Karlsruhe Institute of Technology

Master thesis

# Parallel Propositional Proofs for Clause-Sharing SAT Solvers

Michael Dörr

Date: September 1, 2025

Supervisors: Dr. rer. nat. Dominik Schreiber
Prof. Dr. Peter Sanders

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Generative AI was used in this work for the purposes of spelling, grammar, and style corrections.

Karlsruhe, 1. September 2025

. . . . . .
(Michael Dörr)

**Acknowledgement**

The author gratefully acknowledges the computing time provided on the high-performance computer HoreKa by the National High-Performance Computing Center at KIT (NHR@KIT). This center is jointly supported by the Federal Ministry of Education and Research and the Ministry of Science, Research and the Arts of Baden-Württemberg, as part of the National High-Performance Computing (NHR) joint funding program (https://www.nhr-verein.de/en/our-partners). HoreKa is partly funded by the German Research Foundation (DFG).

# Abstract

In many scientific fields such as electronic circuit design, verification, cryptography, and explainable AI, the *satisfiability problem* (SAT) plays an important role. Parallel clause-sharing SAT solvers (CSSS) are used to solve complex SAT instances relevant to practical applications in these and other fields. To gain confidence in the results of these SAT solvers in the case of unsatisfiable instances, *proofs of unsatisfiability* (UNSAT proofs) are required. While sequential SAT solvers are already capable of generating UNSAT proofs, previous approaches for CSSS have disadvantages in terms of scalability or in persisting proof information.

We address this issue with *Parallel LRUP* (*PalRUP*) as a distributed proof format that can be generated and checked without sequential bottlenecks. PalRUP proofs are generated with our extension of the on-the-fly checker *ImpCheck* [25]. Our proof checker *PalRUP-Check* checks *PalRUP* proofs in a scalable way and is modular, making it a starting point for future formally verified implementations.

We evaluated both C99 implementations on up to 1216 cores and generate PalRUP proofs with 39 % to 54 % running time overhead, whereas pure on-the-fly checking reportedly imposes 37 % to 42 % overhead. In our tests, PalRUPCheck took less than 100 s to check proofs that have a solving time of over 200 s. For sufficiently large instances, we therefore have a proof checker that can check significantly faster than CSSS can solve the corresponding instances.

# Zusammenfassung

In vielen wissenschaftlichen Bereichen, wie dem Schaltkreisentwurf, der formalen Verifikation, der Kryptographie und der erklärbaren KI, spielt das *aussagenlogische Erfüllbarkeitsproblem* (SAT) eine wichtige Rolle. Um praxisrelevante, komplexe SAT-Instanzen zu lösen, die beispielsweise in diesen Bereichen auftreten, werden parallele Klausel-Austausch-SAT-Solver (KASS) eingesetzt. Um Vertrauen in die Ergebnisse dieser SAT-Solver im Fall von unerfüllbaren Instanzen zu erlangen, werden *Unerfüllbarkeits-Beweise* (UNSAT-Beweise) benötigt. Während sequenzielle SAT-Solver bereits UNSAT-Beweise generieren können, haben die bisherigen Ansätze für KASS entweder Nachteile bezüglich der Skalierbarkeit oder persistieren die Beweisinformationen nicht.

Um diese Situation zu verbessern, haben wir *Parallel LRUP* (*PalRUP*) als verteiltes Beweisformat entwickelt, das ohne sequenzielle Flaschenhälse erzeugt und geprüft werden kann. Mit unserer Erweiterung des on-the-fly-Checkers *ImpCheck* [25] können PalRUP-Beweise erzeugt werden. Unser Beweisprüfer *PalRUPCheck* kann PalRUP Beweise skalierbar prüfen und ist modular entworfen, um später als Ausgangspunkt für eine formal verifizierte Implementierung zu dienen.

Die C99-Implementierungen beider Ansätze haben wir auf bis zu 1216 Kernen evaluiert. Dabei wurden PalRUP-Beweise mit einem Laufzeit-Overhead von 39 % bis 54 % generiert, während reines On-the-Fly-Checking einen Overhead von 37 % bis 42 % verursacht. In unseren Tests hat PalRUPCheck für Beweise, die eine Lösezeit von über 200 s haben, weniger als 100 s für die Beweisprüfung benötigt. Für hinreichend große Instanzen haben wir also einen Beweisprüfer, der erstmals deutlich schneller prüfen kann als KASS die dazugehörigen Instanzen lösen können.

# Contents

# 1 Introduction

In Section 1.1, we describe the importance of SAT and the state of proof production for UNSAT proofs. We also present our contributions in Section 1.2.

## 1.1 Motivation

The *Boolean satisfiability problem* (SAT) is the fundamental problem for complexity theory and the P vs. NP problem [6]. As such, it is both well researched and an important problem-solving tool for many application areas such as electronic design, verification, cryptography, and explainable AI [5, 8, 17, 28]. For example, in electronic design automation, it is necessary to be able to test two circuits for equality [32]. It is possible to encode this equality check as a SAT instance [13], which can be satisfied if there is at least one assignment of the inputs for which the outputs of the two circuits are different. Consequently, this means that if the circuits are equal, no variable assignment (therefore input assignment) is found for which the two circuits output different values, and thus a SAT solver responds with unsatisfiable (UNSAT). While instances with satisfiable (SAT) results can be checked instantly by substituting variables in linear time, this is more difficult for UNSAT instances. To have trust in UNSAT results, the core clauses, i.e. those clauses that are necessary to obtain an UNSAT result, that modern SAT solvers generate (learn) at runtime, must be checked for correct derivation by a formally verified checker. Although core clauses only make up a fraction of all generated clauses, proofs can become exceptionally long (depending on the proof system).

DRAT [30] and LRAT [7] are widely used as proof formats for unsatisfiability. In both formats, clauses generated (learned) by various solver techniques are recorded. These clauses always satisfy the *Resolution Asymmetric Tautology* (RAT) property with respect to a subset of previously learned clauses. Unlike LRAT, DRAT does not record which subset of clauses is sufficient to satisfy the RAT property. While checking DRAT proofs, therefore, takes about as long as solving the corresponding instance, LRAT proofs can be checked in only 5 % to 10 % of the time.

Like other algorithms, SAT solving can be accelerated through parallelization [18]. In the best performing parallelization paradigm for SAT solving, several SAT solver threads are started simultaneously for the same instance and exchange (share) clauses periodically. This extends the limits of solvable SAT and UNSAT instances. Errors, such as those in network communication via MPI, can cause solvers to incorrectly return UNSAT for *satis-*

*fiable* instances. Therefore, proof of the unsatisfiability (UNSAT) of the problem instance is desired. However, this kind of parallelization renders the (re-)construction of a coherent proof of unsatisfiability particularly challenging. The reason for this is that shared clauses cannot be checked locally for individual solver threads.

As we will see in Section 2.4, existing approaches still have unresolved issues. On the one hand, there is the possibility of generating monolithic proofs [19]. However, this has a sequential I/O bottleneck. In addition, it can happen that the simultaneously active clauses may require more main memory when checking the proof, than is available on the machine on which the parallel solver solved the instance. On the other hand, there is the scalable option of monitoring the solution process in real-time. However, this approach does not support the generation of a proof that can be viewed by third parties.

We present our *PalRUP* approach with the novel UNSAT proof checker *PalRUPCheck* in this thesis. With PalRUPCheck we now have an unverified UNSAT proof checker that sets new standards in scalability for checking UNSAT proofs of massively parallel SAT solvers. This suggests that further research with the aim of developing a formally verified checker similar to PalRUPCheck would be promising.

## 1.2 Contributions

Now we list our contributions of this thesis.

Firstly, we formally define the *Parallel LRUP* (*PalRUP*) proof format, that is based on LRAT [7]. It is a file format for distributed data and thus *natively* allows the data to be written and processed in parallel. We provide an EBNF context free grammar for this purpose. In addition, we define what restrictions PalRUP imposes on clause IDs to make efficient checks for acyclicity possible.

Secondly, we propose an new extension to the realtime checker ImpCheck [25] that is capable of producing PalRUP proofs with solvers that have support for ImpCheck's interface. That way we have a proof producing approach that can be employed by other clause-sharing SAT solvers than the clause-sharing paralell SAT solver MallobSat [26] for proof production. We outline our approach to PalRUP proof generation and provide an open-source C99 implementation.

Thirdly, we outline our novel UNSAT proof checker approach PalRUPCheck. Several processes with different ranks are started for each step, with the number of processes depending on the number of partial proofs ($P$). For each step, we examine the work that a single process performs in that step. In the first step, each clause of the partial proof is checked for the RUP property. Imported clauses are added as axioms and written to new files, but are not checked for the RUP property. These clauses are deduplicated in a further step and redirected to other files so that they are located with the partial proofs of their original solvers. A final check ensures that the split clauses were generated in the partial proof of their original solver and are identical. To ensure that the final step is a sufficient

check, all files have cryptographic hashes (fingerprints), which are checked in the final step in particular. This ensures that, on the one hand, all learned clauses in the partial proofs satisfy the RUP property and, on the other hand, that the dependencies among the clauses are acyclic and that all shared clauses were learned by a solver thread. Also a proof of correctness of the checking approach is provided.

Fourthly, we also provide a open-source C99 implementation of PalRUPCheck. We ensure that at most $\left\lceil \sqrt{P} \right\rceil$ files are written per folder at the same time to achieve better performance on parallel network file systems. Also for better modularity, we split the three steps from the theoretical approach into five steps. We also analyze every of the five steps in terms of runtime, memory, and storage consumption.

Lastly, we evaluate our implementation of ImpCheck's new PalRUP proof production extension and our new PalRUP UNSAT proof checker. We evaluated the runtime in different steps, memory usage, and storage consumption. To ensure the reproducibility of the tests, the scripts for the experiments and evaluation results are also available online. In evaluation we see that PalRUPCheck on 1216 cores is faster than solving for instances that have a solving time of at least 75 s. For solving times of more than 200 s, relative proof checking time is below 50 % (below 100 s). Also we compare our results with original ImpCheck on the same machine with the same test instances for best comparability. As a result, we see that proof production adds about 2 % to 12 % overhead to realtime checking only. Some of these results can be further improved by reducing some rather high overheads we also report. Also storage consumption of PalRUP proofs must be improved in future, because our largest proof has a size of about 2.8 TB written. However, this has no visible drawbacks to scalability.

## 1.3 Outline

The thesis is structured as follows.

Chapter 2 provides foundations in the SAT problem and parallel SAT-Solving. Also we discuss proofs of unsatisfiability in general and and the previous work to obtain trusted UNSAT results. Chapter 3 discusses the theoretical considerations that influenced the specification of PalRUP, the formal definition of the format, and our approach to generating PalRUP proofs. Chapter 4 discusses the properties of PalRUPCheck and analyzes each step of the checker. In addition, the correctness of the checking approach is proven. Chapter 5 covers the algorithmic details of the most important modules designed for PalRUPCheck. Chapter 6 covers the evaluation of our approach to proof generation and proof checking. Chapter 7 we summarize our work and provide an outlook for future work.

# 2 Preliminaries

This chapter covers the basics of the Boolean satisfiability problem Section 2.1 and modern SAT solving Section 2.2. We then discuss unsatisfiability proofs Section 2.3 and previous work in this area in Section 2.4.

## 2.1 SAT Problem

The Boolean satisfiability problem (SAT problem) or propositional satisfiability problem is the first problem for which it has been proven to be *NP-complete* [6]. An instance of a SAT Problem is a Boolean logic expression in conjunctive normal form (CNF). For the expression, we want to find out whether a variable assignment exists for which the expression is `true`. In context of SAT a variable or negated variable is referred to as a *literal*, whereas a *conjunction of literals* is referred to as a *clause*.

**Definition of the SAT Problem:**
Let $L = \bigcup_{j=0}^{k} l_j \cup \neg l_j$ a set of literals,
$C = \bigcup_{i=0}^{n} c_i$ a set of clauses with clause $c_i \subset L$ and
$A_L(l_j) = b_j$ a total assignment of literals to $b_j \in \{\texttt{true}, \texttt{false}\}$ with
$A(\neg l_j) = \neg A(l_j)$.

For a given set of clauses $C$ a formula $F = \bigwedge_{c_i \in C} (\bigvee_{l_j \in c_i} l_j)$ in CNF is *satisfiable* (SAT) if an total assignment $A_L$ exists under which $F$ evaluates to `true`. Otherwise $F$ is considered *unsatisfiable* (UNSAT).

A formula $F$ can be evaluated in $\mathcal{O}(n \cdot k)$ under a given $A_L$ by checking if all $n$ clauses in $C$ contain at least one literal that is `true`. In the case that $F$ is SAT, such an $A_L$ that evaluates $F$ to `true` can be checked in linear time with the specification of $A_L$. For an UNSAT $F$, it is usually not feasible to check every possible $A_L$. As there are two possible assignments for each literal $l_j$ the possible number of total assignments is $2^k$. Millions of variables $k$ are common for SAT problem instances, therefore another way to check UNSAT instances is needed. As a more practical witness of a formula's unsatisfiability, a *proof of unsatisfiability* is required, as we explain in detail in Section 2.3.

## 2.2 Parallel SAT-Solving

Modern SAT solvers use the *Conflict-Driven Clause Learning* (CDCL) approach [18]. With this approach a solver derives *conflict clauses* that satisfy some well-defined redundancy criterion (see Section 2.3). These clauses allow to prune the search tree for a SAT assignment or can help to derive the *empty clause* if the instance is UNSAT. While there are normally literals in clauses, it can happen that a clause without literals, the empty clause, is learned. This can happen, for example, if the single literal clauses $c_a := \{l_i\}$ and $c_b := \{\neg l_i\}$ were learned in a previous step. From $\{c_a\}$ and $\{c_b\}$ we can now learn $c_{empty} := \{\}$. For an UNSAT formula, $c_{empty}$ can be derived with any complete calculus over Boolean logic. This opens up the possibility for many different clause learning techniques for SAT solvers.

The currently best performing approach to large scale parallelization of SAT solving are *clause-sharing solvers*. Many different and diversified CDCL solvers can be started here. For more efficiency the different solver threads share their best clauses with the other solvers. Recent discoveries have shown that clause-sharing is the primary driver of scalability [18, 26]. Diversification also has a positive influence on the solving time. Schreiber and Sanders [26] have shown that the random time deviations due to the non-deterministic component of distributed programs are sufficient for search space pruning.

In *MallobSat*[26], the best SAT solver in the Cloud track of the SAT Competition since 2020[1], clause-sharing is done periodically and in an all-to-all fashion. The SAT Competition [2] is an international competition in which SAT solvers are entered each year for various disciplines and compete against each other. The Main track is the discipline for sequential SAT solvers. In the Parallel track, SAT solvers run on one node with 64 cores and in the Cloud track on 100 nodes with 16 cores each, making a total of 1600 cores. So far only SAT solvers in the Main track must provide a proof of unsatisfiability. This is due to the fact that no proof format with proof checker has yet been established for parallel clause-sharing solvers.

## 2.3 Proofs of Unsatisfiability

In sequential SAT solving, some solvers are formally verified [16, 27] and therefore do not necessarily require proofs of unsatisfiability (UNSAT proofs) for their results to be trusted. Most SAT solvers remain unverified due to their complexity, but are optimized for solving speed. Since 2016 the Main track of the SAT Competition requires solvers to produce a so-called DRAT proof [3, 12, 30]. Valid clauses can be derived with different techniques

[1] SAT Competition 2020 results: https://satcompetition.github.io/2020/results.html Accessed: 2025-08-30

[2] The International SAT Competition Web Page: https://satcompetition.github.io/ Accessed: 2025-08-30

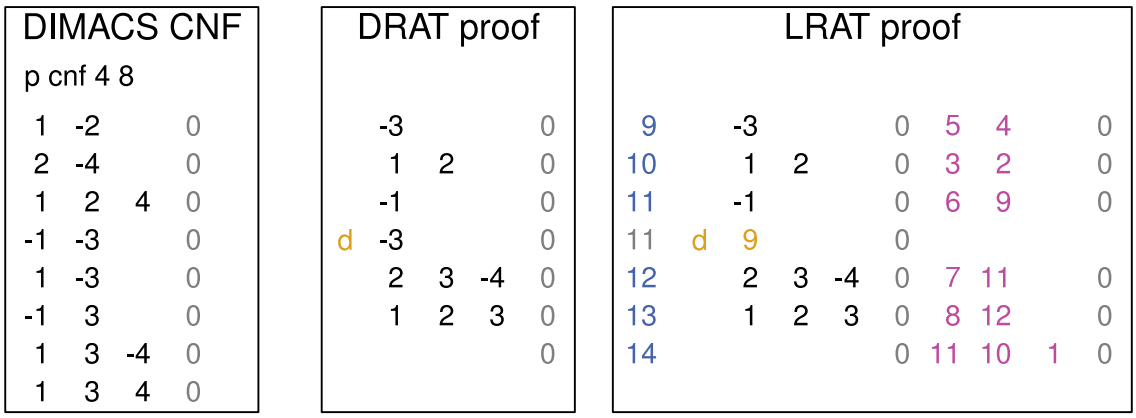


**Figure 2.1:** An example of an UNSAT instance in DIMACS CNF format with corresponding DRAT and LRAT proofs. Literals (black) are separated by zeros (grey) to create clauses. LRAT additionally stores IDs (blue) and hints (purple) in addition to DRAT. (Figure adapted from [18])

by solvers, but they must follow the *Resolution Asymmetric Tautology* (RAT) property, that is an generalization of the *Reverse Unit Propagation* (RUP) property [30]. For a clause $c$ to have the RUP property, $c$ must satisfy the following: If $c$ has all its literals assigned to `false`, then propagating these assignments through the solver's current clause set yields a conflict (contradiction). Therefore the addition of $c$ does not change the satisfiability of $F$. That is also the case with clauses which have the RAT property, not explained in detail here.

CDCL solvers can record their learned conflict clauses in chronological order as *add lines* to produce a DRAT proof. A checker then only has to check each add line for the RAT property with its current clause set. With only this information in a proof, clauses may have to be propagated through the whole clause database to check the RAT criterion. To achieve better performance and less memory consumption at runtime, CDCL solvers delete some learned clauses using heuristics. These *delete lines* must be added to DRAT proofs, to ensure that a proof checker that also has to maintain a clause database does not run out of memory. A second central motivation for deletions is a vast reduction of running times, because the checker does need to look at much smaller clause sets of clauses in each step. That is why checking a DRAT proof takes about as long as it took the solver to solve the problem instance [12].

This is alleviated by *LRAT* [7] though, due to *hints* in add lines. Every learned clause gets an unique ID and a list of dependent clause IDs, functioning as hints. Hints make LRAT checking significantly more efficient than DRAT checking, because clauses do not have to be propagated through the whole clause database anymore. These hints indicate the prerequisite clauses for the new clause; for the case of RUP, they correspond to the sequence of clauses a checker needs to look at to find the necessary conflict. An example for DRAT and LRAT is given in Figure 2.1

Checking proof lines in the same order as they were written by the solver is called *forward checking*. Now LRAT opens up the option for *backward checking*, as implemented by the *Lrat-Trim* toolchain [20], Starting with the last produced clause in the LRAT proof, what should be the empty clause, its ID is pushed to a stack. While the stack is not empty, the clauses belonging to the stack's top ID are checked for the RAT criterion with their hinted clauses. If the RAT criterion is satisfied, all hinted IDs that are not from the clause list of the original problem are pushed to the stack. If the stack is emptied without an error, the proof is considered sound. As only the clauses actually used (*core clauses*) to derive the empty clause need to be checked, backward checking LRAT proofs is much faster than checking DRAT proofs.

While LRAT proofs promise performance improvements, *CaDiCaL* [4, 20] is the only solver with native LRAT output so far. Highly optimized modern SAT solvers use a variety of pre-/inprocessing techniques for which outputting hints can be very challenging and so LRAT imposes significant burden on solver developers compared to DRAT. DRAT proofs can be converted to LRAT proofs via a first (forward) checking pass with the *Drat-Trim* toolchain [12, 30]. LRAT proofs can then be reduced in size by a factor of 2 to 3 and/or checked with *Lrat-Trim*. While Heule, Hunt, and Wetzler [12] already implemented backward checking with the *Drat-Trim* toolchain to reduce DRAT proofs in size, their backward checking on DRAT proofs is more costly than forward checking because of the computation of *core clauses* [12].

## 2.4 Related Work

In 2014, Heule, Manthey, and Philipp [11] presented a first approach to parallel proof checking for clause-sharing solvers. The main idea of their approach is to share all produced clauses of every solver with the *proof master*. The *proof master* takes care of producing the single proof file in one of two modes: directly printing all clauses to the proof file, or printing only unique clauses by detecting duplicate clauses with a counting schema. While the second mode produces much smaller proofs, the counting schema needs all learned clauses to be stored in main memory. This means that the main memory consumption can potentially grow linearly with the number of solvers in the portfolio.

For evaluation the SAT solver *Riss* [2] was modified to support parallel portfolio solving with clause-sharing and proof generation with the proof master. Their evaluation with four solver threads on an eight core cpu node showed that DRUP proofs can be generated with little impact on solving time. The proofs can be checked with *drat-trim* [12, 30] in about four times the solving time.

For large scale portfolio SAT solvers with hundreds of solver threads, this approach does not scale because of two main reasons. Firstly, communication to a single proof master will have a huge negative impact to solving time, because the volume of communication will be too large for a single process to handle. Secondly, either the main memory usage for

duplicate elimination becomes too large for one node, or the single proof file becomes huge. Even with duplicate elimination, a major problem of the approach is the huge corridor of active clauses during checking. So either checking is completely infeasible or takes huge amounts of time. With a linearly growing proof file in $p$, it is not even practical for a single shared-memory machine (see [19]).

Addressing this problem, Manthey and Philipp [15] published their approach *proofcheck* to checking DRAT proofs in parallel using eight cores in 2019. In summary, their approach is based on backward checking, in which the clauses to be checked are colored. At startup, a single checker marks clauses until a preset threshold is met. These clauses are then distributed to the workers based on their colors. In the main loop, the worker who colors a clause checks that clause. Due to race conditions, another worker can overwrite the color, but with this approach, clauses are checked at least once. Also, no locking is required. If a worker runs out of clauses to check, it terminates. When all workers have terminated gracefully (i.e., they have checked all marked clauses), the proof is verified. While *proofcheck* could not outperform *drat-trim* [12, 30], it still showed self-speedup with eight cores at 50 % efficiency.

To reduce size and therefore checking time of distributed unsatisfiability proofs, Michaelson et al. [19] introduced an certified-UNSAT mode to the to the massively parallel clause-sharing solver MallobSat [21, 26]. At solving time each solver prints an partial proof in LRAT [7] format. So far only CaDiCaL [4] supports LRAT output without postprocessing, therefore MallobSat only uses CaDiCaL solvers in its portfolio in certified-UNSAT mode. If a solver reports the instance is UNSAT, the distributed proof production starts with the partial proof files. Like other parallel proof production approaches, only LRUP is supported so far.

Intuitively, Michaelson et al.'s proof production approach entails to parallelize LRAT backward checking over all partial proof files, thus essentially "rewinding" solving, and to merge all proof lines identified as relevant. Each partial proof is merged considering only the clauses that are necessary to produce the empty clause (core clauses). Core clauses are recursively provided as hints by the LRAT format for each core clause. If a hint refers to an original problem clause, the clause is not considered a core clause and is dropped. Whenever a core clause is encountered as a hint for the first time, a deletion line can be inserted to reduce memory usage of a checker later. Core clauses that are provided from another solver by clause-sharing (identified by their ID), are send to the corresponding merger as hint. Mergers are arranged in a k-ary tree. For $k = 2$ a merger is responsible for:

- Merging two partial proofs into a single stream of LRAT lines.
- Merging the resulting stream with two incoming streams into a locally final stream.
- Passing this locally final stream on to the merger of the next level.

Mergers at leaf level do not have (to merge) incoming streams and the merger at root level writes its final stream to disk as monolithic proof. Each merger terminates when no core clause is left. Lastly, because it was read backwards the resulting proof file is reversed to restore the correct order.

In evaluation Michaelson et al. [18] showed that proof production takes about 1.8 times, and checking takes about 2.1 times the solving time at cloud scale (1600 threads), making this the first feasible approach for obtaining proofs from clause-sharing solvers. Some downsides remain: proof production and checking still require a multiple of solving time. Given that the proof is produced by many solvers but checked by only one checker, this is inevitable. Also it can happen, that the amount of simultaneously active clauses in the clause set is so huge, that there is realistically no single node in existence with sufficient amounts of main memory.

The most recent approach to trusted solving, ImpCheck [25], like all parallel proof approaches, supports LRUP. ImpCheck supervises the integrity of the solving process on-the-fly, with only a small impact to performance, but no persistent proof is generated. Each solver thread is assigned an ImpCheck process that monitors its execution for correctness. To do this, each solver thread must send each learned clause $c_i$ to its ImpCheck process, including its ID, literals, and hints. Each ImpCheck process also maintains a clause set, which is used to check whether $c_i$ can be learned with it. If $c_i$ satisfies the RUP property, it is added to the clause set. Clauses that a solver thread intends to share must first be sent to its associated ImpCheck process. These clauses are also checked for the RUP property, and a cryptographic *fingerprint* is generated for each. A fingerprint is a hash of the clause, computed using the clause ID, literals, and a secret key known only to ImpCheck processes. This fingerprint is sent back to the solver thread, which can now share the clause with its fingerprint. The ImpCheck processes of the receiving solver threads can then verify the clause and its fingerprint. If any RUP check fails or a fingerprint cannot be validated, ImpCheck declares the entire solving process invalid.

In summary, there is the possibility to run massively parallel clause-sharing solvers with ImpCheck [25] or to generate persistent monolithic proofs at the expense of scalability with [18]. The monolithic proofs are not always checkable due to the potentially huge amount of simultaneously active clauses and do not scale due to linear components during proof generation and linear checking. ImpCheck scales very well due to the one-to-one relationship from solver thread to checker, but does not generate persistent proofs at all.

# 3 Distributed Proof Format

A scalable way of producing and checking proofs of unsatisfiability requires a new proof format. Our goal is to achieve *the best of both worlds*, that is, a *persistent* proof that can be *generated* and *checked* without sequential bottlenecks. In this chapter we first outline how a scalable checker can be designed in Section 3.1. We also examine possible errors in a proof that a checker should detect in Section 3.2. Section 3.3 formally describes our distributed PalRUP UNSAT proof format. Finally, we elaborate in Section 3.4 how we generate PalRUP UNSAT proofs using our extension for ImpCheck [25].

## 3.1 Overview

As discussed in Section 2.4, there are currently two feasible approaches for trusted SAT solving with clause-sharing solvers. First, *MallobSat* [26] can generate monolithic proofs. However, generating and checking these proofs is computationally expensive. Moreover, some proofs cannot be checked on any realistically existing single node due to memory limitations. This is because memory consumption scales with the number of compute nodes, whereas checking is performed on only a single node. Second, *ImpCheck* [25], an extension for MallobSat, uses on-the-fly checking to ensure trusted solver execution. While this approach incurs only a small overhead over the solving time, no persistent proof is generated.

To develop a procedure that generates persistent proofs with minimal overhead and checks them faster than the corresponding instances can be solved, we propose the following (see Figure 3.1): Each solver thread records partial proof files in LRAT format (see Section 2.3). Thus, after solving, one checker can run per proof file in the *pre-check step*. Assuming, that the imported clauses are valid, each partial proof can be checked sequentially. The challenge lies in validating the imported clauses. For this purpose, the imported clauses are recorded by each checker in a new *export file*. We must check that each clause, including its ID and literals, was indeed learned by its original solver, which is identified by the clause ID. The clauses are redistributed during the *route step* and combined into new IMPORT files. Each IMPORT file contains all clauses learned and shared by the same original solver. In the *resolve step*, each resolver scans a partial proof to ensure that every clause in the corresponding IMPORT files is found in the partial proof. To ensure end-to-end validity, fingerprints are added to critical files. Furthermore, clauses must be sorted by ID to deduplicate them during the *route step* and speed up the *resolve step*.

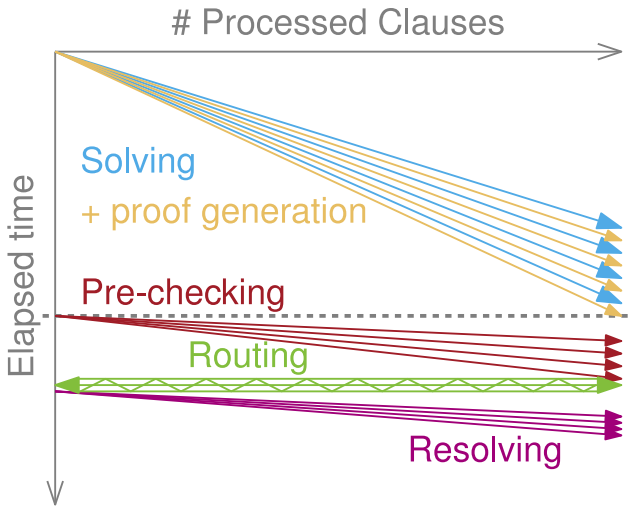


**Figure 3.1:** Abstract workflow of PalRUP proof generation and checking. While solving we generate a PalRUP proof (based on multiple files) on-the-fly. This proof can be checked in three steps: *pre-check*, *route* and *resolve*. The checking workflow is faster than solving of the corresponding instance. (Figure adapted from [25])

## 3.2 Integrity Considerations

In sequential SAT solving, a DRAT or LRAT proof that documents the solver's reasoning is sufficient to prove unsatisfiability. With clause-sharing solvers, additional complexity is introduced to support sharing, since communication libraries may contain errors. Shared clauses are assumed valid by receiving solver threads during solving, without additional hints. Requiring thorough clause checks of receiving solvers would reduce their performance and is not practical due to lack of information about other solvers' learning history. Even if a solver thread were to validate the imported clauses, a proof checker must independently check them after the solving process, as an unverified solver cannot be trusted. Bugs such as buffer overflows, race conditions, memory errors, or communication errors can compromise the validity of shared clauses. Messages exchanged between solver threads can be modified due to these errors [25]. Even purely fictional clauses cannot be excluded.

A clause $c$ is considered invalid if it does not arrive at receiving solvers exactly as it was learned by the sending solver, determined from the clause ID ($ID_c$). When solvers share a clause $c$, they share at least the literals of $c$ (denoted as $c$) and should also share its $ID_c$. Changes in the ID would not necessarily affect the correctness of an UNSAT result, but allowing uncontrolled modification of IDs would prevent efficient use of hints in a proof. Hints are important for fast RUP checking as discussed in Section 2.3 with the LRAT format. Literals of a shared clause are susceptible to many different types of errors. For example, one or more literals may be modified, or the count of literals in a clause may change due to additional or missing literals.

Another problem is the *self-correcting loop scenario*. In this scenario, a solver $S_1$ shares a clause $c$ with solver $S_2$. At some point in the sharing process, $c$ is modified, and $S_2$ receives this modified version $c_w$. Subsequently, $S_2$ can use $c_w$ to learn $c_{w'}$ and share $c_{w'}$ with $S_1$. Finally, $S_1$ uses $c_{w'}$ to learn $c_w$.
As a result, during proof checking, an equality check of the clause's literals is not sufficient to determine whether $S_1$ has learned $c_w$ and could have shared it. In the case of our scenario above, this check would be valid, even though an error occurred. An additional check of the clause ID is also not sufficient, because that would require trust in an unverified solver's ID generation. The addition of explicit clause export statements would merely delay the issue, because $S_1$ can share its newly learned clause $c_w$. Without additional restrictions on the format, dependencies of clauses must be tracked between all solvers to ensure correctness in the sharing process.
A solution that does not require cross-solver dependency tracking and allows for simple ID and literals checking is to enforce strictly larger IDs than their dependencies for new clauses, as described in Section 3.3.

## 3.3 PalRUP Format

Unsatisfiability proofs can easily take up terabytes of space. As shown in Section 2.3 discarding all clauses that are not *core clauses* with backward tracking can reduce the size of the proof, with the drawback of being time-consuming. To accelerate the checking process, the idea is to store each partial proof (from each solver thread) in its own file. A partial proof is checked under the assumption that the imported clauses from other solvers are correct. In subsequent steps, all imports must be resolved to ensure valid sharing. The proposed *Parallel LRUP* (*PalRUP*) format supports this approach by extending *LRAT* (see Section 2.3) with additional import statements and constraints. The format is optimized to support fast checking of correct proofs. If a discrepancy is detected during checking, the checker terminates immediately. The syntactic definition of a partial proof file in *PalRUP* is provided by a context-free grammar in EBNF [14, 31], as shown in Figure 3.2.

**PalRUP EBNF**

⟨*proof*⟩ ::= ⟨*line*⟩*
⟨*line*⟩ ::= (⟨*rat*⟩ | ⟨*delete*⟩ | ⟨*import*⟩) '\n'
⟨*rat*⟩ ::= ⟨*id*⟩ ⟨*clause*⟩ '0 ' ⟨*idlist*⟩ ⟨*res*⟩* '0'
⟨*delete*⟩ ::= ⟨*id*⟩ 'd ' ⟨*idlist*⟩ '0'
⟨*import*⟩ ::= ⟨*id*⟩ 'i ' ⟨*id*⟩ ⟨*clause*⟩ '0'
⟨*res*⟩ ::= ⟨*negNum*⟩ ⟨*idlist*⟩
⟨*idlist*⟩ ::= ⟨*id*⟩*
⟨*id*⟩ ::= ⟨*posNum*⟩
⟨*lit*⟩ ::= ⟨*posNum*⟩ | ⟨*negNum*⟩
⟨*clause*⟩ ::= ⟨*lit*⟩*
⟨*digitWithoutZero*⟩ ::= #'[1-9]'
⟨*digit*⟩ ::= #'[0-9]'
⟨*posNum*⟩ ::= ⟨*digitWithoutZero*⟩ ⟨*digit*⟩* ' '
⟨*negNum*⟩ ::= '-' ⟨*posNum*⟩

**Figure 3.2:** EBNF grammar of PalRUP. PalRUP extends LRAT with minimal changes. In addition to *add* and *delete* lines, *import* lines introduce clauses from clause-sharing as new axioms. Like *delete* lines, *import* lines begin with the same ID used to start a *RAT* line. This allows a checker to verify whether the ID has increased and determine whether a literal is expected next or a character such as 'd' or 'i', indicating a *delete* or *import* line. *Import* lines include the shared clause ID and its literals.

PalRUP proofs need additional semantic constraints that are not defined by the EBNF:

(i) Every generated $ID_c$ of a solver thread with rank $p$ in an environment with $P$ clause-sharing solver threads must satisfy the following condition:

$$ID_c \equiv p \pmod{P} \tag{3.3.1}$$

(ii) The ID for a clause in local clause derivations must be greater than the ID of the previous clause. This ensures clauses in partial proofs remain sorted by ID.

(iii) The $ID_c$ of any new clause $c$ must be greater than all hints $ID_{h_i}$ for that clause.

(iv) With $n_o$ original problem clauses, any valid $ID_c$ must be greater than $n_o$.

(v) Literals in clauses are sorted in ascending order based on their integer value.

(vi) IDs are stored as 64-bit unsigned integers and literals as 32-bit signed integers.

(vii) Each proof file must be stored in `proof_path/`$p$`/out.palrup` for solver $p$.

Constraint (i) ensures that every clause can be tracked back to its original solver thread and therefore partial proof file after clause-sharing.

Thus, shared clauses can be resolved in subsequent checking steps. To support efficient resolving with linear scanning, constraint (ii) requires that RUP lines remain sorted. Correct unsatisfiability proofs must be loop-free. By combining constraints (ii) and (iii), the loop-freeness of the proof can be easily ensured by tracking the last used $ID_c$ and comparing $ID_{h_i}$. Loop-freeness is an essential condition to prevent the *self-correcting loop scenario* described in Section 3.2. IDs up to $n_o$ are used for original problem clauses, hence the need for constraint (iv). That does not mean that $n_o + 1$ is a valid first ID for a rank $0$ solver thread, because (i) still has to be met. At some point during proof checking, clauses must be checked for equality. While the ID check is a simple integer comparison, clauses are considered equal, if they contain exactly the same literals. Without constraint (v), the literals of the first clause $c_1$ must be searched in the second clause $c_2$. In a correct proof $c_1$ is always equal to $c_2$, therefore $|c_1| = |c_2| = |c|$. Even with deletion of found literals in $c_2$, this would require $\mathcal{O}(|c|)$ time per search operation for each of the $|c|$ literals. While it is possible to sort literals in amortized linear time (e.g., using radix sort [29]), this would introduce unnecessary complexity to the checking process. By assuming sorted literals, search operations can be reduced to a single index increment operation and therefore be done in $\mathcal{O}(1)$. Overall, this means that comparisons can be made in $\mathcal{O}(|c|)$ time. Constraint (vi) assigns 64-bit ID space to compensate for the rapid growth of IDs due to (ii) and (iii). (iii) in particular causes skipping of ID ranges, whenever shared clauses of other solvers with greater IDs are used to learn new clauses. To generate IDs that satisfy all three constraints (i), (ii) and (iii) recursive Formula 3.3.2 can be used.

$$\begin{aligned} s &= \mathit{max}(H_{\mathit{max}}, \mathit{ID}_{\mathit{old}}) \\ o &= 1 + s - \mathit{ID}_{\mathit{old}} \\ t &= p - (o \mod p) \\ \mathit{ID}_{\mathit{new}} &= \mathit{ID}_{\mathit{old}} + o + (t \mod p) \end{aligned} \tag{3.3.2}$$

$o$ is an *offset* component to account for (ii) and (iii). It is only necessary to calculate $ID_{new}$ from the greater number of maximum hint ID ($H_{max}$) and last used ID ($ID_{old}$), so $s$ is set to the maximum of these two. $t$ is calculated as the difference from the next valid number. $(t \mod p)$ only has an effect, if $(o \mod p) = 0$, to avoid skipping a valid ID by $p$ unnecessarily.

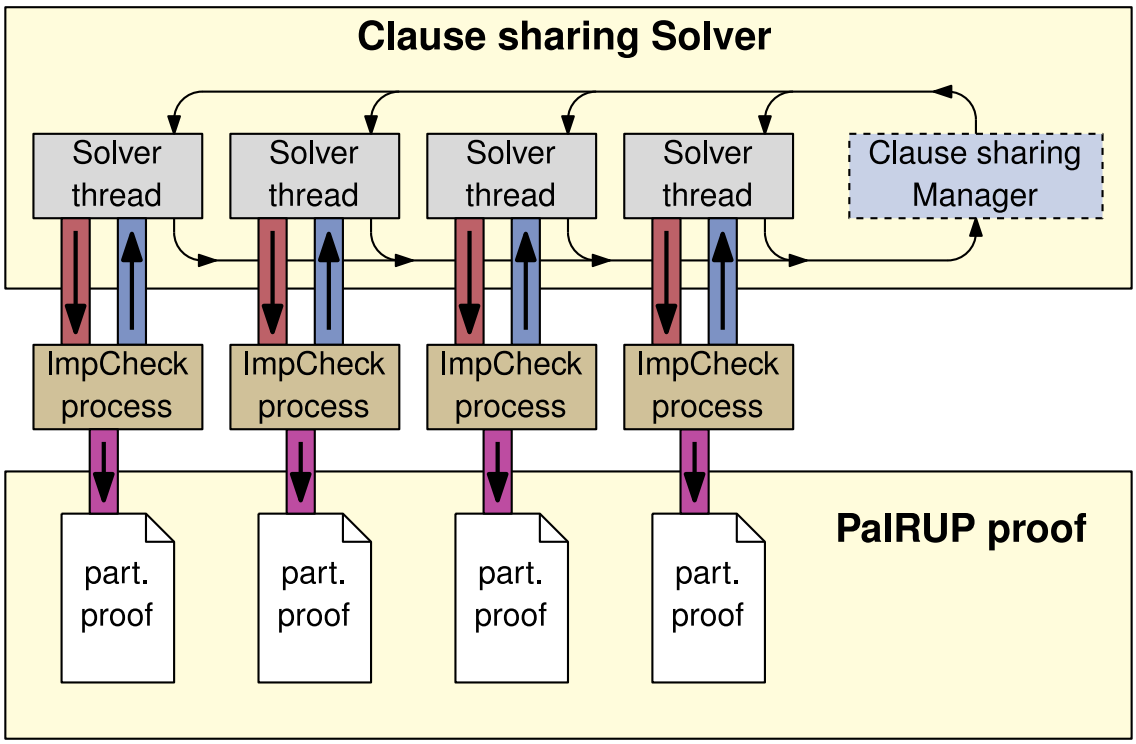


**Figure 3.3:** Clause-sharing solver with ImpCheck in proof producing mode. Each solver thread has an ImpCheck child-process and sends its learned clauses over the red pipe. ImpCheck answers (new ID + fingerprint) are sent back over the blue pipe. All received clauses are saved to a partial proof file (part. proof) for each solver/checker. (Figure adapted from [25])

## 3.4 Producing PalRUP Proofs

After defining a new format for UNSAT proofs for clause-sharing solvers, the next step is to produce PalRUP proofs with one of the existing clause-sharing solvers. *MallobSat* already has support for proof generation with its certified-UNSAT mode and live checking with *ImpCheck* as discussed in Section 2.4. This section describes how ImpCheck is used to produce PalRUP proofs (see Figure 3.3).

While it is possible to implement the necessary changes for PalRUP proofs directly into a solver, the usage of ImpCheck has advantages in terms of portability. ImpCheck is built to support live checking with any SAT solver that can send LRUP proof lines to it over a pipe. One ImpCheck process $I_r$ has to be started for every solver thread $S_r$ and is responsible to monitor its corresponding solver thread $S_r$ for correct execution. Every new clause $S_r$ learns in execution must be sent with hints to $I_r$ which checks if the learned clause is RUP. Also every clause that should be sent for clause-sharing needs to get a fingerprint from ImpCheck before it can be sent. These fingerprints are checked by ImpCheck processes of receiving solver threads upon import. ImpCheck can already record LRUP with additional

import lines for clauses from clause-sharing. Although it matches the format, the additional requirements in Section 3.3 for efficient proof checking are not met. To generate valid PalRUP proofs, some changes were made to ImpCheck and MallobSat.

MallobSat is capable of running ImpCheck with CaDiCaL [4] solver threads. CaDiCaL supports LRAT output with no restrictions [20]. The setup of CaDiCaL and ImpCheck is managed by MallobSat and needs only minor modifications. ImpCheck has to track IDs and modify them accordingly. Initially, logging was directly integrated into old ImpCheck's (oIMP) pipe reading function. Using this approach the main logic of the checker is cleaner, but it is impossible to associate IDs with hints that way. Recording proofs in ImpCheck's new version (nIMP) is done explicitly in the main logic loop after new IDs for learned clauses have been calculated. In MallobSat, IDs already are assigned in a modulo schema, but the starting ID of the first clause learned by $S_0$ is $n_o + 1$. This results in a fixed offset $\delta_o$ for every ID such that $ID_c \equiv p + \delta_o \pmod P$. To fix this, nIMP calculates and adds $\delta = P - \delta_o$ which results in cancellation of $\delta_o$ with respect to $\pmod P$.

$$p + \delta_o + \delta \equiv p + \delta_o + P - \delta_o \equiv p + P \equiv p \pmod P \tag{3.4.1}$$

This results in a small gap of at most $P$ to starting IDs. Compared to the ID shifts needed to give newly learned clauses larger $ID_c$ than $H_{max}$, this gap is negligible.

Solvers are not notified about ID corrections, as this would require extra support from the solver. If solvers would have to wait for an answer from ImpCheck for every learned clause, this would have a huge negative impact on solving performance. However, the lack of this support means that a data structure must store the information for translating original solver IDs $ID_o$ to corrected proof IDs $ID_n$. In nIMP, an offset $o$ is tracked, which is increased if necessary to guarantee $max(ID_h) < ID_o + o$. The following applies: $ID_n := ID_o + o$ and given that the solver automatically increases $ID_o$ while $o$ is never decreased, (ii) is not violated. In the case of hints, it is not clear afterwards which offset was applied when they were learned, as they are only sent by the solver with the original ID. To determine $max(ID_h)$, the corrected ID of each hint must therefore be looked up in the translation data structure. As with the clause database, entries are deleted from the translation table when a solver sends delete lines. This means that the translation table only takes up a linear amount of space in relation to the clause database.

In addition, an ID translation is also required if a solver intends to send a clause to clause-sharing. For each clause to be shared, ImpCheck must generate a fingerprint for the clause with its ID. The ID is translated before the fingerprint is generated so that imported clauses are written to the receiving solver's clause databases and proofs with the correct ID. To make this possible, we extended MallobSat so that not only fingerprints from ImpCheck can be received, but also translated IDs. However, this causes issues with MallobSat's clause-sharing.

In the context of this work, we treat MallobSat's clause export, sharing, and import logic to be a single abstract module we call the Clause-Sharing Manager (CSM). CSM assumes that $ID_c \equiv p + \delta_o \pmod P$ applies to all clauses $ID_c$ of $S_r$. Due to an optimization, the clause is not sent to $S_{p+\delta}$ because the CSM assumes that $S_{p+\delta}$, as the producer, has the clause itself anyway.

This can lead to reduced overall performance of the clause-sharing solver, but a more significant issue is that CSM $S_p$ re-sends the clause with a new ID. This does not cause a conflict in MallobSat. There are therefore two different clauses with different IDs and the same literals. MallobSat deletes the newly duplicated clause with $ID_{c_t} := x \cdot p + \delta_o + \delta + o$ at some point and sends a delete line to ImpCheck. The old clause with $ID_c := x \cdot p + \delta_o$ can still be used for learning new clauses and ImpCheck records $ID_c + \delta + o$ as a hint in the partial proof file. If such a proof is checked, a clause with $ID_{c_t}$ is learned and later also imported. In the optional *lenient* mode of ImpCheck, there is no error if a clause with the same ID and the same literals is imported again. It is only deduplicated then. As soon as the imported clause is deleted during proof checking, there is no longer a clause $ID_{c_t}$. Because $S_r$ only deletes an imported duplicate while solving and not the originally learned clause, $ID_{c_t}$ may still appear in hints in the proof and is deleted again later. If a non-existent hint is used, the proof check fails as expected. This problem must be avoided without modifying the CSM in order to remain as compatible as possible. To solve this problem, it is sufficient to send the clause with $ID_{c_t} - \delta$ to CSM. When importing clauses using ImpCheck, each clause is therefore also saved in the translation table with offset $\delta$. Now the CSM works as intended and ImpCheck can continue to work with the same IDs as MallobSat for live checking. The adjusted IDs therefore only have an effect on the partial proof file and imported clauses, but not on the internal ID assignment of a solver thread's locally learned clauses.

# 4 Distributed Proof Checking

In this chapter we design and analyze the *PalRUP checker* based on ImpCheck that can now check PalRUP proofs described in Section 3.3. First we discuss the aspects of distributed computing we have to keep in mind when designing our algorithm for HPC clusters in Section 4.2. Next we provide an overview of the whole checking procedure end-to-end in Section 4.1. In Section 4.3, each subsection covers a step of the checking procedure in detail. Section 4.4 shows how we use different kinds of *fingerprints* to ensure integrity of files. We examine how the correctness of the checking process is ensured in Section 4.5 and analyze expected runtime and memory consumption of the algorithm in Section 4.6.

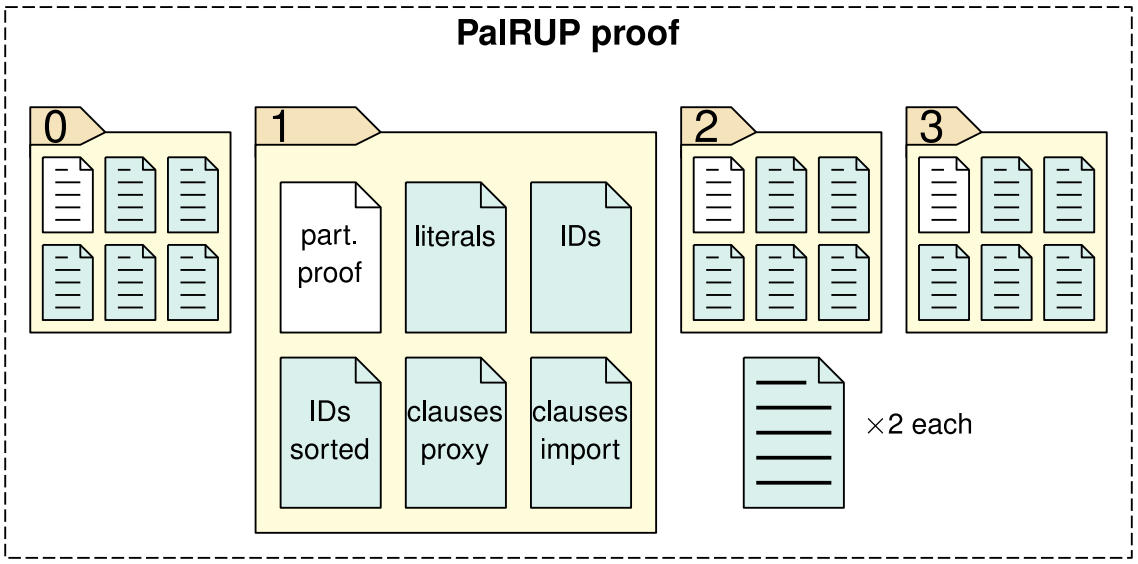


**Figure 4.1:** PalRUP directory structure example for $P = K = 4$ solver threads. Each directory is named using the rank $r$ of its corresponding solver thread. A PalRUP proof by itself only consists of the partial proof files (white). For proof checking, $\sqrt{K}$ files of each type of communication related file (blue) are created in the same $K$ directories, or optionally under a different path with the same name scheme for directories.

## 4.1 Overview

In this section, we present our novel proof checking workflow *PalRUPCheck* and the flow of information between the individual checking stages. As already described in Section 3.1, PalRUP is based on the stages *pre-check* (*CHK*), *route* (*ROT*) and *resolve* (*RSL*). To improve scalability, memory usage, and modularity, the *sort* (*SRT*) and *assemble* (*ASM*) stages have been added.

In the following we assume that $P$ solver threads $S_p$ have generated a PalRUP proof and have written it to disk in $P$ directories $D_p$ into partial proof files PALRUP$_p$. The instances of each step are *CHK*$_p$, *SRT*$_k$, *ASM*$_k$, *ROT*$_k$ and *RSL*$_p$. While $P$ instances *CHK*$_p$ and *RSL*$_p$ are launched, the next square number

$$\begin{aligned} K &:= \left\lceil \sqrt{P} \right\rceil^2 \\ R &:= \sqrt{K} \end{aligned} \tag{4.1.1}$$

is the relevant size for *SRT*$_k$, *ASM*$_k$ and *ROT*$_k$, because it describes the communication square in which the *route* step coordinates the file based all-to-all communication. An example of the directory structure required for communication is given in Figure 4.1.

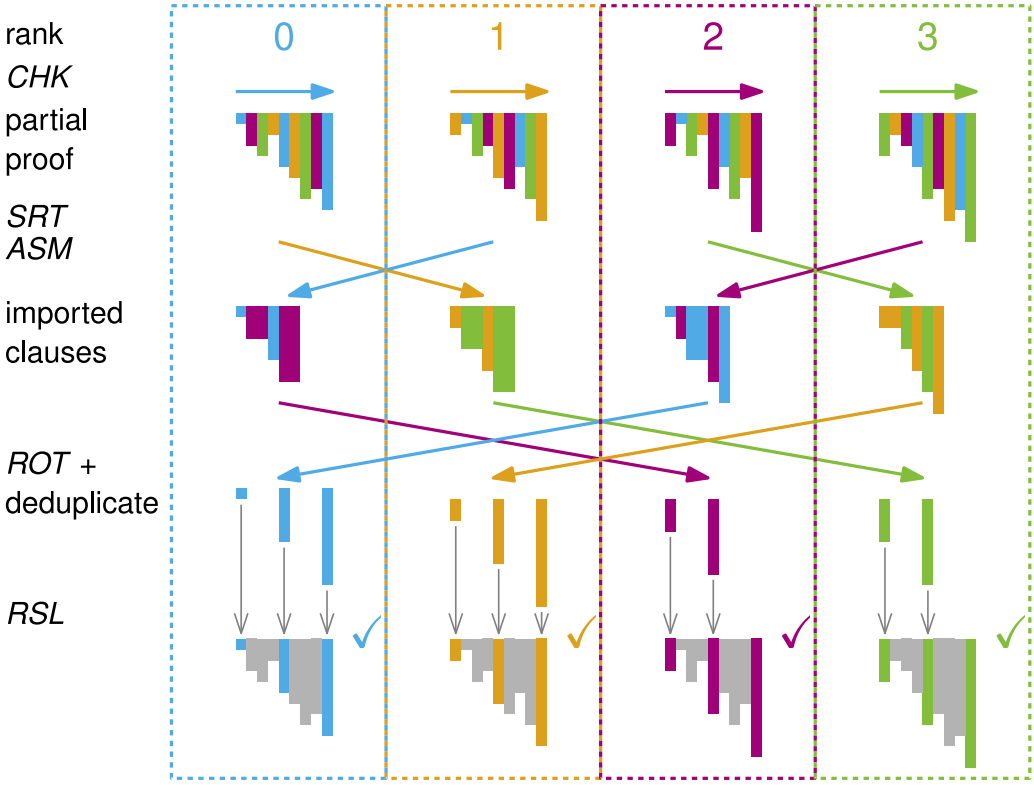


**Figure 4.2:** Overview of *PalRUPCheck*: Four ranks of checkers (*CHK*, arranged vertically) each check a partial proof. Next, the imported clauses are sorted (*SRT*) and assembled (*ASM*) for each rank, and then sent to their proxy rank. In *ROT*, each rank delivers clauses to the rank that generated them. Finally, *RSL* ensures every clause is found in its original partial proof. Ranks 2 and 3 did not share all of their produced clauses, therefore less imported clauses are checked in *RSL*.

Next, we examine how the different steps of PalRUPCheck work and interact with each other (see Figure 4.2).

First, a $\boldsymbol{CHK}_p$ process is started for each PALRUP$_p$. $CHK_p$ checks PALRUP$_p$ line by line, making sure that the RUP property is satisfied for all learned clauses and that the IDs satisfy the requirements of Section 3.3. Each clause from an import line is learned as a new axiom and additionally stored buffered in one of $R$ adjacency arrays $A_x$. Here, $x$ is determined by the routing procedure, which is described in detail in Section 4.3.4.

It suffices for now that all $CHK_p$ and $RSL_p$ are assigned to cells in a $R \times R$ grid, on which the imported clauses of $ROT_k$ are first collected on the x-axis and delivered on the y-axis (see Figure 4.3). For $A_x$ we save a file ID$_x$ with the IDs + indices and a file L$_x$ with literals of the clauses. ID$_x$ is then sorted by ID in $\boldsymbol{SRT}_k$ and saved as SID$_x$. Then $\boldsymbol{ASM}_k$ assembles PROXY$_x$ from SID$_x$ and L$_x$, a file in which import lines are sorted by ID in ascending order.

**Algorithm 1:** MapReduce clause routing example

```
1  def map(KEY, VALUE)
       /* KEY: clause ID                                          */
       /* VALUE: clause literals                                  */
2      EmitIntermediate(KEY, VALUE)
3  end
4  def reduce(KEY, VALUES)
       /* KEY: clause ID                                          */
       /* VALUES: a list of clauses                               */
5      BASECLAUSE ← VALUES[0]
6      foreach c_i in VALUES[1 :] do
7          if c_i ≠ BASECLAUSE then
8              Throw(Error)
9          end
10     end
11     Emit(BASECLAUSE);
12 end
```

$\text{PROXY}_x$ is now read line by line by $\boldsymbol{ROT}_k$ and compared with $\text{PROXY}_x$ of other checkers for deduplication. Each clause is then written once to an $\text{IMPORT}_y$ file, so that it is located at $D_p$ with its original $\text{PALRUP}_p$, which must have produced it. Another way to describe the same end result for routing is using *MapReduce* [9]. Example map() and reduce() functions are given in Algorithm 1.

However, a large general parallelization framework would complicate a later formally verified implementation, which is why a use-case-specific algorithm was designed. Similarly, we avoided using network-based communication libraries like MPI to keep the technology stack of our checking framework at a minimum, only dealing with files (just like sequential checkers).

After routing, $\boldsymbol{RSL}_k$ can search for all clauses from the $R$ $\text{IMPORT}_y$ files in $D_p$ in $\text{PALRUP}_p$ and confirm their existence. If the empty clause was learned in *CHK* and all clauses were found in *RSL*, the proof is checked.

## 4.2 Distributed Computing Considerations

When developing applications for High Performance Computing (HPC) clusters, inter-node communication is a critical bottleneck. Even though many HPC clusters have a fast Infiniband interconnect, communication should be treated as exceptionally slow and having high-latency. For *PalRUPCheck* the imported clauses, which must be traced back to their original partial proof file, would not fit into main memory for longer solving runs. This memory pressure is avoided with our file-based communication (see Section 4.1).

Most HPC clusters offer a shared network storage[1][2] that can be accessed from every node. For scaling reasons, a parallel file system (e.g. Lustre, GPFS, or BeeGFS) is deployed to manage the network storage with multiple disks. Such file systems are highly optimised for workloads where many clients read large files from different directories. However, concurrent directory access can become a bottleneck, due to slower metadata operation performance.

To avoid competitive directory access, files should be created in separate subdirectories. With $P$ partial proof files we anticipate $P^2$ files because every solver thread is expected to import clauses from every other solver thread. $P$ files can naturally be split in $P$ directories, one for each rank. Still $P$ files per folder does not scale well, therefore a square routing schema like in [22] with only $2 \cdot \sqrt{P}$ files per folder is used (see Section 4.3.4).

In addition to at least one globally visible file system, each node has a local storage. A checker should support usage of different storage locations, to remain independent of specific cluster configurations and limitations. Because of that, PalRUPCheck supports distinct locations for the partial proof files and communication files.

[1] HoreKa documentation: https://www.nhr.kit.edu/userdocs/horeka/filesystems/
Accessed: 2025-08-26

[2] SuperMUC-NG dokumentation: https://doku.lrz.de/file-systems-of-supermuc-ng-10746566.html
Accessed: 2025-08-26

## 4.3 Algorithm Details

This section describes the details and design decisions of each of PalRUPCheck's five steps.

### 4.3.1 Pre-Check

Each of the $P$ pre-checking processes $CHK_p$ is started with the following arguments:

- `formula-path` to load the original problem clauses at initialization.
- `solver-id` $p$ to indicate which solver thread's PALRUP$_p$ file has to be checked.
- `num-solvers` $P$.
- `proofs-path-in` the path to the parent directory containing PALRUP$_p$.
- `proofs-path-out` the path to the parent directory where $ID_x$ and $L_x$ should be written.
- `buffer-KB` the size of read and write buffers for PALRUP$_p$ and each $ID_{x,y}$.
- `redistribution-strategy` as 1 for direct all-to-all quadratic communication or 2 to use the *route* step and only write $R$ files (see Equation (4.1.1)).

After the original problem clauses are loaded to the clause set and all other data structures are initialized, a modified version of ImpCheck is used to check PALRUP$_p$. PalRUP currently supports only an uncompressed binary format. For the fingerprint of the whole PALRUP$_p$, we use SipHash [1]. *Commutative hashing* (see Section 5.2) is used for each literals-file (L$_x$).

Every line of PALRUP$_p$ starts with a *type character* to indicate:

- `a` **add line** for a newly learned clause.
- `i` **import line** to introduce a new imported clause.
- `d` **delete line** with IDs of clauses that can be deleted from the clause set.
- `T` to **terminate** the pre-checking process at the end of the file.

Add lines continue with the clause's $ID_c$. $ID_c$ must be greater than the last ID used and satisfy $ID_c \equiv p \pmod P$. The literals are then read, and the clause is checked for the RUP property using the current clause set. For import lines, $CHK_p$ imports the clause unchecked as a new axiom. To check them later, imported clauses are send to PalRUP's *Importer*. Delete lines instruct $CHK_p$ to remove clauses from the clause set. After termination with `T`, each fingerprint is written to its corresponding file (see Section 5.2).

PalRUP's *Importer* inserts clauses based on their IDs into the corresponding $A_x$ list. See Section 5.3 for details.

Upon graceful termination (reading `T`) and confirming no errors in the proof, all $A_x$ are flushed to disk, splitting them into ID$_x$ and L$_x$. The associated fingerprints of $A_x$ are attached to L$_x$, since attaching them to ID$_x$ would cause errors in the *sorting* step.

### 4.3.2 Sort

Sorting is performed using *bsort*[3]. Bsort is an in-place radix sort for fixed-width binary objects and can sort files that exceed main memory size of the executing machine. For each $\mathrm{L}_x$ file generated in *CHK*, a bsort process is spawned, making a total of $K \cdot R$ processes. Because the data to be sorted is distributed across many files, the sorting step can be parallelized, although bsort itself is sequential.

The arguments are:

- `k` as $8$ bytes for 64-bit unsigned integers IDs. Only these 8 bytes are used as the sortable big-endian key. Because of little-endian data types on x86_64 machines, the IDs are reversed byte-wise before writing to disk.
- `r` as $20 = 8{+}8{+}4$ bytes for the object size (record size) with ID, Index, count_literals.
- `i` as input file ($\mathrm{ID}_x$).
- `o` as output file ($\mathrm{sID}_x$).

Validation of the sorting step is unnecessary, as sorting errors will only cause the proof check to fail. Assuming fingerprints do not collide (vanishingly small probability) in the previous and subsequent steps, errors in the sorting algorithm cannot result in an invalid proof being accepted. In the next step, the output files $\mathrm{sID}_x$ are assembled with $\mathrm{L}_x$ to generate a file of sorted import lines.

### 4.3.3 Assemble

To create linearly readable files again, $\mathit{ASM}_k$ combines all $\mathrm{sID}_x$ files with the corresponding $\mathrm{L}_x$ files in folder $k$.

The start parameters for $\mathit{ASM}_k$ therefore are:

- `rank` $k$, which determines the working directory under *path*.
- `num-solvers` $P$, which is only used to calculate $K$ and $R$.
- `path`, the path to the parent directory where $\mathit{SID}_x$ and $L_x$ are located; the resulting $\mathrm{PROXY}_x$ is written to the same directory.
- `buffer-KB` the size of read and write buffers for each file.
- `redistribution-strategy` as $1$ for direct all-to-all quadratic communication or $2$ to use the *route* step and only write $R$ files.

As soon as all file handles are initialized, the *index objects* {$\mathit{ID}_c$, *startIndex*, *countLiterals*} sorted by $\mathit{ID}_c$ are loaded in sequence. Random read accesses to $L_x$, based on *startIndex*, are then performed to load the literals. To efficiently assemble the PROXY files, we use a custom *buffered file reader*. Further details on the *buffered file reader* can be found in Section 5.5.

[3] bsort Github: https://github.com/adamdeprince/bsort
Accessed: 2025-08-26

### 4.3.4 Route

The *route* step is mainly based on the message indirection method of Sanders and Uhl [22]. Application-related changes have been added, such as deduplication of import lines and handling the case of a non-square $P$.

*ROT*$_k$ requires the following parameters:

- `rank` $k$ from which $x$ and $y$ are calculated.
- `num-solvers` $P$, which is only used to calculate $K$ and $R$. Even if $P$ is less than $K$, $K$ instances of *ROT*$_k$ are initialized.
- `path`, the path to the parent directory where PROXY$_x$ is located and IMPORT$_y$ is written.
- `buffer-KB` the size of read and write buffers for each file.
- `redistribution-strategy` as 1 for direct all-to-all quadratic communication or 2 to use the *route* step and only write $R$ files.

To calculate the coordinates $x$ and $y$ for any solver $p$,

$$\begin{aligned} x &:= p \bmod R \\ y &:= p \div R \end{aligned} \tag{4.3.1}$$

can be used with $\div$ as integer division. For *ROT*$_k$, $k$ is defined as $k = y_{src} \cdot R + x_{dst}$. Since *ROT*$_k$ distributes the clauses within a column, we use Equation (5.3.1) to define $k_{imp}$ as follows:

$$k_{imp} := y_{dst} \cdot R + x_{dst} \tag{4.3.2}$$

This definition applies for $y \in [0, R-1]$, allowing buffers for all IMPORT$_y$ to be created during initialization. Directory $k$ contains the PROXY$_x$ files from PALRUP$_p$, where $p = y_0 \cdot R + x$. *ROT*$_k$ places the import lines from PROXY$_x$ into directory $k_{imp}$, specifically in file IMPORT$_y$. For each import line, $y$ is calculated from *ID*$_c$ using Equation (4.3.1). An example for $P = 8$ is given in Figure 4.3.

To keep IMPORT$_y$ sorted, an $R$-way merge of the PROXY$_x$ files must be performed. It can be assumed that "good" clauses from clause-sharing have been imported frequently and that all PROXY$_x$ have almost the same content, except for a few clauses missing in each file. Therefore, the same clause is often found at the head of the *merge queue* for each PROXY$_x$ file. If *ID*$_c$ and $c$ match, $c$ needs to be written only once in IMPORT$_y$. If *ID*$_c$ matches but $c$ does not, the proof is rejected as invalid. Otherwise, the $c$ with the smallest *ID*$_c$ is written to IMPORT$_y$ and removed from the *merge queue*. Details on the *merge queue* can be found in Section 5.4.

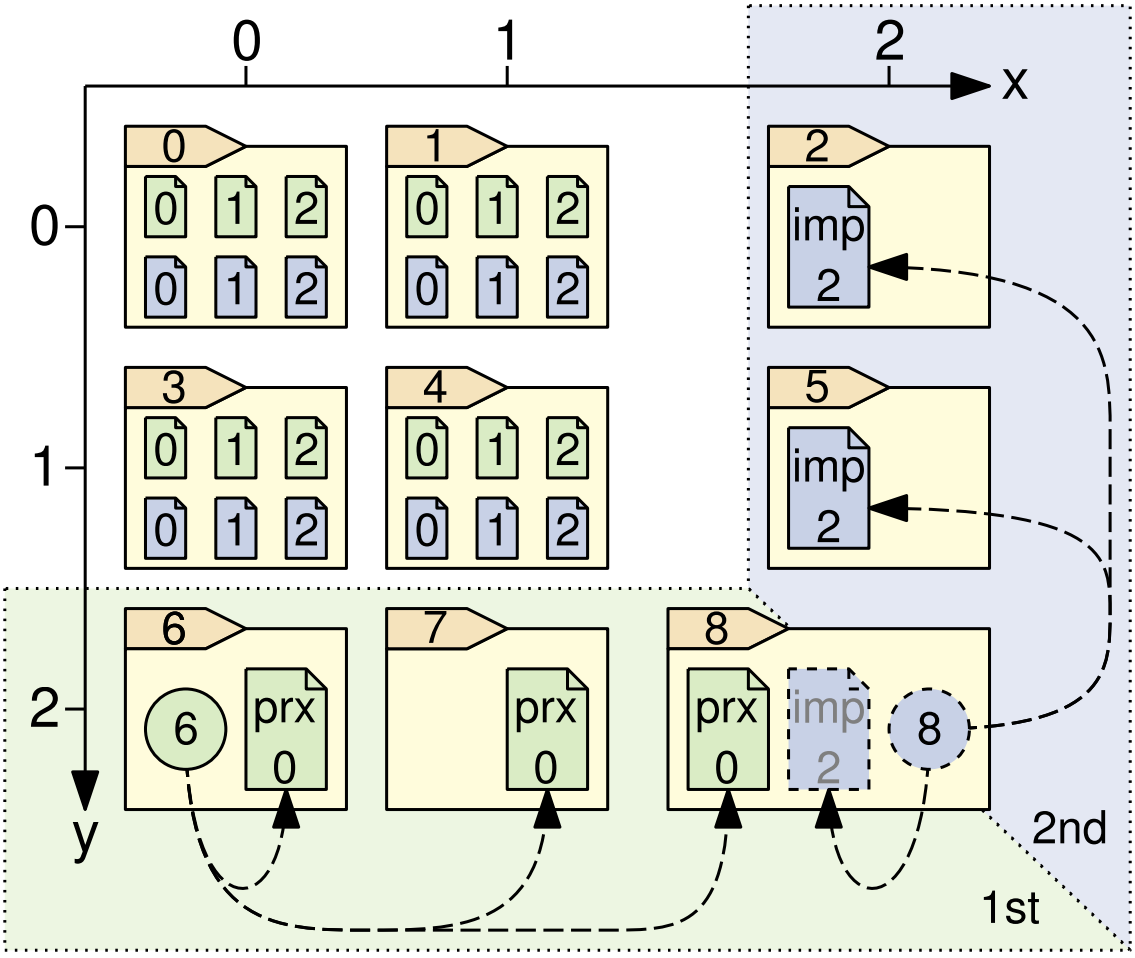


**Figure 4.3:** Example of the routing schema for $P = 8$, $K = 9$. The routing schema is divided into the logical *1st step* (row-wise communication) and *2nd step* (column-wise communication). For clarity, the steps *CHK*, *SRT*, and *ASM* are summarized here, and the PROXY files (green) are treated as the direct output of the *CHK* step. The outputs of the *ROT* step are the IMPORT files (blue). For $K = 9$, $CHK_6$ produces exactly $\sqrt{K} = 3$ PROXY files and stores them in directories 6, 7, and 8. For example, a clause with $ID = 92$ is written to the PROXY file in directory 8, since $92 \equiv 2 \pmod 9$ and according to Equation (4.3.1) and Equation (5.3.2), $k_{prx} = 2 \cdot 3 + 2 = 8$. Although there are no processes with $r = 8$ in the first step, a $ROT_8$ process is created in the second step. This process reads the two $\text{PROXY}_0$ files from directories 6 and 7 in directory 8 to merge and route their contents to the $\text{IMPORT}_2$ files in directories 2 and 5. An empty $\text{IMPORT}_2$ file is created in directory 8 and is ignored by the *RSL* step. The first step is performed for each $p \in [0, P)$, and the second step is performed for each $k \in [0, K)$. At the end of the 2nd step, each directory contains 3 PROXY files and IMPORT files (visualized for directories $0, 1, 3, 4$).

### 4.3.5 Resolve

*RSL*$_p$ must find all clauses from IMPORT$_y$ in PALRUP$_p$ in order to accept the proof. To do this, *RSL*$_p$ is started with the following parameters:

- `solver-id` $p$ to indicate which PALRUP$_p$ file contains the clauses from IMPORT$_y$.
- `num-solvers` $P$.
- `proofs-path` the path to the parent directory containing PALRUP$_p$.
- `communication-path` the path to the parent directory where IMPORT$_y$ is located.
- `buffer-KB` the size of the read buffer.
- `redistribution-strategy` as 1 to expect $P$ files or 2 to expect $R$ IMPORT$_y$ files.

Like the PROXY$_x$ files, the IMPORT$_y$ files for a given $p$ mostly contain the same content. Therefore, it makes sense to use the merge queue again for clause deduplication. Since the learned clauses $C_a$ in PALRUP$_p$ and the clauses $C_i$ in IMPORT$_y$ are sorted, a linear scan over both PALRUP$_p$ and IMPORT$_y$ is sufficient to find all $c_i \in C_i$ in PALRUP$_p$. PALRUP$_p$ is read line by line. While scanning, import and delete lines are skipped. For add lines, *ID*$_{c_i}$ (where $c_i \in C_i$) is compared with *ID*$_{c_a}$ (where $c_a \in C_a$). As long as *ID*$_{c_i}$ is greater than *ID*$_{c_a}$, the next add line is loaded. If *ID*$_{c_i}$ matches *ID*$_{c_a}$, $c_i$ and $c_a$ are compared for equality. If $c_i$ and $c_a$ do not match, the proof is rejected. The proof is also rejected if *ID*$_{c_i}$ is smaller than *ID*$_{c_a}$, as $c_i$ is not found in PALRUP$_p$. Otherwise, the scan continues.

While scanning, the fingerprints of all files are recalculated and compared with the fingerprints stored at the end of each file. If the fingerprints match, it can be assumed that PALRUP$_p$ and IMPORT$_y$ have not been changed between the steps. If the fingerprints do not match, the proof is rejected.

Only if all processes of the steps *CHK*, *ROT*, and *RSL* give positive feedback, the proof is accepted.

## 4.4 Integrity of Proofs

To protect the files from modification, cryptographic fingerprints are created using the clauses in the files and a secret key that *PalRUPCheck* is compiled with. For this purpose, *siphash*, which is already used by ImpCheck, and commutative hashing (*CommHash*) are employed (see Section 5.2). Both types of fingerprints prevent clauses from being modified, added, or removed without changing the fingerprint. Fingerprints created with *CommHash* are invariant with respect to the order of the clauses read in. This invariance is required to create fingerprints for the clauses in PROXY$_x$, as the clause order is changed during sorting. Since the order of clauses in the other files must remain unchanged to maintain their sorted order, *siphash* is used for those files.

We now examine how each step creates and checks fingerprints to ensure the integrity of the checking procedure. Since PALRUP$_p$ is read again in the last step, a fingerprint must be created for it by *CHK*$_p$. The *siphash* fingerprint for PALRUP$_p$ is based on its add lines. Since the other lines are irrelevant to *RSL*$_p$, they are excluded from the fingerprint. Note that the fingerprint does not in itself confirm that PALRUP$_p$ is in a valid state, because removing import lines could result in clauses no longer satisfying the RUP property, even though the fingerprint remains unchanged. However, the existence of a fingerprint at the end of each file confirms that *CHK* executed without finding errors in the proof and therefore the PROXY files contain every necessary imported clause. For the PROXY$_x$ files, *CHK*$_p$ creates *CommHash* fingerprints based on PALRUP$_p$'s import lines and appends them to the corresponding L$_x$.

Neither *SRT*$_k$ nor *ASM*$_k$ create fingerprints or compare them. *ASM*$_k$ only appends the fingerprint from the end of L$_x$ to PROXY$_x$.

*ROT*$_k$ calculates $R$ *CommHash* fingerprints from all PROXY$_x$ files in the merge queue and checks them for equality at the end of each file. If the clauses are sorted by *ID*$_c$ and the fingerprints match, it can be assumed that no errors occurred in the previous steps. Of course, this also requires that all PROXY$_x$ files are found in the $k$ directory as expected. This ensures that every clause imported by *CHK*$_p$ is also forwarded to IMPORT$_y$. For the outgoing IMPORT$_y$ files, *ROT*$_k$ must generate *siphash* fingerprints. These fingerprints are also appended to the end of each IMPORT$_y$ file.

Finally, *RSL*$_p$ must also calculate *siphash* fingerprints for PALRUP$_p$ and each IMPORT$_y$ file. This ensures that the clauses in IMPORT$_y$ are searched for within the same add lines in PALRUP$_p$ that *CHK*$_p$ processed during the RUP check. It also ensures that every clause that *ROT*$_k$ forwarded is present in IMPORT$_y$.

## 4.5 Correctness

Now we will show that if *PalRUPCheck* accepts a proof, the proof is sound. We assume that fingerprints grant perfect authenticity.

**Theorem 1.** *Let $\pi_i \in \Pi$ with $\Pi$ the set of PalRUP proofs for an unsatisfiable CNF formula $F$. If PalRUPCheck accepts $\pi_i$, then $\pi_i$ is a sound proof for $F$.*

*Proof.* If *PalRUPCheck* accepts a proof $\pi_i$, this means that $CHK_p$, $ROT_k$, and $RSL_p$ did not reject $\pi_i$ and explicitly responded positively. In addition, for at least one $p$, $CHK_p$ must have found the *empty clause*.

***CHK* is valid.** $CHK_p$ has determined for all learned clauses that they satisfied the RUP property with respect to the clause database at the time of learning. Each clause depends **only on previously learned** clauses of other solver threads because IDs suffice Constraint (ii) and Constraint (iii) of Section 3.3. Now it only remains to ensure that the imported clauses learned by other solver threads are valid, that is, they must be found in the other $\text{PALRUP}_p$ files.

(i) ***ROT* is valid.** Therefore, each import line from $\text{PALRUP}_p$ is assigned to exactly one $p_{dst}$ and was found by a $ROT_k$. Since all $ROT_k$ know exactly how many $p$ are assigned to them (in their row), they expect exactly one $\text{PROXY}_x$ file for each $p$. Additionally, $ROT_k$ checked the fingerprints of all $\text{PROXY}_x$ files. $ROT_k$ writes each clause exactly once to an $\text{IMPORT}_y$ file. Since all these clauses are read again by an $RSL_p$, every clause from each import line in $\text{PALRUP}_p$ arrives exactly once in $\text{IMPORT}_y$.

(ii) ***RSL* is valid.** Therefore, each $RSL_p$ has found a valid fingerprint at the end of the $\text{PALRUP}_p$ and the $R$ $\text{IMPORT}_y$ files. Additionally, each clause $c$ from $\text{IMPORT}_y$ was found in $\text{PALRUP}_p$.

(iii) $\text{IMPORT}_y$ contains only clauses from import lines of the $P$ PALRUP files where the $ID_c$ is claimed to have been generated by solver thread $p_{dst}$.

Overall, this means that **every**(i) imported clause **arrived** (i) in the **correct file** (iii), was **produced**(ii) by $\text{PALRUP}_{p_{dst}}$, and no clause was **modified** along the way (i)+(ii). This means that all import lines seen by $CHK_p$ can be confirmed as valid. This fulfills the only prerequisite: the validity of the import lines that $CHK_p$ could not check in $\text{PALRUP}_p$.

Since all $\text{PALRUP}_p$ files are valid and an empty clause was found in one of them, $\pi_i$ is sound. □

## 4.6 Theoretical Performance

In this section, we analyze the theoretical runtime and main memory consumption of the different steps in *PalRUPCheck*.

For this purpose, we will use several variables:

- $P$, Number of solver threads. To simplify the analysis, square numbers are assumed for $P$, and therefore $P = K$.
- $R := \sqrt{P}$ for clarity.
- $N$, Number of all learned clauses. We expect $\frac{N}{P}$ clauses to be learned per solver thread. To account for unbalanced clause learning, we use $\kappa$.
- $\kappa$, time for checking all LRUP steps in the longest partial proof.
- $I$, number of all clauses that were shared (passed to the CSM). Note that, $I \ll N$.
- $H$, maximum number of active clauses in a clause database at any point in time. We can expect slightly fewer than $N$ deletions.
- $B$, buffer size for files.

We neglect small constant overheads, e.g. fingerprint hash updates and comparisons, because we are interested in scaling for large $N$ and $P$ here. We also assume a constant clause size, as clauses are relatively small compared to $I$ and are capped by the number of literals in the original CNF problem file.

In the first step, we start $P$ instances of $CHK_p$. Each $CHK_p$ has a constant startup overhead of $R$ *CommHash* objects and creation of PROXY files [4]. The RUP check is performed in $\mathcal{O}(\kappa)$ for all clauses in the longest partial proof. We expect $\mathcal{O}(1)$ lookup time for a clause in the hash table. A deletion operation is also performed in expected $\mathcal{O}(1)$ time. At most $\kappa$ deletion operations are possible with a count of $\kappa$ clauses. An import operation is added to the hash table in $\mathcal{O}(1)$ and also written into one of $R$ files. In worst case here, all $I$ shared clauses were imported by every solver thread and therefore $\frac{I}{R}$ clauses are expected per PROXY file.
In total the runtime is in $\mathcal{O}(R+\kappa+I)$ and the expected memory usage is in $(R+1)B+H \in \mathcal{O}(R \cdot B + H)$. The expected file sizes are $2\frac{N}{P} + I$ lines for $\text{PALRUP}_p$ and $\mathcal{O}(\frac{I}{R})$ for each of the $R \cdot P$ PROXY files ($\mathcal{O}(I \cdot P)$ in total).

The sort step has to sort $R \cdot P$ PROXY files of size $\mathcal{O}(\frac{I}{R})$ with $P$ processing elements (PE). Thus, we assume each $SRT_p$ sorts $R$ files of size $\mathcal{O}(\frac{I}{R})$. As bsort is used as an external tool, some claims might be inaccurate. We do not provide an analysis for memory usage, because bsort is designed to work with files much larger than main memory anyway. With a general runtime for radix sort in $\mathcal{O}(n)$, a file is sorted in $\mathcal{O}(\frac{I}{R})$. With $P$ PEs and $R \cdot P$ files, the expected time of the sorting step is $\mathcal{O}(\frac{R \cdot P}{P} \cdot \frac{I}{R}) = \mathcal{O}(I)$ From $R \cdot P$ PROXY files, $R \cdot P$ sorted $\text{PROXY}_s$ files are created with the same content, resulting in $\mathcal{O}(I \cdot P)$ additional storage space.

[4] For clarity *ID* and $L$ files are referred to as the PROXY file they will be assembled to in step *ASM*.

In $\mathit{ASM}_p$, $R$ files of size $\mathcal{O}(\frac{I}{R})$ are assembled. This is done line-by-line without additional loops. Therefore the runtime of $\mathit{ASM}_p$ is in $\mathcal{O}(R \cdot \frac{I}{R}) = \mathcal{O}(I)$. Since this is done sequentially for each file, memory usage is in $2B \in \mathcal{O}(B)$. As a result, $P \cdot R$ files are created with the same content, using $\mathcal{O}(I \cdot P)$ additional storage.

Each $\mathit{ROT}_p$ has to handle $2R$ files, $R$ PROXY and $R$ IMPORT files. The clauses from the PROXY files must be written to the correct IMPORT file while maintaining the sorted order. The merger takes care of this by using a linear scan to retrieve the clause $c$ with the smallest $\mathit{ID}_c$ from the PROXY files and deleting all duplicates of $c$. This results in $R$ compare operations per unique clause from PROXY files. In our worst case scenario, there are $\mathcal{O}(\frac{I}{R})$ clauses in each PROXY file. However, since all files contain the same clauses, deduplication effectively removes all clauses except those in $\text{PROXY}_0$. This represents the worst case, as skipping a missing clause in any PROXY file avoids only one deletion operation but does not add any extra work. Writing to the IMPORT file is only a $\mathcal{O}(1)$ operation. Overall, this means that the runtime of $\mathit{ROT}_p$ is in $\mathcal{O}(R \cdot \frac{I}{R}) = \mathcal{O}(I)$. The expected memory usage is in $2R \cdot B \in \mathcal{O}(R \cdot B)$. Each of the $R$ output files each have a size of $\mathcal{O}(\frac{I}{R^2}) = \mathcal{O}(\frac{I}{P})$. With a total of $R \cdot P$ IMPORT files for $P$ $\mathit{ROT}_p$ instances, this is an additional storage consumption of $\mathcal{O}(R \cdot I)$.

In the last step, $\mathit{RSL}_p$ reads $\text{PALRUP}_p$ and $R$ IMPORT files. Every $c \in$ IMPORT must be found in $\text{PALRUP}_p$. Again, as in the *route* step before, the merging queue takes care of providing $c$ with the smallest $\mathit{ID}_c$ in a linear merge in $\mathcal{O}(R)$. Since the IMPORT files are expected to contain $R$ duplicates of each $c$, $\mathcal{O}(\frac{I}{P})$ clauses have to be found in $\text{PALRUP}_p$. Note that $P \cdot \frac{I}{P} = I$ for $P$ instances of $\mathit{RSL}_p$ is the amount of clauses we assumed to be shared in total. Since the learned clauses in $\text{PALRUP}_p$ are sorted by $\mathit{ID}_c$, the linear scan to find the imported clauses runs in $\mathcal{O}(\kappa + R \cdot \frac{I}{P}) = \mathcal{O}(\kappa + \frac{I}{R})$ The expected memory usage is in $(R+1)B \in \mathcal{O}(R \cdot B)$, and no files are written.

The runtime per PE in total for all five steps therefore is

$$\begin{aligned} & \mathit{CHK}_p + \mathit{SRT}_p + \mathit{ASM}_p + \mathit{ROT}_p + \mathit{RSL}_p \\ =& R + \kappa + I \\ &+ I \\ &+ I \\ &+ I \\ &+ \kappa + \frac{I}{R} \\ \in& \mathcal{O}(R + \kappa + I) \end{aligned} \tag{4.6.1}$$

The expected maximum memory usage is

$$\begin{aligned} &\mathit{max}((R+1)\cdot B + H \\ &\quad ,2B \\ &\quad ,2R\cdot B \\ &\quad ,(R+1)B \\ &\quad ) \\ =&\mathit{max}((R+1)\cdot B + H, 2R\cdot B) \\ \in&\mathcal{O}(R\cdot B + H) \end{aligned} \tag{4.6.2}$$

With a growing $P$ and therefore $R$, $2R \cdot B$ can surpass $(R+1) \cdot B$ for a fixed $B$. However, we can chose to set $B := \frac{H}{R-1}$, than

$$\begin{aligned} &(R+1)\cdot B + H \\ =&(R+1)\cdot \frac{H}{R-1} + H \\ =&\frac{RH+H}{R-1} + \frac{(R-1)H}{R-1} \\ =&\frac{2RH}{R-1} \\ =&2R\cdot B \end{aligned} \tag{4.6.3}$$

so for every $B \leq \frac{H}{R-1}$, step *CHK* has the highest memory usage.

The total of the written files is

$$\begin{aligned} &P\cdot(2\frac{N}{P} + I) \\ &+ I\cdot P \\ &+ I\cdot P \\ &+ I\cdot P \\ &+ I\cdot R \\ &= 2N + I(4P+R) \\ &\in \mathcal{O}(N + I(P+R)) \end{aligned} \tag{4.6.4}$$

This results in an additional storage consumption of $\mathcal{O}(I(P+R))$ for checking, compared to the original proof.

It is important to remember that $N$, $P$, and $I$ are all interrelated. To solve difficult problem instances with a large $N$, a correspondingly large number of solver threads $P$ should be used to achieve a satisfactory runtime. However, since SAT solving does not scale perfectly with $P$ [24] because many clauses are generated multiple times independently by different solver threads, $N$ also grows with $P$. $I$ also scales with $P$ and $N$, as the number of shared

clauses $I$ increases with $N$ during the solving process. $I$ can be kept small by a clever CSM with deduplication, thereby reducing $P$'s impact on $I$ during periodic clause-sharing. Although $R = \sqrt{P}$ limits runtime scalability depending on the PEs, the constant factors for $I$ and $\frac{N}{P}$ are very high and dominate the runtime. The maximum expected memory usage, $2R \cdot B$, is more of a limiting factor, as $B$ cannot be set arbitrarily small to ensure efficient block-based file reading. This is not a problem so far even with a total of 1216 PEs and a B of 64 MiB (see Section 6.2). Since *PalRUP* does not yet perform any trimming, unlike other methods, the space consumption is comparatively large. Overall, we are satisfied with the scaling in terms of runtime, memory, and storage consumption, none of which are limiting factors when checking PalRUP proofs at present.

# 5 Implementation Details

For the implementation of *PalRUPCheck*, certain modules from ImpCheck like the *hash table* (Section 5.1) and *SipHash* (Section 5.2) have been reused. In *CHK* we utilized *Importer* (Section 5.3) as black box for clause routing. Other modules, such as *CommHash* (Section 5.2), the *Merging Queue* (Section 5.4) and a buffered *File Reader* (Section 5.5) are implemented independently of specific steps, but are utilized in multiple locations. In the following sections, we will examine some of their details more closely.

## 5.1 Hash Table

The clause database in ImpCheck is a linear probing hash table. It supports 64-bit keys and `void*` values. For the clause database in Section 3.4 and Section 4.3.1 therefore $ID_c$ is used as the key and $c$ is linked with a pointer. This hash table is also instantiated as the translation data structure. As translation table, it stores original IDs as keys and offsets as values. Since the offsets are stored as 64-bit unsigned integers and a 64-bit pointer to an array with 32-bit literals is stored for each clause, the translation table is guaranteed to take up less space than the clause database. Another option for a translation data structure is to store the offsets in the clauses of the clause database. This has the advantage of saving 64 bits of memory per clause because the keys do not need to be stored twice. However, storing the offsets in the clause arrays has the disadvantage that a further redirection must be loaded via a pointer to the memory area of the clause, which reduces the cache efficiency and thus the performance of the frequent $max(ID_h)$ calculation. Since the translation table stores data in an array of key-value pairs, reading the value is unlikely to cause a cache miss, as it is located directly after the key in memory.

To support fast insert and search operations, the data array is kept at a maximum load factor of 50 %. If more space is needed, an array of double the size is allocated and all entries are reinserted. To avoid long collision chains when an entry is deleted, the resulting gap is filled with next suitable element in the probe sequence that belongs before or in the gap, if such an element exists. This ensures that searches for elements can terminate upon encountering an empty entry (zero key).

## 5.2 Fingerprints

Two kinds of fingerprints are used in *PalRUP*: *SipHash* [1] is already used by *ImpCheck* and is reused to sign the files generated by *PalRUPCheck*. Schreiber [25] gives more details about its usage there. However, *SipHash* does not support changes in the ordering of input keys. Thus, it is used for PALRUP and IMPORT files. Delete lines in PALRUP do not update the *SipHash* object, as they are not read in *RSL*.

Therefore, another hashing module *commutative hashing* (*CommHash*) was implemented to generate fingerprints that can be generated in *CHK* and validated in *ROT* after the clauses are sorted. To do this, it uses two 64-bit unsigned integer values internally as clause state, which are reinitialized for each clause using a secret key. $ID_c$ and $c$ are then hashed using `murmur3`, and the clause state is updated with the widely used `hash_combine`[1] function. To ensure that *CommHash* remains commutative with respect to clause order, the clause state is added to a fingerprint state after processing each clause. The clause state is then reinitialized for the next clause. Since addition is a commutative operation, the fingerprint remains invariant even if the clauses are permuted during sorting.

## 5.3 Importer

The *Importer* manages the $R$ adjacency arrays $A_x$. $A_x$ refers to the $ID_x$ arrays, which store *index objects* of {$ID_c$, *startIndex*, *countLiterals*}, and $L_x$ arrays which store the associated literals. *startIndex* refers to indices of $\mathrm{L}_x$, where the literals of clause $c$ are stored. Note that *countLiterals* must be saved, as after sorting $\mathrm{ID}_x$ it would be infeasible to recover that information.

Clauses are inserted into $A_x$ based on $ID_c$. For every clause, $p_o := ID_c \bmod P$ is calculated as the index of the original solver $S_{p_o}$. The values of $x$ and $y$ are calculated as in Section 4.3.4:

$$x := p \bmod R$$
$$y := p \div R$$

For better performance on lists of larger objects, sorting algorithms sort *keys* with pointers to these objects [23]. That is why an *adjacency array* is used as the data structure. In our case, $ID_c$ can be used as key for sorting. Since we use *bsort* (see Section 4.3.2), fixed-size keys are required.

[1] Boost, `hash_combine`: https://www.boost.org/doc/libs/1_55_0/doc/html/hash/reference.html. Accessed: 2025-08-11.

Whenever an $A_x$ would exceed its buffer size due to insertion of a new clause, $ID_x$ and $L_x$ are flushed to file. That way memory consumption is capped to a flexible but constant overhead in relation to the solving process. The output location is calculated using the following formula:

$$k = y \cdot R + x \tag{5.3.1}$$

$\mathrm{ID}_x$ and $\mathrm{L}_x$ are located at `proofs-path-out/`$k_{prx}$`/` with:

$$k_{prx} := y_{src} \cdot R + x_{dst} \tag{5.3.2}$$

If necessary, `proofs-path-out/`$k_{prx}$`/` is created, as $k_{prx}$ can range up to $K - 1 > P$, which may exceed $P$. Every clause that is inserted to $A_x$ also updates the order-invariant *CommHash* fingerprint.

## 5.4 Merging Queue

The *merging queue* (MQ) is used in both the route and resolve step. MQ manages the input files using the *file reader* and fingerprint calculation using *commutative hashing* in the routing step and *SipHash* in the resolve step Section 5.2. It also expects output pointers where the smallest clause (with respect to $ID_c$) will be stored. Each time `import_merger_next` is called, the clause ($c$) in the output is updated.

An array is maintained for each of the $R$ input files, storing the smallest clause $c$ from each file. In a linear merge, the currently smallest $c$ is taken to replace its duplicates with new clauses. If a new smallest $c$ is found during the merge, the merge is continued with this $c$. It is not necessary to start duplicate elimination from the beginning when a smaller $c$ is found, as this $c$ has not appeared before. To avoid unnecessary copying of $c$, the pointer to the current $c$ is stored in an output pointer. The next time `import_merger_next` is called, this $c$ is replaced by a new clause from the file. If a file is empty, `MAX_U64` is set as a sentinel for its $ID_c$ entry in the array, which means it is always skipped.

## 5.5 File Reader

The file reader is used for reading every file in *PalRUPCheck*. It uses `fread_unlocked` to read files in `buffer_size` chunks. Most reads are served from the *main buffer*. The file reader returns only copies of *data objects*, as the buffers may change before the read data is fully processed by the caller.

With each read request, an internal *byte counter* is incremented by $n$ bytes read. Before each read, a check is performed to see whether the *byte counter* would exceed the buffer limit. Often *data objects* like a 8-byte $ID_c$, $c$ is split at the end of the buffer. In such cases the *fragment buffer* is used, where the whole object (e.g. $c$) is stored. The first part of the

*fragment buffer* is filled with the remaining data from the *main buffer*. It may happen that a large clause with a large number of literals is loaded and the *main buffer* is too small to accommodate the rest of the requested object. In this case, the *fragment buffer* continues to be filled from the *main buffer* until the object is loaded. Otherwise the rest of the *fragment buffer* is filled with the rest of the object from the next *main buffer*. Note that only one load of a *main buffer* is needed, if sufficiently large main buffers are used.

The procedure described above enables efficient block-based linear reading of files. *RSL* requires a `skip_bytes` function to jump forward in the *main buffer* and reload it when crossing buffer boundaries. This approach is faster than copying and discarding objects.

For *ASM*, the ability to jump backwards in files to byte $s$ with `reader_seek` is also required. Since this functionality is not required by any other step, it is tailored to *ASM*. In *ASM*, shared clauses can be assumed to arrive at least slightly pre-sorted during solving. In MallobSat, solver threads periodically share their best clauses. As solving progresses, newer clauses with higher IDs are typically shared, reflecting their improved quality. Occasionally, clauses with significantly lower IDs may appear, requiring the *buffered file reader* to jump far forward in $\mathrm{L}_x$ and then back to the previous position. Since most previous literals have likely been read, only a tenth of the buffer area is loaded with literals before the index jumped to when jumping back.

# 6 Experimental Evaluation

In this section, we present the evaluation of our approaches for proof generation from Section 3.4 and proof checking from Chapter 4. The source code for ImpCheck (with PalRUP proof generation) and PalRUPCheck[1], as well as the evaluation data[2] are available online.

## 6.1 Setup

All experiments were run on HoreKa[3], a Slurm-managed high-performance cluster in Karlsruhe, Germany. Each node is equipped with two Intel Xeon Platinum 8368 processors, providing a total of 76 cores (152 threads) per node. Each node also has 256 GB of main memory and a 960 GB NVMe SSD as local storage. The underlying connection network is InfiniBand 4X HDR. Several options are available for sharing data across multiple nodes:

(i) **Workspace** is based on two large-scale IBM Spectrum Scale parallel file systems. With a limit of 250 TB and 50 million inodes per user, there is sufficient space for PalRUP proofs and communication files during proof checking, but tests have shown significant fluctuations in read speed, which hinders the reproducibility of the experiments.

(ii) **Full Flash PFS** is a Lustre-based parallel file system that uses only flash storage. While the IOPS rates are improved and more stable compared to Workspace, the user limit is 1 TB, which is not sufficient for proof and checking even with four nodes.

(iii) **BeeOND** is a job-local BeeGFS on-demand parallel file system that can be requested and created when starting a job. It combines the local hard disks of all nodes into a virtual mount point and supports RAID 0-like stripe patterns with up to 32 stripes. Assuming 750 GB of space per local hard disk, this provides almost 3 TB with 4 nodes and 12 TB with 16 nodes, which is sufficient for the experiments. Additionally, it offers a high throughput and low latency thanks to NVMe storage and the InfiniBand interconnect between nodes.

[1] PalRUPCheck source code: `https://github.com/MichaelDoerr/ImpCheck`
[2] PalRUPCheck evaluation data: `https://github.com/MichaelDoerr/PalRUPCheckEval`
[3] HoreKa documentation: `https://www.nhr.kit.edu/userdocs/horeka/`
Accessed: 2025-08-26

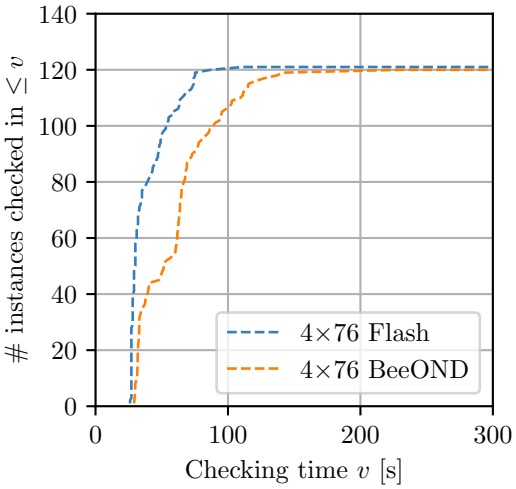


**Figure 6.1:** Proof checking performance of Full Flash PFS (Flash) and BeeOND partitions on 4 nodes.

Although BeeOND is the preferred choice for the reasons mentioned above, the tests were carried out on Full Flash PFS. During the preparation of the experiments, it became apparent that the creation of the BeeOND mount sometimes fails silently on a few nodes. The only solution in this case is to restart the job until Slurm randomly schedules the job on other nodes. However, this resulted in an excessive amount of overhead for job starts, so another solution was needed until the problem is fixed.

To support experiments on Full Flash PFS, PalRUPCheck was extended to allow separate paths for the PalRUP proof and the communication files. Since PalRUP consists of several partial proof files, they can be split across several nodes, so that node $i$ contains $[i \cdot 76, (i+1) \cdot 76 - 1]$ of the files. To make this possible, the randomized rank assignment of the solver threads is also disabled in MallobSat. This means that 76 partial proof files are stored on each node's local disk, while communication files are exchanged between nodes via Full Flash PFS.

As shown in Figure 6.1, Full Flash PFS requires less time for all five steps of the check combined (Checking time) than when BeeOND is used. One possible explanation for the lower performance of BeeOND is the overhead caused by BeeOND's metadata server, which must run on the nodes themselves. Thus, by switching to Full Flash PFS, no proof checking performance was lost, but rather gained. Due to the above-mentioned problems with BeeOND, the tests were not performed with 16 nodes. The performance of BeeOND on a single node was also not compared, as inter-node communication and thus BeeOND is not required in this case.

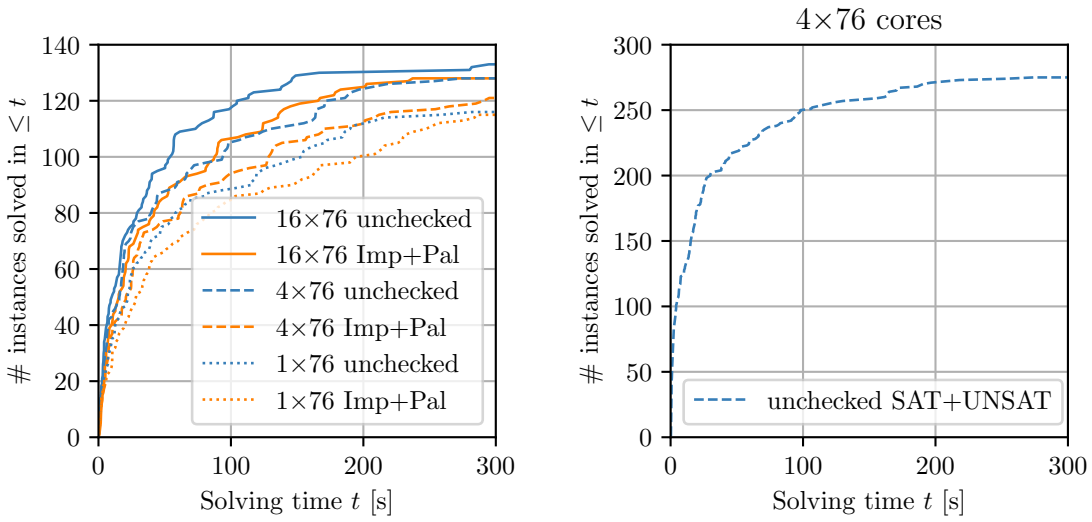


**Figure 6.2:** Solving times of MallobSat without live-checking or proof production (unchecked) and with live-checking and proof production (Imp+Pal) on 1-16 nodes (left). For comparison with [25], pure solving performance on 4 nodes on the full SAT Competition 2023 instance set is given on the right.

To compare our results with [25], we used the 400 instances from the SAT Competition 2023 for the right graph in Figure 6.2. All other experiments used only the UNSAT instances from the SAT Competition 2023, except for 26 instances that were removed [4] from the test set because MallobSat+ImpCheck reliably produced error messages during execution. Only UNSAT instances were used, as SAT instances naturally do not produce valid unsatisfiability proofs. The experiments were run on 1, 4, and 16 nodes with a 300-second timeout for solving, both with and without proof production. An additional 300-second timeout was set for proof checking, but it was never reached; all proofs were checked in 116 seconds or less.

[4] The removed instances are:
all 20 variants of `hash_table_find_safety_size_N`;
`g2-T49.2.0`;
`goldberg03.hard_eq_check.i10mul.miter.used-as.sat04-333`;
`oisc-subrv-and-nested-14`;
`oisc-subrv-sll-nested-15`;
`REGRandom-K4-L3-Seed15`;
`REGRandom-K4-L4-Seed10`;

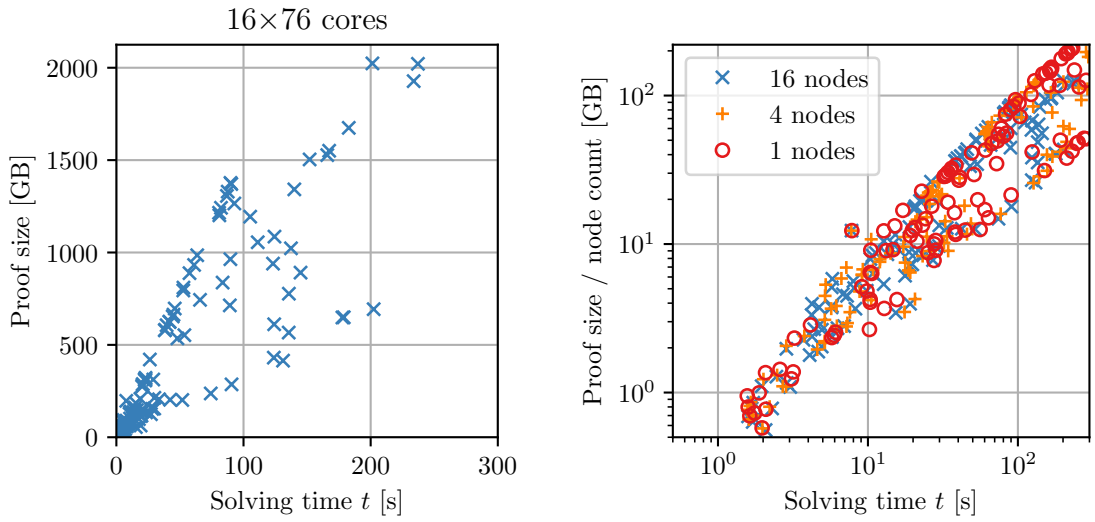


**Figure 6.3:** Size of PalRUP proofs (aggregated partial proofs) on 16 nodes on the left. Scaled proof sizes with space consumption per node on the right.

## 6.2 Results

We discuss proof production in Section 6.2.1 and checking performance in Section 6.2.2.

### 6.2.1 Proof Production and Size

We first examine the overhead our modified, PalRUP proof producing ImpCheck introduces to solving compared to pure MallobSat without checking or proof production. As shown on the right side of Figure 6.2, our parameterization of MallobSat with reduced sharing volume performs similarly to [25] with the same number of solved instances (275). Proof generation significantly impacts solving performance (see left side of Figure 6.2). The median for the relative overhead is 39 % for 1 node, 48 % for 4 nodes, and 54 % for 16 nodes. These values are slightly higher than those reported in [25] (37 % – 42 %). This is primarily due to the additional overhead of lookups and inserts in the translation table for ID calculation.

Figure 6.3 shows that the size of PalRUP proofs depends primarily on the solving time and node count. On the left the largest proof is approximately 2023 GB. On the right, proof size per node appears independent of node count and depends solely on solving time. Overall, the proof size appears to remain between $20 \cdot \frac{\#nodes}{100}$ GB/s as the lower bound and $100 \cdot \frac{\#nodes}{100} = \#nodes$ GB/s as the upper bound for longer runs. This variation depends heavily on the properties of individual SAT instances. Solvers may struggle to learn new clauses, resulting in small proofs, or they may learn a large number of non-core clauses that do not contribute to deriving the empty clause, additionally increasing the proof size.

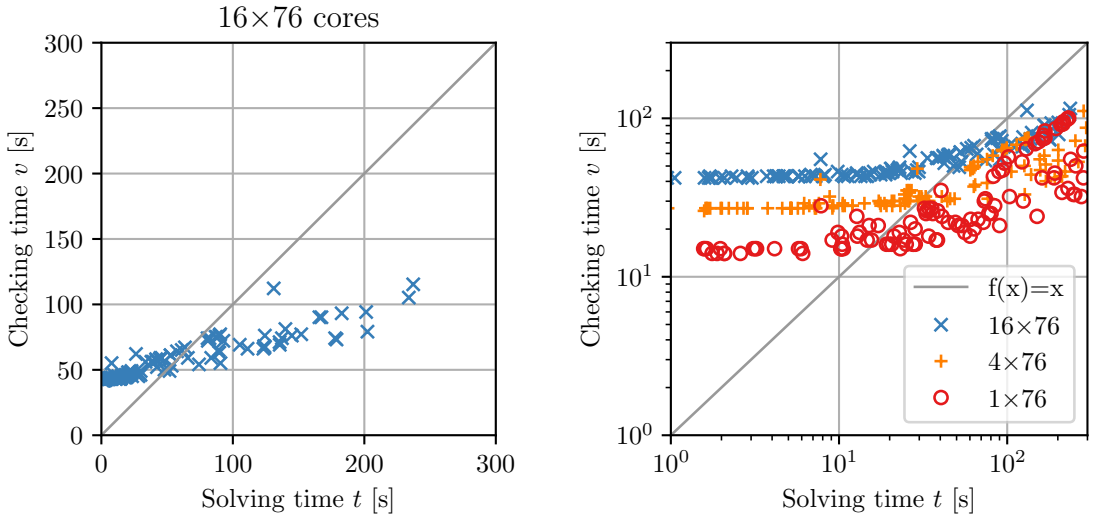


**Figure 6.4:** Time of full proof check in comparison to Solving time. Linear plot for 16 nodes on the left and logarithmic plot for 1-16 nodes on the right.

| # nodes | CHK | SRT | ASM | ROT | RSL |
|---|---|---|---|---|---|
| 1 | 17.13 | 7.06 | 2.58 | 2.04 | 2.42 |
| 4 | 13.65 | 15.11 | 3.65 | 3.13 | 3.70 |
| 16 | 13.57 | 27.57 | 6.42 | 4.11 | 3.60 |

**Table 6.1:** Mean overhead of each proof checking step in [s] at different node counts. The execution time of each step is measured from the start of the `mpirun` call until its return. Thus, only full seconds are reported, despite timing being tracked in nanoseconds.

### 6.2.2 Proof Checking Performance

In this section, we examine the performance of PalRUPCheck when fully checking PalRUP UNSAT proofs. We discuss various aspects including each steps runtime, communication volume, and ratios to solving time.

In Figure 6.4 on the left we can see that for 16 nodes, proof checking is faster than solving for all tested instances that have a solving time of at least 75 s. For 4 nodes, this applies to instances with a solving time of at least 33 s, and for 1 node, 19 s. On the right side, it appears that there is a constant overhead for proof checking. This amounts to approximately 14 s, 27 s, and 42 s for 1, 4, and 16 nodes, respectively.

We use Figure 6.5 to analyze each checking step. Pre-Check has a constant overhead of about 4 s for instances with less than 3 s solving time. This is also the case with 4 s overhead for the Assemble step for instances with less than 87 s solving time. The Sort, Route, and Resolve steps have a constant overhead of 27 s, 4 s and 3 s in our experiments.

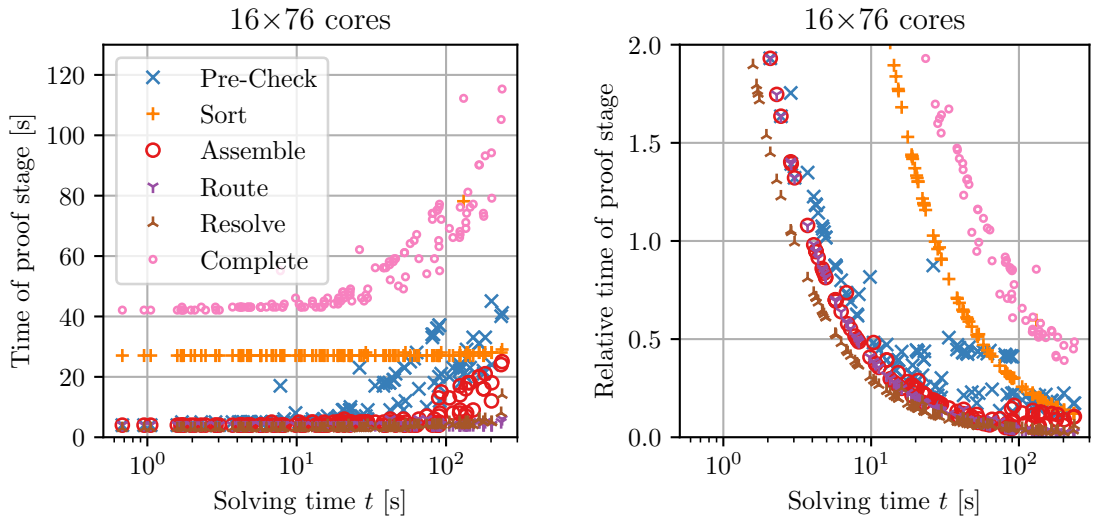


**Figure 6.5:** Time of every proof stage and the cumulative time of every step (Complete) in regards to solving time on 16 nodes. Absolute timings on the left and relative timings on the right.

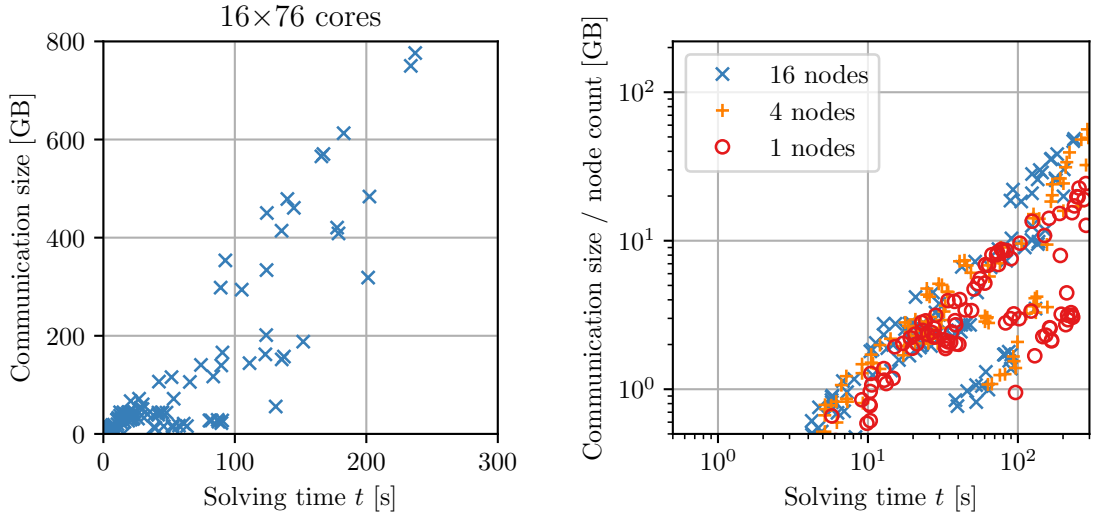


**Figure 6.6:** Total size of all files written for communication purpose. 16 nodes on the left side and size per node for different node counts on the right.

The overhead times for all node counts are shown in Table 6.1. For all node counts, the Sort step has by far the highest overhead and thus offers the highest potential for optimization. One of the main drawbacks of the current workflow of the sort step is that each file to be sorted must be copied from the shared file system to a local node, and the sorted file must

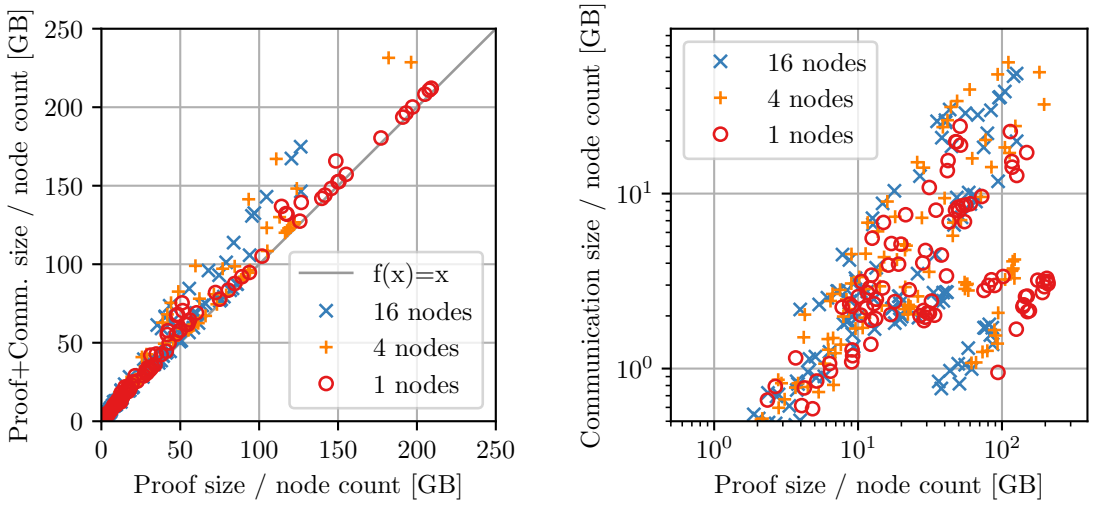


**Figure 6.7:** Left: Communication size relative to proof size. Right: Total written file sizes relative to proof size. All sizes are per node.

then be copied back. Otherwise the performance of bsort with files on remote file systems is even worse. A possible direction for future work is to eliminate the Sort and Assemble step with an external heap in the Pre-Check step. Using an external heap, clauses can be written directly in sorted order to PROXY files, eliminating the need for a dedicated sorting step. An Assemble step is only necessary if the external heap does not support variable-sized objects.

Figure 6.6 shows that communication size correlates linearly with the solving time, as expected, since a longer solving time also means more time for clause-sharing. However, there does not appear to be a high lower bound. Instances with very little communication during checking may either have small proof sizes or involve clauses that are not imported because they have already been learned by all solver threads. On the right, the number of nodes appears to influence communication size per node, as the data points for 16 nodes are typically above those for 1 or 4 nodes.

Figure 6.7 shows no obvious influence of the number of nodes on the ratio between communication size and proof size. However, the right plot resembles the right plot of Figure 6.6, since solving time and proof size are strongly correlated. The left plot shows that communication files can increase storage demand for checking by up to 60 % compared to the proof size alone.

We did not evaluate PalRUPCheck in a case where scale of checking (SOC) $\neq$ scale of solving (SOS). Due to PalRUPCheck's architecture, we expect the following behavior: When $SOC > SOS$, *SRT*, *ASM*, and *ROT* can benefit from this if $P \neq K$. However, with future optimization, *SRT* and *ASM* can be integrated in *CHK*. When $SOC < SOS$, efficiency may improve if steps are scheduled by expected work, e.g., input file size. Memory

consumption should not be an issue, because each process is independent and the different ranks of a step can be started separately.

In summary, PalRUP is a truly scalable proof format. While PalRUP consumes significantly more storage space than other approaches, it offers a scalable method for persistent proof generation. PalRUPCheck also requires substantial storage for communication files but verifies proofs faster than they are generated (for instances exceeding specific solving times). While the file sizes and constant overheads of individual steps can be further optimized, we are encouraged by the results.

# 7 Conclusion

## 7.1 Summary

This work is another step towards scalable verified clause-sharing SAT solving. With our ImpCheck extension, we provide a scalable and bottleneck-free way of UNSAT proof generation for clause-sharing SAT solvers. This approach primarily requires LRUP [7] recording of solving steps via pipe to keep requirements to SAT solvers low. LRUP output is already supported by CaDiCaL in MallobSat, currently the fastest parallel SAT solver[1] and the fastest cloud SAT solver [2] SAT solver, which we use for our evaluation. We evaluate our PalRUP proof production (and checking) as a new ImpCheck extension using MallobSat on a 1216-core setup. Our results confirm that the proof generation approach is bottleneck-free, producing PalRUP proofs with a running time overhead of 39 % to 54 %. In comparison, pure on-the-fly checking incurs an overhead of 37 % to 42 % [25]. The persistent proofs we generate can be checked later or by third parties.

PalRUPCheck can check difficult UNSAT instances with long solving times **faster** than the solving process itself on the same hardware. For each solver thread, the following holds: If solving fits into RAM, then checking also fits into RAM. This property distinguishes PalRUPCheck from monolithic LRAT checkers, which lose this advantage when proofs are merged into a single file. For smaller UNSAT instances on single machines, [18] still provides a well suited workflow for clause-sharing solvers with small, trimmed proofs and fast, efficient single-core checking.

We consider our goal of providing a scalable way to produce and check proofs of unsatisfiability achieved. PalRUPCheck is designed without sequential bottlenecks and utilizes multiple disks when reading proof-related files. With a minimally restrictive ID scheme and fingerprints for all critical files, we ensure integrity of the checking workflow.

[1] SAT Competition 2025 results; https://satcompetition.github.io/2025/satcomp25slides.pdf
Accessed: 2025-08-30

[2] The SAT competition 2025 has no Cloud track, so the results of 2024 apply:
https://satcompetition.github.io/2024/results.html
Accessed: 2025-08-30

## 7.2 Future Work

PalRUPCheck still has some drawbacks that need to be addressed in the future. For example, proof sizes can become enormous. With a solving time of 201 s, the largest proof in our experiments reaches a size of 2023 GB. The second-largest proof, at 2021 GB, requires an additional 776 GB for communication. With more efficient encoding (e.g., variable-byte encoding, where IDs and literals are always encoded in 7-byte payloads and 1-bit continue blocks), significant space can be saved. We also expect this to result in improved checking performance, since PalRUPCheck is mostly limited by hard disk speed. Substantial space savings could be achieved through a trimming process similar to that in [18]. It would be valuable to investigate how much this improves verification performance, despite the likelihood of producing partial proof files of non-uniform sizes.

With the new extension of ImpCheck, there is now a plugin solution for proof generation, but causes significant overhead. Our proof format expects LRUP support from CDCL solvers and has increased requirements for clause IDs. Given that CaDiCaL already supports LRUP logging, a natural next step is to adapt MallobSat and CaDiCaL to natively generate IDs that satisfy PalRUP requirements. This would eliminate the need for a translation table and could likely be implemented with minimal overhead, resulting in significantly less impact on solving times.

On the other hand, most CDCL solvers currently only support DRAT logging. To enable these solvers to be used for PalRUP proof generation, an ImpCheck extension for the live conversion of DRUP to LRUP is a potential solution.

In the future, bounded variable addition [10] could also be supported as a preprocessing technique. This would require new literals to be explicitly assigned to solvers via $l \equiv r \pmod{p}$ to avoid conflicts, and corresponding recorded lines must also be added.

Since the approach is a promising method for fast proof checking of large-scale clause-sharing SAT solvers, a formally verified PalRUPCheck-like checker is desirable.

# Bibliography


[1] J.-P. Aumasson and D. J. Bernstein. „SipHash: A Fast Short-Input PRF“. In: *Progress in Cryptology - INDOCRYPT 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 489–508. ISBN: 978-3-642-34931-7. DOI: 10.1007/978-3-642-34931-7_28.

[2] A. Balint et al., eds. *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*. English. Vol. B-2013-1. Department of Computer Science Series of Publications B. Finland: University of Helsinki, 2013. DOI: 10138/40026.

[3] T. Balyo, M. Heule, and M. Jarvisalo. „SAT Competition 2016: Recent Developments“. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31.1 (Feb. 2017). DOI: 10.1609/aaai.v31i1.10641.

[4] A. Biere et al. „CaDiCaL 2.0“. In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*. Vol. 14681. Lecture Notes in Computer Science. Springer, 2024, pp. 133–152. DOI: 10.1007/978-3-031-65627-9_7.

[5] E. Clarke et al. „Bounded Model Checking Using Satisfiability Solving“. In: *Formal Methods in System Design* 19.1 (July 2001), pp. 7–34. ISSN: 1572-8102. DOI: 10.1023/A:1011276507260.

[6] S. A. Cook. „The complexity of theorem-proving procedures“. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047.

[7] L. Cruz-Filipe et al. „Efficient Certified RAT Verification“. In: *Automated Deduction – CADE 26*. Cham: Springer International Publishing, 2017, pp. 220–236. ISBN: 978-3-319-63046-5. DOI: 10.1007/978-3-319-63046-5_14.

[8] A. Darwiche. „Three modern roles for logic in AI“. In: *Electronic Proceedings in Theoretical Computer Science* 326 (Sept. 2020). ISSN: 2075-2180. DOI: 10.4204/eptcs.326.

[9] J. Dean and S. Ghemawat. „MapReduce: simplified data processing on large clusters“. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[10] A. Haberlandt, H. Green, and M. J. H. Heule. „Effective Auxiliary Variables via Structured Reencoding“. In: *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Vol. 271. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. ISBN: 978-3-95977-286-0. DOI: 10.4230/LIPIcs.SAT.2023.11.

[11] M. Heule, N. Manthey, and T. Philipp. „Validating Unsatisfiability Results of Clause Sharing Parallel SAT Solvers“. In: July 2014. DOI: 10.29007/6vwg.

[12] M. J. Heule, W. A. Hunt, and N. Wetzler. „Trimming while checking clausal proofs“. In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 181–188. DOI: 10.1109/FMCAD.2013.6679408.

[13] D. Kaufmann, A. Biere, and M. Kauers. „Incremental column-wise verification of arithmetic circuits using computer algebra“. In: *Formal Methods in System Design* 56.1 (Dec. 2020), pp. 22–54. ISSN: 1572-8102. DOI: 10.1007/s10703-018-00329-2.

[14] D. E. Knuth. „backus normal form vs. Backus Naur form“. In: *Commun. ACM* 7.12 (Dec. 1964), pp. 735–736. ISSN: 0001-0782. DOI: 10.1145/355588.365140.

[15] N. Manthey and T. Philipp. „Checking Unsatisfiability Proofs in Parallel“. In: *Proceedings of Pragmatics of SAT 2015 and 2018*. Vol. 59. EPiC Series in Computing. EasyChair, 2019, pp. 34–49. DOI: 10.29007/8w4v.

[16] F. Marić. „Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL“. In: *Theoretical Computer Science* 411.50 (2010), pp. 4333–4356. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2010.09.014.

[17] J. P. Marques-Silva and K. A. Sakallah. „Boolean satisfiability in electronic design automation“. In: *Proceedings of the 37th Annual Design Automation Conference*. DAC '00. Los Angeles, California, USA: Association for Computing Machinery, 2000, pp. 675–680. ISBN: 1581131879. DOI: 10.1145/337292.337611.

[18] D. Michaelson et al. „Producing Proofs of Unsatisfiability with Distributed Clause-Sharing SAT Solvers“. In: *Journal of Automated Reasoning* 69.2 (May 2025), p. 12. ISSN: 1573-0670. DOI: 10.1007/s10817-025-09725-w.

[19] D. Michaelson et al. „Unsatisfiability proofs for distributed clause-sharing SAT solvers“. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 2023, pp. 348–366. DOI: 10.1007/978-3-031-30823-9_18.

[20] F. Pollitt, M. Fleury, and A. Biere. „Faster LRAT Checking Than Solving with CaDiCaL“. In: *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Vol. 271. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. ISBN: 978-3-95977-286-0. DOI: 10.4230/LIPIcs.SAT.2023.21.

[21] P. Sanders and D. Schreiber. „Decentralized Online Scheduling of Malleable NP-hard Jobs“. In: *International European Conference on Parallel and Distributed Computing*. Springer. 2022, pp. 119–135. DOI: 10.1007/978-3-031-12597-3_8.

[22] P. Sanders and T. N. Uhl. „Engineering a Distributed-Memory Triangle Counting Algorithm“. In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2023, pp. 702–712. DOI: 10.1109/IPDPS54959.2023.00076.

[23] P. Sanders et al. „Sorting and Selection“. In: *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Cham: Springer International Publishing, 2019, pp. 153–210. ISBN: 978-3-030-25209-0. DOI: 10.1007/978-3-030-25209-0_5.

[24] D. Schreiber. „Scalable SAT Solving and its Application“. PhD thesis. Karlsruhe Institute of Technology, 2023. DOI: 10.5445/IR/1000165224.

[25] D. Schreiber. „Trusted Scalable SAT Solving with On-The-Fly LRAT Checking“. In: *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*. Vol. 305. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 25:1–25:19. ISBN: 978-3-95977-334-8. DOI: 10.4230/LIPIcs.SAT.2024.25.

[26] D. Schreiber and P. Sanders. „MallobSat: Scalable SAT Solving by Clause Sharing“. In: *Journal of Artificial Intelligence Research (JAIR)* (2024). DOI: 10.1613/jair.1.15827.

[27] S. H. Skotåm. „CreuSAT, Using Rust and Creusot to create the world's fastest deductively verified SAT solver“. Master's Thesis. University of Oslo, 2022. DOI: 10852/96757.

[28] M. Soos, K. Nohl, and C. Castelluccia. „Extending SAT Solvers to Cryptographic Problems“. In: *Theory and Applications of Satisfiability Testing - SAT 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 244–257. ISBN: 978-3-642-02777-2. DOI: 10.1007/978-3-642-02777-2_24.

[29] „Sorting and Selection“. In: *Algorithms and Data Structures: The Basic Toolbox*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 99–125. ISBN: 978-3-540-77978-0. DOI: 10.1007/978-3-540-77978-0_5.

[30] N. Wetzler, M. J. H. Heule, and W. A. Hunt. „DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs“. In: *Theory and Applications of Satisfiability Testing – SAT 2014*. Cham: Springer International Publishing, 2014, pp. 422–429. ISBN: 978-3-319-09284-3. DOI: 10.1007/978-3-319-09284-3_31.

[31] N. Wirth. „What can we do about the unnecessary diversity of notation for syntactic definitions?“ In: *Commun. ACM* 20.11 (Nov. 1977), pp. 822–823. ISSN: 0001-0782. DOI: 10.1145/359863.359883.

[32] N. Yogananda Jeppu, T. Melham, and D. Kroening. „Enhancing active model learning with equivalence checking using simulation relations“. In: *Formal Methods in System Design* 61.2 (Dec. 2022), pp. 164–197. ISSN: 1572-8102. DOI: 10.1007/s10703-023-00433-y.