A Natively Parallel Proof Framework for Clause-Sharing SAT Solving

Ruben Götz
Karlsruhe Institute of Technology, Germany

Michael Dörr
Karlsruhe Institute of Technology, Germany

Dominik Schreiber
Karlsruhe Institute of Technology, Germany

Abstract

Unsatisfiability proofs are valuable artifacts in propositional satisfiability (SAT) since they can provide correctness guarantees and thus complete trust in reported results. In powerful parallel and distributed clause-sharing SAT solvers, existing proof technology either funnels all solver threads' relevant reasoning steps into a single proof file, which leads to scalability problems for large setups and long running times, or checks proof information in parallel in real-time, which is fully scalable but leaves no persistent artifact. We suggest an alternative approach to achieve the best of both worlds. Specifically, we consider *parallel proof files* that are logged and also checked in parallel. To this end, we introduce PalRUP– an LRUP-based proof format and a bottleneck-free, decentralized parallel checking procedure that only uses the (parallel) file system and is composed of a set of small, sequential trusted components. In evaluations on up to 3072 cores, we observe that our approach allows for low-overhead proof logging during solving and substantially outscales prior proof producing approaches in terms of checking performance.

2012 ACM Subject Classification Hardware → Theorem proving and SAT solving

Keywords and phrases Satisfiability, Proofs, Distributed computing

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.17

Supplementary Material
Software (Source Code): `https://github.com/rubenGoetz/PalRUP-Check` [13]
archived at `swh:1:dir:f9333873d989dbe91482ece76696a5d3d4d5d4c8`
Software (Source Code): `https://github.com/domschrei/mallob` [33]
archived at `swh:1:dir:63e95b4ba914b33387dbf6e31babbc7b9c3904dc`
Dataset: `https://github.com/rubenGoetz/SAT26_experimental_data` [14]
archived at `swh:1:dir:18d5082193154a230bf5993b5f6d1b6866a69627`

Funding This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), project number 565407588 (Project: Propositional Proofs As Big Data). The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`https://www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Super-computer SuperMUC-NG at Leibniz Supercomputing Centre (`https://www.lrz.de`). Some of this work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research (Germany).

Acknowledgements The authors wish to thank Jakob Nordström and Bart Bogaerts for a fruitful discussion at SAT'24 that provided the initial inspiration for the parallel checking approach presented in this paper.

1 Introduction

Proofs of unsatisfiability have become a core part of the theory, practice, and empirical research surrounding propositional satisfiability (SAT) solving. They are essential artifacts for many reasons: Since proofs provide a *correctness guarantee*, they are valuable for several use cases of SAT solving, such as the verification of critical systems [39]. In mathematical theorem proving, proofs are necessary to reach the high level of confidence and trust that is required in the result and to allow other parties to independently confirm it (see [15, 18, 21, 35]). They serve as certificates for the safety or correctness of a system [40], can aid the debugging of wrong solver behavior [26], and can be used to identify unsatisfiable subsets of a formula [2]. Lastly, gathering proofs from SAT solving runs allows to analyze the behavior, heuristic choices, efficiency, and parallelizability of SAT solving systems [11, 20, 34].

Facing an increasing technological shift towards many-core systems and high-performance computing, parallel and distributed SAT solving has become an increasingly relevant topic [1]. Existing proof support for large-scale *clause-sharing* SAT solvers, which represent the state of the art in general-purpose parallel SAT solving [32], is mostly limited to two approaches. In the first approach, all solver threads log partial proof files and then funnel all of the globally required proof steps into a single, *monolithic* proof file [24]. The second approach checks the solver threads' reasoning in real time and transfers correctness guarantees for clauses across checkers via trusted fingerprints [29]. As such, large-scale proof technology still features an undesirable gap between monolithic proof production, which ultimately does not scale because a potentially huge proof is *sequentially* written and checked, and scalable real-time proof checking, which does not provide a persistent, transferable artifact. We argue that the underlying issue is that distributed solvers are being forced to abide by the standards of sequential proof technology, which has not been designed for this purpose and is therefore insufficient to achieve full scalability. While this sentiment is not entirely new [16, 36], prior efforts on scalable proof checking focused on proofs obtained via *search space splitting* – a parallelization paradigm that allows for relatively simple separation of concerns [18, 25, 36] but is usually being outperformed by clause-sharing solving [1, 32].

In this work, we bridge this research gap by devising a formal, algorithmic, and practical framework for the production and validation of propositional proofs that is *natively* designed for (massively) parallel computations and, as such, fully scalable. Specifically, our proof format PalRUP (Parallel LRUP) defines a proof as a set of p *proof fragments* that can be distributed across machines. In addition to bottleneck-free parallel proof logging, we propose an approach to check PalRUP proofs in parallel, i.e., at the same scale at which they were produced. We prove the soundness of this approach and discuss how we orchestrate our checking setup with a number of small, sequential trusted programs whose pre– and postconditions we connect to each other. We also discuss the employed algorithms of our checking approach and provide rough asymptotic complexities. Experiments at up to 3072 cores show that the running time overhead of PalRUP proof logging during solving is minor, similar to sequential proof logging, whereas our proof checking substantially outperforms conventional checking of proofs obtained from prior approaches. While our implementation is not yet formally verified, it is designed to realistically achieve this feat in future work.

This paper is structured as follows. We provide some necessary background on parallel SAT solving, proofs of unsatisfiability, and their intersection in Section 2. Subsequently, Section 3 introduces our proof format and parallel checking approach and features a proof of its correctness. In Section 4, we present our practical setup and discuss relevant technical as well as algorithmic considerations. Section 5 features a thorough experimental evaluation of our work, and a conclusion with directions to future work is given in Section 6.

2 Background

We assume basic familiarity with propositional logic. In order to decide whether a given propositional formula in Conjunctive Normal Form (CNF; i.e., a set of clauses) is satisfiable, most modern satisiability (SAT) solvers [4] build upon the *Conflict-Driven Clause Learning* (CDCL) paradigm [22], which entails a careful search of the space of partial variable assignments while deriving and maintaining *conflict clauses* from encountered conflicts.

LRUP Proofs. An LRUP[1] [7] proof of unsatisfiability $U = \langle l_1, \ldots, l_n \rangle$ consists of n *proof statements*, each of which either *adds* (a.k.a. produces, learns, derives) or *deletes* clauses. A clause addition $l = (id, c, D_c)$ features a clause ID id, a clause (i.e., set of literals) c, and a sequence of IDs $D_c = \langle d_1, d_2, \ldots, d_k \rangle$ that are called the *dependencies* (a.k.a. requirements, hints) of c. A deletion statement $l = (D)$ simply denotes a list of clause IDs.

An LRUP proof U corresponds to a specific CNF formula $F = c_1 \wedge \ldots \wedge c_o$. F and U together implicitly define the progression of a *clause table* C that maps certain IDs $ids(C)$ to clauses $cls(C)$. Initially, $C := \{i \mapsto c_i \mid 1 \leq i \leq o\}$, i.e., IDs 1 through o are assigned to the o input clauses. An addition $l = (id, c, D_c)$ requires that $id \notin ids(C)$, that $d \in ids(C)$ for each $d \in D_c$, and that c is *entailed* by $cls(C)$. In a sound LRUP proof, this can be checked by setting each literal in c to FALSE and traversing the sequence of clauses referenced by D_c: All non-final clauses have to reduce to unit clauses and the final clasue has to reduce to the empty clause (i.e., a conflict) [7]. This confirms that $cls(C) \wedge \neg c$ is unsatisfiable and therefore $cls(C) \models c$. C is then extended by $C[id] := c$. A deletion $l = (D)$ requires that $d \in ids(C)$ for each $d \in D$ and modifies C by removing the entry for each such d. If U meets the mentioned requirements and features the addition of $c = \emptyset$, then U *certifies* the unsatisfiability of F (since the empty clause transitively follows from F). In contrast to other proof formats (such as DRUP [17]), an LRUP proof's explicit dependencies allow to check each step's validity without the need for propagating assignments through the entire clause set. This allows for highly efficient proof checking [26].

Parallel SAT. There are two major approaches to parallelize SAT solving [1]. The first approach is to split the formula at hand into disjoint sub-problems by assigning complementary values to a subset of its variables. These sub-problems are then solved independently and the results can be combined. This *search space splitting* approach allows for rather scalable proof production, since proofs for independent sub-problems can be stitched together [25] and can even be checked in parallel [16, 36]. Search space splitting is a successful method for solving some long-standing open problems of mathematics on massively parallel hardware, especially with hand-crafted, problem-dependent splitting heuristics [18, 35, 15], but it does not perform convincingly on diverse, cross-application instance sets [1].

The second approach to parallelizing SAT is to run a number of sequential solvers on the original, unmodified problem and to induce orthogonal search behavior by deviating solver configurations (*portfolio*) and/or by exchanging learned clauses between the solvers (*clause sharing*) [1]. Effective clause sharing in particular [32] allows modern distributed SAT solvers to exploit thousands of parallel cores (even if the same solver configuration is employed at all cores [31]) – reducing running times on many hard inputs from hours to minutes or even seconds [12] and allowing to conquer previously infeasible inputs [32].

[1] LRUP is a subset of the more commonly known LRAT proof format. LRUP only supports adding *entailed* clauses, whereas (actual) LRAT reasoning steps are still largely unsupported by parallel solvers.

However, obtaining proof certificates from clause-sharing solvers is a much more complex endeavor than from explicit search space splitting due to dense dependencies between the solvers' lines of reasoning, induced by shared clauses [19, 23]. Earlier approaches on this matter [10, 19] perform synchronized proof logging to a single proof. They have been met with very limited success, mostly resulting in prohibitively high proof checking cost.

Proof technology for clause-sharing solvers. In 2023, Michaelson et al. introduced a viable method to let clause-sharing solvers emit proofs [23, 24]. Each solver thread logs LRUP proof information (without deletion statements) to its own partial proof file, whereas all clause IDs are kept *globally* unique via modulus arithmetic. Each shared clause is transferred together with its ID and can then be referenced in another partial proof file just like an original input clause. Once a solver unit concludes unsatisfiability, the global proof is assembled: Each solver unit reads its partial proof in reverse while collecting all transitive dependencies of the "winning" solver unit's conclusion step. Remote dependencies (due to clause sharing) are sent back to the respective solver unit of origin and then traced from there. All collected derivations are streamed into a single, still *inverted* proof file. Clause deletion statements can be injected whenever a clause ID occurs for the first time in the inverted stream. The proof can be checked by a conventional LRUP checker after un-inverting the file or by a specialized LRUP checker that operates directly on the inverted file [24].

In evaluations with 1520 parallel cores [24], assembling (and checking) a proof roughly took once (and twice) the solving time on average. Funneling all required proof information (which can exceed hundreds of gigabytes) into a single, monolithic proof still constitutes a scalability bottleneck, especially considering that proof volume increases with the degree of parallelism [23]. Similarly, checking does not scale since it is a sequential operation. A more subtle issue is the approach's main memory consumption during checking: While the idiom "*If solving fit into RAM, checking will as well*" is sensible in sequential SAT solving, a combined proof from distributed solving can feature too many concurrently active clauses for a single machine's main memory. Michaelson et al. did observe both checking timeouts and a checker running out of memory [24]. We expect both of these issues to become increasingly common when running solvers at larger scales (e.g., 3072 cores, the largest scale for standalone SAT solving to date [32]) or for longer periods.

In an effort to overcome these bottlenecks, Schreiber suggested to check a distributed solver's LRUP proof information *in real-time* during solving [29]: Each solver thread forwards its LRUP proof steps to a dedicated checker process (one per solver thread) via named (UNIX) pipes.[2] Each checked clause derivation is returned to the solver process together with a *fingerprint* that only the checkers are able to compute. Clauses are then shared together with their fingerprint. To make use of an external clause, a solver thread first needs to forward the clause and its fingerprint to its checker process, which checks the fingerprint to confirm that the clause has already been checked by another checker. Compared to Michaelson et al.'s monolithic proof production [23], this bottleneck-free approach reduces the overhead of proof checking by at least an order of magnitude. That said, the procedure does not yield a persistent artifact that can be (re-)checked later. As such, the approach is only appealing in settings where a high level of confidence is desirable but a persistent proof is expendable. Moreover, the approach relies on trusted communication channels across checker units via fingerprints, which limits the confidence offered by the approach at least at a formal level.

[2] In terms of program code, this inter-process channel is indistinguishable from plain linear file I/O.

3 The PaIRUP Framework

We now present our parallel proof framework PaIRUP (Parallel LRUP).

3.1 Proof Format

A PaIRUP proof $\mathcal{P}$ is a concept that encompasses $p \geq 1$ numbered proof files (one for each solver thread), called *fragments*, which can be distributed over several machines, both during solving and logging as well as during checking. The format of a PaIRUP proof fragment is heavily based on LRUP and, particularly, features statements identical to LRUP for clause addition and deletion. PaIRUP fragments additionally feature *clause import* statements, which provide the ID and literals of a clause but no dependencies. Intuitively, a PaIRUP-producing solver backend outputs an import statement whenever it imports a shared clause learned by another solver unit. The import statement establishes an explicit reference to this imported clause at the proof level – unlike the implicit usage of imported clauses in monolithic proof production [24] but more akin to import directives in ImpCheck's real-time checking [29]. Just like original and learned clauses, imported clauses can be referenced by ID in subsequent derivations. They can also be (locally) deleted and are then no longer referenceable.

Each PaIRUP proof $\mathcal{P}$ is implicitly associated with a certain CNF formula F with o original input clauses. We write $P \in \mathcal{P}$ if P is a proof fragment of $\mathcal{P}$. For any $P \in \mathcal{P}$, we define $L(P) := \{(id, c) : P \text{ features a derivation of } c \text{ with ID } id\}$ and $I(P) := \{(id, c) : P \text{ features an import of } c \text{ with ID } id\}$ as the set of learned and imported clauses in P, respectively. Both definitions extend to the entire $\mathcal{P}$ in a natural manner: $L(\mathcal{P}) = \bigcup_{P \in \mathcal{P}} L(P)$ and $I(\mathcal{P}) = \bigcup_{P \in \mathcal{P}} I(P)$. We further introduce $C(\mathcal{P}) := L(\mathcal{P}) \cup I(\mathcal{P})$ and $C(P) := L(P) \cup I(P)$ as the set of *all* (non-original) clauses in $\mathcal{P}$ and $P \in \mathcal{P}$ respectively.

We introduce four *syntactical properties* that every PaIRUP proof $\mathcal{P}$ needs to satisfy:

- **P0** **Input ID domain**: Any $(id, c) \in C(\mathcal{P})$ must satisfy $id > o$.
- **P1** **Locality of assigned IDs**: In the i-th fragment P_i ($i \in \{0, \ldots, p-1\}$), any $(id, c) \in L(P_i)$ must satisfy $id \equiv i \mod p$.
- **P2** **Monotonicity of assigned IDs**: For each $P \in \mathcal{P}$, any pair $(id, c), (id', c') \in L(P)$ where (id, c) precedes (id', c') must satisfy $id < id'$.
- **P3** **Monotonicity of hints**: For any $(id, c) \in L(\mathcal{P})$, all $d \in D_c$ must satisfy $d < id$.

These properties are trivial to check immediately (and locally) when checking a proof fragment. We later explain how practical solver implementations can be made to obey them with little effort. P1 ensures that each clause ID can be mapped to a unique producer whereas P2 and P3 ensure the unambiguous mapping of each clause to a unique ID and the *acyclicity* of the global proof (i.e., that clauses can never cyclically depend on each another).

Relation to existing formats. Given a PaIRUP proof fragment P for formula F, it is straight forward to construct an equivalent LRUP proof in terms of internal consistency (i.e., all learned clauses being entailed by $F \cup I(P_i)$) by moving all imported clauses to the problem definition F and thus extending their scope to the entire proof (see Appendix A). This measure is necessary since LRUP does not support delayed addition of unchecked clauses. Declaring imports at any point rather than upfront allows PaIRUP to capture dynamic clause sharing with a single, linear file. In fact, a solver running for an extended period can accumulate a very large trace of imports, which may be infeasible to keep in main memory all at once and could therefore render pure, unmodified LRUP checking impractical.

The proof format LIDRUP [8] extends LRUP by incremental user interaction loops, which includes explicit transitions between *solving* and *input* stages and the ability to add input clauses (with custom IDs) during each *input* stage. Exploiting this mechanism, a PaIRUP

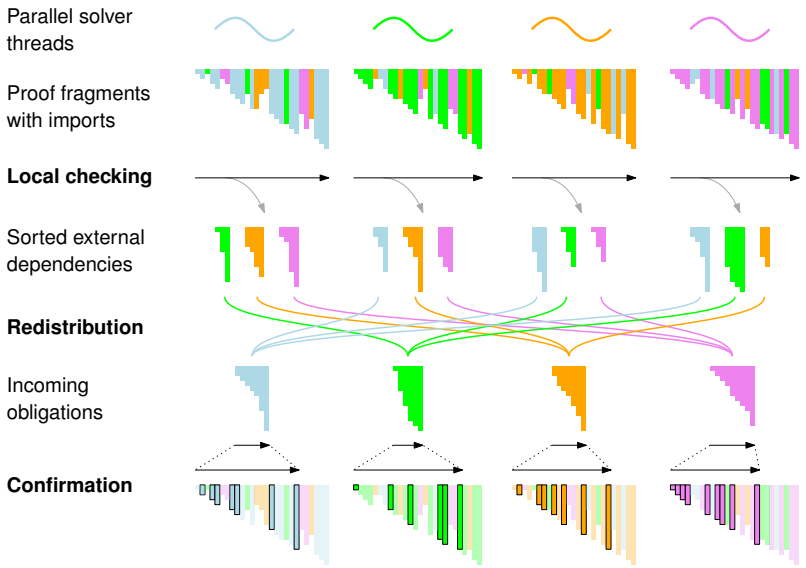


Figure 1 Illustration of PalRUP checking procedure with four solver units (and hence four checker units), chronologically from top to bottom. Each colored bar corresponds to a clause, with the color indicating the producing solver unit and the length indicating the clause ID.

fragment can be translated directly and faithfully to a LIDRUP proof (see Appendix A). That said, the resulting LIDRUP artifact is relatively verbose, and LIDRUP checkers are designed to handle many advanced features that are not required for our purposes.

3.2 Checking

Given a PalRUP proof with fragments $P_0, \ldots, P_{p-1}$, a PalRUP checking procedure runs in parallel p corresponding sequential checker programs, called *pals*. Each pal i processes its associated fragment P_i in three stages as illustrated in Fig. 1:

1. **Local checking:** Traverse P_i. During this traversal, check P0–P3 and check/process each addition and deletion like a conventional LRUP checker. Treat each clause *import* as a learned clause that needs no checking, and accumulate the full set $I(P_i)$.
2. **Redistribution:** Send each clause $(id, c) \in I(P_i)$ to pal $id \bmod p$ and accordingly receive the set $E(P_i) := \{(id, c) \in I(\mathcal{P}) \mid id \equiv i \bmod p\}$ from the other pals.
3. **Matching** (or **Confirmation**): Check $E(P_i) \subseteq L(P_i)$ in a single linear pass.

Subsequently, each pal outputs OK if no errors occurred and additionally outputs UNSAT if the empty clause was learned in its fragment. A proof is considered *accepted* by the parallel procedure if and only if all p pals output OK and at least one pal outputs UNSAT.

The local checking step is similar to running p conventional LRUP proof checkers independently on p LRUP proof files. The main difference are imported clauses, which are added to the pal's clause table without any immediate checks. The redistribution step ensures that each clause imported in some fragment arrives at some (other) fragment. The confirmation step then ensures that all clauses received from the redistribution step are in fact valid learned clauses of the local fragment that, importantly, depend only on *strictly earlier* clauses.

3.3 Correctness

We now show the correctness of PaIRUP, i.e., that any PaIRUP proof accepted by our parallel checking procedure does in fact confirm the unsatisfiability of the corresponding formula.

We call a PaIRUP proof $\mathcal{P}$ for a formula F *sound* iff all pals in a parallel checking run accept $(F, \mathcal{P})$, and we define $\mathcal{P}$ as *valid* iff $\mathcal{P}$ is sound and $(id, \emptyset) \in L(\mathcal{P})$ for some id. The following lemma forms the basis for our approach's correctness:

▶ **Lemma 1.** *If* PaIRUP *proof $\mathcal{P}$ is sound for formula F, $F \models c$ for each $(id, c) \in C(\mathcal{P})$.*

Proof. Since $\mathcal{P}$ is sound, we know that there is an ordering $P_0, \ldots, P_{p-1}$ of the fragments of $\mathcal{P}$ that complies with the basic syntax of PaIRUP and properties P0–P3. The sets $L(\cdot)$, $P(\cdot)$, and $C(\cdot)$ are also well-defined. We define $C^x := \{c \mid (id, c) \in C(\mathcal{P}), id \leq x\}$, i.e., all clauses with an ID no larger than x that occur as learned or imported clauses in any fragment of a PaIRUP proof $\mathcal{P}$. We proceed to prove $\forall\, c \in C^x : F \models c$ via induction over x.

For $x \leq o$, Property P0 (input ID domain) ensures that $C^x = \emptyset$. Next, for any x, we assume $\forall\, c \in C^x : F \models c$ and are left to show that any clause $c \in C^{x+1}$ is entailed by F. If $c \in C^x$, there is nothing to show. Otherwise, $(x+1, c) \in C(\mathcal{P})$. Let us first consider all cases of an import statement $(x+1, c) \in I(P)$ in some $P \in \mathcal{P}$. The pal of P sends $(x+1, c)$ to the pal of $P' := P_{x+1 \bmod p}$. Property P1 (locality of IDs) ensures that P' is the only fragment P_i that may satisfy $(x+1, c) \in L(P_i)$. The pal of P' subsequently checks that $(x+1, c) \in L(P')$. Property P2 (monotonicity of IDs) further ensures that such a clause addition can only occur once in P'. As such, we know that *all* imports with ID $x+1$ in $\mathcal{P}$ originate from one globally unique clause addition in $\mathcal{P}$ and thus all correspond to the exact same literals, c. It remains to show that an addition $l = (x+1, c, D)$ in some P implies $F \models c$ (regardless of whether or not $\mathcal{P}$ features any imports of c). P3 (monotonicity of dependencies) ensures $D \subseteq F \cup C^x$. During local checking, the responsible pal performs a RUP check on l to confirm $D \models c$, implying $F \cup C^x \models c$. Due to induction, we know that $F \models C^x$ and therefore $F \models c$. ◀

▶ **Theorem 2.** *If a* PaIRUP *proof $\mathcal{P}$ is valid for a formula F, then F is unsatisfiable.*

Proof. For $\mathcal{P}$ to be valid, it needs to be sound and contain the empty clause. If this is the case, it follows from Lemma 1 that $F \models \emptyset$ and thus F is unsatisfiable. ◀

4 Implementation

In the following, we present our practical setup as well as the algorithms and techniques that are relevant to render PaIRUP a viable, efficient, and dependable framework in practice.

4.1 Proof Logging

An LRUP-logging solver backend, such as CaDiCaL in prior works [24, 26, 29], can be modified to log PaIRUP proof information via a few small changes. First, the solver needs to log import statements whenever it imports a shared clause. Secondly, P1–P3 must be ensured, either by reworking the solver's internal clause ID assignment or by introducing additional program logic in the proof logging module to map between solver-internal and *adjusted* clause IDs. In our implementation, we applied the latter option for the sake of genericity.

Our proof logger maintains a table H that maps solver-internal IDs to adjusted IDs. While the solver backend continues to use original, unmodified clause IDs, we always use adjusted IDs when exporting the clause for sharing purposes and when writing proof lines.

Given a clause addition (id, c, D) with hints $D = \langle d_1, \ldots, d_k \rangle$, we first use H to replace each hint $d > o$ in D with $H[d]$. We take the previous assigned ID id'_{prev} and find the maximum hint ID $d'_{max} = \max\{D'\}$ to then compute the *target ID* $t = \max(d'_{max}, id'_{prev}) + 1$, i.e., the minimum viable ID that respects both P2 (Monotonicity of assigned IDs) and P3 (Monotonicity of hints). To satisfy P1 (Locality of assigned IDs), we increase t until it is congruent to i mod p, which yields the final *adjusted ID* id'. Finally, we add the mapping $id \mapsto id'$ to H and output the clause addition (id', c, D').

Given a deletion statement for clause IDs $D = \langle d_1, \ldots, d_k \rangle$, we again translate the IDs to D' as above (*before* outputting the statement) and also delete each according entry $d \mapsto d'$ from H. Import statements (id, c) do not need to be modified since id originates from a different solver unit and has therefore already been adjusted in that context.

Similar to existing binary file formats for DRAT [17] and LRAT [7], our file format contains proof information in a compact and lightly compressed binary representation based on a *variable-bytelength* coding. Each number (i.e., literal or ID) is written using seven out of eight bits per byte, where the eighth bit indicates if the number continues in the next byte.

4.2 Proof Checking Setup

We designed our parallel proof checking approach based on the following principles:

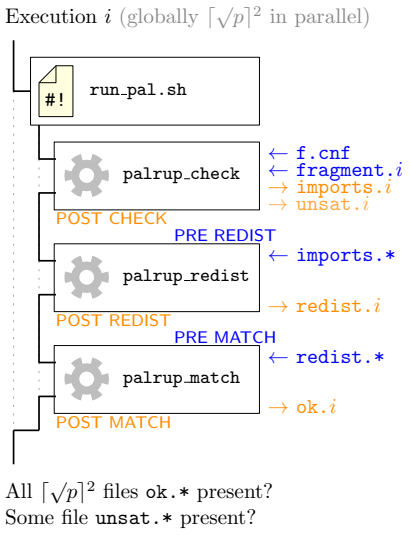
1. Each proof fragment is only accessible at one machine of the distributed environment.
2. Each pal for a fragment P should asymptotically only consume as much main memory as is required for checking its proof (i.e., memory according to the clause table maintained by a LIDRUP checker when checking an equivalent proof, see Section 3.1).
3. Communication between pals is limited to the well-defined exchange of a few plain *files*.

The third principle is motivated by our aim to enable bottleneck-free parallel and distributed proof checking without relying on complex and error-prone network communication libraries. This way, assuming that all checker units have shared access to some (parallel) file system, the mode of operation is comparable to conventional (sequential) proof checking and does not require any added dependencies at the application level. Indeed, a purely sequential execution model for individual trusted programs may greatly facilitate a potential formally verified implementation (cf. [36]), which our setup is specifically designed to prepare for.

At a technical level, our PalRUP checking setup is based on three distinct trusted C programs: `palrup_check`, `palrup_redist`, and `palrup_match`. Each such program is purely sequential and performs plain file I/O with no further interfaces or dependencies. We compose these individually trusted programs with a thin scripting layer that calls $\mathcal{O}(p)$ incarnations of the three programs and connects their pre– and postconditions.

Fig. 2 illustrates this setup and specifies according pre– and postconditions. Firstly, each pal's run of `palrup_check` corresponds to the local checking stage. It has no preconditions and ensures that, *if* it wrote a file `imports.`i, its proof fragment P_i is internally consistent and `imports.`i contains all imports $I(P_i)$. Next, to redistribute clause imports, we arrange all pals in a *quadratic grid*[3] for efficiency reasons (see Section 4.3 for details). Each run of `palrup_redist` reads all imports from its row in this grid and, given the postconditions of its row's checking steps as preconditions, ensures that its successful output `redist.`i contains all clauses imported in its row that originate from its column. Lastly, each run of `palrup_match` relies on the preconditions of its column's `palrup_redist` executions to ensure that, if `ok.`i is present, then *all* pals checked their fragments and all exports $E(P_i)$ from proof fragment

[3] Some "dummy" pals are spawned with empty inputs to make up for the difference between p and $\lceil\sqrt{p}\rceil^2$.



$\mathsf{row}(i) := \{j \in \{0,\dots,\lceil\sqrt{p}\rceil^2 - 1\} \mid i \operatorname{div} \lceil\sqrt{p}\rceil = j \operatorname{div} \lceil\sqrt{p}\rceil\}$
$\mathsf{col}(i) := \{j \in \{0,\dots,\lceil\sqrt{p}\rceil^2 - 1\} \mid i \bmod \lceil\sqrt{p}\rceil = j \bmod \lceil\sqrt{p}\rceil\}$
$\mathsf{SOUND}(i) \equiv \mathit{sound}(P_i) \wedge (\mathit{file}(\texttt{unsat}.i) \Rightarrow \emptyset \in C(P_i))$

POST CHECK:
$\mathit{file}(\texttt{imports}.i) \Rightarrow (\mathsf{SOUND}(i) \wedge I(P_i) \in \texttt{imports}.i)$

PRE REDIST:
$\forall j \in \mathsf{row}(i) : \mathit{file}(\texttt{imports}.j) \Rightarrow$
$(\mathsf{SOUND}(j) \wedge I(P_j) \in \texttt{imports}.j)$

POST REDIST:
$\mathit{file}(\texttt{redist}.i) \Rightarrow \forall j \in \mathsf{row}(i) : (\mathsf{SOUND}(j) \wedge \forall k \in \mathsf{col}(i) :$
$\{(id, c) \in I(P_j) \mid id \equiv k \bmod p\} \in \texttt{redist}.i)$

PRE MATCH:
$\forall j \in \mathsf{col}(i) : \mathit{file}(\texttt{redist}.j) \Rightarrow (\forall k \in \mathsf{row}(j) : \mathsf{SOUND}(k) \wedge$
$\{(id, c) \in I(P_k) \mid id \equiv i \bmod p\} \in \texttt{redist}.j)$

POST MATCH:
$\mathit{file}(\texttt{ok}.i) \Rightarrow \big((\forall j \in \{0,\dots,\lceil\sqrt{p}\rceil^2 - 1\} : \mathsf{SOUND}(j)) \wedge$
$E(P_i) \subseteq L(P_i)\big)$

Figure 2 Left: Schematic PaIRUP checking workflow with input (←) / output (→) files and PRE–/POSTcondition markers. Right: According pre– and postconditions of each trusted checker program. *file*(x) indicates whether file x exists; *sound*(P_i) indicates whether PaIRUP fragment P_i is sound (i.e., its local checking succeeded). An empty fragment P_x for $x \geq p$ is trivially sound.

i are in $L(P_i)$. The end-to-end result is obtained by checking in a surrounding script as to whether all $\lceil\sqrt{p}\rceil^2$ files ok.* and at least one file unsat.* are present, which marks the PaIRUP proof as valid and (due to Theorem 2) the according formula as unsatisfiable.

At a technical level, the i-th pal only writes files $[\dots].i$, which prevents race conditions. In practice, these files are also organized in appropriate sub-directories for better file system performance. Furthermore, any output file expected by another program is created only at the end of a *successful* computation and by *atomically* moving a fully written intermediate file to the destination, which prevents incomplete reads. Accordingly, a pal expecting an input file repeatedly polls the file system until the file is present (or checking times out). As such, our approach is fully decentralized and features no global synchronization barriers.

4.3 Algorithms and Complexity

We now describe the algorithms and asymptotic complexity of each major checking stage.

Local checking. During local checking, the i-th pal outputs $I(P_i)$, i.e., the clauses it had to *assume* to be entailed by F. Depending on the parallel solver, we may expect high overlaps between the $I(\cdot)$ across different solver units, up to the point where all solver units import almost identical clause sets. As such, detecting and eliminating these duplicates as early as possible in the checking procedure can significantly reduce the total workload. To this end, we want to ensure that pal i outputs $I(P_i)$ *sorted* by ID in ascending order. However, the clause imports in P_i are not necessarily sorted by ID, and simply collecting $I(P_i)$ in-memory and sorting it in the end may not be feasible since $I(P_i)$ usually grows linearly with P_i and can therefore exceed the available main memory. Feasible alternatives include (a) the use of a (semi-)external-memory priority queue [27] or (b) outputting $I(P_i)$ in the input order followed by a (semi-)external sorting step.

In terms of complexity, a pal i performs a check for each RUP step in its respective fragment P_i and handles imports as outlined above. We simplifyingly assume constant-size proof lines and define that each proof fragment has at most N clause additions and at most

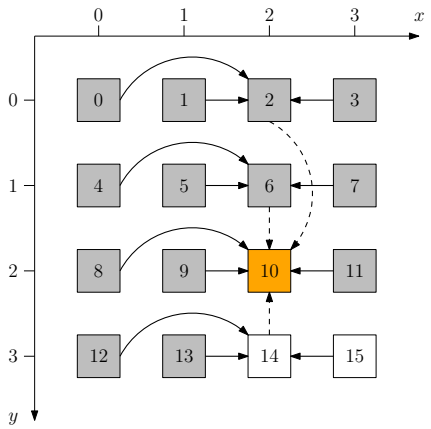


Figure 3 Example information flow to pal 10 in our redistribution scheme for 14 pals. Arrows show the information flow of clauses that were produced by pal 10 and imported by other pals. Solid line arrows show stage one and dashed line arrows show stage 2 of the redistribution. Pals 14 and 15 are spawned as "dummy" pals to serve as relays.

M clause imports. Per pal, this roughly results in $\mathcal{O}(N)$ work to perform the necessary LRUP checks,[4] in addition to $\mathcal{O}(M \log M)$ work to handle clause imports via a priority queue and/or sorting (abstracting away internal vs. external memory).

In our actual implementation, we insert each import $(id, c) \in I(P_i)$ into a priority queue Q (prioritized by id) and flush its $\alpha \cdot |Q|$ smallest elements (e.g., $\alpha = 0.5$) once Q exceeds a certain volume. If every clause import in the fragment is mispositioned by less than $k/2$ positions compared to the sorted sequence of imports, the series of flushed chunks will always form a sorted sequence of IDs. Otherwise, we encounter corner cases where an ID to flush is *smaller* than the last flushed ID. In such cases, we traverse and re-write the output written thus far while merging together its clauses and the clauses to be flushed. Given sufficient space for Q, we expect this workaround to be required very rarely for checking typical proofs, where the IDs of imported clauses generally increase over time.

Redistribution. At an algorithmic level, the redistribution step can be considered a standard *all-to-all* collective operation, i.e., each among p processing elements has a dedicated message for each of its $p - 1$ peers. The challenge is to implement this operation in an efficient and yet simple and mostly technology-independent way.

As a compromise between efficiency and simplicity, we propose a two-stage redistribution procedure based on an all-to-all communication scheme described by Sanders and Uhl [28]: As outlined earlier, we arrange all p pals in a two-dimensional[5], $\lceil\sqrt{p}\rceil \times \lceil\sqrt{p}\rceil$ grid. In a first step, all information is sent along the horizontal direction; i.e., a pal at position (x, y) in the grid sends the clauses for destination (x', y') to the pal at position (x', y). In the second step, all data is sent along the vertical direction and reaches its final destination. If p is not a square number, an additional $\lceil\sqrt{p}\rceil^2 - p$ processes are spawned to serve as relays. Fig. 3 illustrates the information flow of this procedure.

[4] Assuming (expected / amortized) constant-time bookkeeping of clauses, e.g., with a good hash table.
[5] For very large configurations, e.g., $p \geq 10\,000$, the method could be extended to $d > 2$ dimensions in a straight forward manner to reduce the workload per pal at the expense of added communication rounds.

This method requires two communication rounds with a total of $2 \cdot \lceil\sqrt{p}\rceil^2 \cdot \lceil\sqrt{p}\rceil \in \mathcal{O}(p\sqrt{p})$ sent and received messages (files) since each party handles and forwards data from/to $\lceil\sqrt{p}\rceil$ parties per step. Since we ensure that each pal in the local checking step outputs its imported clauses sorted by ID in ascending order, the data in each incoming message during redistribution is sorted. As such, $\lceil\sqrt{p}\rceil$ incoming messages can be efficiently merged (e.g., with a heap of size $\lceil\sqrt{p}\rceil$) and split into $\lceil\sqrt{p}\rceil$ sorted outputs while detecting and discarding duplicates in the process. Each message is bounded by M, implying $\mathcal{O}(\sqrt{p} \cdot M \cdot \log\sqrt{p}) \subseteq \mathcal{O}(\sqrt{p} \cdot M \cdot \log p)$ work per pal.

Confirmation. The confirmation step once again entails p independent traversals of $P_0, \ldots, P_{p-1} \in \mathcal{P}$. This time, however, the RUP steps in a proof fragment P_i do not need to be checked again. Instead, the pal merges its incoming clauses into a single sorted sequence $E(P_i)$ and scans P_i until it encounters the next clause $(id, c) \in E(P_i)$ (or fails).

In terms of complexity, $E(P_i)$ consists of $\lceil\sqrt{p}\rceil$ distinct sorted clause sequences, which can be of size $\min\{\lceil\sqrt{p}\rceil M, N\}$ each ($\lceil\sqrt{p}\rceil M$ if all imported clauses in $\mathcal{P}$ originate from P_i; N if every single learned clause in P_i was imported by another pal). Let us denote this maximum number of *exported* clauses per pal by X. Then after merging and deduplicating these sequences in $\mathcal{O}(\sqrt{p}X \log\sqrt{p}) \subseteq \mathcal{O}(\sqrt{p}X \log p)$ time, the at most N remaining clauses can be matched with the proof fragment in $\mathcal{O}(N)$ time.

Total complexity. Totalling the above asymptotic complexities for each stage, we arrive at $\mathcal{O}(N + M \log M + (M + X)\sqrt{p} \log p)$ total work per pal. Assuming $\sqrt{p} \log p \geqslant \log M$, the work simplifies to $\mathcal{O}(N + (M + X)\sqrt{p} \log p)$. As such, compared to LRUP/ LIDRUP proof checking in $\mathcal{O}(N)$, PalRUP checking expectedly incurs added work in terms of the amount of imported and exported clauses, which scales *sub-linearly* with the degree of parallelism p.

5 Evaluation

We now present the experimental evaluation of our PalRUP logging and checking pipeline. Our checker, PalRUP logging solver and experimental data are available online.

5.1 Setup

We use up to 64 nodes of SuperMUC-NG, an HPC cluster where each node has 48 physical cores and 96 GB of RAM. As in earlier works [31], we run two MallobSat processes per node with 24 solver threads per process. Due to a lack of local disks on SuperMUC-NG's nodes, we use a parallel file system both to log proofs and to exchange information across checker units. For parallel runs, we use two scales per default: one node (48 cores) and 16 nodes (768 cores). We additionally include a run of our approach at 64 nodes (3072 cores) – the largest scale at which any clause-sharing SAT solver has been evaluated to date [31].

We compare our PalRUP proof logging and checking pipeline with three state-of-the-art baselines: (a) the sequential SAT solver CaDiCaL [3] (which won the UNSAT track of SAT Competition 2025), using LRAT proof logging and checking via `lrat-trim` (without proof trimming, for best single-pass performance) [26]; (b) the monolithic proof production approach by Michaelson et al. [24], which entails running the distributed clause-sharing solver MallobSat [32] (with CaDiCaL as its solver backend) followed by assembling and then sequentially checking the produced (byte-wise inverted) proof using the efficient standalone checker bundled with Mallob [24]; and (c) the same configuration of MallobSat but with proof logging disabled entirely. Per run, we limit (a) to 5000 s (excluding the subsequent proof

Table 1 Statistics on total file sizes of successfully logged proofs in GiB. Note that CaDiCaL was run for a substantially longer wallclock time (5000 s vs. 300 s).

Solver	#	p10	median	mean	p90	max
CaDiCaL	156	0.003	0.5	1.4	4.3	14.1
Monolithic 1 node	139	0.001	0.4	2.0	5.2	25.5
Monolithic 16 nodes	152	0.002	1.2	7.4	19.6	129.8
PalRUP 1 node	136	0.001	3.4	5.9	12.1	54.2
PalRUP 16 nodes	155	0.005	33.4	68.2	173.2	797.0
PalRUP 64 nodes	154	0.393	128.4	227.2	583.1	2011.3

checking, which always finishes within a few minutes) and (b) to 300 s of solving followed by up to 1500 s *each* for assembling and for checking a monolithic proof. All other runs are limited to 300 s *each* for solving and for checking (as applicable). All of these limits are measured in terms of wallclock time.

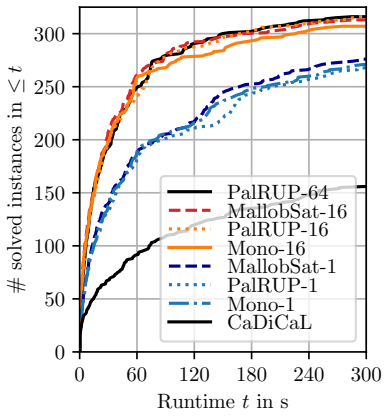
In terms of benchmarks, we use the most recent SAT Competition 2025 benchmark set [6], consisting of 400 diverse inputs that are roughly balanced in terms of satisfiable vs. unsatisfiable inputs. We generally run all approaches on all inputs, including satisfiable ones, to discern the overhead incurred in all cases.

5.2 Results

We now discuss our experimental results in regards to proof logging, its associated overhead on solving times, raw checking times and end-to-end execution times for solving and checking combined.

Proof Logging. Table 1 shows some statistics for the proofs logged in our experiments. Both the monolithic proof production and sequential LRAT proof logging yield a single proof file. For the case of monolithic proof production, the assembly of said proof file involves a backward timming procedure that discards unused clauses and hence eliminates the majority of the originally generated proof data [24]. Our approach, by contrast, does not rely on any postprocessing after solving and instead includes all reasoning steps of all solver threads in the final proof data, which in turn results in significantly larger proofs. The median proof data throughput per second per solver thread during solving is about 1.66 MiB for all scales of PalRUP and about 2 MiB for sequential CaDiCaL with LRAT output. At 3072 cores, this implies a median total PalRUP proof size of about 5 GiB per second of solving time. As such, when increasing the scale of solving, the mean PalRUP proof size increases considerably but not proportionally (e.g., by a factor of 11.6 when going from one to 16 nodes), given that larger scales usually take less time to solve an input (Fig. 7 in the Appendix further illustrates this effect). The largest PalRUP proof produced (at 64 nodes) is comprised of 2 TiB of data and belongs to the instance `cliquecoloring_n14_k7_c6.cnf`, a clique coloring formula with only 273 variables, where solving took 257.6 s and checking took 127.1 s.

Solving times. Next, we consider the impact of PalRUP proof logging on solving times. As Fig. 4 (left) shows, the modifications needed to enable PalRUP logging in a solver backend only introduce minor overhead with little differences to the overhead incurred by proof logging for monolithic proof production. Logging PalRUP proofs, unsurprisingly, does not hinder the scalability of a clause sharing solver and is viable in practice. Fig. 4 (right) shows



Setup		Solver	all #	all PAR	SAT #	SAT PAR	UNSAT #	UNSAT PAR
N	1×48	MallobSat	284	225.5	143	226.8	141	224.2
	16×48	MallobSat	318	159.9	162	156.5	156	163.3
SP	1	CaDiCaL	295	672.0	139	613.2	156	730.7
	1×48	Monolithic	273	231.9	134	233.9	139	229.9
	16×48	Monolithic	308	167.1	156	160.2	152	174.1
PP	1×48	PalRUP	274	238.1	138	235.2	136	241.0
	16×48	PalRUP	317	156.7	162	141.9	155	171.4
	64×48	PalRUP	317	155.6	163	140.3	154	170.9

Figure 4 Left: Solved instances over time for different MallobSat configurations with and without proof logging. Right: Data on solving performance for all configurations with parallel (PP), monolithic/sequential (SP), and no (N) proof logging. In both the plot and the table, *pure solving times* are considered, which also excludes proof assembly in monolithic proof production.

detailed performance data for all runs. Note that the 64-node run only performed slightly better than the 16-node run, as opposed to MallobSat's scaling behavior reported in earlier works [31, 32]. Since we did not include a 64-node run of unchecked MallobSat, we are unsure if this scalability limit is due to massively parallel proof logging to a network file system or simply a consequence of the employed solver setup and considered benchmarks.

Checking times. Let us now compare the checking performance of PalRUP to existing checking approaches. Fig. 5 (left) shows running times of the respective checking procedures in relation to the time needed to solve each instance. Note that the assembly of monolithic proof files is not taken into account. While small monolithic proofs can be checked faster than a distributed PalRUP proof, our approach scales much better to harder instances that need more than ten seconds to solve. Especially when executed on multiple nodes, our approach proves to be orders of magnitude faster than monolithic proof checking in several cases.

Fig. 6 provides more detailed insights into PalRUP checking times by showing the average running times of a pal – broken down into local checking, redistribution and confirmation – at different scales. The redistribution step is orders of magnitude faster than local checking at small scales but makes up for a significant portion of checking times at larger scales. While the size of the proof fragments is limited by the allowed timeout during solving, the communication workload scales with the number of active processors. At even larger scales beyond 3072 cores, additional measures to reduce this work may be in order, e.g., by extending our redistribution scheme by a third dimension. Some more insights on the load balance of our checking approach are given in Fig. 8 (Appendix).

End-to-end checked performance. Fig. 5 (right) illustrates *end-to-end* execution times, i.e., the time needed from starting to solve an instance to the finished validation of a produced proof of unsatisfiability. CaDiCaL's sequential proof production and checking based on LRAT is extremely efficient (as can also be seen in Fig. 5 left) but limited to sequential solving performance. Both the monolithic proof production and our approach significantly outperform the sequential approach on non-trivial instances. Since PalRUP does not feature any sequential bottleneck for the produced proof data, our approach significantly outperforms

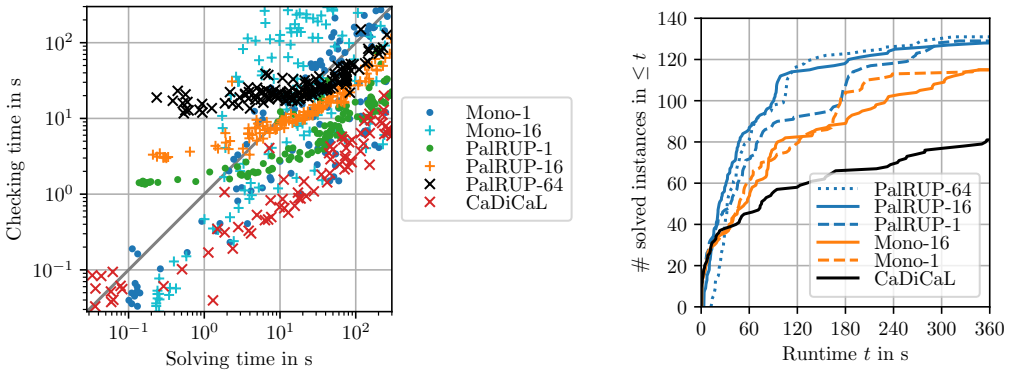


Figure 5 Left: Per-instance comparison of solving vs. checking times, including PalRUP, monolithic proof production (excluding proof assembly), and CaDiCaL with LRAT checking. Right: Solved *and checked* unsatisfiable inputs over time, i.e., including the time needed for solving and subsequently checking the produced proof (also including proof assembly for monolithic proof production).

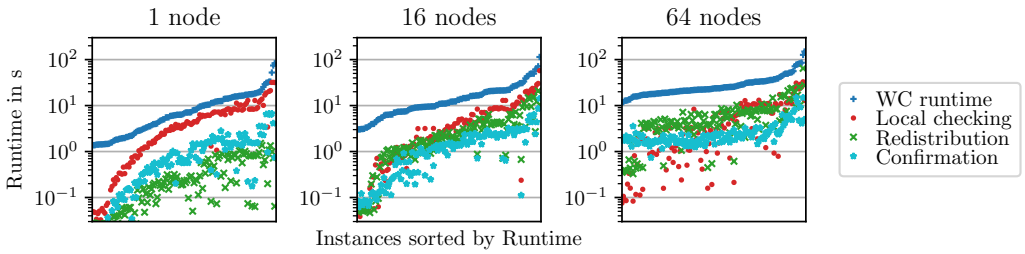


Figure 6 Average running times of our checking procedure per instance, split into stages, at 1, 16, and 64 nodes (48, 768, and 3072 cores respectively).

all prior approaches and, unlike monolithic proof production, mostly shows improved end-to-end performance when increasing the scale of solving (and checking). As indicated by the scalability problems of monolithic proof production already at 768 cores, there are no prior scalable approaches to constructing explicit proofs with a clause-sharing solver at a scale of up to 3072 cores. Our approach does prove highly scalable – oftentimes achieving faster checking than solving for solving times beyond 20–30 s (Fig. 5 left).

Without any dedicated experiments, let us also discuss the overhead of checked SAT solving via PalRUP in relation to real-time checking (without a persistent proof artifact) via ImpCheck [29, 30], the thus far only bottleneck-free proof checking for clause-sharing solvers. The latest and most efficient version of ImpCheck [30] was shown to incur running time overheads of roughly 21% (mostly scale-independent [29]) for non-trivial instances, with a solver setup essentially equivalent to ours. Our approach incurs smaller or similar overhead during solving (13%–24% depending on the scale) whereas post-hoc checking effort varies based on the solving time (see Fig. 5 left). For solving times beyond 60 s, the mean checking overhead is about 11% (29%, 48%) at 1 (16, 64) nodes. In total, we estimate that our approach incurs roughly 2–3× the overhead of ImpCheck in terms of end-to-end performance on sufficiently challenging instances and at large scales.

6 Conclusion

Parallel and distributed clause-sharing SAT solvers are powerful automated reasoning tools, but they are still held back by the lack of truly scalable methods for producing and checking proofs of unsatisfiability. In this work, we introduce the PaIRUP framework for logging and checking *parallel proofs*, including in particular a decentralized parallel proof checking method. Large-scale experiments show the promise of this method – outscaling prior proof logging approaches substantially in terms of checking time.

Formally verifying our approach and devising an according formally verified implementation of our proof checking procedure was out of scope for this work. That said, we specifically designed our setup to allow to achieve this feat in the near future. In particular, we intend to build upon a recent joint project (involving the 3rd author) on verifying the ImpCheck real-time proof checking approach [38]. This project showed that (and how) it is feasible to implement programs in the verified CakeML toolchain [37] that serve as drop-in replacements for trusted sequential core programs (see Fig. 2) while modeling the interactions between different incarnations of them with a foreign function interface (FFI) model. Applying similar techniques, we believe that we can obtain a scalable verified proof checking approach with very few added assumptions (e.g., a correct file system).

Apart from its verification, we are interested to extend and augment the PaIRUP framework regarding a number of aspects. First, we aim to further improve the practical performance of proof checking with further engineering and algorithmic improvements, such as only considering a clause import for redistribution if it was actually used in a subsequent derivation by its importing solver unit. Next, we consider to devise parallel postprocessing methods for PaIRUP proofs, including a backward trimming operation similar to the one used in monolithic proof checking [24] and/or more powerful compression techniques. Lastly, we want to extend PaIRUP to stronger proof formats such as *incremental* proofs [8], LRUP combined with variable addition [9], or (more long term) the VeriPB cutting-planes proof system [5].

References

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

2 Anton Belov, Marijn J. H. Heule, and João Marques-Silva. MUS extraction using clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 48–57, 2014. doi:10.1007/978-3-319-09284-3_5.

3 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Proc. Computer Aided Verification (CAV)*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

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

5 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, 2023. doi:10.1613/jair.1.14296.

6 Cayden Codel, Katalin Fazekas, Marijn J. H. Heule, and Ashlin Iser, editors. *Proceedings of SAT Competition 2025: Solver and Benchmark Descriptions*, 2025. doi:10.34726/10379.

7 Luis Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Automated Deduction – CADE*, pages 220–236. Springer, 2017. doi:10.1007/978-3-319-63046-5_14.

8 Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. Certifying incremental SAT solving. In Nikolaj Bjørner, Marijn Heule, and Andrei Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 100 of *EPiC Series in Computing*, pages 321–340. EasyChair, 2024. doi:10.29007/pdcc.

9 Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. Incremental inprocessing rules beyond resolution. In *Joint Proceedings of the 23rd International Workshop on Satisfiability Modulo Theories (SMT'25) and of the 16th Pragmatics of SAT International Workshop (POS'25), co-located with the 28th International Conference on Theory and Applications of Satisfiability Testing (SAT'25), Glasgow, UK*, 2025.

10 Mathias Fleury and Armin Biere. Scalable proof producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses. In *Proc. Pragmatics of SAT*, 2022. doi:10.48550/arXiv.2207.13577.

11 Rohan Fossé and Laurent Simon. On the non-degeneracy of unsatisfiability proof graphs produced by SAT solvers. In *Proc. Principles and Practice of Constraint Programming (CP)*, pages 128–143. Springer, 2018. doi:10.1007/978-3-319-98334-9_9.

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

13 Ruben Götz, Michael Dörr, and Dominik Schreiber. PalRUP-Check. Software, DFG-56540758, swhId: swh:1:dir:f9333873d989dbe91482ece76696a5d3d4d5d4c8 (visited on 2026-07-01). URL: https://github.com/rubenGoetz/PalRUP-Check, doi:10.4230/artifacts.26960.

14 Ruben Götz, Michael Dörr, and Dominik Schreiber. rubenGoetz/SAT26_experimental_data. Dataset, swhId: swh:1:dir:18d5082193154a230bf5993b5f6d1b6866a69627 (visited on 2026-07-01). URL: https://github.com/rubenGoetz/SAT26_experimental_data, doi:10.4230/artifacts.26962.

15 Marijn J. H. Heule. Schur number five. In *AAAI Conference on Artificial Intelligence*, volume 32, 2018. doi:10.1609/aaai.v32i1.12209.

16 Marijn J. H. Heule and Armin Biere. Compositional propositional proofs. In *Proc. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 444–459. Springer, 2015. doi:10.1007/978-3-662-48899-7_31.

17 Marijn J. H. Heule, Warren Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, pages 181–188. IEEE, 2013. doi:10.1109/fmcad.2013.6679408.

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

19 Marijn J. H. Heule, Norbert Manthey, and Tobias Philipp. Validating unsatisfiability results of clause sharing parallel SAT solvers. In *Proc. Pragmatics of SAT*, pages 12–25, 2014. doi:10.29007/6vwg.

20 George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *Proc. AAAI Conference on Artificial Intelligence*, volume 27, pages 481–488, 2013. doi:10.1609/aaai.v27i1.8660.

21 Boris Konev and Alexei Lisitsa. Computer-aided proof of erdős discrepancy properties. *Artificial Intelligence*, 224:103–118, 2015. doi:10.1016/j.artint.2015.03.004.

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

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

24 Dawn Michaelson, Dominik Schreiber, Marijn J. H. Heule, Benjamin Kiesl-Reiter, and Michael W. Whalen. Producing Proofs of Unsatisfiability with Distributed Clause-Sharing SAT Solvers. *Journal of Automated Reasoning*, 69, 2025. doi:10.1007/s10817-025-09725-w.

25 Abhishek Nair, Saranyu Chattopadhyay, Haoze Wu, Alex Ozdemir, and Clark Barrett. Proofstitch: Proof combination for divide and conquer SAT solvers. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, pages 84–88, 2022. doi:10.34727/2022/isbn.978-3-85448-053-2_14.

26 Florian Pollitt, Mathias Fleury, and Armin Biere. Faster LRAT checking than solving with CaDiCaL. In *Proc. Theory and Applications of Satisfiability Testing (SAT)*, pages 21:1–21:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.21.

27 Peter Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithmics (JEA)*, 5:7–es, 2000. doi:10.1145/351827.384249.

28 Peter Sanders and Tim Niklas Uhl. Engineering a distributed-memory triangle counting algorithm. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 702–712, 2023. doi:10.1109/IPDPS54959.2023.00076.

29 Dominik Schreiber. Trusted Scalable SAT Solving with on-the-fly LRAT Checking. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 25:1–25:19, 2024. doi:10.4230/LIPIcs.SAT.2024.25.

30 Dominik Schreiber, Katalin Fazekas, Mathias Fleury, and Armin Biere. Real-time proof checking for distributed incremental SAT solving. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2026. doi:10.1007/978-3-032-22752-2_18.

31 Dominik Schreiber, Niccolò Rigi-Luperti, and Armin Biere. Streamlining distributed SAT solver design. In *Theory and Applications of Satisfiability Testing (SAT)*, 2025. doi:10.4230/LIPIcs.SAT.2025.27.

32 Dominik Schreiber and Peter Sanders. MallobSat: Scalable SAT Solving by Clause Sharing. *Journal of Artificial Intelligence Research (JAIR)*, 80:1437–1495, 2024. doi:10.1613/jair.1.15827.

33 Dominik Schreiber, Peter Sanders, Niccolò Rigi-Luperti, and Ruben Götz. Mallob. Software, swhId: swh:1:dir:63e95b4ba914b33387dbf6e31babbc7b9c3904dc (visited on 2026-07-01). URL: https://github.com/domschrei/mallob, doi:10.4230/artifacts.26961.

34 Laurent Simon. Post mortem analysis of SAT solver proofs. In *Proc. Pragmatics of SAT*, pages 26–40, 2014. doi:10.29007/gpp8.

35 Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is 15. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 389–406, 2023. doi:10.1007/978-3-031-30823-9_20.

36 Yong Kiam Tan, Marijn JH Heule, and Magnus O Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25(2):167–184, 2023. doi:10.1007/s10009-022-00690-y.

37 Yong Kiam Tan, Magnus O Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2, 2019. doi:10.1017/s0956796818000229.

38 Yong Kiam Tan, Dominik Schreiber, Johannes Åman Pohjola, and Magnus O Myreen. Verified real-time proof checking for large-scale SAT solving. Under review, 2026.

39 Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. In *IEEE*, volume 103, pages 2021–2035, 2015. doi:10.1109/JPROC.2015.2455034.

40 Emily Yu, Nils Froleyks, Armin Biere, and Keijo Heljanko. Towards compositional hardware model checking certification. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, pages 44–54, 2023. doi:10.34727/2023/isbn.978-3-85448-060-0_12.

A Proof Translations

In the following, we sketch how some proof formats related to PalRUP can be transformed into each another.

PalRUP fragment to LRUP. Given a PalRUP fragment P, add $|I(P)|$ to all clause IDs greater than o and then replace all clause import IDs $\{id + |I(P)| : (id, c) \in I(P)\}$ with the IDs $\{o+1, \ldots, o+|I(P)|\}$. Next, remove the imports and deletions of each $(id, c) \in I(P)$ from P and append the clauses in $I(P)$ to F (ordered by ID). The result is an LRUP proof $\widetilde{P}$ for $\widetilde{F} = F \cup \{c \mid (id, c) \in I(P)\}$ with $\widetilde{o} := o + |I(P)|$ clauses, where the scope of each previously imported clause now extends to the entire proof.

PalRUP fragment to LIDRUP. Given a PalRUP fragment P, identify all sections of P that do *not* contain any clause import statements. At the beginning of each such section, insert a LIDRUP *query-sat* statement with an empty set of assumptions. At the end of each section, insert a LIDRUP *status* statement with result UNKNOWN. Rewrite each PalRUP clause import statement as an according LIDRUP *input* statement. Next, prepend all original problem clauses as LIDRUP *input* statements to the proof and append concluding LIDRUP statements at the end of the proof: *status* with result UNSAT and *core* with the empty clause's ID if it exists, and *status* with result UNKNOWN otherwise. The result is a LIDRUP proof $\widetilde{P}$ for an incremental formula that is successively built up from F to feature all imported clauses. The clause checks, insertions, and deletions that need to be performed by the checker are identical between the PalRUP fragment and the LIDRUP proof.

LRUP to PalRUP fragment. Each LRUP proof directly qualifies as a PalRUP proof fragment (with no clause imports) as long as it complies with PalRUP properties P0–P3. The proof can always be made compliant for $i = 0$ and $p = 1$ (i.e., assuming the fragment is the only one in the "parallel" proof) by traversing the proof while replacing each assigned clause ID in an addition with a fresh ID from a separate counter (initialized to $o + 1$) and remapping each ID in subsequent occurrences (in deletions and hints) accordingly.

B Supplementary Figures

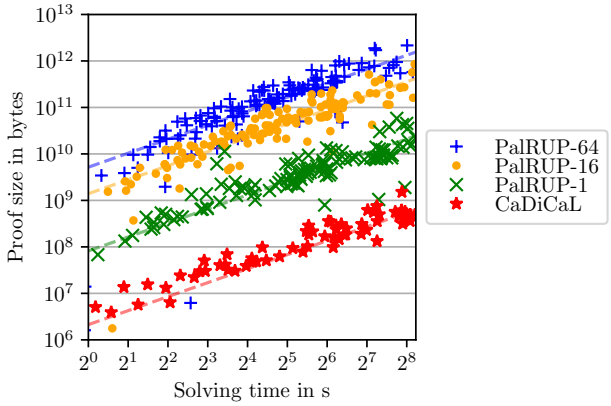


Figure 7 Total proof sizes relative to solving time for PalRUP at 1/16/64 nodes and sequential CaDiCaL with LRAT checking. The according lines are lines through the origin with a gradient of the respective median throughput, which is 2.02 MiB/s for CaDiCaL and 77.99 (1330.17, 4942.87) MiB/s for PalRUP at 1 (16, 64) nodes.

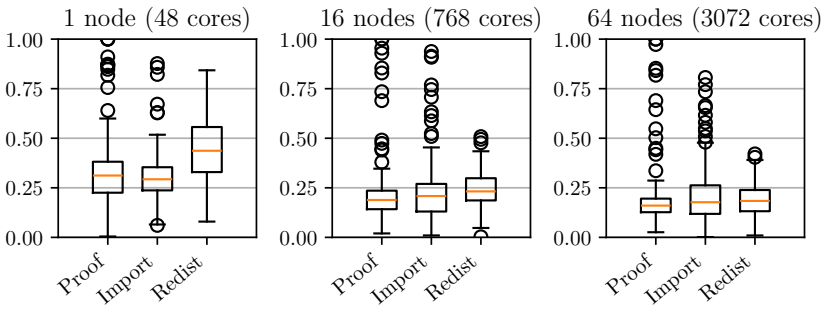


Figure 8 Distributions over the normalized inter quartile range (IQR) for each instance per checker and produced kind of data. *Proof* relates to IQRs of produced PalRUP fragment sizes, *Import* and *Redist* likewise relate to the communication files written after *local checking* and *redistribution*. Since the work done by each of our pals depends on the input file sizes, the shown variance in file sizes can serve as a proxy for the work balance in our approach.