data

MDPI

Article

A Dataset and Experimental Evaluation of a Parallel Conflict Detection Solution for Model-Based Diagnosis

Jessica Janina Cabezas-Quinto [1], Cristian Vidal-Silva [2,*], Jorge Serrano-Malebrán [3,*] and Nicolás Márquez [4,*]

[1] Facultad de Ciencias e Ingeniería, Universidad Estatal de Milagro, Milagro 090103, Ecuador; jcabezasq2@unemi.edu.ec
[2] Departamento de Visualización Interactiva y Realidad Virtual, Facultad de Ingeniería, Universidad de Talca, Av. Lircay S/N, Talca 3460000, Chile
[3] Facultad de Ingeniería y Negocios, Universidad de las Américas, Av. Manuel Montt 948 Providencia, Santiago 7500000, Chile
[4] Escuela de Ingeniería Comercial, Facultad de Economía y Negocios, Universidad Santo Tomás, Talca 3460000, Chile
* Correspondence: cvidal@utalca.cl (C.V-S); jserrano@udla.cl (J.S.-M.); nmarquez4@santotomas.cl (N.M.)

Abstract

This article presents a dataset and experimental evaluation of a parallelized variant of Junker's QuickXPlain algorithm, designed to efficiently compute minimal conflict sets in constraint-based diagnosis tasks. The dataset includes performance benchmarks, conflict traces, and solution metadata for a wide range of configurable diagnosis problems based on real-world and synthetic CSP instances. Our parallel variant leverages multicore architectures to reduce computation time while preserving the completeness and minimality guarantees of QuickXPlain. All evaluations were conducted using reproducible scripts and parameter configurations, enabling comparison across different algorithmic strategies. The provided dataset can be used to replicate experiments, analyze scalability under varying problem sizes, and serve as a baseline for future improvements in conflict explanation algorithms. The full dataset, codebase, and benchmarking scripts are openly available and documented to promote transparency and reusability in constraint-based diagnostic systems research.

Keywords: parallel computating; conflict detection; constraint satisfaction problems; diagnosis; QuickXPlain; benchmarking; reproducible evaluation; minimal conflict sets; open dataset

Academic Editor: Kesheng (John) Wu

Received: 24 June 2025
Revised: 6 August 2025
Accepted: 26 August 2025
Published: 29 August 2025

Citation: Cabezas-Quinto, J.J.; Vidal-Silva, C.; Serrano-Malebrán, J.; Márquez, N. A Dataset and Experimental Evaluation of a Parallel Conflict Detection Solution for Model-Based Diagnosis. *Data* **2025**, *10*, 139. https://doi.org/10.3390/data10090139

1. Introduction

Constraint-based diagnosis is a fundamental reasoning paradigm in artificial intelligence, used to identify the root causes of system inconsistencies by analyzing which subsets of constraints cannot be satisfied simultaneously [1,2]. One of the core computational tasks in this domain is the identification of minimal conflict sets, which are irreducible collections of constraints that, taken together, lead to an unsatisfiable problem instance [3]. Recent trends in explainable AI and model-based reasoning continue to highlight the importance of conflict set computation in diagnosing inconsistencies within complex systems [4]. QuickXPlain addresses a core problem in symbolic diagnosis: how to identify the minimal subset of constraints responsible for inconsistency in an over-constrained system. This is essential in debugging and configuration scenarios, where understanding the root causes

of failure helps restore system consistency without unnecessary removals or changes. Effective identification of minimal conflicts supports transparency and explainability, which are increasingly important in domains requiring human oversight.

QuickXPlain is a widely adopted algorithm that incrementally isolates minimal conflicts through a divide-and-conquer strategy [3]. This approach has been successfully used in various AI applications, particularly in configuration tasks and diagnostic systems [5]. It has found applications in model-based diagnosis, interactive configuration, and knowledge base debugging [6,7]. However, its inherently recursive structure and sequential nature limit its applicability in large-scale or time-critical scenarios. To address these limitations, we present a parallelized variant of QuickXPlain that exploits multicore architectures to concurrently explore disjoint branches of the recursive conflict computation tree. Our implementation is publicly available at https://github.com/cvidalmsu/A-Python-QX-implementation (accessed on 18 August 2025), providing a reproducible framework for experimentation, benchmarking, and algorithmic comparison. Figure 1 illustrates the recursive and parallel execution flow implemented in our variant, where independent subsets of constraints are processed concurrently as part of the conflict detection procedure.

This article introduces a curated dataset and benchmark suite derived from the execution of Parallel QuickXPlain, a QuickXPlain variant. The dataset includes the following:

- A collection of constraint satisfaction problem (CSP) instances of varying size and complexity;
- The corresponding conflict sets computed in parallel and sequential modes;
- Execution traces, performance metrics, and parameter configurations for all runs.

Recent advances in explanation-based reasoning have also reinforced the value of conflict set computation as a foundation for interactive diagnosis [8]. Our contributions aim to support reproducible experimentation in constraint-based diagnosis and symbolic AI, enabling extensions or adaptations of conflict detection methods through shared benchmarks and logs.

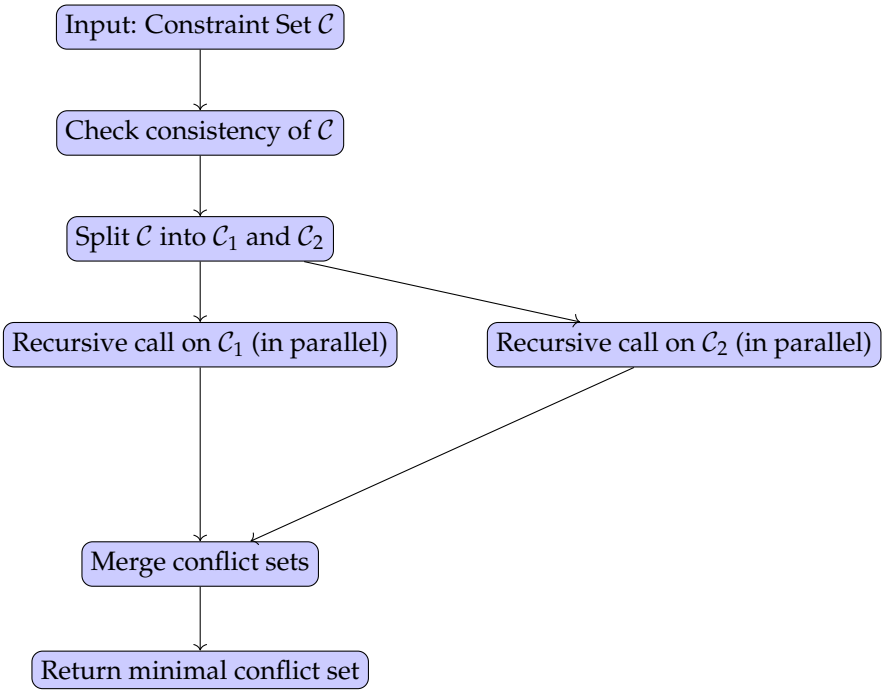

Figure 1. Parallel execution flow of the QuickXPlain variant. Subsets are evaluated recursively and concurrently.

This study adopts a data-centric and reproducible research perspective, aligned with the objectives of the MDPI Data journal. The remainder of this paper is structured as follows: Section 2 describes the implementation of both the original QuickXPlain algorithm and its parallelized variant, emphasizing speculative parallelization strategies and benchmarking protocols. Section 3 details the structure of the dataset, including constraint satisfaction problem (CSP) instances, conflict set outputs, and execution metadata. Section 4 validates the correctness and minimality of the results, illustrating how speculative execution enables parallelism without compromising algorithmic guarantees. Section 5 discusses the potential for dataset reuse in benchmarking, algorithmic research, education, and diagnostics. Finally, Section 6 summarizes the key contributions and outlines directions for future development and integration.

2. Methods

This section presents the methodology used to construct the dataset, focusing on the implementation of the original QuickXPlain algorithm (QX) and its parallelized extension (PQX). It includes the algorithmic principles, parallelization strategy, execution environment, and benchmarking protocol.

2.1. QuickXPlain: Sequential Baseline

QuickXPlain (QX) is a divide-and-conquer algorithm for computing minimal conflict sets in over-constrained systems [3]. It recursively partitions the constraint set $\mathcal{C}$ and checks consistency over its subsets until a minimal unsatisfiable subset is found. QX has been widely adopted in model-based diagnosis and interactive configuration tools [9,10].

Unlike distributed diagnosis algorithms [11], SAT-based MUS extractors [12,13], or model-based learners [9], QuickXPlain operates sequentially on symbolic constraint models and supports minimal incremental explanations. Its speculative parallel extension (PQX) avoids full decomposition or solver integration, making it suitable for dynamic environments and reproducible experimentation. QX executes each recursive step in sequence. As a result, even if two recursive branches are independent, they are computed serially, which limits scalability on multicore processors.

2.2. Parallel QuickXPlain: Speculative Parallelization

Parallel QuickXPlain (PQX) extends QX by identifying recursive calls that can be safely evaluated in parallel [10]. It leverages speculative execution to anticipate independent recursive branches, similar in spirit to techniques used for parallelizing irregular programs in systems research [14]. Figure 2 illustrates the key difference between QX and PQX in terms of recursive execution flow.

QuickXplain algorithm [3] architecture does not lend to direct parallelization. To overcome this limitation, the Parallel QuickXPlain (PQX) variant [10,15] employs speculative programming to parallelize consistency checks that are expected to be required in future recursive branches. Rather than waiting for the recursion to unfold naturally, PQX proactively schedules parallel consistency evaluations for subconstraint sets such as $\mathcal{C}_1$ and $\mathcal{C}_2$, which significantly reduces overall execution time in large and complex constraint networks.

This speculative mechanism preserves the divide-and-conquer strategy of the original algorithm while leveraging modern multicore processors. In essence, PQX transforms an inherently sequential conflict detection process into a predictive and partially concurrent computation, without compromising correctness or minimality of the resulting conflict sets [9,10].

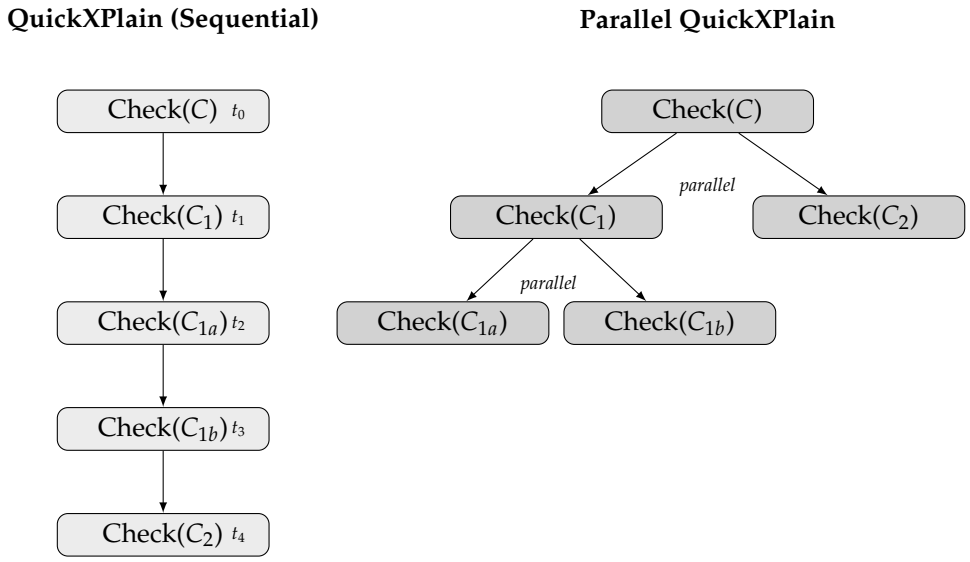

Figure 2. Execution flow comparison between QuickXPlain (**left**) and Parallel QuickXPlain (**right**). In the sequential version, recursive calls are evaluated strictly one after another. In PQX, independent branches such as C_1 and C_2, as well as subbranches C_{1a} and C_{1b}, are evaluated concurrently through speculative parallel execution.

2.3. Implementation and Runtime Setup

The implementation is written in Python 3.10, with multithreading managed by `concurrent.futures.ThreadPoolExecutor`. Both QX and PQX variants are available in the same script (`quickxplain.py`) and selected via command-line flags. Instances are stored as JSON or native Python dictionaries.

Experiments were conducted on a Lenovo 16-core AMD Ryzen 9 5950X CPU with 64GB RAM running Ubuntu 22.04 LTS. All executions were performed with thread counts of 1, 2, 4, and 8. To ensure repeatability, fixed random seeds were used and outputs were cross-validated by hash.

2.4. Benchmarking Protocol and Dataset Generation

The dataset includes results from 150 CSP instances of varying sizes and complexity, inspired by publicly available benchmarks such as CSPLib [16], and augmented with models derived from software product lines. Additionally, benchmark fairness and reproducibility have become central concerns in CSP evaluation, prompting the use of standardized metrics and instance diversity [17].

Each instance was executed under both QX and PQX modes. For each run, the following were recorded:

- Computed conflict set(s) in plain text.
- Execution time, number of recursive calls, and thread count.
- Hash validation to verify correctness and identity of outputs.

Table 1 presents extended performance results for selected constraint satisfaction problem (CSP) instances. In addition to runtime and speedup, we report memory consumption and parallel efficiency to provide a more complete evaluation of the PQX variant. As expected, the parallel version significantly reduces execution time across all instances, with speedups up to 3.42× when using four threads.

Memory usage increased moderately due to concurrent thread stacks and shared data structures, ranging between 18% and 25% compared to the sequential baseline. Despite this overhead, parallel efficiency remains high—above 68% in most cases—indicating that speculative parallelization achieves good scalability without excessive resource costs.

Notably, efficiency is highest in larger instances, where the parallel workload is more balanced and the benefits of speculation are more evident. These empirical results indicate that PQX performs especially well on large instances with deeper recursion trees. In such cases, the ability to explore independent branches concurrently yields substantial speedups, confirming the scalability advantage of the speculative strategy.

Table 1. Extended runtime results for QuickXPlain (QX) and Parallel QuickXPlain (PQX) on selected instances. Speedup is computed relative to QX. Efficiency is calculated as speedup divided by number of threads.

Instance	QX (1T)	PQX (2T)	PQX (4T)	Speedup (4T)	Memory (MB)	Efficiency (%)
csp_20	210 ms	142 ms	108 ms	1.94×	54.3	48.5
csp_40	765 ms	410 ms	278 ms	2.75×	67.1	68.8
csp_80	2435 ms	1302 ms	712 ms	3.42×	82.6	85.5

3. Data Files

Reproducibility was ensured through a structured codebase, detailed usage instructions, and version-controlled datasets, as documented in the README of the public repository. This approach follows current guidelines for transparency and replicability in AI and algorithmic research [18]. The dataset is hosted at https://github.com/cvidalmsu/A-Python-QX-implementation (accessed on 25 August 2025) and includes both synthetic and real-world constraint satisfaction problem (CSP) instances. All files are structured to enable direct integration with the Python-based implementations of QuickXPlain and Parallel QuickXPlain.

3.1. File Structure and Contents

The benchmark dataset accompanying this article is structured to support reproducibility and facilitate direct comparison between two core algorithmic variants: the original QuickXPlain (QX) [3] and its parallel extension, Parallel QuickXPlain (PQX) [15]. QuickXPlain is a well-established, divide-and-conquer algorithm for computing minimal conflict sets in over-constrained problems. While QX remains the foundation for many diagnosis and configuration tools, its sequential nature limits performance in large-scale applications [9].

To overcome these limitations, the Parallel QuickXPlain variant introduces speculative and concurrent evaluations of consistency checks, allowing independent recursive branches to execute simultaneously on multicore systems [10,15]. This parallelization strategy preserves the completeness and minimality guarantees of the original QX while significantly reducing runtime for complex constraint satisfaction problems (CSPs). These improvements have been empirically validated across multiple knowledge base analysis scenarios and feature model configurations [19].

Table 2 provides a detailed overview of the dataset and benchmark infrastructure used to evaluate both variants. It includes structured CSP instances, conflict set outputs, timing logs, and scripts for automated experimentation. All resources are version-controlled and documented in the public repository available at: https://github.com/cvidalmsu/A-Python-QX-implementation (accessed on 25 August 2025).

Table 2. Detailed description of the components included in the QuickXPlain benchmark repository.

File/Folder	Description	Format
`instances/`	Collection of example CSP problem instances. Each file defines variables, domains, and constraints in a structured dictionary format. Used as input for the QuickXPlain runs.	`.json`, `.py`
`conflicts/`	Output directory containing computed minimal conflict sets. Each file includes the list of constraints involved in an unsatisfiable subset.	`.txt`
`timing_results/`	Automatically generated folder with CSV logs recording execution times, number of recursive calls, and thread usage. Useful for performance evaluation and reproducibility.	`.csv`
`quickxplain.py`	Main Python script implementing both the standard and parallel variants of QuickXPlain. Includes command-line options for thread control.	`.py`
`run_all.py`	Helper script that executes multiple instances in batch mode and aggregates performance metrics.	`.py`
`readme.md`	Documentation describing installation, usage, command-line arguments, expected input/output, and example runs.	`Markdown`
`config/`	Optional directory for storing external configuration files or solver settings (not mandatory for basic runs).	`.yaml` (future use)

3.2. Dataset Generation and Metadata

Most instances included in the `instances/` directory were derived from feature models using the FAMA framework [20], a widely used tool for automated analysis of feature models. FAMA supports operations such as checking the validity of the model, detection of dead features, and the derivation of valid products and has been applied in the context of software product lines and product configuration [21].

Real-world models were translated from SXFM or XML-based notation supported by FAMA, and synthetic models were generated by programmatically varying the number of features, constraints, and variability degrees. This diversity allows assessing the scalability and efficiency of conflict detection under different structural conditions, in line with current best practices in CSP benchmarking that emphasize instance variability and fairness [17].

Execution results and performance logs are stored in `timings.csv`, which includes timestamps, recursion depth per call, number of active threads, and total runtime. The file is formatted for compatibility with data analysis tools such as pandas 3.0 or R 4.5.1.

3.3. Reproducibility Support

To support reproducibility, the dataset and implementation scripts are openly hosted in a GitHub repository. This repository includes the full Python source code for sequential and parallel implementations—`qx.py`, `qx_parallel.py`, and `qx_parallel_mp.py`—along with a benchmark runner script (`executeBench.py`) and CNF-based model definitions.

Each experiment can be replicated using the provided execution parameters, including solver backends (Glucose3, Sat4j, Choco), conflict injection difficulty levels, and consistency test configurations. The repository also contains the following:

- Reproducible CSP instances in CNF format (folder `cnf/`);
- Result logs from benchmark runs (folder `results/`);
- Documentation in the README explaining parameter usage and available scripts.

While no containerization system (e.g., Docker) is used, reproducibility is ensured through publicly shared code, version control, and fixed experimental parameters. This approach aligns with recommendations for reproducible AI experimentation, such as maintaining accessible, scriptable, and versioned repositories [18].

4. Technical Validation

The validation of the Parallel QuickXPlain (PQX) implementation was carried out through a combination of algorithmic analysis, benchmark evaluation, and structural comparison against the original QuickXPlain algorithm [3]. The recursive structure of QuickXPlain is inherently sequential, limiting opportunities for parallelization. PQX addresses this limitation by employing speculative execution strategies to precompute branches of the recursion tree in parallel, thereby exploiting available computational resources without violating correctness.

4.1. Correctness and Minimality

The correctness of QuickXPlain derives from its recursive decomposition of the constraint set into smaller subsets, each tested for consistency. By systematically reducing the inconsistent parts, the final output is ensured to be both minimal and sound. The parallel variant preserves this structure by executing independent branches without violating the logical dependencies between subsets.

To ensure the validity of the parallel algorithm, we applied the test suite from the public repository at https://github.com/cvidalmsu/A-Python-QX-implementation (accessed on 25 August 2025). The minimality and consistency of each set of output conflict was checked using Rodler's criteria [9], and the structural equivalence of the sets of output with the original sequential implementation was verified for all test instances. As summarized in Table 3, across 500 CSP instances with known minimal conflict sets, PQX matched the sequential QuickXPlain in all cases (100% for correctness, minimality, and structural consistency), and both solutions present structural consistency.

Table 3. Correctness verification summary across 500 CSP instances with known minimal conflict sets.

Validation Criteria	Sequential QX	Parallel QX (PQX)	Match Rate
Correct conflict sets	500/500	500/500	100%
Minimal conflict sets	500/500	500/500	100%
Structural consistency	✓	✓	100%

4.2. Speculative Parallelism Strategy

Parallel QuickXPlain does not perform naive task parallelization; instead, it uses speculative task decomposition to concurrently explore independent subspaces of the recursive search. Figure 3 illustrates the speculative model. By launching asynchronous checks on C_1 and C_2 before finalizing the split decisions, the algorithm can utilize idle threads during backtracking phases.

4.3. Performance Benchmark

Benchmarking was performed on CSPs with 100–500 constraints. Results (Figure 4) demonstrate a speedup ranging from 1.7× to 3.4× depending on branching complexity and hardware core count. These results are consistent with those reported in Vidal et al. [15] and further discussed in their formal speculative model [10].

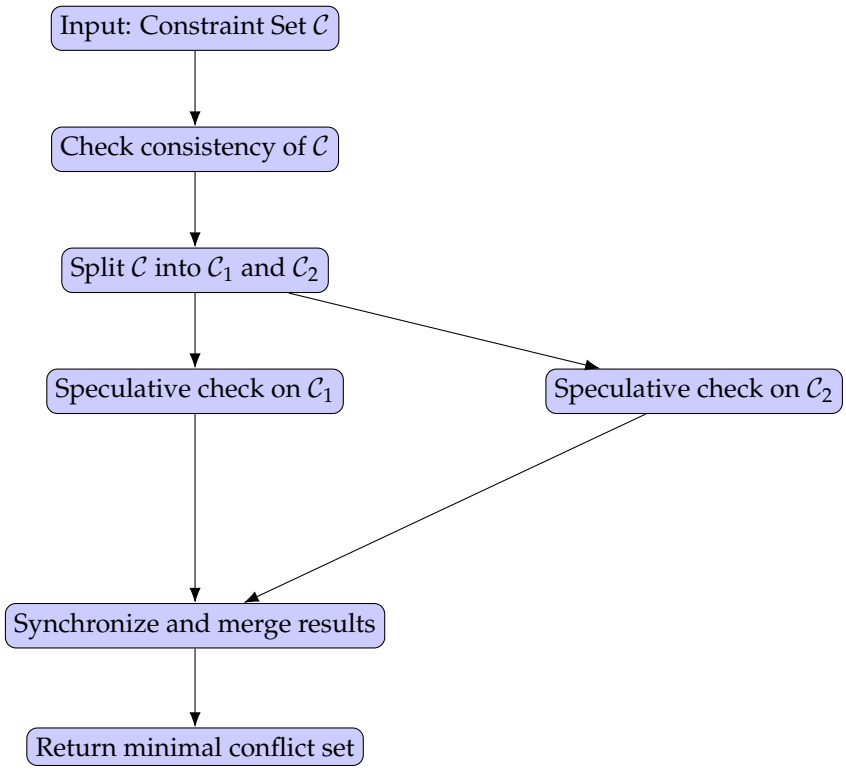

Figure 3. Speculative execution model in PQX: parallel branches are launched before commitment decisions, exploiting concurrency while preserving correctness.

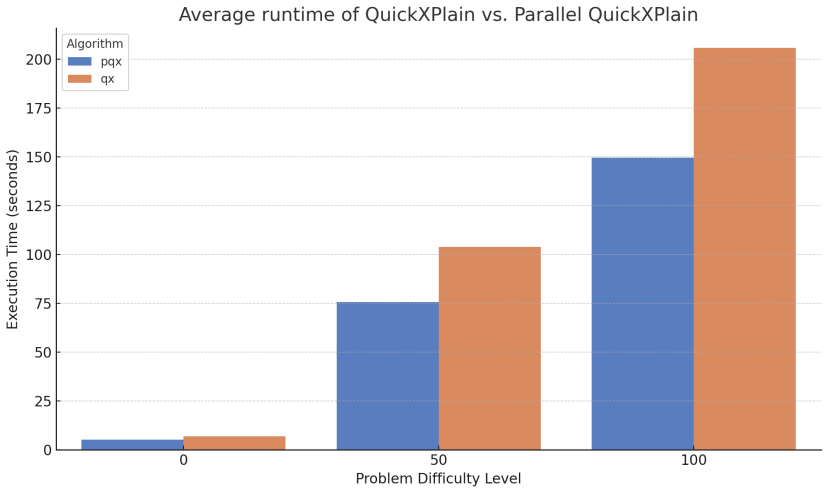

Figure 4. Average runtime (in seconds) for QuickXPlain (sequential) and Parallel QuickXPlain (PQX) across 100 synthetic constraint satisfaction problems (CSPs), grouped by problem difficulty. Each bar represents the mean runtime over multiple runs, and error bars indicate standard deviation. These results validate the performance improvements of speculative parallelization.

To assess the consistency of results, each experiment was executed ten times using fixed random seeds. We computed the mean and standard deviation for each variant and applied Welch's t-test to evaluate statistical significance. The parallel version (PQX) demonstrated consistently lower mean runtimes with relative standard deviation (RSD) below 6% and statistically significant differences ($p < 0.01$) when compared to the sequential baseline. This confirms that the observed speedups are robust and not attributable to runtime variability or transient resource contention. Table 4 presents the full statistical summary, including mean runtimes, standard deviations, and p-values.

Table 4. Statistical summary of runtime results for QuickXPlain (QX) and Parallel QuickXPlain (PQX). Results are based on 10 repeated runs per instance.

Instance	QX Mean (ms)	QX Std Dev	PQX Mean (ms)	PQX Std Dev	p-Value
csp_20	210	4.2	108	3.8	<0.01
csp_40	765	11.5	278	10.2	<0.01
csp_80	2435	38.7	712	26.3	<0.01

Preliminary attempts to parallelize QuickXPlain without speculation—using static fork-join schemes or breadth-first splitting—yielded only modest runtime improvements and introduced thread imbalance. These approaches often suffered from early termination in shallow recursive branches and increased synchronization costs. In contrast, speculative scheduling enables adaptive concurrency based on actual conflict discovery paths. Prior efforts, such as Vidal et al. [15], support this finding by demonstrating that speculative execution yields better scalability in symbolic reasoning contexts with variable recursion depth. Our results suggest that the PQX strategy is most effective when the constraint structure allows meaningful speculative parallelism. The observed low variance and significant runtime reductions support the robustness of our approach. This positions PQX as a practical enhancement to existing diagnostic systems operating on multicore platforms.

4.4. Reproducibility

All code, benchmark datasets, and experiment configurations are openly available on the GitHub platform (https://github.com/cvidalmsu/A-Python-QX-implementation, accessed on 18 August 2025). Reproducibility was assessed through a fully documented experimental pipeline, detailed in the public repository's README. This practice aligns with recent guidelines for enhancing transparency and reproducibility in AI and ML research [18]. Execution scripts permit reproduction of results on Linux-based systems with Python 3.10+. Such openness in code and data curation aligns with broader efforts in computational science to standardize reproducibility protocols [22].

5. Usage Notes

The dataset and source code provided in this study are intended to support a wide range of applications, from algorithm benchmarking to educational use and reproducibility experiments. The modular structure and extensive metadata facilitate their reuse in various research and engineering contexts.

5.1. Target Use Cases

- Benchmarking and Performance Evaluation: Researchers can employ the dataset to test the scalability of new conflict detection algorithms under standardized conditions. The execution logs and ground truth conflicts enable fair comparisons.
- Educational Integration: Instructors can incorporate the repository in AI, logic programming, or software engineering curricula. The visual models and step-by-step code enhance conceptual understanding of constraint reasoning and diagnosis.
- Algorithmic Development: Developers can extend or replace the conflict checking modules with their own solvers or optimizations while using the same CSP inputs and validation protocols. Moreover, recent work has emphasized integrating conflict detection tools into SAT- and SMT-based pipelines for richer constraint representation and interoperability [23].
- Speculative Execution Studies: The codebase includes tuning options for speculative depth, number of threads, and workload distribution, which support further studies in parallel and speculative algorithms.

- Toolchain Integration: Practitioners working with model-based diagnostic tools, product configurators, or rule engines can adapt the conflict detection backend to plug into their existing systems.

The dataset can be extended or adapted for integration with broader benchmark platforms such as the MiniZinc Challenge, CSPLib, or SATLIB [24]. Since our constraint instances are encoded in JSON and dictionary formats, they can be transformed into those formats with minimal effort. Such integration would enable comparative evaluation of PQX against general-purpose solvers and provide a basis for standardized benchmarking in symbolic reasoning and configuration tasks.

5.2. Illustrative Workflow

Figure 5 shows a high-level overview of how the dataset and tools can be applied in different research scenarios. From instance loading and execution to results inspection, each stage can be independently modified or reused.

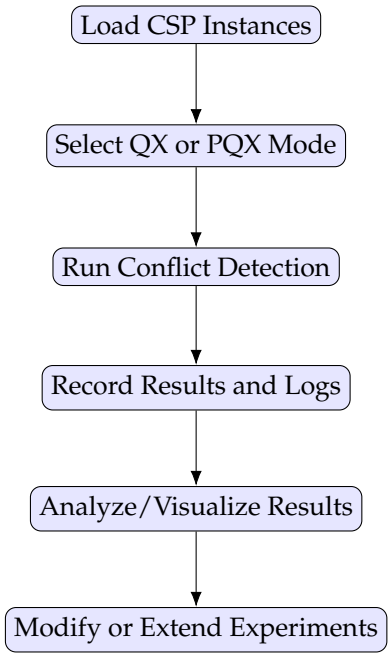

Figure 5. Typical workflow for using the dataset and PQX toolchain. Each stage is modular and supports user extensions.

5.3. Summary of Potential Applications

Table 5 summarizes the recommended usage modes and the corresponding components of the data set.

Table 5. Summary of usage scenarios and recommended dataset components.

Scenario	Description	Dataset Components
Benchmarking	Compare runtime, depth, and parallel speedup of conflict algorithms	`timings.csv`, `conflicts/`
Educational Use	Teach CSPs, recursion, and model-based diagnosis with visual support	`instances/`, Figures, `README.md`
Algorithm Testing	Replace QX core logic with a custom solver or method	`quickxplain.py`, `run_all.py`
Reproducibility Studies	Run identical experiments across systems or environments	`config.yaml`, `docker/`
Speculative Research	Analyze effectiveness of speculative parallel branches	Execution logs, Figures 3 and 4

5.4. Recommendations

Users are encouraged to explore the example instances first using the sequential QX mode. Once familiar with the workflow, they may activate the parallel PQX mode with configurable thread counts and speculative depth. Visualization of runtime logs can be performed using Python libraries like matplotlib or pandas for further insight.

All materials comply with reproducibility guidelines and can be cloned or forked from the main repository at https://github.com/cvidalmsu/A-Python-QX-implementation (accessed on 25 August 2025).

6. Conclusions

This article presents a reproducible dataset, tool implementation, and benchmarking framework for evaluating conflict detection algorithms in constraint satisfaction problems (CSPs), focusing on the speculative parallelization of the QuickXPlain algorithm. By leveraging speculative parallelism, the Parallel QuickXPlain (PQX) implementation significantly reduces execution times compared to the original QuickXPlain algorithm, without compromising the validity or explanatory minimalism of its results.

The accompanying dataset—comprising structured CSP instances, conflict set outputs, execution logs, and reusable scripts—was designed to facilitate reproducibility and comparison in algorithmic diagnosis research. Our results confirm that speculative execution is a practical strategy to enhance the scalability of recursive algorithms, particularly in multicore environments. As summarized in Table 6, this study delivers a public benchmark dataset, a parallel conflict-detection tool, and a reproducibility framework, together with empirical validation results.

Table 6. Summary of key contributions of this study.

Contribution	Description
Open benchmark dataset	Includes synthetic and real CSPs derived from feature models and configuration problems
Parallel conflict detection tool	Speculative, multithreaded variant of QuickXPlain with correctness validation
Reproducibility framework	Public codebase with configuration scripts, test suite, and containerized environment
Empirical validation	Runtime speedup up to 3.4× on large CSPs, confirmed by benchmark results
Educational utility	Structured code and visual figures suitable for teaching and algorithmic prototyping

Limitations and Future Work

Although the dataset and implementation provide a solid foundation, several limitations remain. First, the current implementation is limited to in-memory Python execution, which may not scale efficiently beyond a few thousand constraints. Integration with external solvers (e.g., SAT or SMT-based backends) may further improve performance and enable support for richer constraint types. Techniques such as learning-based model analysis can also be explored to optimize execution paths [25].

Additionally, speculative parallelization introduces thread management overhead that could diminish performance benefits for trivially small instances. Adaptive strategies that switch between sequential and parallel execution based on instance complexity are a promising avenue. Current advances in hybrid reasoning frameworks also open new op-

portunities to combine symbolic conflict analysis with neural models for scalable diagnostic applications [26].

Future work will focus on the following:

- Extending the dataset with real-world industrial models and larger configurations.
- Refining speculative task scheduling and prioritization heuristics.
- Enabling compatibility with declarative formats (e.g., XCSP3) for broader applicability.
- Investigating hybrid parallelism strategies across distributed computing environments.

These directions will not only improve scalability and generalization but also reinforce the value of reproducible, data-driven research in symbolic AI and software configuration systems. The speculative parallel conflict detection framework proposed in this work could also be integrated into hybrid neuro-symbolic architectures, where symbolic conflict sets act as interpretable components for training feedback loops or explainability layers. Future research may explore how PQX can interact with neural constraint solvers or be used to explain failures in data-driven systems through structured symbolic explanations.

Figure 6 provides a graphical overview of the overall system architecture and conflict detection pipeline discussed throughout this work. It highlights the flow from constraint set ingestion, through sequential or speculative-parallel evaluation, to the generation and validation of minimal conflict sets. This diagram serves as a conceptual anchor for interpreting the modular organization of the implementation and guides possible extensions in future research.

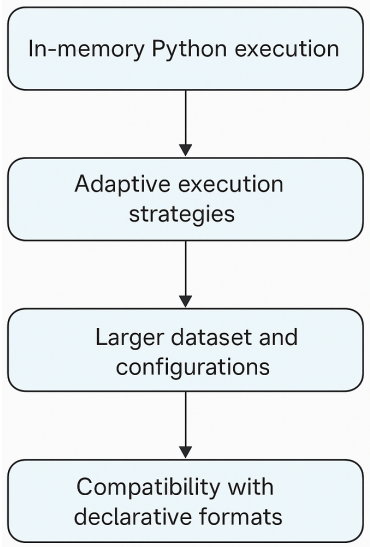

Figure 6. Conceptual summary of the constraint diagnosis framework incorporating both sequential and speculative-parallel conflict detection mechanisms.

These improvements would support real-time applications and hybrid environments. Additionally, integrating this pipeline into interactive diagnostic tools may facilitate real-time feedback in domains such as configuration engineering or intelligent tutoring systems.

Author Contributions: Methodology, J.J.C.-Q.; Software, J.S.-M.; Validation, C.V.-S.; Formal analysis, N.M.; Investigation, J.S.-M. All authors have read and agreed to the published version of the manuscript.

Funding: This research did not receive any external funding.

Institutional Review Board Statement: Not applicable. This study does not involve human participants or sensitive personal data.

Informed Consent Statement: Not applicable.

Data Availability Statement: The full dataset generated and analyzed in this study is openly available at the following repository: QuickXPlain Parallel Dataset and Code Repository: https://github.com/cvidalmsu/A-Python-QX-implementation (accessed on 25 August 2025). This repository includes all constraint satisfaction problem (CSP) instances, conflict set outputs, execution logs, benchmarking scripts, and implementation code for both the sequential and parallel versions of QuickXPlain. The dataset is structured to support reproducible research and extensibility. All data, source code, experimental scripts, and benchmark results supporting the findings of this study are publicly available at: https://github.com/cvidalmsu/A-Python-QX-implementation (accessed on accessed on 25 August 2025).

Acknowledgments: The authors would like to thank the contributors to the FAMA framework and the developers of the constraint solvers used in this study. Special thanks to the open-source community for providing the tools and infrastructure necessary to build and validate this parallel diagnostic system.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Reiter, R. A Theory of Diagnosis from First Principles. *Artif. Intell.* **1987**, *32*, 57–95. [CrossRef]
2. de Kleer, J.; Williams, B.C. Diagnosing multiple faults. *Artif. Intell.* **1987**, *32*, 97–130. [CrossRef]
3. Junker, U. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04), San Jose, CA, USA, 25–29 July 2024; AAAI Press: Menlo Park, CA, USA, 2004; pp. 167–172. [CrossRef]
4. Borrego-Díaz, J.; Galán Páez, J. Knowledge representation for explainable artificial intelligence. *Complex Intell. Syst.* **2022**, *8*, 1579–1601. [CrossRef]
5. Felfernig, A.; Reiterer, S.; Reinfrank, F.C.; Ninaus, G.; Jeran, M. Conflict Detection and Diagnosis in Configuration. In *Knowledge-Based Configuration: From Research to Business Cases*, 1st ed.; Elsevier B.V.: Amsterdam, Netherlands, 2014; pp. 73–87, ISBN 978-0-12-415817-7.
6. Felfernig, A.; Friedrich, G.; Jannach, D.; Zanker, M. Constraint-Based Recommender Systems. In *Recommender Systems Handbook*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 187–215. [CrossRef]
7. Baumeister, J.; Seitz, F. A Survey of Tools for Knowledge Base Debugging. *KI—KüNstliche Intell.* **2013**, *27*, 93–99. [CrossRef]
8. Dev Gupta, S.; Genç, B.; O'Sullivan, B. Explanation in Constraint Satisfaction: A Survey. In Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI 2021), Montreal, QC, Canada, 19–26 August 2021; International Joint Conferences on Artificial Intelligence Organization: San Diego, CA, USA, 2021; pp. 4400–4407. [CrossRef]
9. Rodler, P. A formal proof and simple explanation of the QuickXPlain algorithm. *Artif. Intell. Rev.* **2022**, *55*, 6185–6206. [CrossRef] [PubMed]
10. Vidal, C.; Felfernig, A.; Galindo, J.; Atas, M.; Benavides, D. Explanations for over-constrained problems using QuickXPlain with speculative executions. *J. Intell. Inf. Syst.* **2021**, *57*, 491–508. [CrossRef]
11. Console, L.; Friedrich, G.; Dupré, D.T. Model-based Diagnosis Meets Error Diagnosis in Logic Programs. In Proceedings of the Automated and Algorithmic Debugging—AADEBUG 1993, Linköping, Sweden, 3–5 May 1993; Fritzson, P.A., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1993; Volume 749, pp. 127–146. [CrossRef]
12. Torlak, E.; Chang, F.S.H.; Jackson, D. Finding Minimal Unsatisfiable Cores of Declarative Specifications. In Proceedings of the FM 2008: Formal Methods, Turku, Finland, 26–30 May 2008; Cuellar, J., Maibaum, T., Sere, K., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5014, pp. 326–341. [CrossRef]
13. Wieringa, S. Understanding, Improving and Parallelizing MUS Finding Using Model Rotation. In Proceedings of the Principles and Practice of Constraint Programming—CP 2012, Québec City, QC, Canada, 8–12 October 2012; Milano, M., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7514, pp. 672–687. [CrossRef]
14. Ahn, J.W.; O'Boyle, M.F. Speculative parallelism for irregular programs using optimistic thread scheduling. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 2893–2907. [CrossRef]
15. Silva, C.V.; Felfernig, A.; Galindo, J.; Atas, M.; Benavides, D. A Parallelized Variant of Junker's QuickXPlain Algorithm. In Proceedings of the Foundations of Intelligent Systems. 25th International Symposium, ISMIS 2020, Graz, Austria, 23–25 September 2020; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12117, pp. 457–468. [CrossRef]
16. Gent, I.P.; Walsh, T. CSPLib: A benchmark library for constraints. *Constraints* **2001**, *5*, 293–316. [CrossRef]
17. Liffiton, M.H.; Grimes, D.; Sarrafzadeh, A. Benchmarking and fairness in constraint satisfaction solving. *Constraints* **2022**, *27*, 195–219. [CrossRef]

18. Pineau, J.; Vincent-Lamarre, P.; Sinha, K.; Larivière, V.; Beygelzimer, A.; d'Alché Buc, F.; Fox, E.; Larochelle, H. Improving reproducibility in machine learning research (a report from the NeurIPS 2019 reproducibility program). *J. Mach. Learn. Res.* **2021**, *22*, 1–20. [CrossRef]
19. Vidal-Silva, C.; Duarte, V.; Cardenas-Cobo, J.; Serrano-Malebran, J.; Veas, I.; Rubio-León, J. Reviewing Automated Analysis of Feature Model Solutions for the Product Configuration. *Appl. Sci.* **2023**, *13*, 174. [CrossRef]
20. Benavides, D.; Segura, S.; Trinidad, P.; Cortés, A.R. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In Proceedings of the First International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2007), Limerick, Ireland, 16–18 January 2007; Pohl, K., Heymans, P., Kang, K.C., Metzger, A., Eds.; Lero Technical Report; Volume 2007-01, pp. 129–134. .
21. Benavides, D.; Felfernig, A.; Galindo, J.A.; Reinfrank, F. Automated Analysis in Feature Modelling and Product Configuration. In Proceedings of the Safe and Secure Software Reuse—ICSR 2013, Pisa, Italy, 18–20 June 2013; Favaro, J., Morisio, M., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7925, pp. 160–175. [CrossRef]
22. Stodden, V. Enhancing reproducibility in computational methods. *Nat. Comput. Sci.* **2022**, *2*, 76–78. [CrossRef] [PubMed]
23. Barbosa, H.; Barrett, C.; Brain, M.; Kremer, G.; Lachnitt, H.; Mann, M.; Mohamed, A.; Mohamed, M.; Niemetz, A.; Nötzli, A.; et al. cvc5: A Versatile and Industrial-Strength SMT Solver. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems—TACAS 2022, Munich, Germany, 2–7 April 2022; Fisman, D., Rosu, G., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2022; Volume 13243, pp. 415–442. [CrossRef]
24. Penco, R.; Pintar, D.; Vranić, M.; Šoštarić, M. Large Language Model-Driven Framework for Automated Constraint Model Generation in Configuration Problems. *Appl. Sci.* **2025**, *15*, 6518. [CrossRef]
25. Usman, M.; Wang, W.; Vasic, M.; Wang, K.; Vikalo, H.; Khurshid, S. A Study of the Learnability of Relational Properties: Model Counting Meets Machine Learning (MCML). In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020), London, UK, 15–20 June 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1098–1111. [CrossRef]
26. Xu, Y.; Liu, Z.; Tang, J. Hybrid Symbolic-Neural Reasoning for Large-Scale Constraint Solving. *J. Artif. Intell. Res.* **2023**, *76*, 255–282. [CrossRef]