





Unsatisfiability Proofs for Distributed Clause-Sharing SAT Solvers

Dawn Michaelson² ✉ , Dominik Schreiber³ ✉ , Marijn J.H. Heule^{1,4} ,
Benjamin Kiesl-Reiter¹ , and Michael W. Whalen^{1,2} 

¹ Amazon Web Services

² University of Minnesota

³ Karlsruhe Institute of Technology

⁴ Carnegie Mellon University

Abstract. Distributed clause-sharing SAT solvers can solve problems up to one hundred times faster than sequential SAT solvers by sharing derived information among multiple sequential solvers working on the same problem. Unlike sequential solvers, however, distributed solvers have not been able to produce proofs of unsatisfiability in a scalable manner, which has limited their use in critical applications. In this paper, we present a method to produce unsatisfiability proofs for distributed SAT solvers by combining the partial proofs produced by each sequential solver into a single, linear proof. Our approach is more scalable and general than previous explorations for parallel clause-sharing solvers, allowing use on distributed solvers without shared memory. We propose a simple sequential algorithm as well as a fully distributed algorithm for proof composition. Our empirical evaluation shows that for large-scale distributed solvers (100 nodes of 16 cores each), our distributed approach allows reliable proof composition and checking with reasonable overhead. We analyze the overhead and discuss how and where future efforts may further improve performance.

Keywords: SAT solving · proofs · distributed computing.

1 Introduction

SAT solvers are general-purpose tools for solving complex computational problems. By encoding domain problems into propositional logic, users have successfully applied SAT solvers in various fields such as formal verification [31], automated planning [25], and mathematics [8, 16]. The list of applications has grown significantly over the years, mainly because algorithmic improvements have led to orders of magnitude improvement in the performance of the best sequential solvers (see, e.g., [21] for a comparison).

Despite all this progress, there are still many problems that cannot be solved quickly with even the best sequential solvers, pushing researchers to explore ways of parallelizing SAT solving. One approach that has worked well for specific problem instances is *Cube-and-Conquer* [17, 18], which can achieve near-linear

speedups for thousands of cores but requires domain knowledge about how effectively to split a problem into subproblems. An alternative approach that does not require such knowledge is *clause-sharing portfolio solving*, which has recently led to solvers [12, 28] achieving impressive speedups (10x–100x on a 100x16 core cluster) over the best sequential solvers across broad sets of benchmarks.⁵

Although distributed solvers are demonstrably the most powerful tools for solving hard SAT problems, there is an important caveat: unlike sequential solvers, current distributed clause-sharing solvers cannot produce proofs of unsatisfiability. While there has been foundational work in producing proofs for shared-memory clause-sharing SAT solvers [14], existing approaches are neither scalable nor general enough for large-scale distributed solvers. This is not just a theoretical problem—for four problems in the 2020 and 2021 SAT competitions, distributed solvers produced incorrect answers that were not discovered until the 2022 competition because they could not be independently verified.⁶

In this paper, we deal with this issue and present the first scalable approach for generating proofs for distributed SAT solvers. To construct proofs, we maintain *provenance* information about shared clauses in order to track how they are used in the global solving process, and we use the recently-developed LRAT proof format [9] to track dependencies among partial proofs produced by solver instances. By exploiting these dependencies, we are then able to reconstruct a single linear proof from all the partial proofs produced by the sequential solvers. We first present a simple sequential algorithm for proof reconstruction before devising a parallel algorithm that can even be implemented in a distributed way. Both algorithms produce independently-verifiable proofs in the LRAT format. We demonstrate our approaches using an LRAT-producing version of the sequential SAT solver CaDiCaL [5] to turn it into a clause-sharing solver, and then modify the distributed solver Mallob [28] to orchestrate a portfolio of such CaDiCaL instances while tracking the IDs of all shared clauses.

We conduct an evaluation of our approaches from the perspective of efficiency, benchmarking the performance of our clause-sharing portfolio solver against the winners of the cloud track, parallel track, and sequential track from the SAT Competition 2022. Adding proof support introduces several kinds of overhead for clause-sharing portfolios in terms of solving, proof reconstruction, and proof checking, which we examine in detail. We show that even with this overhead, distributed solving and proving is much faster than the best sequential approaches. We also demonstrate that our approach dramatically outperforms previous work on proof production for clause-sharing portfolios [14]. We argue that much of the overhead of our current setup can be compensated, among other measures, by improving support for LRAT in solver backends. We thus hope that our work provides an impetus for researchers to add LRAT support to other solvers.

Our main contributions are as follows:

⁵ c.f.: the SAT Competition 2022 results:

<https://satcompetition.github.io/2022/downloads/sc2022-detailed-results.zip>

⁶ The incorrectly scored problems were `SAT_MS_sat_nurikabe_p08.pddl_71.cnf`, `randomG-Mix-n18-d05.cnf`, `php12e12.cnf`, and `Cake_9_20.cnf`.

- We present the first effective and scalable approach for proof generation in distributed SAT solving.
- We implement our approach on top of the state-of-the-art solvers CaDiCaL and Mallob.
- We perform a large-scale empirical evaluation analyzing the overhead introduced by proof production as compared to state-of-the-art portfolios.
- We demonstrate that our approach dramatically outperforms previous work in parallel proof production, and that it remains substantially more scalable than the best sequential solvers.

The rest of this paper is structured as follows. In Section 2, we present the background required to understand the rest of our paper and discuss related work. In Section 3, we describe the general problem of producing proofs for distributed SAT solving and a simple algorithm for proof combination. In Section 4, we describe a much more efficient distributed version of our algorithm before discussing implementation details in Section 5. Finally, we present the results of our empirical evaluation in Section 6 and conclude with a summary and an outlook for future work in Section 7.

2 Background and Related Work

The Boolean satisfiability problem (SAT) asks whether a Boolean formula can be satisfied by some assignment of truth values to its variables. An overview can be found in [6]. We consider formulas in *conjunctive normal form* (CNF). As such, a formula F is a conjunction (logical “AND”) of disjunctions (logical “OR”) of literals, where a literal is a Boolean variable or its negation. For example, $(\bar{a} \vee b \vee c) \wedge (b \vee \bar{c}) \wedge (a)$ is a formula with variables a, b, c and three clauses. A *truth assignment* \mathcal{A} maps each variable to a Boolean value (true or false). A formula F is *satisfied* by an assignment \mathcal{A} if F evaluates to true under \mathcal{A} , and F is *satisfiable* if such an assignment exists. Otherwise, F is called *unsatisfiable*.

If a formula F is found to be satisfiable, modern SAT solvers commonly output a truth assignment; users can easily evaluate F under the assignment in linear time to verify that F is indeed satisfiable. In contrast, if a formula turns out unsatisfiable, sequential SAT solvers produce an independently-checkable proof that there exists no assignment that satisfies the formula.

File Formats in Practical SAT Solving. In practical SAT solving, formulas are specified in the DIMACS format. DIMACS files feature a header of the form ‘`p cnf #variables #clauses`’ followed by a list of clauses, one clause per line. For example, the clause $(x_1 \vee \bar{x}_2 \vee x_3)$ is represented as ‘`1 -2 3 0`’. An example formula in DIMACS format is given in Figure 1.

The current standard format for proofs is DRAT [15]. DRAT files are similar to DIMACS files, with each line containing a proof statement that is either an *addition* or a *deletion*. Additions are lines that represent clauses like in the DIMACS format; they identify clauses that were derived (“learned”) by the solver. Each clause addition must preserve satisfiability by adhering to the so-called

DIMACS					DRAT			LRAT											
p	cnf	4	8																
1	-2			0						9		-3	0	5	4	0			
	2		-4	0		1	2	0		10		1	2	0	3	2	0		
1	2		4	0				-1	0	11				-1	0	6	9	0	
-1		-3		0	d			-3	0	11	d							9	0
1		-3		0		2	3	-4	0	12	2	3	-4	0		7	11	0	
-1		3		0		1	2	3	0	13	1	2	3	0		8	12	0	
1		3	-4	0					0	14				0	11	10	1	0	
1		3	4	0															

Fig. 1: DIMACS formula and corresponding proofs in DRAT and LRAT format.

RAT criterion—as the details of RAT are not essential to our paper, we refer the reader to the respective literature for more details [20]. Deletions are lines that start with a ‘d’, followed by a clause; they identify clauses that were deleted by the solver because they were not deemed necessary anymore. Clause deletions can only make a formula “more satisfiable”, meaning that they aren’t required for deriving unsatisfiability, but they drastically speed up proof checking. A valid DRAT proof of unsatisfiability ends with the derivation of the empty clause. As the empty clause is trivially unsatisfiable (and since each proof step preserves satisfiability) the unsatisfiability of the original formula can then be concluded. An example DRAT proof is given in Figure 1.

The more recent LRAT proof format [9] augments each clause-addition step with so-called *hints*, which identify the clauses that were required to derive the current clause. This makes proof checking more efficient, and in fact the usual pipeline for trusted proof checking is to first use an efficient but unverified tool (like DRAT-trim [15]) to transform a DRAT proof into an LRAT proof, and then check the resulting LRAT proof with a formally verified proof checker (c.f., [9, 13, 22, 30]). Figure 1 shows an LRAT proof corresponding to a DRAT proof. Each proof line starts with a clause ID. The numbering starts with 9 because the eight clauses of the original formula are assigned the IDs 1 to 8. Each clause addition first lists the literals of the clause, then a terminating 0, followed by hints (in the form of clause IDs), and finally another 0. For example, clause 9 contains the literal -3 and can be derived from the clauses 4 and 5 of the original formula. Clause deletions just state the clause ID of the clause that is to be deleted, as in the later deletion of clause 9. In our work, we exploit the hints of LRAT to determine dependencies among distributed solvers.

Parallel and Distributed SAT Solving. One way to parallelize SAT solving is to run a portfolio of sequential solvers in parallel and to consider a problem solved as soon as one of the solvers finishes (c.f. [1, 4, 5, 11, 12, 18, 23, 29, 32]). Given that the solvers are sufficiently diverse, portfolio solving is already effective if all of the sequential solvers work independently, but performance and scalability can be boosted significantly by having the solvers share information in the form of learned clauses [4, 12]. This approach is taken by the distributed solver Mallob [28], which won the cloud track of the last three SAT competitions [2, 3, 27]. As opposed to other solvers, Mallob relies on a communication-efficient aggrega-

tion strategy to collect the globally most useful learned clauses and to reliably filter duplicates as well as previously shared clauses [27]. With this strategy, which aims to maximize the density and utility of the communicated data, Mallob scored first place in all four eligible subtracks for unsatisfiable problems at the 2022 SAT Competition.

As we discuss in more detail later, the drawback of clause sharing is that a local proof written by an individual solver may contain clauses whose derivations cannot be justified because they rely on clauses imported from another solver. Previous work focuses on writing DRAT proofs for clause-sharing parallel solvers [14]. In that work, solvers write to the same shared proof as they learn clauses. However, since the clauses are shared, one solver deleting a clause could invalidate a later clause-addition by another solver that is still holding the clause. To handle this, the parallel solver moderates deletion statements, only writing them to the proof once all solvers have deleted a clause, which leads to poor scalability during proof search. In our approach, solvers write proof files fully independently—only when the unsatisfiability of the problem has been determined do we combine all proofs into a single valid proof.

Other recent work includes reconstructing proofs from divide-and-conquer solvers [24] and from a particular shared-memory parallel solver [10] whereas we aim to exploit distributed portfolio solving.

3 Basic Proof Production

Our goal is to produce checkable unsatisfiability proofs for problems solved by distributed clause-sharing SAT solvers. We propose to reuse the work done on proofs for sequential solvers by having each solver produce a partial proof containing the clauses it learned. These partial proofs are invalid in general because each sequential solver can rely on clauses shared by other solvers when learning new clauses. For example, when solver A derives a new clause, it might rely on clauses from solvers B and C , which in turn relied on clauses from solvers D and E , and so on. The justification of A 's clause derivation is thus spread across multiple partial proofs. We need to combine the partial proofs into a single valid proof in which the clauses are in *dependency order*, meaning that each clause can be derived from previous clauses.

To generate an efficiently-checkable combined proof in a scalable way, we must solve three challenges:

1. Provide metadata to identify which solver produced each learned clause.
2. Efficiently sort learned clauses in dependency order across all solvers.
3. Reduce proof size by removing unnecessary clauses.

Switching from DRAT to the LRAT proof format provides the mechanism to unlock all three challenges. First, we specialize the clause-numbering scheme used by LRAT in order to distinguish the clauses produced by each solver. Second, we use the dependency information from LRAT to construct a complete proof from the partial proofs produced by each solver. Finally, we determine which clauses are unnecessary (or used only for certain parts of the proof) to delete clauses from the proof as soon as they are no longer required.

Algorithm 1 Algorithm for combining partial proofs

```

1: function COMBINE(partial proofs  $p_1, p_2, \dots, p_n$ , number of original clauses  $o$ )
2:    $i \leftarrow 1$ 
3:   while true do
4:     if  $p_i.hasNext()$  then
5:        $\langle id, type, clause, proofHint \rangle \leftarrow p_i.peekNext()$ 
6:       if  $dependenciesSatisfied(proofHint)$  then
7:          $emit \langle id, type, clause, proofHint \rangle$ 
8:          $p_i.next()$  ▷ Line completed
9:         if  $clause = \emptyset$  then ▷ Derived empty clause
10:        return
11:      else ▷ Leave the line and move to next partial proof
12:         $i \leftarrow (i \bmod n) + 1$ 
13:      else ▷ Move to next partial proof if current is done
14:         $i \leftarrow (i \bmod n) + 1$ 

```

We update the clause-distribution mechanism in the distributed solver to broadcast the clause ID with each learned clause. A receiving solver stores the clause with its ID and uses the ID in proof hints when the clause is used locally, as it does with locally-derived clauses. Unlike locally-derived clauses, we add no derivation lines for remote clauses to the local proof. Instead, these derivations will be added to the final proof when combining the partial proofs.

3.1 Solver Partial Proof Production

To combine the partial proofs into a complete proof, we modify the mechanism producing LRAT proofs in each of the component solvers. We assign to each clause an ID that is unique across solvers and identifies which solver originally derived it. The following mapping from clauses to IDs achieves this:

Definition 1. *Let o be the number of clauses in the original formula and let n be the number of sequential solvers. Then, the ID of the k -th derived clause ($k \geq 0$) of solver i is defined as $ID_k^i = o + i + nk$.*

Given ID_k^i , we can easily determine the solver ID i using modular arithmetic.

3.2 Partial Proof Combination

Once the distributed solver has concluded the input formula is unsatisfiable, we have n partial proofs. The clause derivations in these proofs refer to clauses of other partial proofs, but they are, locally, in dependency order. We can therefore combine the partial proofs without reordering their clauses beforehand. We can simply interleave their clauses so the resulting proof is also in dependency order, ignoring any deletions in the partial proofs.

Our algorithm goes through the partial proofs round-robin, at each step emitting all the clauses from each file where the dependencies of the clause have

INSTANCE 1					INSTANCE 2					COMBINED				
9		-3	0	5 4 0	10	1	2	0	3 2 0	9		-3	0	5 4 0
11		-1	0	6 9 0	12	2	3	-4 0	7 11 0	11		-1	0	6 9 0
11	d	9	0		14			0 11 10 1 0	10	1	2	0	3 2 0	
13	1	2	3 0	8 12 0						12	2	3	-4 0	7 11 0
										14			0 11 10 1 0	

Fig. 2: Partial proofs and combined proof of unsatisfiability.

already been emitted. It ends when the empty clause is emitted. The procedure is shown in Algorithm 1. For each partial proof, we maintain an iterator over the learned clauses. We add the next clause from the current partial proof (p_i) to the final proof if its dependencies are satisfied (determined by comparing each hint to the last clause emitted from the partial proof whence it originated); otherwise it cycles to the next partial proof. It emits the line and moves to the next clause in the file. The algorithm terminates when it emits the empty clause (line 10).

Example 1. Suppose that two solver instances (instance 1 and instance 2) determined together that the formula from Figure 1 is unsatisfiable, with the two partial proofs shown in Figure 2. We start with instance 1. As clause 9 only relies on original clauses, we emit it. Clause 11 relies on original clause 6 and emitted clause 9, so we emit it. Clause 13 relies on clauses 8 and 12, which is not emitted, so we cannot emit clause 13 and move to instance 2. Clause 10 can be emitted, as can clause 12, which relies on an original and an emitted clause. Clause 14 relies on emitted clauses 11 and 10 and on original clause 1, so we can emit it as well. Since clause 14 is the empty clause, we finish with a complete proof, shown in Figure 2(c). Notice that clause 13 was not added to the combined proof, since it was not required to satisfy any dependencies of the empty clause.

3.3 Proof Pruning

The combined proof produced by our procedure is valid but not efficiently checkable because (1) it can contain clauses that are not required to derive the empty clause and (2) it does not contain deletion lines, meaning that a proof checker must maintain *all* learned clauses in memory throughout the checking process. To reduce size and to improve proof-checking performance, we prune our combined proof toward a minimal proof containing only necessary clauses, and we add deletion statements for clauses as soon as they are not needed anymore.

Algorithm 2 shows our pruning algorithm that walks the combined proof in reverse (similar to backward checking of DRAT proofs [19]). We maintain a set of clauses *required* in the proof, initialized to the empty clause alone. We then process all clauses in reverse order, including the empty clause, ignoring all clauses not in the required set. For each required clause, we check its dependencies to see if this is the first time (from the proof's end) a dependency is seen; if so, we emit a deletion line for the dependency since it will never be used again in the proof. After checking all its dependencies, we output the clause itself. The

Algorithm 2 Algorithm for pruning proofs

```

1: function PRUNE(combined and reversed proof  $p$ , number of original clauses  $o$ )
2:    $required \leftarrow \{p.peekNextId()\}$   $\triangleright$  Must be empty clause, which is required
3:   while  $p.hasNext()$  do
4:      $\langle id, type, clause, proofHint \rangle \leftarrow p.readNext()$ 
5:     if  $id \in required$  then  $\triangleright$  Only process a line if it is required later
6:       for  $hint \in proofHint$  do
7:         if  $hint > o \wedge hint \notin required$  then  $\triangleright$  Not used later
8:            $required \leftarrow required \cup \{hint\}$ 
9:            $emit \langle id, delete, hint \rangle$ 
10:     $emit \langle id, add, clause, proofHint \rangle$ 

```

final output of the algorithm is a proof in reversed order, where each clause is required for some derivation and deleted as soon as it is no longer required.

Example 2. Consider the combined proof from Figure 2. After applying Algorithm 2, working backward from clause 14, we determine that clause 12 is not required, so it is removed. Additionally, prior to clause 11, clause 9 is not in the required set, so it can be deleted after processing clause 11. On larger proofs, as discussed in Section 6, pruning can reduce the size of the proof by 10x or more.

4 Distributed Proof Production

The proof production as described above is sequential and may process huge amounts of data, all of which needs to be accessible from the machine that executes the procedure. In addition, maintaining the required clause IDs during the procedure may require a prohibitive amount of memory for large proofs. In the following, we propose an efficient distributed approach to proof production.

4.1 Overview

Our previous sequential proof-combination algorithm first combines all partial proofs into a single proof and then prunes unneeded proof lines. In contrast, our distributed algorithm first prunes all partial proofs in parallel and only then merges them into a single file.

We have m processes with c solver instances each, amounting to a total of $n = mc$ solvers. We make use of the fact that the solvers exchange clauses in periodic intervals (one second by default). We refer to these intervals between subsequent sharing operations as *epochs*. Consider Fig. 3 (left): Clause 118 was produced by S_2 in epoch 1. Its derivation may depend on local clause 114 and on any of the 11 clauses produced in epoch 0, but it cannot depend, e.g., on clause 109 or 111 since these clauses have been produced after the last clause sharing. More generally, a clause c produced by instance i during epoch e can only depend on (i) earlier clauses by instance i produced during epoch e or earlier, and (ii) clauses by instances $j \neq i$ produced *before* epoch e .

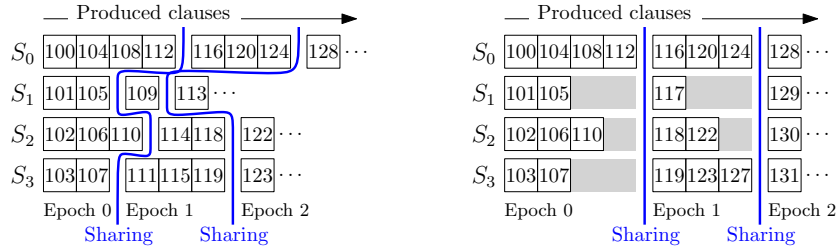


Fig. 3: Four solvers work on a formula with 99 original clauses, produce new clauses (depicted by their ID), and share clauses periodically, without (left) and with (right) aligning clause IDs.

Using this knowledge, we can essentially *rewind* the solving procedure. Each process reads its partial proofs in reverse order, outputs each line which adds a required clause, and adds the hints of each such clause to the required clauses. Required remote clauses produced in epoch e are transferred to their process of origin before any proof lines from epoch e are read. As such, whenever a process reads a proof line, it knows whether the clause is required. The outputs of all processes can be merged into a single valid proof (Section 4.3).

4.2 Distributed Pruning

Clause ID Alignment. To synchronize the reading and redistribution of clause IDs in our distributed pruning, we need a way to decide from which epoch a remote clause ID originates. However, solvers generally produce clauses with different speeds, so the IDs by different solvers will likely be in dissimilar ranges within the same epoch over time. For instance, in Fig. 3 (left) instance S_3 has no way of knowing from which epoch clause 118 originates. To solve this issue, we propose to align all produced clause IDs after each sharing. During the solving procedure, we add a certain offset δ_i^e to each ID produced by instance i in epoch e . As such, we can associate each epoch e with a global interval $[A_e, A_{e+1})$ that contains all clause IDs produced in that epoch. In Fig. 3 (right), $A_0 = 100$, $A_1 = 116$, and $A_2 = 128$. Clause 118 on the left has been aligned to 122 on the right ($\delta_2^1 = 4$) and due to $A_1 \leq 122 < A_2$ all instances know that this clause originates from epoch 1.

Initially, $\delta_i^0 := 0$ for all i . Let I_i^e be the first original (unaligned) ID produced by instance i in epoch e . With the sharing that initiates epoch $e > 0$, we compute the common start of epoch e , $A_e := \max_i \{I_i^e + \delta_i^{e-1} - i\}$, as the lowest possible value that is larger than all clause IDs from epoch $e-1$. We then compute offsets δ_i^e in such a way that $I_i^e + \delta_i^e = A_e + i$, which yields $\delta_i^e := (A_e + i) - I_i^e$. If we then export a clause produced during e by instance i , we add δ_i^e to its ID, and if we import shared clauses to i , we filter any clauses produced by i itself. Note that we do not modify the solvers' internal ID counters or the proofs they output. Later, when reading the partial proof of solver i at epoch e , we need to add δ_i^e to each ID originating from i . All other clause IDs are already aligned.

Rewinding the Solve Procedure. Assume that instance $u \in \{1, \dots, n\}$ has derived the empty clause in epoch \hat{e} . For each local solver i , each process has a *frontier* F_i of required clauses produced by i . In addition, each process has a *backlog* B of remote required clauses. B and F_i are collections of clause IDs and can be thought of as maximum-first priority queues. Initially, F_u contains the ID of the empty clause while all other frontiers and backlogs are empty. Iteration $x \geq 0$ of our algorithm processes epoch $\hat{e} - x$ and features two stages:

1. *Processing:* Each process continues to read its partial proofs in reverse order from the last introduced clause of the current epoch. If a line from solver i is read whose clause ID is at the top of F_i , then the ID is removed from F_i , the line is output, and each clause ID hint h in the line is treated as follows:

- h is inserted in F_j if local solver j (possibly $j = i$) produced h .
- h is inserted in B if a remote solver produced h .
- h is dropped if h is an ID of an original clause of the problem.

Reading stops as soon as a line’s ID precedes epoch $e = \hat{e} - x$. Each F_i as well as B now only contain clauses produced *before* e .

2. *Task redistribution:* Each process extracts all clause IDs from B that were produced during $\hat{e} - x - 1$. These clause IDs are aggregated among all processes, eliminating duplicates in the same manner as Mallob’s clause sharing detects duplicate clauses [28]. Each process traverses the aggregated clause IDs, and each clause produced by a local solver i is added to F_i .

Our algorithm stops in iteration \hat{e} after the Processing stage, at which point all frontiers and backlogs are empty and all relevant proof lines have been output.

Analysis. In terms of total work performed, all partial proofs are read completely. For each required clause we may perform an insertion into some B , a deletion from said B , an insertion into some F_i , and a deletion from said F_i . If we assume logarithmic work for each insertion and deletion, the work for these operations is linear in the combined size of all partial proofs and loglinear in the size of the output proof. In addition, we have \hat{e} iterations of communication whose overall volume is bounded by the communication done during solving. In fact, since only a subset of shared clauses are required and we only share 64 bits per clause, we expect strictly less communication than during solving. Computing A_e for each epoch e during solving is negligible since the necessary aggregation and broadcast can be integrated into an existing collective operation. Regarding memory usage, the size of each B and each F_i can be proportional to the combined size of all required lines of the according partial proofs. However, we can make use of external data structures which keep their content on disk except for a few buffers.

4.3 Merging Step

For each partial proof processed during the pruning step, we have a stream of proof lines sorted in reverse chronological order, i.e., starting with the highest clause ID. The remaining task is to merge all these lines into a single, sorted proof file. As shown in Fig. 4 (left), we arrange all processes in a tree. We can easily merge a number of sorted input streams into a single sorted output stream

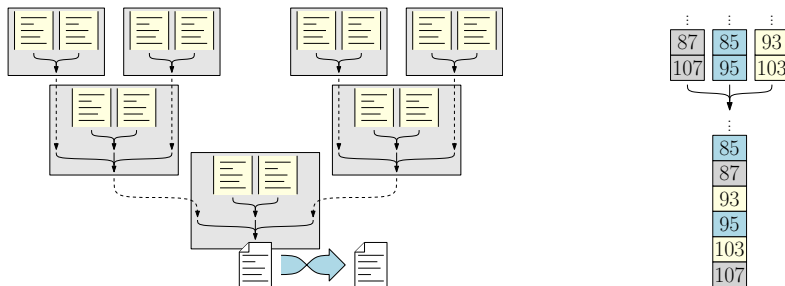


Fig. 4: Left: Proof merging with seven processes and 14 solvers. Each box represents a process with two local proof sources. Dashed arrows denote communication. Right: Example of merging three streams of LRAT lines into a single stream. Each number i represents an LRAT line describing a clause of ID i .

by repeatedly outputting the line with the highest ID among all inputs (Fig. 4 right). This way, we can hierarchically merge all streams along the tree. At the tree’s root, the output stream is directed into a file. This is a sequential I/O task that limits the speed of merging. Finally, since the produced file is in reverse order, a buffered operation reverses the file’s content.

A final challenge is to add clause deletions to the final proof. Before a line is written to the combined proof file, we can scan its hints and output a deletion line for each hint we did not encounter before (see Section 3.3). However, implementing this in an exact manner requires maintaining a set of clause IDs which scales with the final proof size. Since our proof remains valid even if we omit some clause deletions, we can use an approximate membership query (AMQ) structure with fixed size and a small false positive rate, e.g., a Bloom filter [7].

5 Implementation

We employ a solver portfolio based on the sequential SAT solver CaDiCaL [5]. We modified CaDiCaL to output LRAT proof lines and to assign clause IDs as described in Section 3.1. To ensure sound LRAT proof logging, some features of CaDiCaL currently need to be turned off, such as bounded variable elimination, hyper-ternary resolution, and vivification. Similarly, Mallob’s original portfolio of CaDiCaL configurations features several options that are incompatible with our proof logging as of yet. Therefore, we created a smaller portfolio of “safe” configurations that include shuffling variable priorities, adjusted restart intervals, and disabled inprocessing. We also use different random seeds and use Mallob’s diversification based on randomized initial variable polarities.

We modified Mallob to associate each clause with a 64-bit clause ID. For consistent bookkeeping of sharing epochs, we defer clause sharing until all processes have fully initialized their solvers. While several solvers may derive the empty clause simultaneously, only one of them is selected to be the “winner” whose empty clause will be traced. The distributed proof production features

communication similar to Mallob’s clause sharing. To realize the frontier F_i and the backlog B described in Section 4.2, we implemented an external-memory data structure which writes clause IDs to disk, categorized by their epoch. Upon reaching a new epoch, all clause IDs from this epoch are read from disk and inserted into an internal priority queue to allow for efficient polling and insertion. To merge the pruned partial proofs, we use point-to-point messages to query and send buffers of proof lines between processes. We interleave this merging with the pruning procedure in order to avoid writing the intermediate output to disk. We use a fixed-size Bloom filter to add some deletion lines to the final proof.

6 Evaluation

In this section, we present an evaluation of our proof production approaches. We provide the associated software as well as a digital appendix online.⁷

6.1 Experimental Setup

Supporting proofs introduces several kinds of performance overhead for clause-sharing portfolios in terms of solving, proof reconstruction, and proof checking. We wish to examine how well our proof-producing solver performs against (1) best-of-breed parallel and cloud solvers that do not produce proofs, (2) previous approaches to proof-producing parallel solvers, and (3) best-of-breed sequential solvers. We analyze the overhead introduced by each phase of the process, and we discuss how and where future efforts might improve performance.

We use the following pipeline for our proof-producing solvers: First, the input formula is preprocessed by performing exhaustive unit propagation. This is necessary due to a technical limitation of our LRAT-producing modification of CaDiCaL. Second, we execute our proof-producing variant of Mallob on the preprocessed formula. Third, we prune and combine all partial proofs, using either our sequential proof production or our distributed proof production. Fourth, we merge the preprocessor’s proof and our produced proof and syntactically transform the result to bring the set of clause IDs into compact shape. Fifth and finally, we run `lratt-check`⁸ to check the final proof. Only steps two and three of our pipeline are parallelized (step three depending on the particular experiment). We will refer to the first two steps as *solving*, the third step as *assembly*, the fourth step as *postprocessing*, and the fifth step as *checking*.

To examine performance overhead for proof-producing parallel and distributed solvers, we compare our proof-producing cloud and parallel solvers (`mallob-cacld-p` and `mallob-capar-p`) against six solvers. First, we include the winners of the 2022 SAT competition cloud track (`mallob-kicaliglu`, using Kissat+CaDiCaL+Lingeling+Glucose), parallel track (`parkissat-rs`, using Kissat), and sequential track (`Kissat_MAB-HyWalk`), as well as the second place

⁷ <https://github.com/domschrei/mallob/tree/certified-unsat>

⁸ <https://github.com/marijnheule/drat-trim>

Table 1: Overview of solved instances: (S)equential, (P)arallel, and (C)loud

Solver	Type	Solved	SAT	UNSAT	PAR-2 score
Kissat_MAB-HyWalk	S	218	118	100	1065.7
parkissat-rs	P	299	155	144	603.0
mallob-ki	P	260	113	147	827.6
mallob-capar	P	292	145	147	641.6
mallob-capar-p (Seq.)	P	279	140	139	719.8
mallob-capar-p (Par.)	P	276	141	135	731.4
mallob-kicaliglu	C	341	165	176	344.8
mallob-cacld	C	333	163	170	378.0
mallob-cacld-p	C	314	159	155	484.1

solver from the parallel track (`mallob-ki`, using Lingeling⁹). We then run a parallel and cloud version of Mallob that runs our described CaDiCaL portfolio *without* proof production (`mallob-capar` and `mallob-cacld`).

Following the SAT competition setup, each cloud solver runs on 100 m6i.4xlarge EC2 instances (16 core, 64GB RAM), each parallel solver runs on a single m6i.16xlarge EC2 instance (64 core, 256GB RAM), and the sequential `Kissat_MAB-HyWalk` runs on a single m6i.4xlarge EC2 instance. For each solver, we run the full benchmark suite from the SAT-Competition 2022 (400 formulas) containing both SAT and UNSAT examples. The timeout for the solving step is 1000 seconds, and the timeout for all subsequent steps is set to 4000 seconds.

Since earlier work [14] is no longer competitive in terms of solving time, we only compare proof-checking times. Specifically, we measure the overhead of checking un-pruned DRAT proofs as the ones produced by [14]. As such, we can get a picture of the performance of the earlier approach if it was realized with state-of-the-art solving techniques. We generate un-pruned DRAT proofs from the original (un-pruned) LRAT proof by stripping out the dependency information and adding delete lines for the last use of each clause.

6.2 Results

First we examine the performance overhead of changing portfolios to enable proof generation as described in Section 5 on the *solving process only*. Fig. 5 (left) and Table 1 show this data. The PAR-2 metric takes the average time to solve each problem, but counts a timeout result as a 2x penalty (e.g., given our timeout of 1000 seconds, a timeout is scored as taking 2000 seconds). We can see that our CaDiCaL portfolio `mallob-capar` outperforms the Lingeling-based `mallob-ki` significantly and is almost on par with `parkissat-rs`. Similarly, `mallob-cacld` solves eight instances less compared to `mallob-kicaliglu` but performs almost equally well otherwise. In both cases, we have constructed solvers which are,

⁹ `mallob-ki` employed a Lingeling-based portfolio due to a misconfiguration, see: <http://algo2.iti.kit.edu/schreiber/downloads/mallob-ki-mallob-li.pdf>

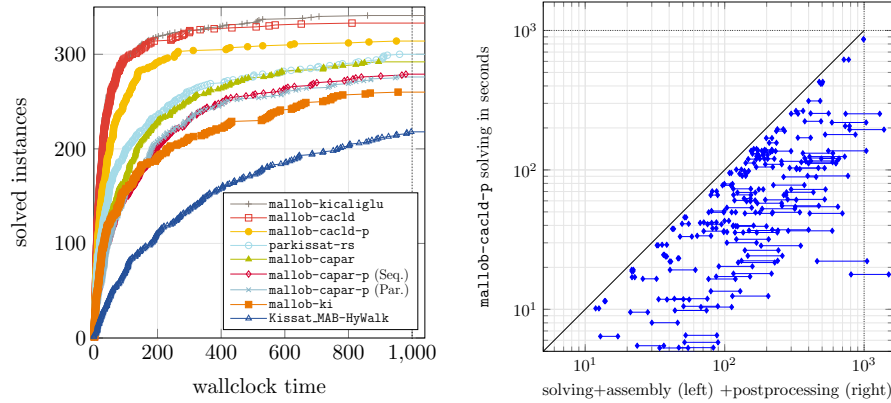


Fig. 5: Left: Comparison of solving times. Right: Relation of solving times to assembly and postprocessing times for `mallob-cacld-p`. Each pair of points corresponds to one instance, the y coordinate denoting the solving time. The left x coordinate denotes solving and assembly time and the right x coordinate denotes solving, assembly, and postprocessing time.

up to a small margin, on par with the state of the art. For our actual proof-producing solvers, `mallob-capar-p` and `mallob-cacld-p`, we noticed a more pronounced decline in solving performance. On top of the overhead introduced by proof logging and our preprocessing, we experienced a few technical problems, including memory issues¹⁰, which resulted in a drop in the number of instances solved and also caused `mallob-capar-p` with parallel proof production to solve three instances less than with sequential proof production. We believe that we can overcome these issues in future versions of our system. That being said, our proof-producing solvers already outperform any of the solvers at a lower scale.

Second, we examine statistics on proof reconstruction and checking, showing results in Table 2. Since we want to investigate our approaches’ overhead compared to pure solving, we measure run times as a multiple of the solving time. (We provide absolute run times in the Appendix, Table 1.) The prefix “Seq.” denotes `mallob-capar-p` with sequential proof production, “Par.” denotes `mallob-capar-p` with distributed proof production run on a single machine, and “Cld.” denotes `mallob-cacld-p` with distributed proof production.

DRAT checking succeeded in 81 out of 139 cases and timed out in 58 cases. For the successful cases, DRAT checking took $24.8\times$ the solving time on average whereas our sequential assembly, postprocessing and checking combined succeeded in 139 cases and only took $3.8\times$ the solving time on average. This result confirms that our approach successfully overcomes the major scalability problems of earlier work [14]. In terms of uncompressed proof sizes, our LRAT

¹⁰ We disabled Mallob’s memory panic mode to ensure consistent proof logging.

Table 2: Statistics on proof production and checking. All properties except for file sizes and pruning factor are given as a multiple of the solving time. We list minima, maxima, medians, arithmetic means, and the 10th and 90th percentiles.

Property	#	min	p10	med	mean	p90	max
DRAT check	81	0.512	1.725	7.442	24.815	67.065	169.869
Seq. assembly	139	0.019	0.305	1.376	2.324	5.747	13.289
Seq. postprocessing	139	0.001	0.012	0.131	0.263	0.790	2.218
Seq. checking	139	0.007	0.043	0.572	1.252	3.970	10.980
Seq. asm+post+chk	139	0.037	0.412	2.110	3.840	10.834	26.487
Par. assembly	135	0.059	0.080	0.365	0.805	2.227	7.475
Par. postprocessing	135	0.001	0.016	0.156	0.293	0.861	2.300
Par. checking	135	0.007	0.042	0.622	1.241	3.540	11.645
Par. asm+post+chk	135	0.067	0.167	1.097	2.339	6.611	21.420
Cld. assembly	155	0.114	0.185	1.412	2.444	5.410	44.268
Cld. postprocessing	155	0.003	0.060	0.696	2.046	4.785	39.096
Cld. checking	155	0.033	0.189	3.291	8.883	21.974	170.378
Cld. asm+post+chk	155	0.168	0.577	5.110	13.373	32.484	253.742
DRAT proof size (GiB)	139	0.012	0.366	1.236	3.246	8.395	29.308
Seq. proof size (GiB)	139	0.016	0.223	2.379	5.384	16.082	46.986
Par. proof size (GiB)	135	0.006	0.173	2.034	5.345	13.164	57.739
Cld. proof size (GiB)	155	0.016	0.342	3.940	10.533	30.130	89.106
Cld. pruning factor	155	2.374	5.379	17.826	293.762	337.486	12466.700

proofs can be about twice as large as the DRAT proofs, which seems more than acceptable considering the dramatic difference in performance. Given that DRAT-based checking was ineffective at the scale of parallel solvers, we decided to omit it in our distributed experiments which feature even larger proofs.

Regarding `mallob-capar-p` with parallel proof production, we can see that the assembly time is reduced from $2.32\times$ down to $0.81\times$ the solving time on average, which also improves overall performance ($3.84\times$ to $2.34\times$).

The results for `mallob-cacl-d-p` demonstrate that our proof assembly is feasible, taking around $2.5\times$ the solving time on average. We visualized this overhead and how it relates to the postprocessing overhead in Fig. 5 (right). The proofs produced are about twice as large as for `mallob-capar-p`. Considering that the proofs originate from 25 times as many solvers, this increase in size is quite modest, which can be explained by our proof pruning. We captured the *pruning factor* — the number of clauses in all partial proofs divided by the number of clauses in the combined proof — for each instance. Our pruning reduces the derived clauses by a factor of 293.8 on average (17.8 for the median instance), showing that it is a crucial technique to obtain proofs that are feasible to check. As such, we also managed to produce and check a proof of unsatisfiability for a formula whose unsatisfiability has not been verified before (`PancakeVsInsertSort_8_7.cnf`).

Lastly, to compare our approach at the largest scale with the state of the art in sequential solving, we computed speedups of `mallob-cacl-d-p`, solv-

ing times only, over `Kissat_MAB-HyWalk` and arrived at a median speedup of 11.5 (Appendix, Table 2). We also analyzed `drat-trim` checking times of `Kissat_MAB-HyWalk`, kindly provided by the competition organizers, and arrived at a median overhead of $1.1\times$ its own solving time (Appendix, Table 3). Going by these measures, `Kissat_MAB-HyWalk` takes around $11.5 \cdot 2.1 \approx 24.2\times$ the solving time of `mallob-cacld-p` to arrive at a checked result while our complete pipeline only takes $5.1\times$ the solving time for the median instance. This indicates that our approach is considerably faster than the best available sequential solvers.

We can see that the bottleneck of our pipeline shifts from the assembly step further to the postprocessing and checking steps when increasing the degree of parallelism. This is to be expected since the latter steps are, so far, inherently sequential whereas our proof assembly is scalable. While the postprocessing step is a technical necessity in our current setup, we believe that large portions of it can be eliminated in the future with further engineering. For instance, enhancing the LRAT support of our modified CaDiCaL to natively handle unit clauses in the input would allow us to skip preprocessing and simplify postprocessing.

7 Conclusion and Future Work

Distributed clause-sharing solvers are currently the fastest tools for solving a wide range of difficult SAT problems. Nevertheless, they have previously not supported proof-generation techniques, leading to potential soundness concerns. In this paper, we have examined mechanisms to add efficient support for proof generation to clause-sharing portfolio solvers. Our results demonstrate that we can, with reasonable efficiency, add support to these solvers to have full confidence that the results they produce are correct.

Following our research, more work is required to reduce overhead in the different steps involved and to improve scalability of the end-to-end procedure. This may include designing more efficient (perhaps even parallel) LRAT checkers, examining proof-streaming techniques to eliminate most I/O operations, and improving LRAT support in solver backends. In fact, it might be possible to generalize our approach to DRAT-based solvers by adding additional metadata, and this might allow easier retrofitting of the approach onto larger portfolios of solvers. We also intend to investigate producing proofs in Mallob for the case where many problems are solved at once and jobs are rescaled dynamically [26].

Acknowledgments

We would like to thank Mario Carneiro for providing help for his FRAT-supporting variant of CaDiCaL; Markus Iser for providing competition data on proof checking; Andrew Gacek for his suggestions to early drafts of this paper; and the reviewers for their helpful feedback. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agr. No. 882500). This project was partially supported by the U.S. National Science Foundation grant CCF-2015445.



References

1. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8561, pp. 197–205. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_15
2. Balyo, T., Froleyks, N., Heule, M., Iser, M., Jarvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. Department of Computer Science Report Series B, Department of Computer Science, University of Helsinki, Finland (2021)
3. Balyo, T., Froleyks, N., Heule, M., Iser, M., Jarvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions. Department of Computer Science Report Series B, Department of Computer Science, University of Helsinki, Finland (2020)
4. Balyo, T., Sanders, P., Sinz, C.: HordeSat: A massively parallel portfolio SAT solver. In: Heule, M., Weaver, S. (eds.) Theory and Applications of Satisfiability Testing – SAT 2015. pp. 156–172. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_12
5. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Jarvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009), <http://dblp.uni-trier.de/db/series/faia/faia185.html>
7. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970). <https://doi.org/10.1145/362686.362692>
8. Brakensiek, J., Heule, M., Mackey, J., Narváez, D.E.: The resolution of Keller’s conjecture. J. Autom. Reason. **66**(3), 277–300 (2022). <https://doi.org/10.1007/s10817-022-09623-5>
9. Cruz-Filipe, L., Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 220–236. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_14
10. Fleury, M., Biere, A.: Scalable proof producing multi-threaded SAT solving with Gimsat through sharing instead of copying clauses. In: Pragmatics of SAT (2022). <https://doi.org/10.48550/arXiv.2207.13577>
11. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1-2), 43–62 (2001). [https://doi.org/10.1016/S0004-3702\(00\)00081-3](https://doi.org/10.1016/S0004-3702(00)00081-3)
12. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. J. Satisf. Boolean Model. Comput. **6**(4), 245–262 (2009). <https://doi.org/10.3233/sat190070>
13. Heule, M., Jr., W.A.H., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September

- 26-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10499, pp. 269–284. Springer (2017). https://doi.org/10.1007/978-3-319-66107-0_18
14. Heule, M., Manthey, N., Philipp, T.: Validating unsatisfiability results of clause sharing parallel sat solvers. In: POS@ SAT. pp. 12–25 (2014)
 15. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **abs/1610.06229** (2016). <https://doi.org/10.48550/arXiv.1610.06229>
 16. Heule, M.J.H.: Schur number five. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18). pp. 6598–6606. AAAI Press (2018). <https://doi.org/10.1609/aaai.v32i1.12209>
 17. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 228–245. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_15
 18. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Hardware and Software: Verification and Testing. pp. 50–65. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34188-5_8
 19. Heule, M.J., Hunt, W.A., Wetzler, N.: Trimming while checking clausal proofs. In: 2013 Formal Methods in Computer-Aided Design. pp. 181–188 (2013). <https://doi.org/10.1109/FMCAD.2013.6679408>
 20. Järvisalo, M., Heule, M.J., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7364, pp. 355–370. Springer (2012). https://doi.org/10.1007/978-3-642-31365-3_28
 21. Kissat SAT solver. <http://fmv.jku.at/kissat/>, accessed: 2022-08-17
 22. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020). <https://doi.org/10.1007/s10817-019-09525-z>
 23. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: Painless: A framework for parallel SAT solving. In: Gaspers, S., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing – SAT 2017. pp. 233–250. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_15
 24. Nair, A., Chattopadhyay, S., Wu, H., Ozdemir, A., Barrett, C.: Proof-stitch: Proof combination for divide and conquer SAT solvers. In: Formal Methods in Computer-Aided Design. pp. 84–88 (2022). <https://doi.org/10.34727/2022/isbn.978-3-85448-053-2>
 25. Rintanen, J.: Planning and SAT. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 483–504. IOS Press (2009). <https://doi.org/10.3233/978-1-58603-929-5-483>
 26. Sanders, P., Schreiber, D.: Decentralized online scheduling of malleable NP-hard jobs. In: European Conference on Parallel Processing. pp. 119–135. Springer (2022). https://doi.org/10.1007/978-3-031-12597-3_8

27. Schreiber, D.: Mallob in the SAT competition 2022. In: Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions. pp. 46–47. Department of Computer Science Report Series B, University of Helsinki (2022)
28. Schreiber, D., Sanders, P.: Scalable SAT solving in the cloud. In: Li, C.M., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing – SAT 2021. pp. 518–534. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_35
29. Schubert, T., Lewis, M., Becker, B.: Pamiraxt: Parallel SAT solving with threads and message passing. *J. Satisf. Boolean Model. Comput.* **6**(4), 203–222 (2009). <https://doi.org/10.3233/sat190068>
30. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in cakeml. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 223–241. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_12
31. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE* **103**(11), 2021–2035 (2015). <https://doi.org/10.1109/JPROC.2015.2455034>
32. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.* **32**, 565–606 (2008). <https://doi.org/10.1613/jair.2490>