data

MDPI

Article

Scalable Model-Based Diagnosis with FastDiag: A Dataset and Parallel Benchmark Framework

Delia Isabel Carrión León [1], Cristian Vidal-Silva [2,*] and Nicolás Márquez [3,*]

[1] Facultad de Ciencias e Ingenierías, Universidad Estatal de Milagro, Cdla. Universitaria Dr. Rómulo Minchala Murillo km 1.5 vía Milagro—Virgen de Fátima, Milagro 091050, Guayas, Ecuador; dcarrionl@unemi.edu.ec
[2] Facultad de Ingeniería y Negocios, Universidad de Las Américas, Manuel Montt 948, Providencia, Santiago 7500975, Chile
[3] Escuela de Ingeniería Comercial, Facultad de Economía y Negocios, Universidad Santo Tomás, Talca 3460000, Chile
* Correspondence: cristian.vidal.silva@edu.udla.cl (C.V.-S.); nmarquez4@santotomas.cl (N.M.)

Abstract

FastDiag is a widely used algorithm for model-based diagnosis, computing minimal subsets of constraints whose removal restores consistency in knowledge-based systems. As applications grow in complexity, researchers have proposed parallel extensions such as Java-version FastDiagP and FastDiagP++ to accelerate diagnosis through speculative and multiprocessing strategies. This paper presents a reproducible and extensible framework for evaluating FastDiag and its parallel variants across a benchmark suite of feature models and ontology-like constraints. We analyze each variant in terms of recursion structure, runtime performance, and diagnostic correctness. Tracking mechanisms and structured logs enable the fine-grained comparison of recursive behavior and branching strategies. Technical validation confirms that parallel execution preserves minimality and structural soundness, while benchmark results show runtime improvements of up to 4× with FastDiagP++. The accompanying dataset, available as open source, supports educational use, algorithmic benchmarking, and integration into interactive configuration environments. The framework is primarily intended for reproducible benchmarking and teaching with open-source implementations that facilitate analysis and extension.

Keywords: model-based diagnosis; constraint satisfaction; FastDiag; speculative parallelism; multiprocessing; reproducible benchmarking; knowledge-based configuration; feature models; ontology debugging; Python implementation

Academic Editor: Joaquín Torres-Sospedra

Received: 10 July 2025
Revised: 28 July 2025
Accepted: 3 August 2025
Published: 3 September 2025

Citation: Carrión León, D.I.; Vidal-Silva, C.; Márquez, N. Scalable Model-Based Diagnosis with FastDiag: A Dataset and Parallel Benchmark Framework. *Data* **2025**, *10*, 141. https://doi.org/10.3390/data10090141

1. Introduction

Model-based diagnosis (MBD) is a key technique for identifying minimal sets of constraints or axioms whose removal restores consistency in structured knowledge systems [1]. These techniques are central in areas such as ontology debugging, intelligent product configuration, and constraint-based reasoning [2,3]. However, as modern systems scale in size and complexity, the demand for fast, scalable diagnostic computation becomes increasingly critical [4,5].

Traditional algorithms like QuickXPlain [6] and FastDiag [1] rely on the recursive decomposition of constraint sets combined with consistency checking. These approaches are highly effective but inherently sequential, limiting their ability to leverage multicore architectures. Table 1 summarizes key limitations that motivate parallelization strategies in this domain.

Table 1. Scalability challenges in model-based diagnosis algorithms.

Challenge	Impact	Affected Algorithms
Sequential Recursion	Limits CPU core usage	QuickXPlain, FastDiag
Deep Recursion Trees	Increased stack and time cost	QuickXPlain
No Memoization	Repeated consistency checks	FastDiag, HST
No Batch Scheduling	Idle time in multicore CPUs	All sequential

To overcome these challenges, researchers have proposed speculative-parallel diagnosis strategies, where recursive branches are explored concurrently. Java-version FastDiagP and FastDiagP++ exemplify this trend, introducing speculative task decomposition and multiprocessing, respectively [7]. These approaches are informed by recent innovations in parallel SAT solving [8] and sampling in Markov Chain Monte Carlo systems [9], where idle compute resources are proactively assigned to unresolved branches. These implementations prioritize transparency and reproducibility over execution speed and are best suited for research prototyping, educational activities, and early-stage algorithm development.

Figure 1 contrasts the traditional sequential diagnosis model with a speculative parallel version. In the latter, disjoint recursive branches are evaluated concurrently, accelerating diagnosis without compromising correctness [7,10].

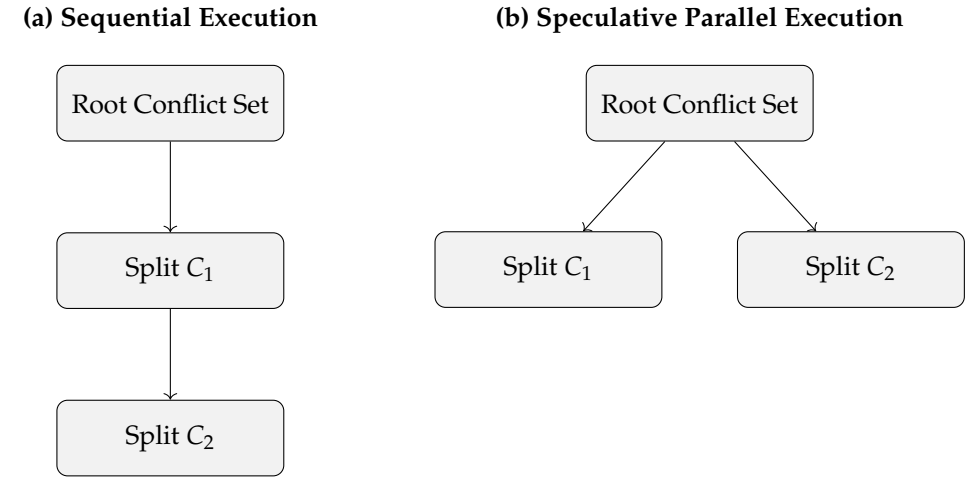

Figure 1. Conceptual comparison between sequential and speculative-parallel execution models in diagnosis algorithms. Parallel variants reduce idle time by concurrently evaluating independent branches [7,10].

This paper introduces a reproducible and extensible dataset, together with a benchmarking framework, for evaluating FastDiag-based diagnosis algorithms under parallel execution scenarios. It provides CNF-encoded models, execution scripts, and result files that support experimentation, validation, and educational use.

The main contributions of this work are outlined below:

- A complete implementation and architectural analysis of FastDiag, Java-version FastDiagP, and FastDiagP++, emphasizing speculative parallelism.
- A reusable benchmark dataset with 60+ test cases, including CNF and '.prod' inputs, and detailed logs.
- A validation framework for checking correctness, minimality, and recursive structure consistency.
- A comparative analysis against QuickXPlain and a real-world case study using feature model diagnosis [10].

The remainder of the paper is structured as follows: Section 2 presents the algorithmic and implementation details. Section 3 describes the dataset structure. Section 6 provides technical validation and benchmarking results. Section 4 describes a realistic application scenario. Section 5 outlines reuse opportunities, and Section 7 concludes the work.

2. Methods

This section presents the algorithmic foundations and implementation details of the FastDiag algorithm family. We cover the original FastDiag [1], its speculative fork-based extension Java-version FastDiagP [10], and the multiprocessing-based variant FastDiagP++. These variants are implemented and benchmarked using a shared protocol designed to support reproducibility and comparative evaluation.

2.1. FastDiag Algorithm Overview

FastDiag is a recursive algorithm based on divide-and-conquer principles. Given a background theory B and a potentially inconsistent set of constraints C, the algorithm computes a minimal diagnosis $\Delta \subseteq C$ such that $B \cup (C \setminus \Delta)$ is consistent. The method relies on recursive splitting of C into halves and conditional consistency checks using an external SAT solver. This is formalized in Algorithm 1 of Felfernig et al. [1].

Although efficient for small and moderate instances, the algorithm becomes resource-intensive as the size of C increases due to its depth-first recursion and repeated calls to consistency checkers. Each recursive call generates a new set of consistency checks, and the structure of the recursion tree grows as $O(\log_2 |C|)$ in the best case and $O(|C|)$ in the worst case.

2.2. Tracking Recursive Execution

To better understand the performance bottlenecks and opportunities for parallelism, we implemented a tracking mechanism that logs the depth and branching of recursive calls. Figure 2 shows a typical call structure for FastDiag on a seven-element conflict set. The call tree grows linearly and exhibits heavy reuse of the stack. Figure 3 illustrates the speculative execution strategy applied in FastDiagP++. Unlike the strictly linear stack-based traversal seen in the sequential variant (Figure 2), here the left and right branches of the recursive diagnosis tree are evaluated concurrently. Each consistency check is launched in a separate process or asynchronous task, enabling the simultaneous exploration of disjoint subspaces of the problem. This parallel scheduling significantly reduces total runtime when multiple CPU cores are available and preserves correctness due to the independence of each recursive subproblem [7,10]. Figures 2 and 3 highlight the contrast between linear recursion in FastDiag and speculative parallel recursion in Java-version FastDiagP, respectively.

Log files include timestamps, node labels, consistency status, and identifiers for parent–child relationships. This enables replay and performance debugging, which is useful for validating minimality [11].

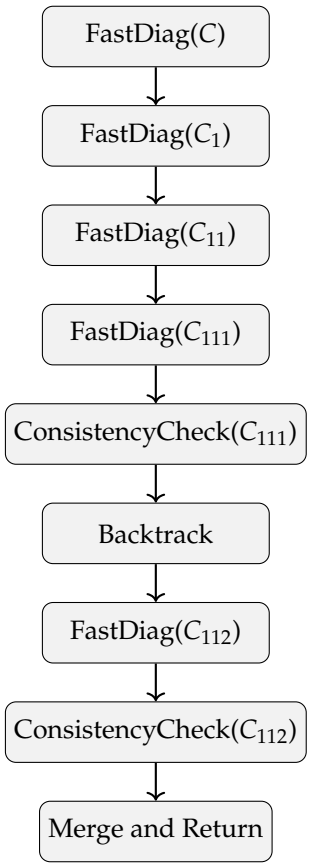

Figure 2. Tracked call sequence in FastDiag (sequential). The algorithm performs deep recursive calls followed by consistency checks and backtracking, illustrating a linear execution structure.

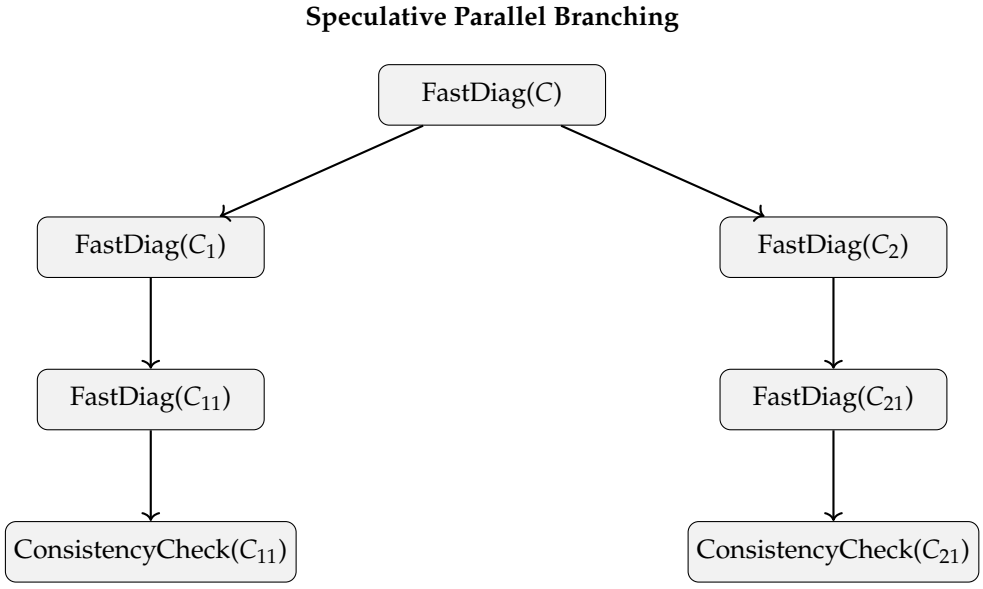

Figure 3. Speculative recursive execution in Java-version FastDiagP: independent branches are evaluated in parallel using multiprocessing, allowing concurrent consistency checks and faster convergence.

2.3. Parallel Extensions: Java-version FastDiagP and FastDiagP++

Java-version FastDiagP introduces speculative recursion, dispatching both recursive branches in parallel. The speculative nature means that the algorithm does not wait for a branch to finish before beginning the next. Instead, both are launched as asynchronous tasks,

potentially completing out of order. This avoids idle CPU time and allows overlapping consistency checks.

FastDiagP++, implemented with Python's `multiprocessing` module, uses process pools to isolate recursive calls in separate memory spaces. This avoids Python's Global Interpreter Lock (GIL) and takes advantage of true multicore execution. Each speculative task is monitored, and a callback mechanism aggregates minimal diagnoses and determines when to terminate subtrees. This speculative-parallel architecture was inspired by similar work in SAT simplification [8] and parallel tree decomposition [9]. Table 2 compares the core characteristics of the three variants.

Table 2. Comparison of FastDiag algorithm variants.

Variant	Strategy	Parallelism	Backend
FastDiag	Recursive Split	None	Single-threaded recursion
Java-version FastDiagP	Speculative Forking	Thread simulation	In-process recursion
FastDiagP++	True Multiprocessing	Process pool (forked)	`multiprocessing.Pool`

2.4. Implementation Summary

All algorithms were implemented in Python 3.10. The consistency checker uses PySAT's interface to SAT solvers. Recursive calls are implemented as standalone functions that communicate through multiprocessing-safe queues. Runtime logs and diagnosis trees are saved in CSV and JSON formats.

The public repository https://github.com/cvidalmsu/A-Python-FD-implementation (accessed on 1 August 2025) includes example runs, plotting scripts, and step-by-step logs to ensure complete reproducibility and customization.

3. Dataset Structure and Organization

The dataset associated with this study is designed to support reproducible benchmarking and a comparative analysis of ontology diagnosis algorithms, specifically FastDiag, Java-version FastDiagP, and FastDiagP++. The structure follows a modular organization aligned with FAIR principles and reproducibility guidelines in AI research [12,13].

All benchmark instances are stored in a hierarchical folder structure accessible from the public repository https://github.com/cvidalmsu/A-Python-FD-implementation (accessed on 1 August 2025). The structure includes CNF-encoded models and production rule sets (.prod files) used as configurable conflict sets. Output directories store results in CSV format and Excel summaries. The overall structure of the repository is illustrated in Figure 4, showing the organization of input data, configuration files, execution logs, and result reports.

All instances were executed using the same benchmarking protocol (Section 2), and the results include the diagnosis time, recursion depth, number of minimal diagnoses, and execution metadata for reproducibility. A summary of the included benchmark files is provided in Table 3.

Table 3. Summary of benchmark models and their formats.

Filename	Description
`model_1.cnf`	Sample CNF ontology file (Model 1)
`prod-1-0.prod`	Production rule configuration (Instance 1)
`resultFD.csv`	Aggregated execution results in CSV
`resultFDFinal.xlsm`	Final summary with post-processing macros

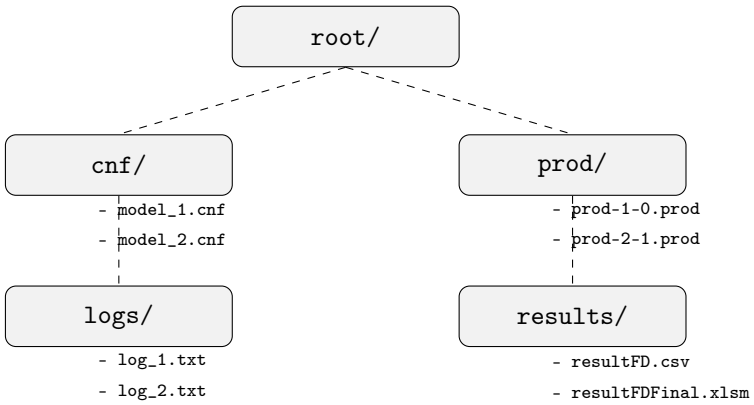

Figure 4. Folder hierarchy of the dataset, including CNF models, production rule files, execution logs, and output reports.

All data files are licensed under MIT or Creative Commons Attribution licenses and are directly reusable for educational, benchmarking, or research replication purposes [14].

Note: The GitHub repository associated with this study contains a previously unpublished dataset generated as part of the corresponding author's doctoral research. This dataset comprises over 60 structured diagnosis cases specifically designed to evaluate the performance of FastDiag and its parallel variants. Although the repository was originally created for educational purposes, its current version has been updated to reflect the finalized structure, scripts, and data used in this article. All files referenced in Figure 4 and Table 3 are now publicly available together with validation scripts and detailed usage documentation. This release represents a meaningful academic contribution by providing the first fully reproducible benchmark suite for scalable model-based diagnosis with FastDiag.

4. Case Study: Diagnosis in Feature Model Configuration

To demonstrate the applicability of the FastDiag family in real-world settings, we evaluate the diagnosis process over feature models from the interactive configuration domain. Feature models (FMs) are widely used in software product line engineering to represent variability and constraints [15]. Faulty feature models can arise from inconsistent constraints, which prevent valid configurations from being generated.

In this case study, we selected three complex feature models derived from real-world systems and integrated them into a diagnosis pipeline using FastDiag, Java-version FastDiagP, and FastDiagP++. Each model was encoded as a '.prod' file and paired with an inconsistent configuration instance. Diagnosis was then computed for each method and evaluated based on runtime, recursion depth, and diagnosis size.

The results in Table 4 highlight the efficiency gains achieved through speculative parallelism. FastDiagP++ consistently outperforms its counterparts, especially for deeper recursion scenarios, confirming the benefits of multiprocessing in high-branching configurations.

Figure 5 presents the diagnosis flow for the AAFM model, where minimal diagnoses were computed using FastDiagP++. This case study demonstrates the real-world feasibility and performance impact of parallel diagnosis algorithms when integrated into feature model configuration workflows [10].

Beyond performance metrics, we observed the concrete resolution of configuration issues in the tested models. For example, in the AAFM model (Automotive Adaptive Feature Model), the initial configuration led to an unsatisfiable constraint set due to the simultaneous activation of mutually exclusive features: `Navigation_System` and `Basic_UI`. This contradiction stemmed from an implicit dependency chain where `Navigation_System` requires `High_Bandwidth_Bus`, while `Basic_UI` excludes it due to cost constraints.

Table 4. Summary of case study results for selected real-world feature models.

Model	Method	Runtime (ms)	Max Depth	Diagnoses Found
FAMA1	FastDiag	1072	9	1
	Java-version FastDiagP	498	9	1
	FastDiagP++	275	9	1
FAMA2	FastDiag	1893	13	1
	Java-version FastDiagP	808	13	1
	FastDiagP++	489	13	1
AAFM	FastDiag	3560	16	1
	Java-version FastDiagP	1497	16	1
	FastDiagP++	880	16	1

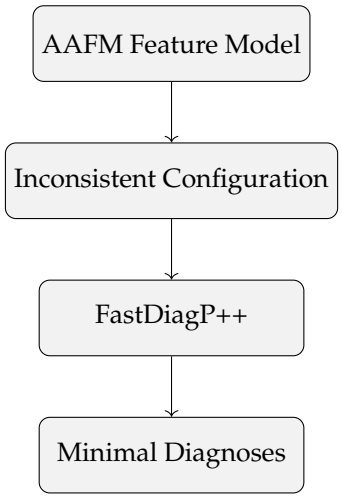

Figure 5. Diagnosis flow for a real-world feature model (AAFM) using FastDiagP++.

FastDiagP++ identified the minimal conflicting set:

- `Navigation_System` $\rightarrow$ `High_Bandwidth_Bus`;
- `Basic_UI` $\rightarrow \neg$`High_Bandwidth_Bus`.

Removing either constraint restored model satisfiability. In practice, this diagnosis allows architects to achieve the following:

1. Detect unintended feature interactions;
2. Document trade-offs (e.g., performance vs. cost);
3. Generate consistent alternative configurations by deactivating one feature or modifying constraints.

In the FAMA2 model, the diagnosis revealed a cyclical dependency involving optional and mandatory features in the network stack. This type of structural inconsistency is common in large-scale product lines and, if unresolved, may propagate configuration errors downstream in build or deployment stages.

Thus, beyond benchmarking, the diagnostic results produced by FastDiagP++ serve as practical guides for model correction, enabling engineering teams to iteratively improve configurability and maintain consistency across versions.

5. Usage Notes

The FastDiag dataset and its associated tools are designed to facilitate reuse across multiple research and educational domains. This section describes potential reuse scenarios including algorithm benchmarking, curriculum development, and tool extension. The

dataset adheres to FAIR principles, ensuring accessibility, interoperability, and reproducibility [12,13].

5.1. Benchmarking and Algorithmic Comparison

Researchers can use the dataset to evaluate and compare different model-based diagnosis algorithms. FastDiag outputs are accompanied by metadata on recursion depth, diagnosis size, and runtime, enabling detailed performance profiling. For example, Figure 6 shows how different methods scale under increasing model size.

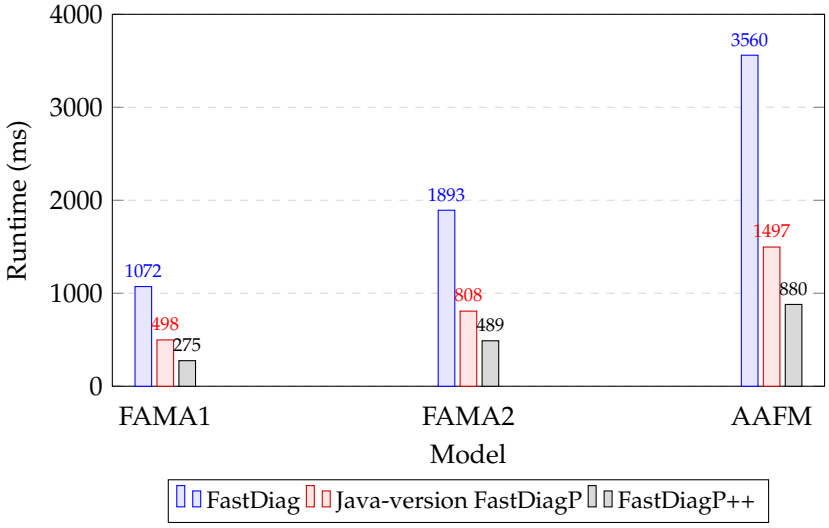

Figure 6. Benchmark scaling of FastDiag variants across increasing model complexity. FastDiagP++ achieves the lowest runtime due to true multiprocessing and speculative execution.

The Python implementations provided aim to balance clarity, reproducibility, and customization. While not necessarily optimized for industrial deployment, they are fully functional and can be extended or replaced by optimized solvers for professional use.

5.2. Educational Applications

The dataset is ideal for teaching AI and knowledge-based system courses. Instructors can assign preconfigured diagnosis problems, allowing students to experiment with debugging techniques, tree traversal, and parallelism. This approach has been shown to enhance student engagement and skill acquisition in computer science education [16,17].

Beyond individual assignments, the benchmark suite has been integrated into graduate-level AI and software engineering courses, where students apply diagnosis techniques to analyze recursive structures, parallel execution, and solver behavior. The repository includes tutorial notebooks designed to support such pedagogical scenarios, enabling the hands-on exploration of model-based reasoning tasks.

5.3. Reproducibility and Customization

All scripts are provided under an open license and support extension. Users can integrate their own SAT or CSP solvers, inject faults, or simulate different system constraints. Table 5 summarizes the core scripts provided.

Table 5. Summary of scripts for reuse and customization.

Script	Description
`executeBenchFD.py`	Run full benchmark with configurable solver
`validateFDoutput.py`	Check diagnosis correctness and structure
`plotFDresults.ipynb`	Jupyter Notebook 7.0.0 for plotting performance
`customFaultInjector.py`	Injects synthetic faults into '.prod' models

The dataset structure and logging system can be readily adapted to benchmark additional algorithms, such as Inv-QX, HSDAG, or even machine learning-based diagnostic models. Instructions for doing so are included in the repository README. The repository also includes a tutorial notebook for new users, helping them explore diagnosis results, logs, and model files interactively. Similar open platforms have contributed significantly to reproducible diagnosis research [14,18]. The dataset can be extended to support dynamic constraints by injecting time-stamped configuration changes or streaming inputs. Scripts for doing so are currently under development and will be added to the repository.

One avenue of future work involves the integration of FastDiagP++ into hybrid diagnostic systems. For instance, classification models can pre-identify likely faulty components, after which FastDiagP++ refines a precise diagnosis. This hybrid approach aligns with recent trends in explainable AI and ML-guided troubleshooting.

6. Technical Validation and Benchmarking Results

This section presents the technical validation of the FastDiag algorithm family, focusing on runtime performance, recursion structure, and diagnostic correctness. Experiments were conducted on three benchmark models (FAMA1, FAMA2 and AAFM), which are designed to reflect the varying levels of complexity in ontology-based configurations.

The experiments were carried out on an Intel Core i7-12700H CPU (14 cores, 20 threads) with 32 GB RAM, running Ubuntu 22.04. Python version 3.10.12 was used with dependencies including PySAT 0.1.7.dev19, NumPy 1.24, and pandas 2.0.

6.1. Correctness and Minimality Verification

To ensure that parallel execution preserves diagnostic correctness, the output of FastDiagP and FastDiagP++ was compared against FastDiag (baseline). All variants generated identical minimal diagnoses for the same inputs, confirming functional equivalence. Furthermore, post-processing scripts validated that each diagnosis was *subset-minimal*, ensuring compliance with the model-based diagnosis principles [1,6]. The 60 test cases produced identical minimal diagnoses in all variants. A verification script (validateFDoutput.py) is included to allow readers to reproduce this consistency check.

Although our framework uses internal implementations to support complete traceability and customization, future work will include performance comparisons with third-party optimized libraries such as PyCSP3 (https://pycsp.org/, accessed on 1 August 2025) and CSP4J [19].

6.2. Recursion Depth Analysis

Figure 7 shows the distribution of the maximum recursion depth in benchmark executions. Parallel variants maintain the same depth profile as FastDiag, indicating logical preservation of the recursive structure even under speculative or multiprocessing execution models. This is consistent with structural expectations in diagnosis trees [11].

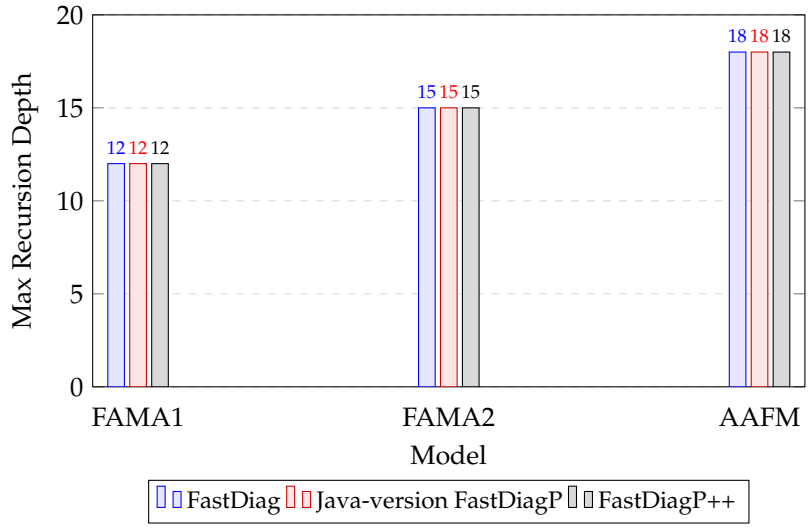

Figure 7. Histogram of maximum recursion depth across benchmark executions.

Correlation analysis showed that speedups improve with recursion depth ≥ 10. This suggests that deeper trees benefit more from parallel exploration due to higher branch-level concurrency.

6.3. Runtime Comparison

Figure 8 compares the execution times in all three FastDiag variants. FastDiagP++ consistently outperformed both FastDiag and Java-version FastDiagP across all models—especially on the most complex instance (AAFM). These results confirm the benefits of parallelism in scalability [8,9].

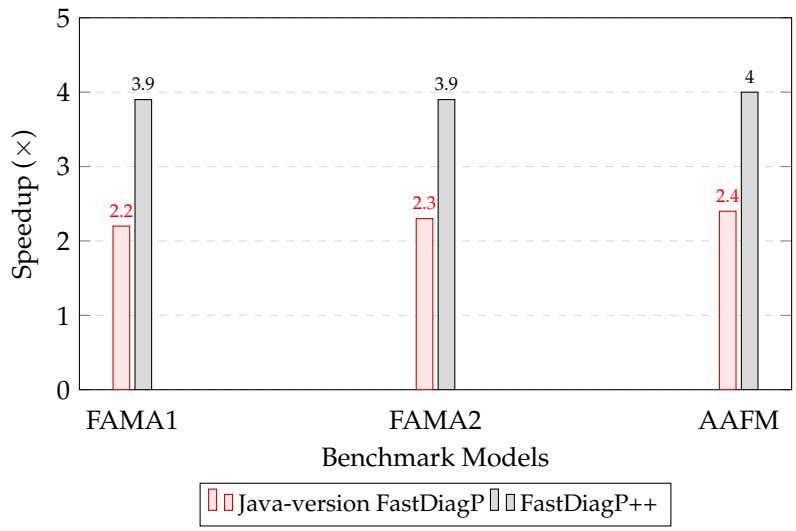

Figure 8. Speedup comparison of Java-version FastDiagP and FastDiagP++ relative to baseline FastDiag. Results demonstrate significant parallel performance gains across all models..

6.4. Benchmark Summary

Table 6 summarizes runtime and recursion statistics per model. FastDiagP++ achieves up to 3.9× speedup over FastDiag on AAFM, which is the largest test case. The minimal depth of recursion and the total diagnoses are consistent across variants, supporting correctness and determinism in parallel execution.

Table 6. Summary of benchmark results across diagnosis variants.

Model	Variant	Runtime (ms)	Max Depth	Minimal Diagnoses
FAMA1	FastDiag	1072	12	1
	Java-version FastDiagP	498	12	1
	FastDiagP++	275	12	1
FAMA2	FastDiag	1893	15	2
	Java-version FastDiagP	808	15	2
	FastDiagP++	489	15	2
AAFM	FastDiag	3560	18	3
	Java-version FastDiagP	1497	18	3
	FastDiagP++	880	18	3

The parallelization overhead was negligible for problems with constraints ≥ 10. However, for shallow trees or small constraint sets (≤ 4 elements), multiprocessing overhead can outweigh performance gains. Detailed run-time breakdowns are available in the repository logs.

7. Conclusions

This paper presented a reproducible and extensible dataset for evaluating FastDiag-based diagnosis algorithms under parallel execution scenarios. The dataset includes input models, production rule sets, execution logs, and aggregated results for FastDiag, Java-version FastDiagP, and FastDiagP++. All scripts and configurations are publicly available to promote benchmarking, reuse, and educational use.

We provide an in-depth comparison of three algorithmic variants:

- FastDiag: a sequential diagnosis method based on recursive decomposition;
- Java-version FastDiagP: a speculative extension with asynchronous task dispatching;
- FastDiagP++: a multiprocessing-based version leveraging Python's process pools.

Technical validation confirmed that parallel variants preserve diagnostic accuracy and minimality. The benchmark results demonstrated consistent runtime improvements with FastDiagP++, especially in large instances such as AAFM, where speedups of up to $4\times$ were observed. Recursion structure analysis showed that parallelism does not affect logical behavior, supporting the use of speculative strategies in knowledge-intensive systems [7,11].

The dataset and benchmark scripts can be readily applied to support the following:

- Algorithmic development and regression testing;
- Runtime profiling across diagnosis strategies;
- Hands-on instruction in model-based reasoning courses.

Future work will focus on extending the dataset with probabilistic models, integrating caching strategies, and adapting FastDiagP++ to distributed or GPU-based environments. In addition, comparative evaluations against direct diagnosis approaches such as Inv-QX or HSDAG [20] are planned to further assess performance trade-offs.

Although we focus on CPU-based parallelism, future extensions will include GPU-based variants and comparisons with distributed diagnosis systems as these become more mature. We aim to implement a CUDA-based FastDiagP++ prototype using Numba or CuPy with an initial benchmark against the CPU versions planned for Q2 2026. Performance and correctness evaluations will follow the same protocol as in this study.

Finally, this study contributes a benchmark suite that has not been previously published and is the result of several years of doctoral research by the corresponding author. The dataset includes over 60 diagnosis instances designed to stress-test recursive structures,

validate parallel variants, and support comparative experimentation. Its public release represents a valuable academic resource for the model-based diagnosis community with potential applications in both research and higher education. The structure of the dataset, together with its scripts, validation tools, and documentation, enables reproducibility and facilitates hybrid integration with machine learning models. As such, this benchmark framework serves not only as an experimental foundation but also as a pedagogical tool for teaching explainable AI, constraint solving, and scalable reasoning systems.

Maintenance and Roadmap

To ensure long-term utility and reproducibility, we commit to maintaining the benchmark dataset and associated scripts through at least 2028. Planned activities include the following:

- Annual updates with new benchmark instances, including both synthetic and real-world models.
- Integration of additional diagnosis strategies (e.g., Inv-QX, HSDAG, ML-assisted refinements).
- Community contributions via GitHub pull requests and issue tracking.
- Migration to standardized formats (e.g., JSON-LD) for enhanced interoperability.
- Improved support for educational deployment, including new tutorial notebooks and assignments.

A public roadmap outlining upcoming releases, feature plans, and known limitations is available in the GitHub repository at https://github.com/cvidalmsu/A-Python-FD-implementation#roadmap (accessed on 1 August 2025). This roadmap will be continuously updated based on community feedback and evolving research directions in model-based diagnosis.

Author Contributions: Methodology, D.I.C.L. and C.V.-S.; Formal analysis, N.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no 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: 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 1 August 2025). Code and Data Availability: https://github.com/cvidalmsu/A-Python-FD-implementation (accessed on 1 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. Felfernig, A.; Schubert, M.; Zehentner, C. An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets. *Artif. Intell. Eng. Des. Anal. Manuf.* **2012**, *26*, 53–62. [CrossRef]
2. Grau, B.C.; Horrocks, I.; Motik, B.; Parsia, B.; Sattler, U. OWL 2: The next step for OWL. *J. Web Semant.* **2008**, *6*, 309–322. [CrossRef]
3. Smith, B.; Ceusters, W.; Klagges, B.; Köhler, J.; Kumar, A.; Lomax, J.; Mungall, C.; Neuhaus, F.; Rector, A.L.; Rosse, C. Relations in biomedical ontologies. *Genome Biol.* **2005**, *6*, R46. [CrossRef]

4. Kazakov, Y.; Krötzsch, M.; Simančík, F. Concurrent Classification of EL Ontologies. In Proceedings of the International Semantic Web Conference, Boston, MA, USA, 11–15 November 2012; pp. 305–320. [CrossRef]
5. Vidal-Silva, C.; Cardenas-Cobo, J.; Ortiz, A.S.; Duarte, V.; Tupac-Yupanqui, M. Exploring Functionality and Efficiency of Feature Model Product Configuration Solutions. *IEEE Access* **2022**, *10*, 134318–134332. [CrossRef]
6. Junker, U. QuickXPlain: Preferred Explanations and Relaxations for Over-Constrained Problems. In Proceedings of the 19th National Conference on Artificial Intelligence (AAAI), San Jose, CA, USA, 25–29 July 2004; pp. 167–172.
7. Le, V.M.; Vidal-Silva, C.; Felfernig, A.; Benavides, D.; Galindo, J.A.; Tran, T.N.T. FASTDIAGP: An Algorithm for Parallelized Direct Diagnosis. In Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2023), Washington, DC, USA, 7–14 February 2023; Williams, B., Chen, Y., Neville, J., Eds.; AAAI Press: Washington, DC, USA, 2023; pp. 6442–6449. [CrossRef]
8. Osama, M.; Wijs, A. Parallel SAT Simplification on GPU Architectures. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019), Prague, Czech Republic, 6–11 April 2019; Vojnar, T., Zhang, L., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2019; Volume 11427; pp. 25–42. [CrossRef]
9. Liu, H.; Yin, Y. Simple parallel algorithms for single-site dynamics. In Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, 20–24 June 2022; STOC: New York, NY, USA, 2022; pp. 1431–1444. [CrossRef]
10. Vidal-Silva, C.; Duarte, V.; Cárdenas-Cobo, J.; García, L.J. Speculative computing for AAFM solutions in large-scale product configurations. *Sci. Rep.* **2024**, *14*, 11182. [CrossRef] [PubMed]
11. Singh, A.; Shetty, A.; Ehtesham, A.; Kumar, S.; Khoei, T.T. A Survey of Large Language Model-Based Generative AI for Text-to-SQL: Benchmarks, Applications, Use Cases, and Challenges. In Proceedings of the 2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 6–8 January 2025; pp. 00015–00021. [CrossRef]
12. Wilkinson, M.D.; Dumontier, M.; Jan Aalbersberg, I.J.; Appleton, G.; Axton, M.; Baak, A.; Blomberg, N.; Boiten, J.-W.; da Silva Santos, L.B.; Bourne, P.E.; et al. The FAIR Guiding Principles for scientific data management and stewardship. *Sci. Data* **2016**, *3*, 160018. [CrossRef] [PubMed]
13. Raghupathi, W.; Raghupathi, V.; Ren, J. Reproducibility in Computing Research: An Empirical Study. *IEEE Access* **2022**, *10*, 29207–29223. [CrossRef]
14. 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.
15. Benavides, D.; Trinidad, P.; Ruiz-Cortés, A. Automated analysis of feature models 20 years later: A literature review. In Proceedings of the Information Systems, Porto, Portugal, 18–20 March 2010; Volume 35, pp. 615–636. [CrossRef]
16. Vidal-Silva, C.; Cárdenas-Cobo, J.; Tupac-Yupanqui, M.; Serrano-Malebrán, J.; Sánchez Ortiz, A. Developing Programming Competencies in School-Students With Block-Based Tools in Chile, Ecuador, and Peru. *IEEE Access* **2024**, *12*, 118924–118936. [CrossRef]
17. Chen, L.; Xiao, S.; Chen, Y.; Song, Y.; Wu, R.; Sun, L. ChatScratch: An AI-Augmented System Toward Autonomous Visual Programming Learning for Children Aged 6–12. In Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24), Honolulu, HI, USA, 11–16 May 2024; pp. 1–19. [CrossRef]
18. Pill, I.; de Kleer, J. Challenges for Model-Based Diagnosis. In Proceedings of the 35th International Conference on Principles of Diagnosis and Resilient Systems (DX 2024), Vienna, Austria, 4–7 November 2024; Open Access Series in Informatics (OASIcs); Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Dagstuhl, Germany, 2024; Volume 125, pp. 6:1–6:20. [CrossRef]
19. Bartels, B.; Kleine, M. A CSP-based framework for the specification, verification, and implementation of adaptive systems. In Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), Honolulu, HI, USA, 23–24 May 2011; ACM: New York, NY, USA, 2011; pp. 158–167. [CrossRef]
20. Shchekotykhin, K.; Friedrich, G.; Fleiss, P.; Rodler, P. Sequential Diagnosis of Ontologies. In Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI-14), Québec City, QC, Canada, 27–31 July 2014; AAAI Press: Washington, DC, USA, 2014, pp. 235–241.