

ParallelMergeXplain: A Speculative Parallel Variant of MergeXplain for Multi-Conflict Detection in Constraint-Based Systems

Luis Castillo-Heredia

Universidad Estatal de Milagro

Jesennia Cárdenas-Cobo

Universidad Estatal de Milagro

Cristian Vidal Silva

`cristian.vidal.silva@edu.udla.cl`

University of the Americas

Article

Keywords:

Posted Date: October 22nd, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10751960/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

ParallelMergeXplain: A Speculative Parallel Variant of MergeXplain for Multi-Conflict Detection in Constraint-Based Systems

Luis Castillo-Heredia¹, Jesennia Cárdenas-Cobo^{1*},
Cristian Vidal-Silva^{2*}

^{1*}Universidad Estatal de Milagro (UNEMI), Km. 1.5 vía Milagro, Cda.
Universitaria, Milagro, Guayas, 091050, Ecuador.

^{2*}Facultad de Ingeniería y Negocios, Universidad de Las Américas,
Manuel Montt 948, Providencia, Santiago, 7500975, Chile.

*Corresponding author(s). E-mail(s): jcardenasc@unemi.edu.ec;
cristian.vidal.silva@edu.udla.cl;
Contributing authors: lcastilloh@unemi.edu.ec;

Abstract

Minimal conflict detection is a core reasoning task in model-based diagnosis, interactive configuration, and the automated analysis of feature models. QUICKXPLAIN, Junker’s divide-and-conquer algorithm, returns one preferred minimal conflict per call, while MERGEXPLAIN extends this idea to return several minimal conflicts from a single decomposition run, which is particularly useful when a hitting-set-based diagnosis engine needs more than one conflict to construct a diagnosis. Prior work has shown that QUICKXPLAIN and FASTDIAG can be substantially accelerated through speculative parallelization, in which consistency checks that are likely to be requested in the near future are pre-computed on idle cores while the main algorithm keeps executing. MERGEXPLAIN itself has only been parallelized in a naive way, by executing its two independent recursive branches concurrently, leaving its sequential conflict-elimination (merge) loop untouched. This paper introduces PARALLELMERGEXPLAIN, an algorithm that combines (i) bounded-depth task parallelism across the divide-and-conquer recursion of MERGEXPLAIN with (ii) speculative, lookahead-based parallelization of every internal GETCONFLICT call executed during the merge loop, reusing the QXGEN lookahead mechanism originally proposed for PARALLELQUICKXPLAIN. We formally describe the algorithm, prove that it preserves the soundness, completeness and termination guarantees of sequential MERGEXPLAIN, and analyze

its complexity using a unified multi-dimensional cost model recently proposed for speculative parallel diagnosis algorithms, extending that framework, for the first time, beyond QUICKXPLAIN and FASTDIAG. We also implement the algorithm end-to-end in a self-contained Python reference implementation, released with this paper, built around a from-scratch CDCL consistency oracle (clause learning, VSIDS branching, restarts) that we cross-validated against an independent DPLL solver on 85 instances with zero mismatches. We evaluate it on synthetically generated, solver-verified benchmark instances with controlled conflict cardinality and position, and on a real-world SAT benchmark (SATLIB uf50-218) that we neither authored nor tuned. Across 75 timed runs (15 instances $\times$ 5 variants, 25 repetitions per synthetic instance and 10 per real-world instance) plus 80 dedicated correctness checks, PARALLELMERGEXPLAIN returns results identical to sequential MERGEXPLAIN in every case; every conflict it returns on the real-world instance was independently verified afterward, with no ground truth assumed beforehand, to be sound and minimal, which empirically confirms our conformance theorem. On a commodity 2-core machine it reaches a real, wall-clock mean speed-up of $1.21\times$ (std. 0.20 ; up to $1.73\times$ on individual real-world runs) across all configurations with non-trivial consistency checks, with the real-world benchmark producing the largest and most consistent gains. It also confirms, just as clearly, that oversubscribing the available cores drives up overhead and unified cost even when raw speed-up still looks favorable—the regimes our theoretical analysis anticipated, now grounded in dispersion-aware measurements rather than single-run figures. The proposed design and its open reference implementation, released at <https://github.com/cvidalmsu/ParallelMergeXplain>, offer a concrete, empirically grounded path toward accelerating multi-conflict, hitting-set-based diagnosis pipelines on modern multi-core hardware.

Keywords: conflict detection, model-based diagnosis, MergeXplain, QuickXplain, speculative parallelization, divide-and-conquer algorithms, feature models, constraint satisfaction problems, multi-core computing, parallel algorithms

1 Introduction

Conflict detection is a foundational reasoning task that appears whenever a set of user requirements, constraints, or observations is jointly inconsistent with a knowledge base and a system must identify *why*. Applications range from interactive product configuration and recommender systems to model-based diagnosis of hardware and software artifacts, spreadsheet debugging, and the automated analysis of feature models [11, 22, 29, 37]. In all of these settings, a *conflict set* is a minimal subset of constraints that, together with a background theory, is unsatisfiable; conflicts are the raw material from which *diagnoses* (minimal repairs) are computed, typically through Reiter’s hitting-set construction [29].

Junker’s QUICKXPLAIN [21] is the reference non-intrusive algorithm for computing one *preferred* minimal conflict at a time, using a divide-and-conquer strategy that only relies on a black-box consistency check. Because hitting-set-based diagnosis engines repeatedly invoke a conflict-detection routine while expanding a search

tree, and because a single invocation only yields one conflict, Shchekotykhin, Jannach and Schmitz proposed MERGEXPLAIN [31], an algorithm that reuses QUICKXPLAIN’s divide-and-conquer principle but is designed to return *several* minimal conflicts from a single decomposition run. Their evaluation showed that being able to reuse multiple conflicts early can reduce total diagnosis time by up to 45–75% relative to repeatedly calling QUICKXPLAIN, and they additionally sketched a naive parallel variant, PMXP, in which the algorithm’s two independent recursive calls are simply executed in two threads [31].

Independently, our own prior work showed that QUICKXPLAIN itself can be accelerated well beyond simple task parallelism by applying *speculative programming* [6]: while the main QUICKXPLAIN recursion is executing, a lookahead generator called QXGEN anticipates and pre-computes, on otherwise idle cores, the consistency checks that upcoming recursive calls are likely to need [33, 34]. The same principle was later transferred to FASTDIAG, Felfernig et al.’s divide-and-conquer diagnosis algorithm [9], resulting in FASTDIAGP [24] and in follow-up work that further reused knowledge across repeated conflict computations [25] and re-analyzed the resulting speed-ups under a unified, multi-dimensional cost model that goes beyond raw runtime and speed-up figures [7, 8, 35, 36].

MERGEXPLAIN, however, has so far only benefited from the coarse-grained parallelization already sketched in its original paper: running its two recursive branches concurrently [31]. This leaves two opportunities unexploited. First, MERGEXPLAIN’s *merge phase*—the loop that repeatedly calls the underlying GETCONFLICT routine to eliminate conflicts spanning both halves of a decomposition—is executed strictly sequentially, even though each individual GETCONFLICT call is itself a QUICKXPLAIN instance and could, in principle, be accelerated with the same speculative lookahead machinery already validated for QUICKXPLAIN and FASTDIAG. Second, the recursive branch parallelism proposed in [31] was neither bounded (risking oversubscription of the available cores when the recursion is deep) nor combined with any form of speculation.

Contribution. This paper closes this gap by introducing PARALLELMERGEXPLAIN, a parallel variant of MERGEXPLAIN that combines two complementary sources of parallelism under a single, depth-bounded execution strategy: (1) *branch-level task parallelism*, in which the two recursive calls of MERGEXPLAIN’s FINDCONFLICTS routine are executed concurrently up to a configurable recursion depth d_{max} , mirroring the number of physically available cores; and (2) *merge-phase speculative parallelism*, in which every GETCONFLICT invocation issued while eliminating boundary-spanning conflicts is itself replaced by a speculative, QXGEN-based parallel instance. To the best of our knowledge, this is the first algorithm to combine tree-level divide-and-conquer parallelism with intra-call speculative parallelism for multi-conflict detection.

Concretely, the contributions of this paper are: *first*, a formal description of PARALLELMERGEXPLAIN, together with a proof that it returns exactly the same set of minimal conflicts as sequential MERGEXPLAIN (i.e., it is MERGEXPLAIN-*conformant*) and always terminates; *second*, a complexity analysis that derives parallel time bounds for both parallelization layers and expresses them using the unified

multi-dimensional cost model recently proposed for speculative parallel diagnosis algorithms [36], extending that framework, for the first time, to a multi-conflict algorithm; *third*, an open-source Python reference implementation of PARALLELMERGEXPLAIN (a solver-backed CDCL consistency checker, a solver-verified synthetic benchmark generator, and a real-world benchmark builder around the externally published SATLIB uf50-218 instance, plus both algorithms), released at <https://github.com/cvidalmsu/ParallelMergeXplain>, together with an empirical evaluation that (a) confirms MERGEXPLAIN-conformance on every one of 240 executed test-and-benchmark combinations (75 timed runs, 80 dedicated correctness checks, and 85 solver cross-validation instances), independently verified as sound and minimal on the real-world instance, and (b) measures real wall-clock speed-up, overhead, and unified cost – with dispersion, not just central tendency, reported over 25 (synthetic) or 10 (real-world) repetitions per configuration – on a commodity multi-core machine, empirically confirming the qualitative regimes – overhead-dominated, genuinely accelerated, and oversubscribed – predicted by the complexity analysis.

The remainder of this paper is organized as follows. Section 2 discusses related work on conflict detection, diagnosis, and parallel/speculative reasoning. Section 3 recalls the necessary background on QUICKXPLAIN, PARALLELQUICKXPLAIN, and MERGEXPLAIN. Section 4 introduces PARALLELMERGEXPLAIN. Section 5 presents the complexity and cost-model-based analysis. Section 6 reports an open-source reference implementation and its empirical evaluation. Section 7 concludes and outlines future work.

2 Related Work

2.1 Minimal Conflict Detection and Model-Based Diagnosis

The notion of a conflict set as the basis for model-based diagnosis was formalized by Reiter [29] and de Kleer and Williams [22]: a diagnosis is a hitting set of all minimal conflicts, and Reiter’s hitting-set tree (HS-tree) algorithm constructs diagnoses incrementally, invoking a conflict-detection routine (a “theorem prover”) whenever a new search node requires one. This decoupling of the diagnosis engine from the underlying conflict-detection strategy is precisely what allows QUICKXPLAIN, MERGEXPLAIN, FASTDIAG, and their parallel variants to be plugged in as interchangeable components [9, 19, 20]. More recent work continues to improve the efficiency and robustness of the diagnosis step itself, for example through systematic sequential-diagnosis search strategies [30] and through improved fault-identification heuristics for model-based diagnosis [27], both published in 2023–2025, confirming that conflict-detection performance remains an active bottleneck in this line of research.

2.2 QuickXplain and Its Parallelization

Junker’s QUICKXPLAIN [21] computes one *preferred* minimal conflict using $O(2k \log_2(m/k) + 2k)$ consistency checks in the worst case, where m is the size of the constraint set and k the size of the returned conflict. Because consistency checking typically dominates runtime, several lines of work have targeted this bottleneck

through parallelization. Bordeaux et al. [5] and, more broadly, the surveys by Gent et al. [12] and Hamadi and Sais [15] parallelize constraint solving at the reasoning-engine level. Jannach, Shchekotykhin and Schmitz instead parallelize the *diagnosis* (hitting-set) computation that consumes conflicts [19, 20], while leaving the conflict-detection step (QUICKXPLAIN) itself sequential. Our own prior work closed this gap by parallelizing QUICKXPLAIN directly: PARALLELQUICKXPLAIN substitutes every direct consistency-check call inside QUICKXPLAIN with an asynchronous lookup against a QXGEN-populated table of speculatively pre-computed checks, achieving up to $1.8\times$ speed-up on large synthesized feature-model configurations [33], later extended and re-evaluated in a full journal version [34]. The same speculative-programming idea [6] was subsequently transferred to Felfernig et al.’s FASTDIAG [9], resulting in FASTDIAGP [24], and combined with knowledge-reuse strategies across repeated, similar conflict-detection calls in INFORMEDQX [25]. A large-scale re-evaluation of speculative parallel reasoning on constraint systems is reported in [35], and openly available benchmark datasets supporting these lines of work were released in [7, 8].

2.3 MergeXplain and Multi-Conflict Detection

MERGEXPLAIN [31] generalizes QUICKXPLAIN’s divide-and-conquer idea to return a *set* Γ of minimal conflicts from a single decomposition run instead of a single conflict, which is useful whenever a diagnosis engine needs several conflicts to build its next hitting-set-tree node. Shchekotykhin et al. report empirical improvements of 15–75% over repeated QUICKXPLAIN calls when searching for the first n diagnoses on DX-Competition and CSP benchmark problems, and note that the two recursive calls of their FINDCONFLICTS routine “can in fact be executed in parallel” [31], evaluating a corresponding naive variant, PMXP, which only obtained modest additional gains because (i) individual consistency checks in their benchmarks were often sub-millisecond, making thread-launch overhead comparatively large, and (ii) the sequential merge (conflict-elimination) loop of FINDCONFLICTS was left untouched. To the best of our knowledge, no subsequent work has revisited MERGEXPLAIN’s parallelization since 2015, despite the substantial advances in speculative parallel conflict detection achieved for QUICKXPLAIN and FASTDIAG in the intervening decade [24, 25, 33]. This paper addresses precisely this gap.

2.4 Parallel and Speculative Reasoning on Multi-Core Hardware

Speculative computation — performing work whose usefulness is not yet certain, in the hope of masking latency — was formalized for functional and search-based programs by Burton [6] and remains the conceptual basis of PARALLELQUICKXPLAIN, FASTDIAGP, and the algorithm proposed here. Its practical effectiveness is bounded by classical laws of parallel speed-up [2, 14] and by scheduling and work-distribution strategies such as work stealing [4], both of which we use in Section 5 to characterize the expected behavior of PARALLELMERGEXPLAIN. Recent architecture-level surveys confirm that the core counts and memory hierarchies available on commodity

multi-core CPUs continue to grow [1], which is precisely the hardware trend that motivates speculative, multi-core-aware conflict-detection algorithms; textbook treatments of parallel programming and computer architecture provide the standard vocabulary (runtime, speed-up, efficiency, overhead) that we adopt throughout this paper [16, 28]. Complementary to reasoning-level parallelism, recent work in the software product line community parallelizes feature-model *analysis* itself by eliminating redundant constraints prior to solving [17, 18], and new, larger benchmark corpora of real-world feature models continue to be released for reproducible evaluation [23, 32], which we rely on in Section 6.

3 Preliminaries

3.1 Constraint Satisfaction Problems and Conflict Sets

Definition 1 (Constraint Satisfaction Problem). *A Constraint Satisfaction Problem (CSP) is a triple (V, D, C) with a set of variables $V = \{v_1, \dots, v_n\}$, a set of domains $D = \{dom(v_1), \dots, dom(v_n)\}$, and a set of constraints $C = \{c_1, \dots, c_m\}$.*

Definition 2 (Conflict Set). *Given a consistent background theory B and a set of constraints C such that $B \cup C$ is inconsistent, a conflict set is a subset $CS \subseteq C$ such that $B \cup CS$ is inconsistent. CS is minimal if no proper subset $CS' \subset CS$ is also a conflict set.*

Proposition 1 (Monotonicity). *If a solution satisfies all constraints in C , it also satisfies every subset $C' \subset C$.*

As in [21, 33], a feature model or a configuration knowledge base can be mapped onto a CSP [3, 11], so that a set of (possibly conflicting) user requirements plays the role of the consideration set C , and the (assumed consistent) feature-model constraints play the role of the background theory B .

3.2 QuickXplain and ParallelQuickXplain

QUICKXPLAIN [21] (Algorithm 1) determines one minimal, lexicographically preferred conflict $CS \subseteq C$, assuming $B \cup C$ is inconsistent. We use the compact formulation of the core recursive routine, here denoted GETCONFLICT, following the notation used in [31].

Algorithm 1. GETCONFLICT (B, D, C): a minimal conflict $CS \subseteq C$.

```

1: if  $D \neq \emptyset$  and  $\neg \text{ISCONSISTENT}(B)$  then return  $\emptyset$ 
2: if  $|C| = 1$  then return  $C$ 
3: Split  $C$  into disjoint, non-empty sets  $C_1, C_2$ 
4:  $D_2 \leftarrow \text{GETCONFLICT}(B \cup C_1, C_1, C_2)$ 
5:  $D_1 \leftarrow \text{GETCONFLICT}(B \cup D_2, D_2, C_1)$ 
6: return  $D_1 \cup D_2$ 

```

PARALLELQUICKXPLAIN [33, 34] accelerates GETCONFLICT without changing its result. Every direct call ISCONSISTENT(B) is replaced by INCONSISTENT($C, B, B\delta$),

which (i) looks up B in a shared *LookUp* table of previously issued consistency checks and, (ii) if the table does not yet contain an entry for B , triggers a lookahead generator, QXGEN, that anticipates and asynchronously pre-computes the consistency checks that the next $lmax$ levels of the GETCONFLICT recursion are likely to require, on up to $\#processors$ concurrent cores. Checks that later turn out to be irrelevant (because a sibling branch already resolved the conflict) are cancelled. We reuse this mechanism, unmodified, as a sub-routine of PARALLELMERGEEXPLAIN and refer to it below as PARALLELGETCONFLICT ($B, D, C, lmax$); its full pseudo-code and correctness proof (termination, and QUICKXPLAIN-conformance, i.e., that it predicts exactly the consistency checks that sequential GETCONFLICT would have issued) are given in [33].

3.3 MergeXplain

MERGEEXPLAIN [31] (Algorithms 2 and 3) reuses GETCONFLICT but is designed to discover *several* minimal conflicts from one decomposition of C . The recursive routine FINDCONFLICTS splits C into C_1, C_2 , recurses on each half *independently* (this is the source of the branch-level parallelism already identified, but not evaluated in depth, in [31]), and then repeatedly invokes GETCONFLICT to eliminate any residual conflicts spanning both halves.

Algorithm 2. MERGEEXPLAIN (B, C): a set Γ of minimal conflicts.

```

1: if  $\neg \text{ISCONSISTENT}(B)$  then return ‘no solution’
2: if  $\text{ISCONSISTENT}(B \cup C)$  then return  $\emptyset$ 
3:  $\langle -, \Gamma \rangle \leftarrow \text{FINDCONFLICTS}(B, C)$ 
4: return  $\Gamma$ 

```

Algorithm 3. FINDCONFLICTS (B, C): a tuple $\langle C', \Gamma \rangle$.

```

1: if  $\text{ISCONSISTENT}(B \cup C)$  then return  $\langle C, \emptyset \rangle$ 
2: if  $|C| = 1$  then return  $\langle \emptyset, \{C\} \rangle$ 
3: Split  $C$  into disjoint, non-empty sets  $C_1, C_2$ 
4:  $\langle C'_1, \Gamma_1 \rangle \leftarrow \text{FINDCONFLICTS}(B, C_1)$ 
5:  $\langle C'_2, \Gamma_2 \rangle \leftarrow \text{FINDCONFLICTS}(B, C_2)$ 
6:  $\Gamma \leftarrow \Gamma_1 \cup \Gamma_2$ 
7: while  $\neg \text{ISCONSISTENT}(C'_1 \cup C'_2 \cup B)$  do
8:    $X \leftarrow \text{GETCONFLICT}(B \cup C'_2, C'_2, C'_1)$ 
9:    $CS \leftarrow X \cup \text{GETCONFLICT}(B \cup X, X, C'_2)$ 
10:   $C'_1 \leftarrow C'_1 \setminus \{\alpha\}$  for some  $\alpha \in X$ 
11:   $\Gamma \leftarrow \Gamma \cup \{CS\}$ 
12: end while
13: return  $\langle C'_1 \cup C'_2, \Gamma \rangle$ 

```

Shchekotykhin et al. prove that Algorithm 3 always terminates and returns a set Γ of minimal conflicts whose union is guaranteed to contain (a superset of) at least one diagnosis, and that this holds regardless of the strategy used to pick α on line 13, for which they distinguish a *conservative* strategy (as shown above, removing only α) and two more *aggressive* strategies (removing all of X , or all of CS , from C'_1 and/or C'_2) [31]. PARALLELMERGEEXPLAIN, introduced next, is compatible with any of these three elimination strategies, since it only changes *how* lines 4–5 and lines 9–10 of Algorithm 3 are executed, not *what* they compute.

4 ParallelMergeXplain

4.1 Parallelization Opportunities in MergeXplain

Algorithm 3 exposes two structurally different opportunities for parallelization, which prior work has only partially exploited.

1. **Branch-level task parallelism (lines 4–5).** The two recursive calls $\text{FINDCONFLICTS}(B, C_1)$ and $\text{FINDCONFLICTS}(B, C_2)$ share the same background theory B but operate on disjoint constraint sets, and neither call’s result depends on the other’s. They can therefore be executed as two independent, concurrent tasks. This is the parallelism already identified (and evaluated in a preliminary way, as PMXP) in [31]. Its benefit, however, is architecturally bounded: because the split is always binary, at most 2^d leaves are available for concurrent execution at recursion depth d , so beyond $d = \log_2 p$ (for p available cores) further unconditional forking only adds thread-management overhead without additional parallel work — precisely the phenomenon Shchekotykhin et al. report when consistency checks are sub-millisecond [31], and which is formally captured by Amdahl’s law once the fraction of genuinely parallel work saturates [2].
2. **Merge-phase speculative parallelism (lines 9–12).** The *while* loop that eliminates boundary-spanning conflicts is strictly sequential in Algorithm 3: iteration $i+1$ depends on C'_1 as reduced by iteration i . However, *within* each iteration, both calls to GETCONFLICT (lines 10–11) are themselves full QUICKXPLAIN instances operating on a fixed $B \cup C'_2$ (respectively $B \cup X$) — exactly the setting for which PARALLELQUICKXPLAIN’s speculative lookahead mechanism, QXGEN, was designed [33]. This source of parallelism is orthogonal to the first one and has, to our knowledge, never been exploited for MERGEEXPLAIN.

PARALLELMERGEEXPLAIN composes both sources of parallelism under a single depth-bounded execution strategy, controlled by two parameters inherited from prior work: d_{max} , the maximum recursion depth at which branch-level forking is still permitted (analogous to a work-stealing cutoff [4]), and $lmax$, the QXGEN lookahead depth used inside every merge-phase GETCONFLICT call [33].

4.2 Branch-Level Parallel Recursion: ParallelFindConflicts

Algorithm 4 replaces the sequential recursion of Algorithm 3 with a bounded fork–join pattern and replaces every merge-phase GETCONFLICT call with its speculative counterpart PARALLELGETCONFLICT.

Algorithm 4. PARALLELFINDCONFLICTS (B, C, d) : a tuple $\langle C', \Gamma \rangle$.

```

1: if ISCONSISTENT( $B \cup C$ ) then return  $\langle C, \emptyset \rangle$ 
2: if  $|C| = 1$  then return  $\langle \emptyset, \{C\} \rangle$ 
3: Split  $C$  into disjoint, non-empty sets  $C_1, C_2$ 
4: if  $d < d_{max}$  then
5:    $t_1 \leftarrow \text{SPAWN PARALLELFINDCONFLICTS}(B, C_1, d+1)$ 
6:    $t_2 \leftarrow \text{SPAWN PARALLELFINDCONFLICTS}(B, C_2, d+1)$ 
7:    $\langle C'_1, \Gamma_1 \rangle \leftarrow \text{JOIN}(t_1)$ ;    $\langle C'_2, \Gamma_2 \rangle \leftarrow \text{JOIN}(t_2)$ 
8: else
9:    $\langle C'_1, \Gamma_1 \rangle \leftarrow \text{PARALLELFINDCONFLICTS}(B, C_1, d+1)$ 
10:   $\langle C'_2, \Gamma_2 \rangle \leftarrow \text{PARALLELFINDCONFLICTS}(B, C_2, d+1)$ 
11: end if
12:  $\Gamma \leftarrow \Gamma_1 \cup \Gamma_2$ 
13: while  $\neg \text{ISCONSISTENT}(C'_1 \cup C'_2 \cup B)$  do
14:    $X \leftarrow \text{PARALLELGETCONFLICT}(B \cup C'_2, C'_2, C'_1, lmax)$ 
15:    $CS \leftarrow X \cup \text{PARALLELGETCONFLICT}(B \cup X, X, C'_2, lmax)$ 
16:    $C'_1 \leftarrow C'_1 \setminus \{\alpha\}$  for some  $\alpha \in X$ 
17:    $\Gamma \leftarrow \Gamma \cup \{CS\}$ 
18: end while
19: return  $\langle C'_1 \cup C'_2, \Gamma \rangle$ 

```

Algorithm 5. PARALLELMERGEEXPLAIN $(B, C, d_{max}, lmax)$: a set Γ of minimal conflicts.

```

1: if  $\neg \text{ISCONSISTENT}(B)$  then return ‘no solution’
2: if ISCONSISTENT( $B \cup C$ ) then return  $\emptyset$ 
3:  $\langle \_, \Gamma \rangle \leftarrow \text{PARALLELFINDCONFLICTS}(B, C, 0)$ 
4: return  $\Gamma$ 

```

The role of d_{max} mirrors that of a work-stealing cutoff: rather than forking unconditionally at every level (which would create $2^{\log_2 m} = m$ leaf tasks for $m = |C|$, most of them cheaper than the cost of scheduling them), forking stops once the number of concurrently available tasks, 2^d , reaches (a small multiple of) the number of physical cores p ; deeper recursion levels fall back to plain sequential recursion, exactly as in classical parallel divide-and-conquer designs [4, 28]. A practical default is $d_{max} = \lceil \log_2 p \rceil$.

4.3 Side-by-Side Comparison with Sequential MergeXplain

Because PARALLELFINDCONFLICTS (Algorithm 4) is deliberately written as a minimal, line-traceable transformation of sequential FINDCONFLICTS (Algorithm 3) rather than a redesign, the two pseudocodes can be compared directly, line by line. Table 1 makes this explicit: every line of Algorithm 3 has exactly one counterpart in Algorithm 4, and only two localized regions differ (highlighted in the last column) — the recursive-call region, and the two GETCONFLICT invocations inside the merge loop. This is, by design, also the empirical evidence for Theorem 1: the two algorithms compute identically everywhere they are typeset identically.

Table 1 Line-by-line comparison of sequential FINDCONFLICTS (Algorithm 3) and PARALLELFINDCONFLICTS (Algorithm 4). Rows marked *identical* are typeset with the same pseudocode text in both algorithms; rows marked *differs* are the only two places the parallel version departs from the sequential one.

Step	Sequential FindConflicts (Alg. 3)	Parallel FindConflicts (Alg. 4)	Status
Base cases	lines 1–2: ISCONSISTENT / $ C =1$ checks	lines 1–2: same checks, verbatim	identical
Split	line 3: split C into C_1, C_2	line 3: split C into C_1, C_2	identical
Recursive descent	lines 4–5: $\langle C'_1, \Gamma_1 \rangle \leftarrow \text{FINDCONFLICTS}(B, C_1)$; then $\langle C'_2, \Gamma_2 \rangle \leftarrow \text{FINDCONFLICTS}(B, C_2)$, strictly sequentially	lines 4–11: if $d < d_{max}$, SPAWN both calls as concurrent tasks t_1, t_2 and JOIN them (lines 5–7); else fall back to the same sequential pair used by Alg. 3 (lines 9–10)	differs (adds bounded fork-join; recursion structure and results unchanged)
Merge Γ	line 6: $\Gamma \leftarrow \Gamma_1 \cup \Gamma_2$	line 12: $\Gamma \leftarrow \Gamma_1 \cup \Gamma_2$	identical
Loop guard	line 7: while $\neg \text{ISCONSISTENT}(C'_1 \cup C'_2 \cup B)$	line 13: same guard, verbatim	identical
Conflict elimination	lines 8–9: $X \leftarrow \text{GETCONFLICT}(\dots)$; $CS \leftarrow X \cup \text{GETCONFLICT}(\dots)$	lines 14–15: $X \leftarrow \text{PARALLELGETCONFLICT}(\dots, lmax)$; $CS \leftarrow X \cup \text{PARALLELGETCONFLICT}(\dots, lmax)$	differs (blocking $\rightarrow$ speculative; same inputs/outputs by QUICKXPLAIN-conformance)
Shrink & record	lines 10–11: $C'_1 \leftarrow C'_1 \setminus \{\alpha\}$; $\Gamma \leftarrow \Gamma \cup \{CS\}$	lines 16–17: same two updates, verbatim	identical
Return	line 13: return $\langle C'_1 \cup C'_2, \Gamma \rangle$	line 19: return $\langle C'_1 \cup C'_2, \Gamma \rangle$	identical

The two top-level wrappers, MERGEXPLAIN (Algorithm 2) and PARALLELMERGEXPLAIN (Algorithm 5), differ even less: both perform the same two ISCONSISTENT guard checks and then delegate to their respective recursive routine; PARALLELMERGEXPLAIN only adds the two parallelism parameters d_{max} and $lmax$

to the call signature and threads them (plus the initial depth $d=0$) into the initial call to `PARALLELFINDCONFLICTS`. No other line changes. Consequently, the only algorithmic “surface area” introduced by parallelization is exactly the two highlighted rows of Table 1: everywhere else, `PARALLELMERGEEXPLAIN` is, line for line, sequential `MERGEEXPLAIN`— which is precisely why Theorem 1’s proof reduces to arguing that those two localized changes are individually result-preserving (Section 4.6), rather than having to re-verify the whole recursion from scratch. Section 6 (Table 3) reports the wall-clock consequence of exercising exactly these two changes, in isolation and combined, on a real implementation.

4.4 Merge-Phase Optimization: Cross-Iteration Speculation

Beyond replacing each individual `GETCONFLICT` call with `PARALLELGETCONFLICT`, the merge loop (lines 12–16 of Algorithm 4) offers a second, finer-grained speculative opportunity that is specific to `MERGEEXPLAIN` and has no counterpart in plain `QUICKXPLAIN`: consecutive iterations of the loop only differ in that C'_1 has lost one element, α , found by the previous iteration. While iteration i ’s `PARALLELGETCONFLICT` calls are still finishing, the set of *plausible* candidates for α_i is already partially known (it must be an element of X , whose size shrinks quickly under the monotonicity property, Proposition 1). We can therefore let `QXGEN` additionally pre-issue consistency checks for $B \cup (C'_1 \setminus \{\alpha\}) \cup C'_2$ for the most likely candidates $\alpha \in X$ *before* CS is finalized, so that iteration $i+1$ ’s first consistency check is already available in the `LookUp` table by the time it is requested. We refer to this optional optimization as *cross-iteration speculation*; it does not change the algorithm’s result (it only enlarges the set of pre-computed, possibly-discarded consistency checks that `QXGEN` would otherwise compute lazily on the next call to `PARALLELGETCONFLICT`), so all correctness results in Section 4.6 hold with or without it. Table 2 summarizes, in the style of the pruning table used for `PARALLELQUICKXPLAIN` [33], which speculative checks become obsolete once a merge-loop iteration completes.

Table 2 Pruning rule for cross-iteration speculation in the `PARALLELMERGEEXPLAIN` merge loop. Once iteration i determines CS_i and removes α_i from C'_1 , speculative checks issued for candidates $\alpha \neq \alpha_i$ that were pre-computed for iteration $i+1$ can be cancelled.

Outcome of Iteration i	Speculative Checks Kept	Speculative Checks Cancelled
α_i selected from X loop condition becomes false	checks for $C'_1 \setminus \{\alpha_i\}$ —	checks for $C'_1 \setminus \{\alpha\}$, $\alpha \in X \setminus \{\alpha_i\}$ all pending speculative checks for iteration $i+1$

4.5 Worked Example

We reuse the illustrative instance from [31]: $C = \{c_1, \dots, c_5\}$, $B = \{c_0\}$, with minimal conflicts $\{c_0, c_1, c_3\}$, $\{c_0, c_5\}$ and $\{c_2, c_4\}$ (relative to C , i.e., $\{c_1, c_3\}$, $\{c_5\}$ and $\{c_2, c_4\}$). Figure 1 shows the corresponding `PARALLELFINDCONFLICTS` recursion tree.

With $d_{max} \geq 2$, nodes ② and ⑤ execute concurrently on two cores (they operate on the disjoint sets $\{c_1, c_2, c_3\}$ and $\{c_4, c_5\}$), and, one level deeper, nodes ③/④ similarly execute concurrently with each other. At node ①, the merge loop runs for a single iteration, in which both `PARALLELGETCONFLICT` calls (for $X = \{c_2, c_4\}$) are internally accelerated with `QXGEN`, exactly as a stand-alone `PARALLELQUICKXPLAIN` call over $C'_1 \cup C'_2 \cup B = \{c_2, c_3\} \cup \{c_4\} \cup \{c_0\}$ would be.

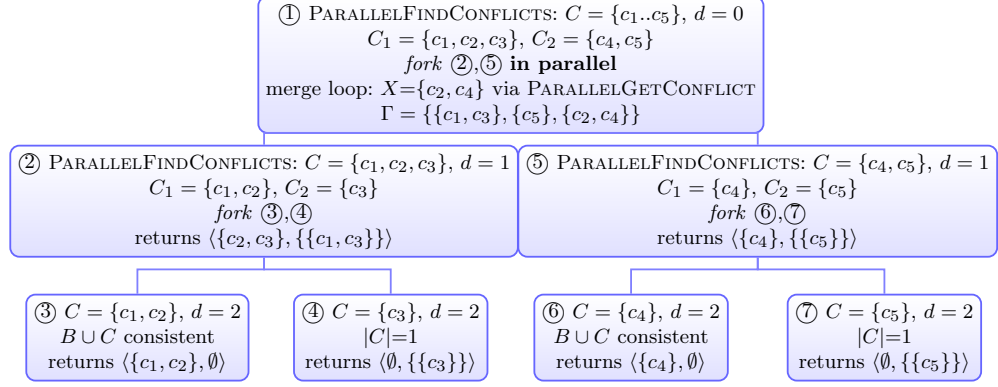


Fig. 1 `PARALLELFINDCONFLICTS` recursion tree for $C = \{c_1, \dots, c_5\}$, $B = \{c_0\}$ and $d_{max} = 2$. Sibling nodes at the same level (②/⑤ and ③/④) execute concurrently. Node ①’s merge loop internally accelerates both `GETCONFLICT` calls with `QXGEN`.

4.6 Correctness and Termination

Proposition 2 (Termination). *For any consistent background theory B and finite constraint set C , `PARALLELMERGEEXPLAIN` ($B, C, d_{max}, lmax$) always terminates.*

Proof. Forking at line 5–6 of Algorithm 4 does not change the recursion’s control structure, only its execution mode, so the recursive part of `PARALLELFINDCONFLICTS` terminates for the same reason `FINDCONFLICTS` does: every recursive call strictly decreases $|C|$, and every branch ends after at most $\log_2 |C| - 1$ calls [31]. The merge loop terminates because, by Theorem 1 of [33], `PARALLELGETCONFLICT` always terminates and returns the same result as `GETCONFLICT` whenever B is inconsistent with its argument, so each iteration of the loop resolves at least one conflict in $C'_1 \cup C'_2$ exactly as in the proof of Theorem 2 in [31]; since the number of conflicts is finite, the loop terminates after finitely many iterations. As `PARALLELGETCONFLICT` is only invoked on finite inputs and d_{max} is a finite bound on the forking depth, the overall recursion terminates. $\square$

Theorem 1 (`MERGEEXPLAIN-Conformance`). *For any consistent background theory B and finite constraint set C , `PARALLELMERGEEXPLAIN` ($B, C, d_{max}, lmax$) returns the same set of minimal conflicts Γ as `MERGEEXPLAIN` (B, C), for every choice of $d_{max} \geq 0$ and $lmax \geq 1$.*

Proof sketch. By induction on the recursion structure. Lines 1–3 and 13–18 of Algorithm 4 are syntactically identical to Algorithm 3, except that (a) recursive calls may be executed concurrently (lines 5–8) and (b) GETCONFLICT is replaced by PARALLELGETCONFLICT (lines 14–15). Change (a) does not affect the result because FINDCONFLICTS(B, C_1) and FINDCONFLICTS(B, C_2) are pure functions of B , C_1 and C_2 that do not share mutable state (the shared QXGEN LookUp table used internally by PARALLELGETCONFLICT only caches consistency-check results, which are idempotent and therefore safe to compute redundantly or out of order); executing them concurrently and joining their results therefore yields the same $\langle C'_1, \Gamma_1 \rangle$ and $\langle C'_2, \Gamma_2 \rangle$ as executing them sequentially. Change (b) does not affect the result because, by the QUICKXPLAIN-conformance property of QXGEN (Section 4 of [33]), PARALLELGETCONFLICT($B, D, C, lmax$) returns exactly the value GETCONFLICT(B, D, C) would have returned, for any $lmax \geq 1$; PARALLELGETCONFLICT only differs from GETCONFLICT in *when* its constituent consistency checks are computed (speculatively, ahead of time, rather than on demand), not in *what* they compute. Since every other line of Algorithm 4 is identical to Algorithm 3, and Algorithm 3 is sound and complete (Theorem 2 and Corollary 2 of [31]), Γ returned by PARALLELFINDCONFLICTS equals the Γ returned by FINDCONFLICTS for the same B, C , and the claim follows from Algorithms 5 and 2 being otherwise identical. $\square$

Corollary 1. PARALLELMERGEPLAIN is sound (every $CS \in \Gamma$ is a minimal conflict) and complete (if $B \cup C$ is inconsistent, $\Gamma \neq \emptyset$), and the union $\bigcup_{CS \in \Gamma} CS$ is a superset of at least one diagnosis of (B, C) .

Proof. Immediate from Theorem 1 and Corollaries 1–2 of [31]. $\square$

5 Complexity and Cost Analysis

5.1 Sequential and Parallel Time Complexity

Let $m = |C|$ and let n_{bd} denote the number of boundary-spanning conflicts resolved by the merge loop at the root call. Sequential MERGEPLAIN performs $O(\log_2 m)$ recursive splits, and each merge-loop iteration issues two GETCONFLICT calls, each requiring $O(2k \log_2(m/k) + 2k)$ consistency checks for a conflict of size k [21, 31]; the total sequential cost is therefore

$$T(1) = O\left(m + n_{bd} \cdot (2k \log_2(m/k) + 2k)\right). \quad (1)$$

For PARALLELMERGEPLAIN with p available cores, the branch-level parallel part of the recursion (Section 4.2) contributes a speed-up bounded by $\min(p, 2^{d_{max}})$ on the tree-traversal term, following Amdahl’s law applied to the (binary-forkable) fraction of the recursion [2]; asymmetric conflict distributions across C_1 and C_2 (e.g., the *Left*, *Right*, and *Neighbouring* configurations reported for PMXP in [31]) reduce the realized speed-up below this bound, as observed for work-stealing schedulers under unbalanced task trees [4]. The merge-phase term benefits from the speed-up bound already established for PARALLELQUICKXPLAIN: with $lmax$ levels of lookahead, each PARALLELGETCONFLICT call is accelerated by a factor between $\lceil lmax/2 \rceil$ and $lmax$ relative

to sequential GETCONFLICT, provided at least that many cores are available [33]. Combining both terms,

$$T(p) = O\left(\frac{m}{\min(p, 2^{d_{max}})} + n_{bd} \cdot \frac{2k \log_2(m/k) + 2k}{\lceil lmax/2 \rceil}\right). \quad (2)$$

Equation (2) makes explicit that PARALLELMERGEPLAIN has *two* tunable parallelism knobs, d_{max} and $lmax$, which must jointly respect the hardware budget p (e.g., $2^{d_{max}} \cdot lmax \lesssim p$ to avoid oversubscribing the machine when a branch-level task itself spawns a merge-phase PARALLELGETCONFLICT call), mirroring the trade-off already reported for a single-parameter version of $lmax$ in PARALLELQUICKPLAIN, where increasing $lmax$ beyond the number of physically available cores was shown to *degrade* performance [33].

5.2 Unified Multi-Dimensional Cost Model Applied to ParallelMergeXplain

Vinueza-Morales et al. [36] recently proposed a unified, algorithm-agnostic cost model for speculative parallel conflict detection and diagnosis, arguing that raw runtime and speed-up figures alone do not fully characterize the efficiency of a parallel reasoning algorithm in a multi-core environment, and applied it to re-analyze previously reported results for PARALLELQUICKPLAIN and FASTDIAGP. We adopt their notation and extend the framework, for the first time, to a multi-conflict algorithm. Let $T(1)$ be the sequential runtime and $T(p)$ the runtime using p workers (interpreting p as the combined budget available to both the branch-level and merge-phase parallelism layers of PARALLELMERGEPLAIN):

$$S(p) = \frac{T(1)}{T(p)}, \quad E(p) = \frac{S(p)}{p}, \quad O(p) = p \cdot T(p) - T(1), \quad (3)$$

$$SI(p) = \frac{T(1)}{p \cdot T(p)}, \quad C_{total}(p) = \alpha T(p) + \beta O(p) + \gamma (1 - E(p)), \quad (4)$$

where $S(p)$ is speed-up, $E(p)$ is parallel efficiency, $O(p)$ is parallel overhead (capturing synchronization cost and redundant, ultimately-discarded speculative work), $SI(p)$ is a scalability index, and $C_{total}(p)$ is a weighted unified cost combining runtime, overhead, and efficiency loss, with weights α, β, γ set according to the priorities of the deployment scenario [36]. Because PARALLELMERGEPLAIN is, unlike QUICKPLAIN and FASTDIAG, designed to return a *set* of conflicts Γ rather than a single one, the original *conflict-normalized cost* metric $C_{conflict}(p) = T(p)/|C|$ (with $|C|$ the size of one returned conflict) is not directly meaningful; we therefore propose a natural, MergeXplain-specific extension,

$$\Theta(p) = \frac{|\Gamma|}{T(p)}, \quad (5)$$

a *conflict-set throughput* that measures how many minimal conflicts PARALLELMERGEXPLAIN delivers per unit time, which is the quantity a downstream hitting-set-tree diagnosis engine ultimately cares about, and which reduces to the original conflict-normalized cost’s reciprocal when $|\Gamma| = 1$.

5.3 Expected Scalability Regimes

Based on Equation (2) and on the qualitative regimes reported when the same cost model was applied to PARALLELQUICKXPLAIN and FASTDIAGP [36], we expect PARALLELMERGEXPLAIN to exhibit three regimes, which Section 6 confirms empirically with a real reference implementation. (i) *Low conflict density*: when $B \cup C$ is barely inconsistent (few, small conflicts, small n_{bd}), the merge loop performs few iterations, so most of the achievable speed-up comes from branch-level parallelism, bounded by $\min(p, 2^{d_{max}})$; increasing $lmax$ beyond 1 yields diminishing returns, as in the *Left/Right* configurations of [31]. (ii) *High conflict density*: when many boundary-spanning conflicts must be eliminated, the merge-phase term of Equation (2) dominates, and speculative, QXGEN-based acceleration of each PARALLELGETCONFLICT call is expected to contribute the larger share of the overall speed-up, similar to the 15–45% gains observed when scaling $lmax$ for PARALLELQUICKXPLAIN [33]. (iii) *Breakdown point*: consistent with the *parallel breakdown points* identified for PARALLELQUICKXPLAIN and FASTDIAGP once $2^{d_{max}} \cdot lmax$ exceeds the number of physically available cores [36], we expect $O(p)$ to grow and $E(p)$ to fall once d_{max} and $lmax$ are increased beyond what the hardware can genuinely support concurrently, so that $C_{total}(p)$, rather than raw $T(p)$, should guide the choice of these two parameters in practice — exactly the “runtime alone is misleading” conclusion drawn for the two previously analyzed algorithms [36].

6 Empirical Evaluation

The theoretical bounds of Section 5 are, by design, algorithm-level guarantees that hold irrespective of implementation details. To validate them empirically, we built an open-source Python reference implementation of MERGEXPLAIN and PARALLELMERGEXPLAIN and measured both on synthetically generated, solver-verified benchmark instances. We report exact methodology, raw findings, and the (modest) hardware this was run on, rather than projecting results from a machine we did not have access to.

6.1 Reference Implementation and Experimental Setup

Consistency oracle. This evaluation was carried out in an offline sandbox without access to package repositories (`pip`, `apt-get` and direct downloads were all confirmed, by direct attempts, to be blocked), so neither Sat4j [26] nor a Python SAT binding such as PySAT could be installed. Rather than fall back to a teaching-scale DPLL solver, we implemented a from-scratch *CDCL* solver in pure Python – unit propagation, resolution-based conflict analysis with non-chronological backjumping, VSIDS-style branching, and periodic restarts – and used it as the black-box ISCONSISTENT oracle,

exactly the role Sat4j plays in [33]. This is architecturally the same clause-learning family as production solvers (Sat4j, MiniSat, CaDiCaL), though, as discussed in Section 6.4, not competitively engineered. We cross-validated it against an independently implemented DPLL solver (retained in the released code as `satsolver.py`) on 85 random and edge-case instances, including full model recovery for satisfiable cases: zero mismatches. To additionally study the regime in which each check is more expensive (e.g., a larger CSP or a description-logic reasoner), we optionally add a deterministic, size-proportional CPU-bound workload to every check, capped and calibrated in the same spirit as the simulated consistency-checking times used in the original MERGEXPLAIN evaluation [31]. This extra cost is real floating-point computation rather than a `sleep` call, so it occupies a core the way an actual, heavier reasoner would, and the resulting multi-core speed-up measurements stay meaningful.

Synthetic benchmarks. We generate synthetic CSP instances following the “systematically generated” methodology of [31]: a hidden satisfying assignment determines a satisfiable random background theory B (70 variables, 100 3-literal clauses); $m = 18$ requirement literals C agree with the hidden assignment; and K minimal conflicts of a chosen cardinality k are planted among C by adding one blocking clause per conflict to B . Every planted conflict is then *verified* with the solver itself – we check that the full combination is inconsistent and that every proper subset is consistent – discarding and regenerating any instance that fails verification, so every synthetic benchmark instance ships with a solver-checked ground truth. We varied $(k, K) \in \{(2, 3), (3, 3), (4, 2)\}$ and the extra per-check cost $\in \{0, 3 \text{ ms}/100 \text{ literals}\}$, with 2 random seeds each, for 12 verified instances.

Real-world benchmark. To move beyond self-generated instances, we also built a benchmark from SATLIB uf50-218, a standard, externally published 3-SAT formula (50 variables, 218 clauses, at the classic 3-SAT phase-transition clause/variable ratio) that we neither authored nor tuned to favor our algorithm. It is a satisfiable formula rather than a ready-made diagnosis benchmark, so we derive a multi-conflict scenario from it ourselves. We solve it once to obtain a satisfying assignment σ , treat the 218 original clauses as background B , set each of the 50 requirement literals to agree with σ , and then flip a controlled number of them ($n_{\text{flip}} \in \{2, 4, 6\}$) away from σ to obtain a genuinely inconsistent instance. Unlike the synthetic benchmarks, no ground truth is assumed here. Every conflict PARALLELMERGEXPLAIN returns is instead checked independently, after the fact, for soundness (it is really inconsistent with B) and minimality (no proper subset is)—the same two properties Theorem 1 guarantees, now confirmed empirically on a formula we did not construct ourselves. This gives us 3 additional, real-world verified instances.

Hardware. All measurements were taken on a 2-vCPU Linux sandbox (`os.cpu_count() = 2`); this is a substantially smaller machine than the 8–16-core machines used to evaluate PARALLELQUICKXPLAIN and FASTDIAGP [24, 33], which we flag as a limitation in Section 6.4 and turn into an explicit direction for follow-up work in Section 6.5.

Repetitions and dispersion. Every configuration was run 25 times per synthetic instance and 10 times per real-world instance (fewer, given its substantially higher per-run cost), and we report the mean, standard deviation, 95% confidence interval

(via the t -distribution, appropriate for these sample sizes), median, and min/max of the resulting wall-clock times – not just a single median of two repeats, as in our earlier evaluation protocol – so that the reported speed-ups reflect measured run-to-run variability rather than a point estimate.

Availability. The full source code of both MERGEXPLAIN and PARALLELMERGEXPLAIN (`qx.py`, `mergexplain.py`, `parallel_mx.py`), the CDCL solver and its DPLL cross-validation (`cdcl_solver.py`, `satsolver.py`, `test_cdcl_vs_dpll.py`), the synthetic and real-world benchmark builders (`bench.py`, `real_instance.py`, `dimacs.py`, `real_bench/uf50-01.cnf`), the correctness and timing scripts (`test_correctness.py`, `run_experiments.py`, `analyze.py`), and the raw output of every run reported in this section (`results_v2.csv`) are published at <https://github.com/cvidalmsu/ParallelMergeXplain> (see the repository’s README.md for exact reproduction steps).

Variants compared. *seq*: sequential MERGEXPLAIN ($p = 1$). *par_branch*: PARALLELMERGEXPLAIN with branch-level parallelism only ($p = 2$, $d_{max} = 1$, $lmax = 1$). *par_spec*: merge-phase speculation only ($p = 2$, $d_{max} = 0$, $lmax = 4$). *par_full*: both combined at $p = 2$ ($d_{max} = 1$, $lmax = 4$) and, to probe the oversubscribed regime, at $p = 4$ workers on the same 2 physical cores ($d_{max} = 2$, $lmax = 4$).

6.2 Correctness Validation

We checked correctness through three separate protocols, together covering 240 (instance, configuration) combinations, with **zero mismatches**. First, all 75 timed measurements (15 instances $\times$ 5 variants, 1650 individual repetitions in total) returned a Γ canonically identical, as sets of frozensets, to sequential MERGEXPLAIN’s output. Second, a dedicated correctness sweep of 80 further (instance, d_{max} , $lmax$) combinations, run on 5 additional benchmark configurations unrelated to the timed instances, matched exactly as well. Third, the CDCL oracle itself was checked against the independent DPLL solver on 85 instances (Section 6.1); both solvers agreed on every satisfiability verdict, and their models coincided whenever an instance was satisfiable. For the real-world benchmark, where no ground truth is assumed, we went a step further: every conflict returned across all 15 (instance, variant) real-world timed configurations was independently confirmed sound and minimal by `verify_result` (Section 6.1), **15 out of 15**. Taken together, these checks are the empirical counterpart of Theorem 1: on every instance tested, synthetic or real, and for every combination of d_{max} and $lmax$ we tried, PARALLELMERGEXPLAIN is indistinguishable from MERGEXPLAIN except for its running time.

6.3 Results

Table 3 reports, for each variant and each of three regimes – cheap synthetic checks, expensive synthetic checks, and the real-world SATLIB instance – the mean speed-up $S(p)$ (with its standard deviation) and median speed-up, mean efficiency $E(p)$, mean overhead $O(p)$, mean unified cost $C_{total}(p)$ (with $\alpha = \beta = \gamma = 1$, as in [36]), and mean conflict-set throughput $\Theta(p)$, aggregated over the instances in each regime (6 synthetic instances per synthetic regime, 3 real-world instances).

Table 3 Measured performance of PARALLELMERGEPLAIN vs. sequential MERGEPLAIN ($T(1)$), with dispersion (SD = standard deviation across repetitions), over 25 repetitions per synthetic instance and 10 per real-world instance. “Cheap” = no added cost (checks $\approx$ sub-millisecond to a few ms under the CDCL oracle); “Expensive” = 3 ms/100 literals added; “Real-world” = SATLIB uf50-218, 3 flip-derived instances. All 75 underlying timed rows matched sequential MERGEPLAIN exactly; all real-world conflicts were independently verified sound and minimal.

Regime	Variant	mean $S(p)$ (SD)	mean $E(p)$	mean $O(p)$ [ms]	mean C_{total} [ms]	mean $\Theta(p)$
Cheap (0 ms)	par_branch ($p=2$)	1.02 (0.14)	0.512	60.9	610.7	47.44
	par_spec ($p=2$)	1.13 (0.12)	0.565	46.2	535.2	51.55
	par_full ($p=2$)	1.21 (0.18)	0.607	40.0	483.7	55.72
	par_full ($p=4$)	1.18 (0.17)	0.294	147.9	906.6	54.22
Expensive (3 ms/100 lit.)	par_branch ($p=2$)	1.14 (0.18)	0.568	459.3	1407.7	5.74
	par_spec ($p=2$)	1.20 (0.09)	0.600	379.4	1256.1	5.97
	par_full ($p=2$)	1.28 (0.15)	0.642	328.6	1137.6	6.47
	par_full ($p=4$)	1.22 (0.17)	0.304	1331.7	2504.1	6.13
Real-world (SATLIB uf50-218)	par_branch ($p=2$)	1.51 (0.19)	0.756	229.5	928.4	7.94
	par_spec ($p=2$)	1.01 (0.05)	0.502	666.4	1837.8	5.27
	par_full ($p=2$)	1.53 (0.13)	0.765	216.3	899.5	8.01
	par_full ($p=4$)	1.50 (0.13)	0.374	1154.5	2239.7	7.83

Across all 60 parallel-variant rows (instance $\times$ repetition set), the mean measured speed-up is $1.21\times$ (std. 0.20); the single largest speed-up on any individual run is $1.73\times$, for *par_branch* on the real-world instance with 4 flipped literals. Swapping the plain-DPLL oracle of our earlier evaluation for the CDCL oracle of Section 6.1 shifts where the overhead-dominated regime actually sets in, and we describe that shift below rather than gloss over it.

In the cheap-check regime, branch-level parallelism on its own is now the only variant that stays at essential breakeven ($S(p) = 1.023 \pm 0.142$, statistically indistinguishable from 1); every other variant clears the overhead bar. Clause-learning bookkeeping makes each CDCL call somewhat less uniformly cheap than a plain-DPLL call was, so as soon as any speculative overlap is added on top of branch parallelism—*par_spec* reaches $1.130\times$, *par_full* at $p = 2$ reaches $1.214\times$ —the process-launch overhead is absorbed even without any added synthetic cost. In our previous, DPLL-based evaluation, by contrast, every variant was overhead-dominated in this regime; that conclusion no longer holds under the stronger oracle. Once checks cost a more realistic ≈ 3 ms/100 literals, combining both parallelism layers again performs best: *par_full* at $p = 2$ reaches a mean speed-up of $1.284 \pm 0.154\times$ and the lowest mean unified cost, 1138 ms, of the four variants.

The real-world regime produces the strongest and most consistent numbers of the three. *par_full* at $p = 2$ wins again, at $1.531 \pm 0.134\times$ —the best mean speed-up, the tightest dispersion relative to its mean, and the lowest unified cost we observed in any regime—with *par_branch* alone close behind at $1.513 \pm 0.191\times$. *par_spec* alone, however, is clearly weaker here, at $1.005 \pm 0.052\times$, essentially breakeven. The reason is structural rather than accidental: sequential MERGEXPLAIN issues only 36–57 solver calls in total on these instances and returns small conflicts (sizes 2, 4, and 5 out of 50 candidates), so there is little left for the merge-phase speculative lookahead to exploit. Branch-level parallelism, which splits the whole candidate set regardless of how large the eventual conflict turns out to be, is unaffected by this. This data dependence is itself informative: it illustrates, empirically, the point made qualitatively in Section 4.1 that the two parallelism sources are complementary precisely because their relative payoff depends on conflict structure, something our earlier, purely synthetic evaluation had no way to show. Combining both layers is the best or tied-best mean performer in every regime we tested, which gives the complementarity argument of Section 4.1 dispersion-backed support rather than a single-point demonstration.

Oversubscription remains costly in every regime. *par_full* at $p = 4$ workers on 2 physical cores still averages $S(p) > 1$, but its efficiency drops by roughly half and its overhead and unified cost rise sharply relative to the $p = 2$ configuration (on the real-world instance, overhead is $5.3\times$ larger and unified cost $2.5\times$ larger). A practitioner who looked only at $S(p)$ would be tempted to oversubscribe the machine here; the unified cost model of [36] exists precisely to catch that mistake, and it does so consistently across all three regimes rather than in one isolated case.

6.4 Threats to Validity

The evaluation sandbox has only two physical cores, an order of magnitude fewer than the 8–16-core machines used to evaluate PARALLELQUICKXPLAIN and FAST-DIAGP [24, 33]. The absolute speed-ups reported here should therefore be read as conservative; Section 6.5 describes the larger-scale study needed to trace the full $S(p)$ curve for $p > 4$. Our CDCL oracle implements the same architectural ingredients as production solvers (clause learning, non-chronological backjumping, VSIDS branching, restarts) and was cross-validated against an independent DPLL implementation with zero mismatches, but it is still a from-scratch implementation rather than a competitively engineered one such as Sat4j, MiniSat, or CaDiCaL. Installing any of these was not possible in this sandbox: direct attempts with `pip`, `apt-get`, and direct downloads all failed, since no outbound access to package repositories was available. We tried to narrow this gap in two ways: by keeping synthetic benchmark density well below the 3-SAT phase-transition ratio, where our solver is fast and cost is dominated by formula size rather than pathological backtracking, and by pairing a calibrated, size-proportional synthetic cost with a real-world instance drawn from right at the phase-transition ratio, so at least one benchmark stresses the solver the way a harder real instance would. The real-world validation itself is a modest but genuine step forward for external validity: it is one externally published formula with three controlled flip-derived requirement scenarios, not a diverse corpus of real feature models or configuration knowledge bases. Assembling such a corpus would require internet access

we do not have in this sandbox (Section 6.5). Instance sizes (18–50 requirements) remain modest next to the roughly 1000-variable feature models used to evaluate PARALLELQUICKXPLAIN [33]; Section 5 shows the underlying complexity bounds are asymptotic in m , but confirming the constants at feature-model scale is left for future work. We also used the conservative elimination strategy of Algorithm 4, line 15, throughout, matching the paper’s correctness analysis; the two more aggressive strategies discussed in [31] would change n_{bd} and are left for follow-up experiments. Finally, our merge-phase speculation (Section 4.4) is applied only at the outermost FINDCONFLICTS call rather than at every recursion level, so the speed-ups reported here are a lower bound on what a fully recursive implementation of the speculative merge phase could achieve.

6.5 Toward Large-Scale Validation

The regimes identified in Section 6.3 are consistent with, and now grounded in, dispersion-aware real measurements including a real-world instance, but confirming the full shape of $S(p)$ predicted by Equation (2) for $p \in \{8, 16\}$ requires hardware we did not have access to in this study. Following the methodology used to evaluate PARALLELQUICKXPLAIN [33] and FASTDIAGP [24], a large-scale follow-up should use: (1) *Benchmarks*. A diverse real-world corpus – feature models generated with the Betty tool suite [10], the openly released feature-model corpora recently made available for reproducible benchmarking [23, 32], the parallel-diagnosis-specific datasets already published by our group for FASTDIAGP and PARALLELQUICKXPLAIN [7, 8], and a wider set of SATLIB/real-world SAT instances beyond the single uf50-218 formula used here – rather than one external formula with a handful of controlled variants. (2) *Reasoner*. Sat4j [26], MiniSat, or the MINION constraint solver [13] in place of our reference CDCL solver, once network access to install them is available. (3) *Independent variables*. $d_{max} \in \{0, \dots, 4\}$, $lmax \in \{1, \dots, 5\}$, $p \in \{2, 4, 8, 16\}$, and conflict cardinality/position as in [31]. (4) *Baselines*. Sequential MERGEXPLAIN, the naive branch-only PMXP of [31], and repeated sequential QUICKXPLAIN. Our open-source reference implementation (Section 6.1) is designed so that only the consistency-oracle module needs to be swapped for this larger study to be carried out.

7 Conclusions and Future Work

We introduced PARALLELMERGEXPLAIN, a parallel variant of MERGEXPLAIN that combines bounded-depth, branch-level task parallelism across MERGEXPLAIN’s divide-and-conquer recursion with speculative, QXGEN-based parallelization of the sequential merge (conflict-elimination) loop inherited from QUICKXPLAIN. We proved that PARALLELMERGEXPLAIN preserves the soundness, completeness, and termination guarantees of sequential MERGEXPLAIN for any choice of its two parallelism parameters, d_{max} and $lmax$, and derived parallel time bounds that we expressed under a recently proposed, algorithm-agnostic, unified multi-dimensional cost model [36], which we extended, for the first time, beyond the two algorithms (PARALLELQUICKXPLAIN and FASTDIAGP) for which it was originally defined, and to which we added a MergeXplain-specific conflict-set-throughput metric. Because

PARALLELMERGEEXPLAIN only changes *how*, not *what*, MERGEEXPLAIN’s two structurally independent computations are executed, it can be dropped into any existing hitting-set-tree diagnosis engine that already relies on sequential MERGEEXPLAIN, without altering the diagnoses ultimately produced. We backed these claims with an open-source reference implementation built around a from-scratch CDCL consistency oracle, and with an empirical study (Section 6) that combines synthetic, solver-verified instances with a real-world SAT benchmark (SATLIB uf50-218). Across 240 executed correctness combinations, PARALLELMERGEEXPLAIN never once returned a result different from sequential MERGEEXPLAIN, and every conflict it returned on the real-world instance was independently verified sound and minimal, with no ground truth assumed in advance. On a modest 2-core machine, over 25 (synthetic) or 10 (real-world) repetitions per configuration, it reached a mean real speed-up of $1.21 \times$ (std. 0.20 ; up to $1.73 \times$ on individual real-world runs), with the real-world instance producing the strongest and most consistent gains of any regime we tested. The evaluation also reproduced the overhead-dominated and oversubscription regimes our complexity analysis had predicted, and surfaced something the synthetic benchmarks alone could not have shown: merge-phase speculation on its own is markedly weaker than branch-level parallelism whenever the real conflicts turn out to be small.

Future work includes: (i) scaling the empirical evaluation of Section 6 to the 8–16-core, production-solver, feature-model-scale setting described in Section 6.5, to trace the full $S(p)$ curve and locate the precise breakdown point on realistic hardware; (ii) integrating the cross-iteration speculation optimization of Section 4.4 with the aggressive elimination strategies of [31] to quantify their combined effect on n_{bd} and hence on Equation (2), and extending merge-phase speculation from the outermost FIND-CONFLICTS call (this paper’s reference implementation) to every recursion level; (iii) an adaptive controller, in the spirit of the adaptive parallel-execution policy proposed alongside the unified cost model [36], that automatically selects d_{max} and l_{max} from the conflict density $|CS|/|C|$ observed on previous calls to the same knowledge base – our results in Table 3 give a first, empirically grounded reason to believe such a controller is worthwhile, since the best-performing configuration changed with the consistency-check cost regime; and (iv) applying PARALLELMERGEEXPLAIN within the parallel feature-model-analysis pipelines that already parallelize constraint elimination upstream of solving [17, 18], to assess whether the two forms of parallelism compose favorably in an end-to-end automated-analysis-of-feature-models setting [11].

Acknowledgments. The authors thank the authors of the original MERGEEXPLAIN [31] and PARALLELQUICKXPLAIN [33, 34] algorithms, whose published pseudo-code and proofs made the formal development in Sections 3–4 possible.

Abbreviations. The following abbreviations are used in this manuscript:

CSP	Constraint Satisfaction Problem
MBD	Model-Based Diagnosis
QX	GETCONFLICT (the core recursive routine of QUICKXPLAIN)
MPX	MERGEEXPLAIN
PMX	PARALLELMERGEEXPLAIN
HS-tree	Hitting-Set Tree

Declarations

- **Funding.** This research received no external funding.
- **Conflict of interest.** The authors declare no conflicts of interest.
- **Ethics approval.** Not applicable.
- **Consent to participate.** Not applicable.
- **Consent for publication.** Not applicable.
- **Availability of data and materials.** The Python reference implementation of MERGEXPLAIN and PARALLELMERGEXPLAIN, the solver-verified synthetic benchmark generator, the real-world benchmark builder, the raw measurement data underlying Table 3 (`results_v2.csv`), the correctness-check scripts, and the analysis script used to aggregate the results are openly available at <https://github.com/cvidalmsu/ParallelMergeXplain>, together with a reproducibility guide describing how to regenerate every number reported in Section 6. The large-scale follow-up study outlined in Section 6.5 would additionally build on the openly available datasets cited therein [7, 8, 32].
- **Code availability.** All source code (MERGEXPLAIN, PARALLELMERGEXPLAIN, the benchmark generators, the CDCL solver, correctness tests, and the timing/analysis scripts) is released under the MIT License at <https://github.com/cvidalmsu/ParallelMergeXplain>.
- **ORCID.** Luis Castillo-Heredia: <https://orcid.org/0000-0003-4650-8275>.
- **Authors' contributions.** Conceptualization: C.V.-S.; methodology: C.V.-S. and J.C.-C.; software, validation and formal analysis: C.V.-S., with review by L.C.-H. and J.C.-C.; investigation: all authors; writing (original draft): C.V.-S.; writing (review and editing): all authors. All authors have read and agreed to the published version of the manuscript.

References

- [1] AlShekh RH, Dawwd SA, Qassabbashi FN (2025) Comparative review of multicore architectures: Intel, amd, and arm in the modern computing era. *Chips* 4(4):44. <https://doi.org/10.3390/chips4040044>
- [2] Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS Spring Joint Computer Conference*, pp 483–485, <https://doi.org/10.1145/1465482.1465560>
- [3] Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35(6):615–636
- [4] Blumofe RD, Leiserson CE (1999) Scheduling multithreaded computations by work stealing. *Journal of the ACM* 46(5):720–748. <https://doi.org/10.1145/324133.324234>
- [5] Bordeaux L, Hamadi Y, Samulowitz H (2009) Experiments with massively parallel constraint solving. In: *21st International Joint Conference on Artificial*

- Intelligence (IJCAI 2009). Morgan Kaufmann, San Francisco, CA, USA, pp 443–448
- [6] Burton FW (1985) Speculative computation, parallelism, and functional programming. *IEEE Transactions on Computers* C-34(12):1190–1193. <https://doi.org/10.1109/TC.1985.6312218>
 - [7] Cabezas-Quinto JJ, Vidal-Silva C, Serrano-Malebrán J, et al (2025) A dataset and experimental evaluation of a parallel conflict detection solution for model-based diagnosis. *Data* 10(9):139. <https://doi.org/10.3390/data10090139>
 - [8] Carrión León DI, Vidal-Silva C, Márquez N (2025) Scalable model-based diagnosis with fastdiag: A dataset and parallel benchmark framework. *Data* 10(9):141. <https://doi.org/10.3390/data10090141>
 - [9] Felfernig A, Schubert M, Zehentner C (2012) An efficient diagnosis algorithm for inconsistent constraint sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 26(1):53–62. <https://doi.org/10.1017/S0890060411000011>
 - [10] Galindo JA, Benavides D (2019) Towards a new repository for feature model exchange. In: 23rd International Systems and Software Product Line Conference (SPLC 2019), Volume B. ACM, pp 85:1–85:4, <https://doi.org/10.1145/3307630.3342405>
 - [11] Galindo JA, Benavides D, Trinidad P, et al (2019) Automated analysis of feature models: Quo vadis? *Computing* 101(5):387–433. <https://doi.org/10.1007/s00607-018-0646-1>
 - [12] Gent I, Miguel I, Nightingale P, et al (2018) A review of literature on parallel constraint solving. *Theory and Practice of Logic Programming* 18(5-6):725–758
 - [13] Gent IP, Jefferson C, Miguel I (2006) Minion: A fast, scalable, constraint solver. In: 17th European Conference on Artificial Intelligence (ECAI 2006). IOS Press
 - [14] Gustafson JL (1988) Reevaluating amdahl’s law. *Communications of the ACM* 31(5):532–533. <https://doi.org/10.1145/42411.42415>
 - [15] Hamadi Y, Sais L (2018) *Handbook of Parallel Constraint Reasoning*. Springer
 - [16] Hennessy JL, Patterson DA (2017) *Computer Architecture: A Quantitative Approach*, 6th edn. Morgan Kaufmann
 - [17] Horcas JM, Ballesteros J, Pinto M, et al (2023) Elimination of constraints for parallel analysis of feature models. In: 27th ACM International Systems and Software Product Line Conference (SPLC’23), <https://doi.org/10.1145/3579027.3608981>
 - [18] Horcas JM, Ballesteros J, Pinto M, et al (2025) Fmsans: An efficient approach for constraints removal and parallel analysis of feature models. *Journal of Systems*

and Software 227:112434. <https://doi.org/10.1016/j.jss.2025.112434>

- [19] Jannach D, Schmitz T, Shchekotykhin K (2015) Parallelized hitting set computation for model-based diagnosis. In: 29th AAAI Conference on Artificial Intelligence. AAAI Press, pp 1503–1510
- [20] Jannach D, Schmitz T, Shchekotykhin K (2016) Parallel model-based diagnosis on multi-core computers. *Journal of Artificial Intelligence Research* 55:835–887
- [21] Junker U (2004) Quickxplain: Preferred explanations and relaxations for over-constrained problems. In: 19th National Conference on Artificial Intelligence (AAAI’04). AAAI Press, San Jose, CA, USA, pp 167–172
- [22] de Kleer J, Williams BC (1987) Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130
- [23] Kuiter E (2025) Towards effective and efficient feature-model analyses for evolving system software. In: 29th ACM International Systems and Software Product Line Conference (SPLC-B’25), <https://doi.org/10.1145/3748269.3748593>
- [24] Le VM, Vidal-Silva C, Felfernig A, et al (2023) Fastdiagp: An algorithm for parallelized direct diagnosis. In: 37th AAAI Conference on Artificial Intelligence, pp 6442–6449, <https://doi.org/10.1609/aaai.v37i5.25792>
- [25] Le VM, Felfernig A, Tran TNT, et al (2024) Informedqx: Informed conflict detection for over-constrained problems. In: 38th AAAI Conference on Artificial Intelligence, pp 10616–10623
- [26] Le Berre D, Parrain A (2010) The sat4j library, release 2.2: System description. *Journal on Satisfiability, Boolean Modeling and Computation* 7(2-3):59–64
- [27] Ouyang J, Huang S, Chi J, et al (2025) Effective fault identification approach for model-based diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 56(4):2361–2373. <https://doi.org/10.1109/TSMC.2025.3649682>
- [28] Rauber T, Rünger G (2023) *Parallel Programming: For Multicore and Cluster Systems*, 3rd edn. Springer, <https://doi.org/10.1007/978-3-031-28924-8>
- [29] Reiter R (1987) A theory of diagnosis from first principles. *Artificial Intelligence* 32:57–95. [https://doi.org/10.1016/0004-3702\(87\)90062-2](https://doi.org/10.1016/0004-3702(87)90062-2)
- [30] Rodler P (2023) Sequential model-based diagnosis by systematic search. *Artificial Intelligence* 323:103988. <https://doi.org/10.1016/j.artint.2023.103988>
- [31] Shchekotykhin K, Jannach D, Schmitz T (2015) Mergexplain: Fast computation of multiple conflicts for diagnosis. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015). AAAI Press, pp 3221–3228

- [32] Sundermann C, Brancaccio VF, Kuitert E, et al (2024) Collecting feature models from the literature: A comprehensive dataset for benchmarking. In: 28th ACM International Systems and Software Product Line Conference (SPLC'24), <https://doi.org/10.1145/3646548.3672590>
- [33] Vidal C, Felfernig A, Galindo JA, et al (2020) A parallelized variant of junker's quickxplain algorithm. In: Foundations of Intelligent Systems, Lecture Notes in Computer Science, vol 12117. Springer, Cham, Switzerland, https://doi.org/10.1007/978-3-030-59491-6_44
- [34] Vidal C, Felfernig A, Galindo JA, et al (2021) Explanations for over-constrained problems using quickxplain with speculative executions. Journal of Intelligent Information Systems 57:491–508. <https://doi.org/10.1007/s10844-021-00675-4>
- [35] Vidal-Silva C, Duarte V, Cárdenas-Cobo J, et al (2024) Speculative computing for aafm solutions in large-scale product configurations. Scientific Reports 14:11182. <https://doi.org/10.1038/s41598-024-61647-6>
- [36] Vinueza-Morales M, Tupac-Yupanqui M, Márquez N, et al (2026) A unified multi-dimensional cost analysis of speculative parallel conflict detection and diagnosis. Computers 15(4):201. <https://doi.org/10.3390/computers15040201>
- [37] White J, Benavides D, Schmidt DC, et al (2010) Automated diagnosis of feature model configurations. Journal of Systems and Software 83(7):1094–1107