling, with a speedup of 451.5 and an according resource efficiency of 14.7%. This discrepancy can be explained by the fact that KCL performs noticeably better than our new approach in the first 10–20 s of solving, for reasons discussed above (i.e., due to Lingeling and YalSAT). The corresponding short running times cause relatively high speedups. In terms of *total* speedups (Table 1), which emphasize longer running times, our approach consistently achieves the best results.

7 Discussion

In the following, we provide a brief discussion of the obtained insights and results.

Our work results in a slimmed down and yet highly competitive solver system. Let us attempt to provide a coarse quantitative estimate of the achieved reduction by considering the involved lines of code. We counted roughly 25k effective lines of code (ELOC; disregarding blank lines and comments) in Lingeling's main source file `lglib.c`, 30k ELOC in CaDiCaL's source files (`*.cpp`) and also 30k ELOC in Kissat's source files (`*.c`). We estimate that roughly 20k ELOC of the Mallob system are in use. Our changes to Mallob amount to few hundred lines of code. Going by these measures, the code to be considered decreases from 105k ELOC to only 50k ELOC via our new setup. Moreover, the code executed by the parallel SAT solving threads decreases by roughly 13k ELOC (when excluding Kissat's source files related to pre–/inprocessing). While these measures are not fully reliable (as each solver backend already features some "dead code" / unused procedures), we believe it is safe to assume that our changes to MallobSat cut the lines of executed code roughly in half and, in particular, drastically reduce the SAT solving program executed in parallel.

Comparing our setup to MallobSat's KCL portfolio, we noticed some remaining short-

[2] The weak scaling as we compute it heavily depends on the sets of instances a certain approach is able to solve. The weak scaling curves of the KCL configuration are therefore difficult to compare to the other configurations due to the different kinds of instances solved (mostly by Lingeling/YalSAT). We do provide a weak scaling plot of the KCL configuration in the appendix (Fig. 14).

comings of our somewhat radical approach, mostly due to foregoing dedicated Stochastic Local Search (SLS) threads and due to missing special techniques for certain hard inputs. This motivates us to (a) reintegrate SLS in our approach, which presents only a minor increase in complexity (especially in terms of information flow), and (b) to integrate advanced preprocessing techniques (as in ref. [60]) that are required to conquer the kinds of instances which our current setup still struggles to solve. Specifically, our new approach to preprocessing can be adapted well to preprocessors that exploit advanced proof systems, such as extended resolution (which our preprocessing already features via bounded variable addition) [29], symmetry breaking (e.g., BreakID [18, 23]), or propagation redundancy (e.g., PReLearn [46]). Following our evaluation, we have made first simple steps in this direction by adding Lingeling's preprocessing and 5% YalSAT threads to our setup; we observed significantly improved performance at a shared-memory level (see appendix Fig. 15).

On a related note, transferring our findings to a *CaDiCaL-only setup* within MallobSat will allow to exploit MallobSat's proof production and checking capabilities (which, as of now, rely on CaDiCaL's LRUP proof output [45]). While it appears challenging to generalize distributed proof production [42] and real-time checking [53] to proof systems beyond resolution, our pragmatic approach to preprocessing allows to "outsource" advanced reasoning to a strict, sequential prefix of the distributed solver's reasoning, which could greatly facilitate combining scalable proof technology with modern proof systems.

Lastly, and perhaps most significantly, the presence of a *competitive* "pure" clause-sharing solver may have notable consequences for future research in parallel and distributed SAT solving. Rather than assuming a gathering of diverse and opaque reasoners who periodically exchange insights [55], we may now rather accurately consider the central building block of modern parallel SAT solving to be a *uniform parallel CDCL* procedure, parallelized by clause sharing. This more precise and transparent algorithmic model lends itself for establishing new theory, heuristics, and algorithms surrounding parallel SAT solving.

8 Conclusion

In the work at hand, we investigate modern distributed clause-sharing SAT solver design. In particular, we empirically confirm earlier suspicions [20] that such distributed solvers are being held back by performing non-scalable tasks fully redundantly, such as pre– and inprocessing techniques. We also show conclusively that LBD values have no measurable significance for MallobSat's distributed clause sharing. At the same time, we show that sequential preprocessing employed next to search-only solving can still greatly benefit distributed solving. Our practical result is a clean distributed clause-sharing solver whose solver threads are "pure" CDCL solving engines with very lightweight and uniform diversification. This system is able to achieve competitive performance, even compared to a mixed portfolio consisting of many different search strategies and advanced reasoning techniques.

In terms of future work, we intend to explore scalable algorithms for selected pre– and/or inprocessing tasks, such as vivification, variable elimination, or SAT sweeping. Distributed computing may allow for much more powerful simplification procedures while the parallel search logic could remain separate, uniform, and fully scalable. Orthogonally, we intend to retrofit our new setup from this work to MallobSat's CaDiCaL backend, which (as of yet) is MallobSat's only modern engine for incremental SAT solving [51] and for proofs of unsatisfiability [42, 53]. Lastly, it may be promising to allow sharing (certain) clauses between an original task j_o and a preprocessed task j_p, since this may allow to transfer made progress from j_o to j_p following preprocessing.

References

1 Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 200–213. Springer, 2012. doi:10.1007/978-3-642-31612-8_16.

2 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 399–404, 2009.

3 Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 197–205. Springer, 2014. doi:10.1007/978-3-319-09284-3_15.

4 Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio SAT solver. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 156–172. Springer, 2015. doi:10.1007/978-3-319-24318-4_12.

5 Tomáš Balyo and Carsten Sinz. Parallel satisfiability. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*. Springer, 2018. doi:10.1007/978-3-319-63516-3_1.

6 Armin Biere. Lingeling and friends entering the SAT challenge 2012. In *Proc. SAT Challenge*, pages 33–34, 2012.

7 Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In *SAT Competition 2013: Solver and Benchmark Descriptions*, page 1, 2013.

8 Armin Biere. Two pidgeons per hole problem. In *SAT Competition 2013: Solver and Benchmark Descriptions*, page 103, 2013.

9 Armin Biere. Yet another local search solver and lingeling and friends entering the SAT competition 2014. In *SAT Competition 2014: Solver and Benchmark Descriptions*, page 65, 2014.

10 Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 285–301. Springer, 2014. doi:10.1007/978-3-319-09284-3_22.

11 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In *International Conference on Computer Aided Verification*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

12 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT competition 2024. *International SAT Competition 2024: Solver and Benchmark Descriptions*, page 8, 2024.

13 Armin Biere, Katalin Fazekas, Mathias Fleury, and Nils Froleyks. Clausal congruence closure. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 6:1–6:25. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.6.

14 Armin Biere, Katalin Fazekas, Mathias Fleury, and Nils Froleyks. Clausal equivalence sweeping. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 236–241. TU Wien Academic Press, 2024. doi:10.34727/2024/isbn.978-3-85448-065-5_29.

15 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *SAT Competition 2020: Solver and Benchmark Descriptions*, page 50, 2020.

16 Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2 edition, 2021. doi:10.3233/faia336.

17 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In *Handbook of Satisfiability*, pages 391–435. IOS Press, 2021. doi:10.3233/faia200987.

18 Bart Bogaerts, Jakob Nordström, Andy Oertel, and Cagrı Uluç Yıldırımoglu. BreakID-kissat in SAT competition 2023 (system description). *SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions*, page 25, 2023.

19 Bart Bogaerts, Jakob Nordström, Andy Oertel, and Cagrı Uluç Yıldırımoglu. Crafted benchmark formulas requiring symmetry breaking and/or parity reasoning. In *SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions*, page 67, 2023.

20 Jannick Borowitz, Dominik Schreiber, and Peter Sanders. An empirical study on learned clause overlaps in distributed SAT solving. In *Pragmatics of SAT (PoS)*, 2024. `doi:https://satres.kikit.kit.edu/papers/2024-pos-empirical.pdf`.

21 Zhihan Chen, Xindi Zhang, Yuhang Qian, and Shaowei Cai. PRS: A new parallel/distributed framework for SAT. *SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, page 39, 2023.

22 Byron Cook. Automated reasoning's scientific frontiers. `https://www.amazon.science/blog/automated-reasonings-scientific-frontiers`, 2021. Amazon Science.

23 Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 104–122. Springer, 2016. `doi:10.1007/978-3-319-40970-2_8`.

24 Thorsten Ehlers and Dirk Nowotka. Tuning parallel SAT solvers. In *Pragmatics of SAT*, volume 59, pages 127–143, 2019. `doi:10.29007/z3g2`.

25 Thorsten Ehlers, Dirk Nowotka, and Philipp Sieweck. Communication in massively-parallel SAT solving. In *Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, pages 709–716. IEEE, 2014. `doi:10.1109/ictai.2014.111`.

26 Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition 2020. *Artificial Intelligence*, 301:103572, 2021. `doi:10.1016/j.artint.2021.103572`.

27 Kilian Gebhardt and Norbert Manthey. Parallel variable elimination on CNF formulas. In *Annual Conference on Artificial Intelligence*, pages 61–73. Springer, 2013. `doi:10.1007/978-3-642-40942-4_6`.

28 Andrew Haberlandt and Harrison Green. Python function register allocation benchmarks. In *SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, page 56, 2023.

29 Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective auxiliary variables via structured reencoding. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 11:1–11:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPIcs.SAT.2023.11`.

30 Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *JSAT*, 6(4):245–262, 2010. `doi:10.3233/sat190070`.

31 Maximilian Heisinger, Mathias Fleury, and Armin Biere. Distributed cube and conquer with paracooba. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 114–122. Springer, 2020. `doi:10.1007/978-3-030-51825-7_9`.

32 Marijn Heule. Avoiding monochromatic triangles and L-shapes. In *SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*, pages 41–42, 2024.

33 Marijn J. H. Heule, Oliver Kullmann, and Victor Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 228–245. Springer, 2016. `doi:10.1007/978-3-319-40970-2_15`.

34 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65. Springer, 2011. `doi:10.1007/978-3-642-34188-5_8`.

35 Yoichiro Iida, Tomohiro Sonobe, and Mary Inaba. Parallel Clause Sharing Strategy Based on Graph Structure of SAT Problem. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 17:1–17:18, 2024. `doi:10.4230/LIPIcs.SAT.2024.17`.

36 Yoichiro Iida, Tomohiro Sonobe, and Mary Inaba. Wance: Learnt clause evaluation method for SAT solver using graph structure. In *International Conference on Learning and Intelligent Optimization*, pages 190–204. Springer, 2024. `doi:10.1007/978-3-031-75623-8_15`.

37 Markus Iser and Christoph Jabs. Global Benchmark Database. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 18:1–18:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.18.

38 Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. PaInleSS: a framework for parallel SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 233–250. Springer, 2017. doi:10.1007/978-3-319-66263-3_15.

39 Mao Luo, Chu-Min Li, Fan Xiao, Felip Manya, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *Proc. IJCAI*, pages 703–711, 2017. doi:10.24963/ijcai.2017/98.

40 Norbert Manthey, Marijn JH Heule, and Armin Biere. Automated reencoding of Boolean formulas. In *Haifa Verification Conference*, pages 102–117. Springer, 2012. doi:10.1007/978-3-642-39611-3_14.

41 João Marques-Silva, Inês Lynce, and Sharad Malik. CDCL SAT solving. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2021. doi:10.3233/faia200987.

42 Dawn Michaelson, Dominik Schreiber, Marijn J. H. Heule, Benjamin Kiesl-Reiter, and Michael W. Whalen. Unsatisfiability proofs for distributed clause-sharing SAT solvers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 348–366. Springer, 2023. doi:10.1007/978-3-031-30823-9_18.

43 Chanseok Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 307–323. Springer, 2015. doi:10.1007/978-3-319-24318-4_23.

44 Muhammad Osama, Anton Wijs, and Armin Biere. Certified SAT solving with GPU accelerated inprocessing. *Formal Methods in System Design*, 62:79–118, 2023. doi:10.1007/s10703-023-00432-z.

45 Florian Pollitt, Mathias Fleury, and Armin Biere. Faster lrat checking than solving with CaDiCaL. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.21.

46 Joseph E. Reeves and Randal E. Bryant. Preprocessors PReLearn and ReEncode entering the SAT Competition 2023. In *SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions*, page 23, 2023.

47 Peter Sanders and Dominik Schreiber. Decentralized online scheduling of malleable NP-hard jobs. In *Euro-Par 2022: Parallel Processing*, pages 119–135. Springer, 2022. doi:10.1007/978-3-031-12597-3_8.

48 Mazigh Saoudi, Souheib Baarir, Julien Sopena, Thibault Lejemble, and Sabrine Saouli. Painless in SAT Competition 2024 with PL-PRS-BVA-KISSAT and PL-PRS-MPI. *Proc. SAT Competition 2024*, page 24.

49 Mazigh Saoudi, Thibault Lejemble, Souheib Baarir, and Julien Sopena. PL-PRS-BVA-KISSAT in SAT Competition 2024. In *Pragmatics of SAT*, 2024. URL: https://dl.lre.epita.fr/papers/saoudi.24.pos.pdf.

50 Dominik Schreiber. Mallob{32,64,1600} in the SAT competition 2023. In *SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions*, pages 46–47, 2023.

51 Dominik Schreiber. *Scalable SAT Solving and its Application*. PhD thesis, Karlsruhe Institute of Technology, 2023. https://doi.org/10.5445/IR/1000165224.

52 Dominik Schreiber. MallobSat and MallobSat-ImpCheck in the SAT Competition 2024. In *SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*, pages 21–22, 2024. URL: http://hdl.handle.net/10138/584822.

53 Dominik Schreiber. Trusted scalable SAT solving with on-the-fly LRAT checking. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 25:1–25:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.25.

54 Dominik Schreiber and Peter Sanders. Scalable SAT solving in the cloud. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 518–534. Springer, 2021. doi:10.1007/978-3-030-80223-3_35.

55 Dominik Schreiber and Peter Sanders. MallobSat: Scalable SAT solving by clause sharing. *Journal of Artificial Intelligence Research (JAIR)*, 2024. Presented at Pragmatics of SAT (PoS) 2024. doi:10.1613/jair.1.15827.

56 Sven Schulz and Wolfgang Blochinger. Cooperate and compete! A hybrid solving strategy for task-parallel SAT solving on peer-to-peer desktop grids. In *Int. Conf. HPC & Simulation*, pages 314–323. IEEE, 2010.

57 Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. PaSAT – parallel SAT-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001. doi:10.1016/s1571-0653(04)00323-3.

58 Vincent Vallade, Julien Sopena, and Souheib Baarir. Enhancing state-of-the-art parallel SAT solvers through optimized sharing policies. In *Pragmatics of SAT*, pages 35–45, 2023. URL: https://ceur-ws.org/Vol-3545/paper3.pdf.

59 Siert Wieringa and Keijo Heljanko. Concurrent clause strengthening. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 116–132. Springer, 2013. doi:10.1007/978-3-642-39071-5_10.

60 Xindi Zhang, Zhihan Chen, and Shaowei Cai. ParKissat: Random shuffle based and pre-processing extended parallel solvers with clause sharing. In *SAT Competition 2022: Benchmark and Solver Descriptions*, page 51, 2022.

61 Xindi Zhang, Zhihan Chen, and Shaowei Cai. PRS: A new parallel/distributed framework for SAT. In *Proc. SAT Competition*, pages 39–40, 2023. URL: https://helda.helsinki.fi/bitstream/10138/563824/1/sc2023_proceedings.pdf#page=40.

62 Jiongzhi Zheng, Mingming Jin, Kun He, Zhuo Chen, and Jinghui Xue. SAT instances based on the set covering problem with conflict. In *SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, page 80, 2023.

A Supplementary Results and Figures

Family	#	Avg. time	Speedup
miter-unsat	14	34.86	0.30
profitable-robust-production-sat	5	7.19	0.35
heule-nol-sat	7	18.50	0.43
social-golfer-sat	6	8.23	0.52
grs-fp-comm-unsat	8	75.60	0.62
scheduling-unsat	9	26.66	0.80
software-verification-unsat	10	56.29	0.97
cryptography-simon-sat	8	0.13	0.98
random-circuits-sat	5	24.89	1.01
hamiltonian-unsat	11	7.11	1.05
cryptography-ascon-unsat	6	8.58	1.05
argumentation-unsat	18	7.45	1.06
satcoin-unsat	5	16.36	1.28
brent-equations-sat	7	0.96	1.29
hamiltonian-sat	12	2.14	1.35
maxsat-optimum-sat	5	5.78	1.36
scheduling-sat	9	40.00	1.59
heule-folkman-sat	5	81.82	1.82
school-timetabling-sat	8	21.01	2.00
cryptography-sat	6	12.77	2.05
set-covering-sat	14	11.07	2.09
mutilated-chessboard-unsat	6	31.48	2.27

Table 2 Geometric mean speedup of "search-only" configuration over default Kissat-only configuration, at 768 cores, separated by GBD benchmark family and result, showing all such categories where the number of considered instances ("#") is at least five. "Avg. time" denotes the average running time of the default configuration on the respective instances.

Family	#	Avg. time	Speedup
set-covering-sat	6	0.21	0.04
heule-folkman-sat	6	5.82	0.08
register-allocation-unsat	11	0.12	0.10
random-circuits-sat	10	23.27	0.60
rbsat-sat	5	13.93	0.68
maxsat-optimum-unsat	5	19.11	0.70
hamiltonian-unsat	9	5.19	0.78
brent-equations-sat	9	0.63	0.81
argumentation-unsat	16	12.00	0.84
profitable-robust-production-sat	8	70.91	0.84
cryptography-ascon-unsat	10	8.12	0.88
quantum-kochen-specker-unsat	6	9.92	0.90
hamiltonian-sat	8	1.46	1.03
scheduling-unsat	6	28.64	1.04
scheduling-sat	17	24.12	1.08
satcoin-unsat	10	13.73	1.16
cryptography-sat	8	24.14	1.18
school-timetabling-sat	9	11.98	1.25
miter-sat	9	87.61	1.25
miter-unsat	23	29.96	1.26
coloring-unsat	5	24.48	1.26
software-verification-unsat	5	34.61	1.38
hashtable-safety-unsat	7	100.41	1.48
cryptography-ascon-sat	8	3.00	1.57
grs-fp-comm-unsat	6	65.93	1.63

Table 3 Geometric mean speedup of our new setup over KCL, at 3072 cores, split as in Table 2. "Avg. time" denotes the average running time of KCL on the respective instances.

Procedure	Kissat	768×d	768×s-o
backbone	0.12	0.18	—
congruence	0.46	1.85	—
eliminate	1.01	1.33	—
extend	0.00	0.00	—
factor	0.55	—	—
fastel	0.30	2.96	—
focused	43.39	44.65	49.97
lucky	0.08	1.34	1.59
parse	0.15	—	—
preprocess	0.83	4.77	—
probe	11.15	13.52	—
reduce	1.37	2.68	3.24
search	86.32	79.15	95.41
simplify	12.61	13.86	1.77
stable	42.93	34.50	45.44
substitute	0.66	1.39	—
subsume	0.39	0.36	—
sweep	2.18	1.54	—
transitive	0.17	0.20	—
vivify	7.03	8.35	—
walking	0.96	0.65	1.77

Table 4 Percentages of total ("CPU") time spent in different procedures of Kissat's SAT solving, for sequential Kissat, 768-core default setup, and 768-core search-only setup. Note that some categories subsume each another (e.g., focused and stable are disjoint sub-categories of search).

Here we report two additional parameter studies. They were done with the setting denoted "Ours", with one difference: original LBDs were still used instead of the Deactivated-LBDs setting. Each run consisted of 396 instances with 300 s timeout on 8 nodes (384 cores).

The first study explores clause database reduction. Kissat offers two parameters, `reduce-low` (default 500) and `reduce-high` (default 900) which control the aggressiveness of database reductions. The default parameters correspond to reductions by 50% early in the run and by up to 90% later in the run. We tested four parameter settings, described briefly.

Reduce setting	#	+	−	PAR2
Default	323	153	170	137.9
Point	323	154	169	139.2
Gaussian	321	151	170	140.8
Range	318	149	169	144.2

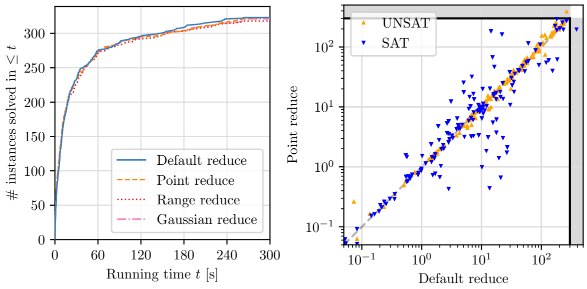


Figure 12 Effects of diversifying Kissat's `reduce-low` and `reduce-high` parameters.

Default reduce: Left the reduce parameters at their default. **Point reduce**: A value $r \in [0, ..., 1000]$ is uniformly sampled per solver and both `reduce-low` and `reduce-high` are set to it. This creates some extreme solvers that keep all clauses forever ($r = 0$) and others that delete every clause almost immediately ($r = 1000$). **Range reduce**: A value $r \in [-200, 1200]$ is uniformly sampled per thread; then we set `reduce-low=max(0,r-200)` and `reduce-high=min(1000,r+200)`. Intuitively this slides the default interval [500,900] randomly to higher or lower values and leaves per solver the flexiblity of shifting from low to high reductions. **Gaussian reduce**: A value r is sampled from a Gaussian distribution with mean 700 and standard deviation 150, then r is clipped to be within [300,980] and both `reduce-low` and `reduce-high` are set to it. This sampling specifically avoids the extremes from the other two settings.

The results of the four reduce settings are shown in Fig. 12. Default reduce performs overall best, whereas Point reduce performs strongly on some SAT instances, which might be due to some aggressive solvers being allowed to eliminate almost all learned clauses.

The second study focuses on the decay of (E)VSIDS scores controlled by the `decay` parameter. Its default value is 50 (per mille), corresponding to an update of variables activity scores to 95% of their former value. Higher `decay` results in more aggressive updates.

Decay setting	#	+	−	PAR2
Uniform [1,50]	324	154	170	135.5
Default [50]	323	153	170	137.9
Uniform [1,200]	322	152	170	140.1
Uniform [50,200]	321	151	170	141.1

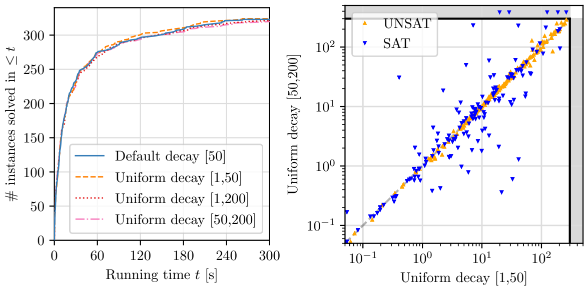


Figure 13 Effects of diversifying Kissat's `decay` parameter.

Kissat accepts values in the range of [1,...,200]. We explore this full range by testing three settings: **Uniform[1,50]**, **Uniform[1,200]** and **Uniform[50,200]**. In each setting every solver thread samples its `decay` value uniformly from the given interval. The third setting is thus much more eager than the default, while the first is more conservative.

The results of the different decay settings are shown in Fig. 13. Regarding PAR2 scores, the conservative updating with `decay` at or below 50 performs better than the more aggressive choices. However, similar to the database reductions, the more eager approaches perform better on some SAT instances, observable in the direct comparison plot.

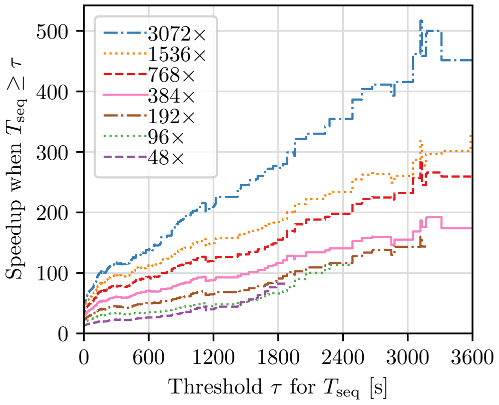
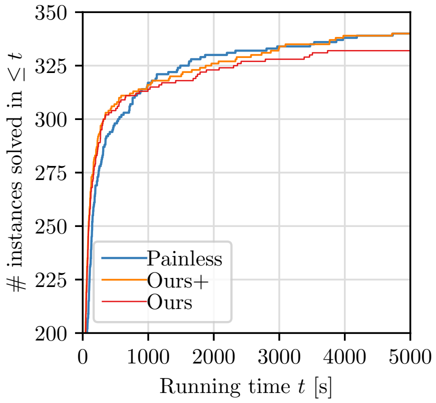


Figure 14 Weak Scaling of KCL configuration (as in Fig. 11).



	#	+	−	PAR2
Ours	332	148	184	1857.0
Ours+	**340**	154	**186**	**1705.3**
Painless	**340**	**159**	181	1721.4

Figure 15 Performance of our approach from the paper ("Ours"), an enhanced version ("Ours+") with additional Lingeling-based preprocessing and each 20th thread running YalSAT instead of CDCL, and the state-of-the-art shared-memory solver PL-PRS-BVA-KISSAT, at a single node (48 cores) and, notably, for up to 5000 s of running time as in the SAT Competition Parallel tracks.