D-Painless: A Framework for Distributed Portfolio SAT Solving

Mazigh Saoudi[1 [0009−0004−6074−9686]], Souheib Baarir[2,3], Julien Sopena[2], and Thibault Lejemble[1 [0000−0001−5902−6094]]

[1] EPITA Research Laboratory (LRE), France mazigh.saoudi@epita.fr
[2] Computer Science Laboratory of Sorbonne University (LIP6), France
[3] Université Paris Nanterre, France

Abstract. In the evolving landscape of SAT solving, leveraging parallel computation has become increasingly significant. The portfolio strategy, combined with clause sharing, has emerged as the leading approach for both local and distributed parallelization on CPUs. Frameworks such as Mallob exemplify the effectiveness of this strategy by providing a straightforward method to deploy portfolio parallel solvers across various computing environments. Similarly, the *Painless* framework specializes in local parallelization, offering diverse strategies for task sharing and parallel execution. This enables the adoption of complex hybrid local parallelization techniques, including portfolio, divide-and-conquer, and cube-and-conquer methods.
This paper presents *D-Painless*, a new extension of the *Painless* framework to include the distributed portfolio strategy and clause sharing. Our enhancement aims to broaden *Painless*'s functionality, enabling more effective and comprehensive distributed SAT solving methodologies.

Keywords: Parallel SAT solving, Distributed computing, Painless, Tool

1 Introduction

Currently, SAT formulas are essential for encoding complex problems across various fields such as circuit verification [25], cryptography [30], automated planning [35], and software verification [26]. The complexity of these encoded problems ranges from simple to extremely challenging. One effective method for solving hard instances is the use of parallelism.

According to the annual SAT competition [17], the Portfolio clause sharing parallelization strategy has been the gold standard for parallel SAT solving for several years. However, many SAT formulae remain too difficult to solver. As a result, exploiting distributed computing environments to reduce solving times has become a crucial research focus, inspiring the development of several software solutions and frameworks [36,5,15,23]. To the best of our knowledge, existing competitive solutions lack the flexibility in distributed sharing and solving strategies that the *Painless* framework provides for (local) multi-core environments [28].

In this work, we extend the *Painless* framework to accommodate distributed clause sharing SAT solvers. This extension, namely *D-Painless*, allows the incorporation of various clause-sharing strategies and node topologies, resulting in a versatile framework that eases distributed SAT solvers development.

D-Painless is designed as a simple, generic, and extensible platform for parallel and distributed SAT solvers. Its primary goal is to empower researchers to easily test and develop new sharing and parallelization strategies without needing to master the complexities of distributed programming for SAT solving. While it is not intended to outperform existing solvers, *D-Painless* serves as a research tool capable of replicating their performance, while providing a versatile environment for rapid prototyping and experimentation with innovative strategies.

This paper is organized as follows: Section 2 introduces preliminary concepts, followed by a review of related work in Section 3. Section 4 details the architecture of *Painless* and its new extension, *D-Painless*. In Section 5, we assess the new architecture with some implemented sharing strategies. Finally, we conclude with a discussion of potential future works in Section 6.

2 Sequential, Parallel and Distributed SAT solving

The Conflict-Driven Clause Learning (CDCL) [38] algorithm has emerged as the predominant method in SAT solving, consistently delivering superior performance across a wide spectrum of SAT challenges. This algorithm has benefited from various enhancements, encompassing sophisticated heuristics [33], advanced pre-processing and in-processing strategies [11], and efficient data structures [31].

Parallelizing CDCL poses significant challenges. State-of-the-art methods primarily involve running multiple instances of the algorithm concurrently. This is typically achieved by altering initial variable assignments [24,19] or by diversifying the heuristics and configurations within a portfolio of sequential solvers [22]. The portfolio approach when combined with clause-sharing mechanisms [22], has proven to be the most effective for parallelizing CDCL solvers. Here, clauses learned by one solver are shared with others, reducing redundant mistakes and utilizing collective knowledge to speed up the solving process. According to the recent work [37], clause sharing is the primary factor in the success of distributed SAT solving, particularly when scaling to hundreds of solvers. While solver diversification remains beneficial, its impact becomes secondary compared to the collaborative learning achieved through clause sharing.

In distributed computing, we handle multiple computational units (or nodes) connected through a network. Implementing a distributed SAT solver based on the portfolio approach requires strategic diversification of CDCL solvers across nodes, ensuring effective clause sharing, both within nodes (*intra-node*) and between nodes (*inter-node*). It is crucial to mitigate communication latency and address the challenges posed by the asynchronous nature of efficient distributed systems, such as distinguishing between a node's failure and its delayed response.

Challenges of Distributed SAT Solving: Scaling up a parallel SAT solver to distributed systems presents two main challenges:

– Technical limitations that increase the difficulty of certain aspects of development, such as deployment, termination detection and fault management.
– Adapting parallelization and clause sharing to efficiently utilize all available computing power.

Thus, developing distributed SAT solvers presents significant challenges that could be addressed through specialized tools abstracting away technical complexities. The key is creating a flexible, modular architecture that enables both simplicity and extensibility. This modularity allows users to easily locate and modify specific behaviors while containing feature logic within well-defined interfaces.

3 Related Works

SAT solving can be parallelized using the divide-and-conquer strategy [19]. One of the pioneering efforts in distributed SAT solving, `PSATO` [42] based on the sequential solver SATO [43], adopts this strategy with a master-slave model for communication and load-balancing. `GridSAT` [16] follows the same idea. In this solver, a node (referred to as a "client" in the original paper) alerts the master node to divide its search tree and assigns a portion to a new node when it encounters a problem that is too challenging or anticipates a memory overflow. This new node, initiated by the master, receives a subset of the problem directly from the initiating node, facilitating problem-solving in a distributed manner. *Paracooba* [23] advances this concept further by employing a Cube&Conquer [24] strategy for distributed SAT solving, along with an adaptable load-balancing mechanism. In Cube&Conquer, the problem is divided into thousands (or millions) of *simpler* sub-problems using look ahead [7, Chapter 5] techniques. Each sub-problem represents a portion of the original problem where some decisions have been pre-made, allowing CDCL solvers to focus on exploring specific parts of the search space efficiently.

At the same time, other solvers have explored the portfolio approach [22], a strategy also explored in other fields, such as in Software Verification [6]. For instance, the annual SAT competitions [17] highlight *Mallob* [37] as a forefront solution in modern distributed SAT solving. *Mallob* distinguishes itself through an adaptable scheduling mechanism. It can schedule different types of jobs, such as distributed portfolio instances of parallel SAT solvers. Thus, it is proficient in handling multiple SAT problems concurrently, adaptively redistributing resources based on the real-time difficulty assessment of each task. *Mallob* employs a distributed clause sharing solver architecture, wherein each SAT problem (job) is tackled by a coordinated group of nodes specifically allocated for it. The clause sharing strategy is to gather the globally best clauses across all the solvers of a given job.

Introduced at the 2023 SAT Competition [4], the *PRS-Distributed* framework [15] aims to simplify the construction of local parallel and distributed SAT solvers. It has demonstrated performance on par with the leading *Mallob* system. The framework organizes computational nodes into distinct groups, each configured with a specific solver setup. For clause sharing, the version presented at the competition utilizes a circular ring architecture per group. Additionally,

the framework can broadcast clauses from any given node to all others within its group using a tree-based structure.

The Parallel Instantiable SAT Solver, known as *Painless* [28], is a framework designed for the instantiation of parallel SAT solvers in multi-core environments. Over the years, *Painless* has successfully integrated a variety of sequential solvers, including *Kissat* [10], *CaDiCaL* [9], *Lingeling* [8], *MapleCOMSPS* [29], *GlucoseSyrup* [2], and *MiniSat* [18]. It incorporates the two principal multi-threaded parallel strategies: Portfolio [22] and Divide and Conquer [19]. The adaptability and utility of *Painless* are attributed to its straightforward architecture (see Figure 1). This has enabled the implementation and execution of diverse parallel solving strategies, making *Painless* a significant tool in parallel SAT solving.

As depicted in Figure 1, the original architecture of *Painless* meticulously outlines the interactions among the various components within a parallel SAT solver, prioritizing simplicity, modularity, and flexibility. The behaviours related to parallelization are encapsulated by a tree of *WorkingStrategy* implementations, with the *SequentialWorker* implementations at the leaves. These *SequentialWorker* instances interface with sequential solvers through the *SolverInterface*. This interface provides all necessary interactions for the *SequentialWorker* to control the solving process, while the *SharingStrategy* facilitates clause sharing among its producers and consumers. The *Sharer* represents the thread that executes the chosen *SharingStrategy*. For an in-depth explanation, please refer to [28].

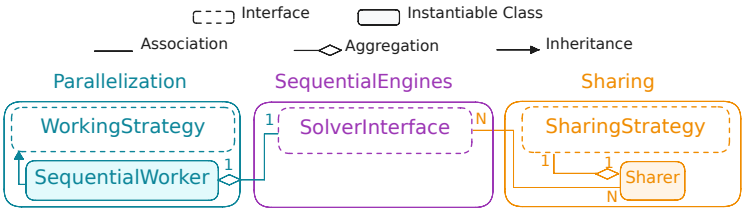


Figure 1: The architecture of *Painless*

The architecture of *Painless* showed its effectiveness in instantiating parallel SAT solvers via its results in the annual SAT Competitions [17]. The simplicity and extensibility it offers permitted the construction of multiple solvers such as those in works [39,40,41]. However, the current architecture lacks sufficient abstractions and flexibility to permit efficient and straightforward instantiation of distributed solvers with diverse clause-sharing strategies.

In the *Mallob*, *Painless* and *PRS-Distributed* solvers, the logic for clause sharing is split between several entities in these solvers. For instance, filtering clauses based on size or LBD [1] value was done within the sequential solvers interaction interfaces' export functions. To elevate this issue, we propose the new framework *D-Painless* that fixes the actual shortcomings of *Painless* and extend its functionalities to support distributed SAT solvers.

Algorithm 1 The `main` Function

```
1: function MAIN
2:     timeout, ... ← parseParameters()                    ▷ global variables
3:     finalResult ← UNKNOWN                               ▷ global variable
4:     finalModel ← ∅                                      ▷ global variable
5:     globalEnding ← false                                ▷ global variable
6:     condGlobalEnd ← initializeCondVariable()            ▷ global variable
7:     mpiRank, worldSize ← initializeMpi()                ▷ global variables
8:     workingStrat ← new workingStrat()
9:     workingStrat.solve()                                ▷ launches new threads
10:    wakeUpState ← sleep(condGlobalEnd, timeout)
11:    if wakeUpState = timedOut then
12:        finalResult ← TIMEOUT
13:    end if
14:    workingStrat.finalize()                             ▷ destructor
15:    finalizeMPI()
16:    if finalResult = SAT then
17:        print(finalModel)
18:    end if                                              ▷ If UNSAT it just returns the result
19:    return finalResult
20: end function
```

4 The Architecture of *D-Painless*

D-Painless is a new framework that enables the creation of distributed portfolios by instantiating and controlling sequential solvers across different nodes. This framework facilitates communication for clause exchange both via shared memory and message passing. In the following section, we will first discuss how the new software architecture mitigates the technical complexities of distributed systems. Then, we will explain how it simplifies the development of new sharing strategies.

4.1 Distributed Deployment In *D-Painless*

In order to manage the multiple nodes and establish communication between them, we use the Message Passing Interface (*MPI*). It proved its effectiveness in state-of-the-art distributed SAT solvers such as *Mallob* [37] and *PRS-Distributed* [32]. A big part of the deployment of the distributed environment is managed by *MPI*, it mainly requires a `hostfile` identifying the different machines to be used. After the deployment, each node gets a unique identifier, the *mpiRank*.

In *D-Painless*, we updated the *main* function of *Painless* to manage the *MPI* initialization and finalization. This function is the one responsible for launching the corresponding *WorkingStrategy*, waiting for its completion, and stopping it if the timeout is reached. In order to ensure flexibility, we also reduced the number of global variables, allowing the *WorkingStrategy* implementations to manage the different instantiated components. Multiple working strategies can exist in parallel, each owning and managing its *Sharers* working with its *SequentialWorkers*. This design simplifies resource management and enhances efficiency.

Algorithm 2 Termination detection in *D-Painless*

```
1:      ▷ The function uses globalEnding, finalResult, mpiRank and worldSize from main()
2: function DetectEnd
3:     rankWinner ← 0
4:     toSendResult ← UNKNOWN
5:     if mpiRank ≠ 0 then
6:         if globalEnding then
7:             send finalResult to root
8:         end if
9:     else                                          ▷ mpiRank = 0 i.e. it is the root
10:        if globalEnding then
11:            toSendResult ← finalResult
12:        else
13:            toSendResult ← checkReceivedResult()      ▷ UNKNOWN if no message
14:        end if
15:        send toSendResult to everyone
16:    end if
       return (toSendResult ≠ UNKNOWN)              ▷ returns true if it should end
17: end function
```

Because nodes may not have local access to the SAT problem file, we implemented a formula-sharing mechanism within the `mpiutils` namespace. To handle large problem instances that can reach several gigabytes, we compress the numerical data using Zlib [21]. This mechanism is optional; the *WorkingStrategy* implementation can choose to use or bypass these functions as needed.

Once the distributed solver finds a solution or a timeout is reached, all its nodes must be notified. And, in a distributed setting, termination is not that evident. Thus, to levitate this hurdle we have the new *main* function shown in Algorithm 1 and the termination detection algorithm described in Algorithm 2.

Algorithm 1 begins by parsing the arguments to define the desired solver to instantiate (line 2), followed by initializing essential variables for termination management (lines 3-6). The thread executing this *main* function sleeps with a timeout on the conditional variable *condGlobalEnd* (line 10) after launching a *WorkingStrategy*. When a running thread finds the answer (SAT or UNSAT), it sets *globalEnding* and notifies the *main* via *condGlobalEnd* and updates the variables *finalResult* and *finalModel*. In case timeout is reached, the *main* wakes up and sets *finalResult* accordingly. The *MPI* finalization (line 15) occurs after the *WorkingStrategy* concludes (line 14), ensuring no messages are left unprocessed. When *main* wakes up, it prints the model when required (lines 16-17) and exits by returning the *finalResult*'s value (SAT, UNSAT or TIMEOUT).

Algorithm 2 is a simple synchronization algorithm, that should be run periodically. The node with $mpiRank = 0$ acts as the master node and is notified by other nodes when they find a solution (line 7). This master node checks if it has found a solution (lines 10-11) and if a worker node sent a termination notification (line 13). Once the master knows whether a solution was found or not, it sends a message to all other nodes (line 15) indicating whether they should continue

or stop. If the function returns `true`, the component executing this algorithm must wake up the node's *main* function via (*condGlobalEnd*).

MPI offers a rich and high-level API for the development of distributed applications. However, developing a portfolio distributed SAT solver remains complex, as it involves dealing with communication synchronization, data bufferization, and multiple threads. Our framework's architecture hides this complexity by clearly separating the different needed behaviors in well-defined interfaces, guiding the developer in using network communication only where needed. The different aspects of the distributed solver can be changed independently. We present the interfaces of our framework in Section 4.2 with some implementation examples of sharing strategies in Section 5.1.

4.2 Clause Sharing in *D-Painless*

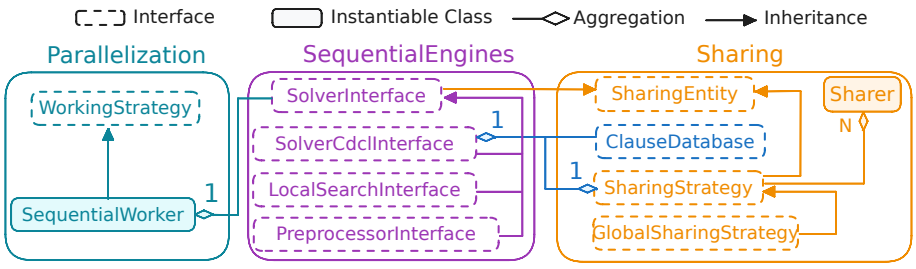


Figure 2: The new architecture of Painless

Clause sharing in distributed SAT solvers can be managed in two main ways: using a single sharing strategy for both inter-node (global) and intra-node (local) sharing, or employing separate strategies for each scope. We want our framework to support both approaches, but the current architecture of *Painless* (Figure 1) does not accommodate the latter, limiting its flexibility. For example, to enable collaboration between global and local strategies, the global strategy must act as both a producer and a consumer within the local strategy, requiring it to implement the *SolverInterface* interface from Figure 1. This highlights a deficiency in the current architecture's ability to support more generic sharing strategies.

To address these limitations and support more flexible sharing mechanisms, we propose the new architecture depicted in Figure 2. This revised structure establishes a coherent hierarchy and interaction schema for both locally parallel and distributed SAT solvers, enabling more robust clause sharing mechanisms. A key advantage of this new architecture is its ability to mix inter-node and intra-node strategies without the need to implement entirely new strategies that handle both aspects. This modular approach allows for greater flexibility and efficiency, as developers can combine existing strategies in various ways without having to implement all possible combinations.

The proposed architecture introduces three new interfaces for sharing: *ClauseDatabase*, *GlobalSharingStrategy*, and *SharingEntity*. Additionally, we have updated the *Sharer* worker and the *SharingStrategy* interface. Furthermore, the

SolverInterface has been made more abstract by adding specialized interfaces: *SolverCdclInterface*, *LocalSearchInterface*, and *PreprocessorInterface*.

SharingEntity	**SharingStrategy**
Public Interface	
+ *importClause()* + *importClauses()* + *addClient()* + *removeClient()* + getSharingId()	+ *doSharing()* + *getSleepingTime()* + *addProducer()* + *removeProducer()* + *connectProducer()*
Protected Interface	
# exportClause() # exportClauses() # *exportClauseToClient()*	# *exportClauseToClient()* (override)
Protected Members	
# m_clients: list<SharingEntity*>	# m_producers: list<SharingEntity*> # m_clauseDB: ClauseDatabase*

Table 1: Combined Abstract Interfaces for *SharingEntity* and *SharingStrategy*. (+) Public method, (#) Protected method or member, (*italics*) Virtual method, (override) redefinition of virtual methods.

In the new architecture, clause importing and exporting functionalities have been transferred from *SolverInterface* to the new *SharingEntity* interface, which is inherited by both *SharingStrategy* and *SolverCdclInterface*. The primary methods of *SharingEntity*, detailed in Table 1, offer flexibility for derived classes, easing the implementation of customized clause filtering and storage strategies.

The *SharingEntity* maintains a dynamic list of clients (subscribers) to which it exports clauses. This list can be safely updated at any time using the *addClient* and *removeClient* methods, ensuring thread-safe operations in concurrent environments. The export process is facilitated by two key methods: *exportClause* for individual clauses and *exportClauses* for bulk exports. Both methods safely access the current client list, preventing race conditions during clause export.

These methods transmit clauses to each client using the *exportClauseToClient* function, which primarily calls the client's public *importClause* method. This adheres to a key design principle: the producer does not judge the usefulness of a clause; instead, this responsibility is left to the consumer. By invoking the client's *importClause* method, we ensure that each client can implement its own sharing logic and clause filtering.

As illustrated in Figure 2, both *SharingStrategy* and *SolverCdclInterface* maintain dedicated *ClauseDatabase* instances to store imported clauses. Each *ClauseDatabase* implementation applies specific storage policies that may discard clauses based on memory constraints or other criteria, effectively adding a second filtering layer after *SharingStrategy*'s initial filter. The *SharingStrategy* additionally maintains a list of producers from which it obtains clauses. Two key methods manage producer interactions:

– *addProducer*: Initializes the necessary data structures for a new producer.

– *connectProducer*: Adds the strategy as a client in the producer's client list.

This two-step process ensures proper initialization order, mitigating potential concurrency issues.

In this revised architecture, the *Sharer* invokes two essential methods defined by the *SharingStrategy*: *doSharing*, which handles the periodic selection and export of clauses to the strategy's clients, and *getSleepingTime*, which determines the wait time for the executor between two consecutive sharing operations. Additionally, the *Sharer* can now manage multiple *SharingStrategy* instances, invoking their *doSharing* methods in a round-robin fashion.

For inter-node sharing strategies, we introduced the specialized *GlobalSharingStrategy* interface, which adds distributed initialization and finalization through two additional methods: *initMpiVariables* and *joinProcess*. Its *doSharing* method incorporates a termination-detection algorithm for the distributed solver, as detailed in Algorithm 2. While implementations can use this algorithm, they are free to develop custom approaches. Developers can also bypass the *GlobalSharingStrategy* interface entirely and directly implement the new *SharingStrategy* interface, enabling highly customized sharing strategies.

It is important to note that a sharing strategy may not always be necessary; in some cases, simply connecting different solvers via the client list is enough, eliminating the need for an additional thread to manage the sharing process.

These enhancements streamline the handling of challenging aspects in distributed environments, such as deployment and termination detection. The framework's simplicity and extensibility facilitate rapid development of new distributed portfolio solvers, while its modular design enables researchers to conduct fair comparisons of different heuristics by maintaining consistency across other solver components.

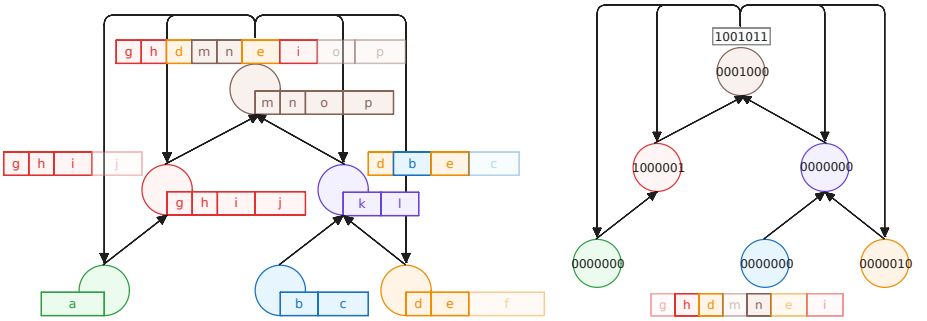
5 Assessing The New Distributed *D-Painless* Framework

To demonstrate the scientific relevance of our framework, we present a study of various parallelization and sharing strategies inspired by state-of-the-art distributed SAT solvers, namely *Mallob* [37] and *PRS-Distributed* [32]. Additionally, we compare novel combinations of global and local sharing strategies that have not been previously explored.

5.1 Implementations of *GlobalSharingStrategy*

We have developed three distributed sharing strategies for evaluating *D-Painless*: *GenericGlobalSharing*, *AllGatherSharing* and *MallobSharing*. They all serialize the clauses to be shared along with their Literal Block Distance (LBD) values [1][4], enabling the receiving node's consumer to assess the usefulness of the received clauses. However, the LBD value makes sense only locally to the solver that produced the clause, sharing it may help, but it has not that much importance [37, Section 5.1]. The size of the serializing buffer is configurable, which is crucial for controlling latency in inter-node communication.

[4] The LBD value is a heuristic used to estimate the importance of a clause, for more details you can refer to [1].



(a) Gathering of clause buffers to the root and broadcasting it.

(b) Generation of the filter bit vector.

Figure 3: The two phases of *MallobSharing* strategy

GenericGlobalSharing implements a flexible approach to clause sharing among distributed SAT solver instances. It utilizes user-defined subscription and subscriber lists to establish communication patterns between nodes, allowing for customizable topologies including ring structures. The strategy employs **non-blocking** sends (`MPI_Isend`) to distribute clauses to subscribers and waits for incoming clauses by probing (`MPI_Probe`), using dynamic receive buffers to efficiently collect clauses from subscriptions. To optimize sharing, the implementation incorporates two Bloom filters [12]: one to prevent resending previously shared clauses, and another to avoid deserializing duplicate received clauses. We judged receiving duplicates of locally produced clauses from a subscription not harmful to the local solvers.

AllGatherSharing implements a straightforward strategy in which, during each round, every node broadcasts its clauses to all other nodes (as in [5]. The `MPI_Allgather` function serves as a synchronization point for all n participating nodes in the distributed system. To expedite the sharing process, we use a uniform static buffer size s across all nodes, padding with zeros if necessary, instead of pre-sending buffer sizes using `MPI_Allgatherv`. After collecting all buffers, we merge them into a single unified buffer, ensuring that each node ends the sharing round with an identical buffer of size $n \times s$. A Bloom filter is employed to prevent redundant serialization and deserialization of clauses that have already been transmitted [5]. Upon deserialization, the clauses are immediately *exported* to the strategy's clients.

MallobSharing strategy uses a binary tree topology for message passing, inspired by the sharing phase in *Mallob* [37]. Each sharing round consists of two phases: *clause merge* and *clause filtering*. Communication is handled using `MPI_Send`, `MPI_Probe`, and `MPI_Recv`. The *importClause* method filters clauses based on their size and LBD values against predefined *maxSize* and *maxLBD* before storing them in the *ClauseDatabase*.

In the first phase (see Figure 3a), nodes merge clause buffers received from their children with their own. Clauses are ordered by size, using LBD for tie-breaking, allowing an ordered merge to select the best clauses. The buffer size

$l(u)$ at each node is limited, except for clauses smaller than *freeSize*, which are not counted when serialized to the buffer [37]:

$$l(u) = \beta_\infty - (\beta_\infty - \beta_0) \cdot e^{\frac{\beta_0}{\beta_0 - \beta_\infty} \cdot (u-1)}$$

where β_∞ is the maximum buffer size, β_0 is the minimum buffer size, and u is the number of merged buffers so far. Each node passes its value of u to its parent via the clause buffer.

The root node's merged buffer contains the best clauses of the sharing round. This buffer is then broadcast to all nodes via the binary tree, with each node sending the buffer to its children.

In the second phase (see Figure 3b), each node generates a bit vector indicating which clauses were self-produced and shared in the previous z rounds. A custom distributed exact filter [37] detects duplicates, allowing clauses to be reshared after sufficient time. The bit vectors are combined via an OR operation up to the root and then broadcast to all nodes. Using the merged bit vector, nodes determine which clauses to export to their local clients.

Although the filter phase can lead to some wasted buffer space, a *compensation* mechanism dynamically adjusts the buffer size in future sharing rounds to mitigate this issue [37].

The first strategy developed was *AllGatherSharing*, consisting of 192 lines of code (LoC) in its `.cpp` file. The *GenericGlobalSharing* file contains 196 LoC, while *MallobSharing* is larger at 728 LoC. Since initialization, serialization, and deserialization processes are consistent across all strategies, both *GenericGlobalSharing* and *MallobSharing* inherit approximately 80 LoC from *AllGatherSharing*. The exact filter methods, adding another 83 LoC, are now integrated into the *MallobSharing* class, as there is currently no separate filter interface.

5.2 Portfolio implementations of *WorkingStrategy*

To evaluate distributed portfolio solvers, we require implementations of: *SolverInterface* to solve formulas, *WorkingStrategy* to enforce the portfolio strategy, and *ClauseDatabase* for clause management.

D-Painless offers multiple *SolverInterface* implementations, including *CadicalSolver* [9], *KissatSolver* [10], *KissatMABSolver* [34], *KissatINCSolver* [13], *LingelingSolver* [8], *MapleCOMSPSSolver* [29], *MiniSatSolver* [18], and *GlucoseSyrupSolver* [3]. For *ClauseDatabase*, notable implementations include *ClauseDatabaseMallob*, which respects a certain literal capacity and manages space for better clauses [37]; *ClauseDatabaseBufferPerEntity*, designed to manage buffers for individual entities without requiring a lock mechanism; and *ClauseDatabasePerSize*, which organizes clauses based on their size.

There is also an existing implementation of the intra-node sharing strategy *HordeSatSharing* [5]. It filters produced clauses based on their LBD: it dynamically adjusts the solver's LBD threshold to maintain optimal sharing if a solver shares too few or too many clauses.

We implement two portfolio strategies using the *WorkingStrategy* interface: *PortfolioSimple* and *PortfolioPRS*. Both leverage the *mpiRank* variable to di-

versify solvers via unique `globalIds`. The *SolverInterface* manages a general ID and type ID, enabling complete diversification.

PortfolioSimple offers a flexible, general-purpose implementation supporting both CDCL and local search solvers. It features local and global sharing mechanisms, a `mallob` mode replicating the sharing strategy from [37], and easy customization of solver types and strategies.

PortfolioPRS is a specialized portfolio based on *PRS-Distributed* [32]. It executes pre-processing techniques such as Unit Propagation, Equivalent-literal Substitution, and Resolution Checking [14] in a leader node. Then, it divides all nodes into five groups: SAT, UNSAT, MAPLE, LGL, and DEFAULT. By following the implementation of [32], the group sizes are calculated based on the total number of nodes n, and are: $\lfloor \frac{n}{8} \rfloor$, $\lfloor \frac{n}{4} \rfloor$, $\lfloor \frac{n}{8} \rfloor$, 1, $n - 2 \cdot \lfloor \frac{n}{8} \rfloor - \lfloor \frac{n}{4} \rfloor - 1$, respectively.

5.3 Evaluated Solvers

In Section 5.4 we evaluate different solvers, with different sharing strategies. The first two solvers emulate state-of-the-art solvers, while the two last explore new solvers implementing original sharing strategies.

PL-Mallob-kc: is an instantiation of *PortfolioSimple* that emulates the setup of *Mallob* [37]. The solvers set includes *Kissat* (v3.1.1) and *CaDiCaL* (v1.9.1)[5]. For the sharing strategy, all solvers act as both producers and clients of the *MallobSharing* strategy. It is configured in concordance to the one used by *Mallob* in the 2024 International SAT Competition:

$$\beta_0 = \frac{400 \cdot solverCount}{roundsPerSecond}, with\ roundsPerSecond = 2$$

The *freeSize* parameter is set to 1 for unlimited unit clause sharing, the maximum clause size *maxSize* and LBD value *maxLBD* are both set to 60. Lastly, the clause database *ClauseDatabaseMallob* is employed, as outlined in [37] with the maximum size set to *maxSize* and the capacity defined as $10 * \beta_0$. β_∞ is set to 250000, while z is defined as 15 seconds, equivalent to 30 rounds. z represents the number of rounds during which a clause is remembered as being shared. If a clause is reshared after this period, it won't be filtered as redundant. However, following *Mallob* terminology, we used an option in seconds.

PL-HordeGeneric-PRS: is an instantiation of *PortfolioPRS* that emulates *PRS-Distributed* [32]. The SAT and UNSAT groups use *KissatINC* (v1.0.3) solvers configured for their respective instance types. The MAPLE group uses *MapleCOMSPS* solvers, and the LGL group uses *Lingeling* (v1.0.0) solvers. Finally, the DEFAULT group uses *KissatINC* solvers with no specialization. For the sharing strategy, *PRS-Distributed* requires sharing clauses from local solvers to the next neighbour, exporting received clauses from the previous neighbour to local solvers and to the next neighbour. Local solvers export clauses according to

[5] The integration of *Lingeling* wasn't used due to a bug on our side when sharing clauses with other solvers (this has been fixed). However, it was used in *PRS* because *Lingeling* solvers shared clauses only among themselves.

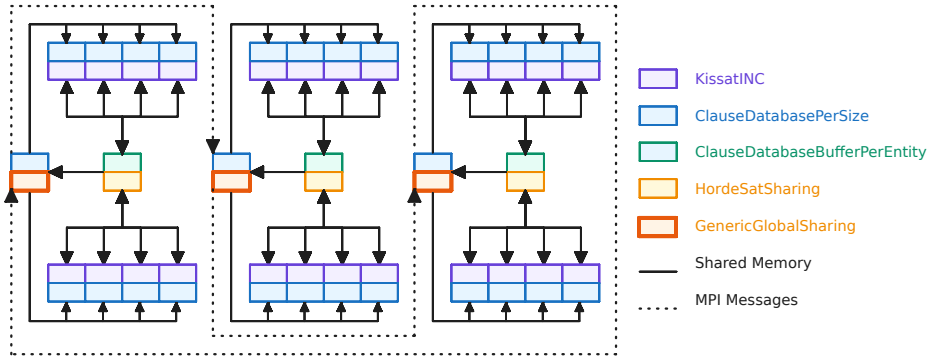


Figure 4: Ring topology of the DEFAULT group with 3 nodes in *PortfolioPRS*

the hordeSat strategy [5]. To emulate this behaviour, we use a *HordeSatSharing* instance having all solvers as clients and producers, and an instance of *GenericGlobalSharing* configured for in-group ring sharing as a client (See Figure 4). To allow solvers receiving clauses from the previous neighbour, they subscribe to the *GenericGlobalSharing*.

The *GenericGlobalSharing* sends clauses obtained from the solvers via the *HordeSatSharing* strategy to the next neighbour and exports received clauses from the previous neighbour to the solvers. To resend clauses received from the previous neighbour to the next neighbour, the *GenericGlobalSharing* needs to *import* clauses that it *exports* to its clients; thus, it adds itself to its list of clients. Finally, following the implementation of *PRS-Distributed*, the *GenericGlobalSharing* strategy uses an unlimited buffer size and a single *Sharer* instance executes both strategies.

PL-HordeMallob-kc: is an instantiation of *PortfolioSimple* with *KissatSolver* and *CadicalSolver* solvers, all connected to *HordeSatSharing*. A *MallobSharing* instance is used for inter-node sharing with the same parameters as *PL-Mallob-kc*. It is solely connected to *HordeSatSharing*, and both of them run on different threads.

PL-HordeAllgather-kc: is an instantiation of *PortfolioSimple* with *KissatSolver* and *CadicalSolver* solvers, all connected to *HordeSatSharing*. The *AllGatherSharing* is used for inter-node sharing, solely connected to *HordeSatSharing*. The *AllGatherSharing* buffer size is set to $1500 \cdot solverCount$ and each sharing strategy is executed by a separate *Sharer* thread.

The initial LBD value in *HordeSatSharing* is set to 2, with each producer having an upper limit of 1500 literals per sharing round. *HordeSatSharing* uses *ClauseDatabaseBufferPerEntity* for concurrency management, while each solver uses *ClauseDatabasePerSize* to store the imported clauses.

5.4 Evaluation

To assess the stability and performance of our new architecture, we used the 400 instances from the SAT'2024 competition, each with a timeout limit of 500 seconds. The tests were conducted on nodes in the "paravance" cluster of the

Instance	Solved	PAR2	SAT	UNSAT	SMAPE-VBS (%)
VBS-prs	318	215.394	154	164	0
PRS-Distributed	304	254.615	143	161	**13.718**
PL-HordeGeneric-PRS	**313**	**237.338**	**150**	**163**	17.894
VBS-mallob-kc	311	218.365	151	160	0
Mallob-kc	**309**	**230.978**	**149**	**160**	**8.069**
PL-Mallob-kc	308	234.252	151	157	20.050
PL-HordeMallob-kc	307	244.905	153	154	-
PL-HordeAllgather-kc	306	247.948	151	155	-

Table 2: PAR2 and number of resolved instances on GRID5000. The winner and best values in each category are shown in bold font.

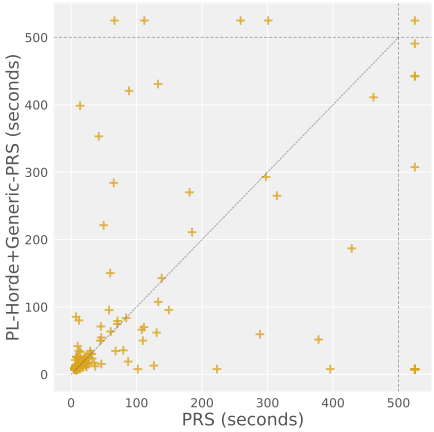
Grid5000 infrastructure. Each node is equipped with two Intel Xeon E5-2630 v3 processors, featuring 8 physical cores each, and 128 GB of RAM, operating under a Non-Uniform Memory Access (NUMA) architecture with each CPU paired with 64 GB of RAM. All evaluations were done using 25 machines, thus having 50 CPUs, totaling 400 cores. All tested solvers use OpenMPI [20], and for each CPU an MPI process is instantiated.

To evaluate our emulations of *Mallob*[6] and *PRS-Distributed*[7], we ran the versions they submitted to the 2024 SAT Competition. *Mallob* and *PL-Mallob-kc* are configured as described earlier Section 5.3, please note that *Mallob* was used in its mono mode, *i.e.* only one SAT problem is solved at a time. With 400 cores, the portfolios consist of multiple sequential solvers: 200 *Kissat* and 200 *CaDiCaL*. *PRS-Distributed* and *PL-HordeGeneric-PRS* both follow a group distribution strategy (Section 5.2): 6 nodes with SAT-specialized *KissatINC* (48 cores), 12 nodes with UNSAT-specialized *KissatINC* (96 cores), 6 nodes with *MapleCOMSPS* (48 cores), 1 node with *Lingeling* (8 cores), and 25 nodes with general-purpose *KissatINC* (200 cores).

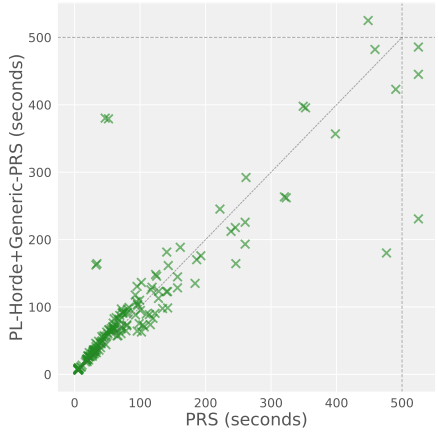
The version of *D-Painless* used to instantiate these solvers is available on GitHub[8]. The results of the evaluation are presented in Table 2, with the top row of the two first groups showing their *relative* Virtual Best Solver (VBS).

The solvers are compared using two metrics: the number of solved SAT problems and their *PAR2 score*. The PAR2 score penalizes unsolved instances by counting them as 2 times the timeout value. While traditionally PAR2 was calculated as a sum, in recent SAT competitions it is reported as an average on the number of benchmark instances. Additionally, to quantify how closely a solver's S execution times $S_{time}(i)$ approaches the ones of its relative VBS across the n tested instances, we use the *Symmetric Mean Absolute Percentage Error* (SMAPE) [27], defined as:

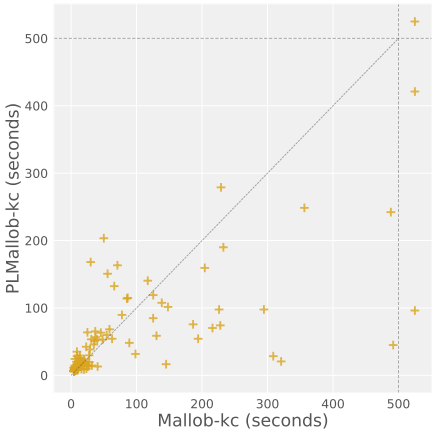
[6] https://github.com/domschrei/mallob/tree/08898345effa904c87d82a73dbee339049467d61
[7] https://github.com/shaowei-cai-group/PRS-sc24/tree/main/PRS-distributed-sc24
[8] https://github.com/lip6/painless/releases/tag/v1.24.10



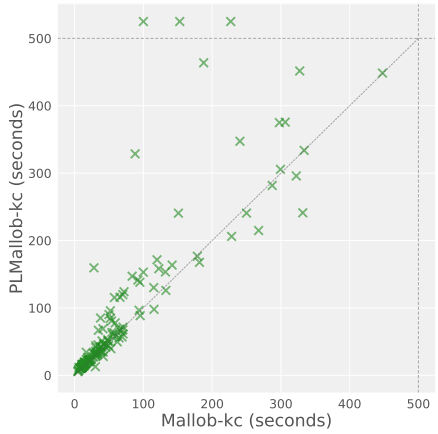
(a) Comparison on SAT instances between *PRS-Distributed* and *PL-HordeGeneric-PRS*.



(b) Comparison on UNSAT instances between *PRS-Distributed* and *PL-HordeGeneric-PRS*.



(c) Comparison on SAT instances between *Mallob-kc* and *PL-Mallob-kc*.

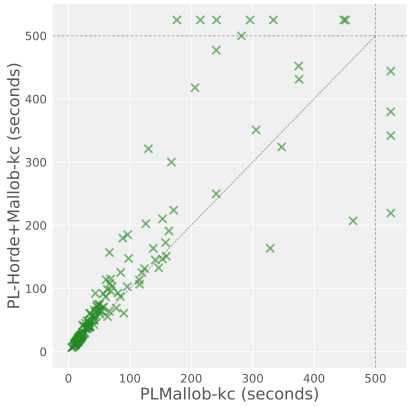


(d) Comparison on UNSAT instances between *Mallob-kc* and *PL-Mallob-kc*.

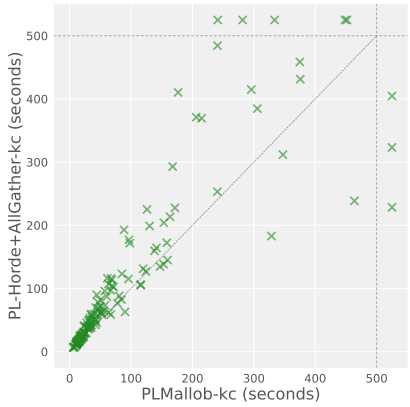
Figure 5: Execution time comparisons of the emulations

$$\text{SMAPE-VBS}(S) = \frac{100}{n} * \sum_{i=1}^{n} \frac{|S_{time}(i) - VBS_{time}(i)| * 2}{S_{time}(i) + VBS_{time}(i)}$$

Table 2 tells us that the best overall solver is *PL-HordeGeneric-PRS*, which solves the most instances. This is quite logical, since it uses three different state-of-the-art solvers: *Lingeling*, *CaDiCaL*, and *Kissat*, each with different preprocessing and in-processing techniques, enabling it to solve different types of formulae. After it, *Mallob-kc* comes second. Even though it only uses *Kissat* and *CaDiCaL*, it achieves excellent performance thanks to its clever clause sharing, showing the importance of efficient clause sharing in solving SAT problems. *PL-*



(a) Comparison on UNSAT instances between *PL-Mallob-kc* and *PL-HordeMallob-kc*.



(b) Comparison on UNSAT instances between *PL-Mallob-kc* and *PL-HordeAllgather-kc*.

Figure 6: Execution times on UNSAT instances comparison with *PL-Mallob-kc*

Mallob-kc comes very close to *Mallob-kc*, with a single instance difference in the overall score, but the UNSAT column shows us that it struggles to match the performance of *Mallob-kc* in UNSAT instances. The two solvers *PL-HordeMallob-kc* and *PL-HordeAllgather-kc* perform worse than *PL-Mallob-kc*, likely due to their more aggressive clause sharing. The *PRS-Distributed* solver performs the worst; however, it was able to solve instances that were not solved by the best solver *PL-HordeGeneric-PRS*, as shown by their relative VBS.

When comparing *PRS-Distributed* with our emulation (*PL-HordeGeneric-PRS*), the scatter plots in Figure 5b show that our emulation solves nearly the same UNSAT instances as *PRS-Distributed*, with improved performance on SAT instances (Figure 5a). It is worth noting that even if *PL-HordeGeneric-PRS* achieves better results than *PRS-Distributed*, its resolution times are a little bit behind the best ones, as it is shown by the new metric *SMAPE-VBS* in Table 2. They are still close in performance, and thus we can say that we were able to emulate the *PRS-Distributed* solver successfully with a more complex software architecture.

When comparing *Mallob-kc* to our emulation (*PL-Mallob-kc*), we observe that our solver is a bit less efficient on UNSAT instances (Figure 5d). However, as illustrated in Figure 5c, our solver resolves many SAT instances more efficiently. The *SMAPE-VBS* values in Table 2 confirms the current shortcomings, since the *PL-Mallob-kc* solver is quite far from its relative VBS. These observations suggest that the current implementation may suffer from limited diversification or sub-optimal clause sharing.

In *PL-HordeMallob-kc* and *PL-HordeAllgather-kc*, we implement more aggressive clause sharing, utilizing the *HordeSatSharing* strategy for local sharing and the *MallobSharing* or *AllGatherSharing* strategy for global sharing. Although their PAR2 scores are higher than that of *PL-Mallob-kc*, Figure 6b

and Figure 6a show that these configurations are able to solve some UNSAT instances that *PL-Mallob-kc* could not. The increased average solving time is likely a consequence of the aggressive clause sharing, which overloads the solvers with excessive clauses, thereby slowing down unit propagation. However, *PL-HordeMallob-kc* with its more balanced global sharing via *MallobSharing* allows to achieve better overall performance.

In summary, these experiments demonstrate that the solvers instantiated from the *D-Painless* framework can perform on par with state-of-the-art solvers. Our results show that *D-Painless* can successfully emulate existing solvers using different clause-sharing strategies with minimal performance loss. This highlights the framework's ability to offer the flexibility, modularity, and generality for which it was designed. Ultimately, *D-Painless* simplifies the development of new distributed portfolio SAT solvers, making it easier to experiment with novel sharing strategies.

6 Conclusion

Taking advantage of massively parallel hardware is essential for addressing hard SAT problems. Current tools offer excellent performance and scalability on distributed systems for portfolio strategies but lack flexibility in their inter-node communication strategies. We propose a new extension for *Painless*, namely *D-Painless*, enabling clause sharing in a distributed system. This extension makes the framework capable of instantiating distributed portfolio solvers using various clause-sharing strategies.

An initial assessment shows that the new *D-Painless* architecture allows for the construction of distributed portfolios, offering performance close to that of state-of-the-art distributed SAT solvers. The modular and flexible nature of this new architecture simplifies experimenting with different sharing strategies.

The current architecture lacks a reliable mechanism to detect errors, making it vulnerable to node failures. Thus, to improve robustness, we aim to refine our strategies, introduce new ones like divide-and-conquer, and develop fault-tolerant algorithms that simplify the management of distributed workflows. In addition to this, we will work on optimizing the performance of the various existing components to further enhance the overall efficiency of the system.

Acknowledgements

The experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several Universities as well as other organizations[9].

[9] `https://www.grid5000.fr`.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI. vol. 9, pp. 399–404 (2009)
2. Audemard, G., Simon, L.: Glucose in the SAT 2014 competition. SAT COMPETITION 2014 p. 31 (2014)
3. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: int. conf. on Theory and Applications of Satisfiability Testing. pp. 197–205. Springer (2014)
4. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions. Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland (2023)
5. Balyo, T., Sanders, P., Sinz, C.: Hordesat: A massively parallel portfolio SAT solver. In: Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 156–172. Springer (2015)
6. Beyer, D., Kanav, S., Richter, C.: Construction of verifier combinations based on off-the-shelf verifiers. In: Johnsen, E.B., Wimmer, M. (eds.) Fundamental Approaches to Software Engineering. pp. 49–70. Springer International Publishing, Cham (2022)
7. Biere, A., van Maaren, H.: Handbook of satisfiability: Second edition. IOS Press, Amsterdam, NY (May 2021)
8. Biere, A.: Splatz, lingeling, plingeling, treengeling, yalsat entering the SAT competition 2016. SAT COMPETITION 2016 p. 44 (2016)
9. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2017-1, pp. 14–15. University of Helsinki (2017)
10. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
11. Biere, A., Järvisalo, M., Kiesl, B.: Preprocessing in SAT solving. In: Handbook of Satisfiability (2021)
12. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
13. Chen, Z., Zhang, X., Cai, S., Lu, P.: Cdcl solvers with improved local search cooperation and pre-processing. In: Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland (2022)
14. Chen, Z., Zhang, X., Cai, S., Lu, P.: Cdcl solvers with improved local search cooperation and pre-processing. In: SAT COMPETITION 2022 Proceedings. pp. 37,38 (2022)
15. Chen, Z., Zhang, X., Qian, Y., Cai, S.: Prs: A new parallel/distributed framework for SAT. In: SAT COMPETITION 2023 Proceedings. pp. 39,40 (2023)
16. Chrabakh, W., Wolski, R.: Gridsat: A chaff-based distributed SAT solver for the grid. ACM/IEEE SC 2003 Conference (SC'03) pp. 37–37 (2003)
17. Conference, S.: SAT competition website, `https://satcompetition.github.io/`

18. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 502–518. Springer (2003)
19. Frioux, L.L., Baarir, S., Sopena, J., Kordon, F.: Modular and efficient divide-and-conquer SAT solver on top of the painless framework. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems (2019)
20. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting. pp. 97–104. Budapest, Hungary (2004)
21. Gailly, J.l., Adler, M.: zlib (1995), `https://www.zlib.net/`
22. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation **6**, 245–262 (2009)
23. Heisinger, M., Fleury, M., Biere, A.: Distributed cube and conquer with paracooba. In: SAT. Lecture Notes in Computer Science, vol. 12178, pp. 114–122. Springer (2020)
24. Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding cdcl SAT solvers by lookaheads. In: Haifa Verification Conference. pp. 50–65. Springer (2011)
25. Hu, K., Chu, Z.: An efficient circuit-based SAT solver and its application in logic equivalence checking. Microelectronics Journal **142**, 106005 (2023)
26. Kleine Büning, M., Balyo, T., Sinz, C.: Using dimspec for bounded and unbounded software model checking. In: Ait-Ameur, Y., Qin, S. (eds.) Formal Methods and Software Engineering. pp. 19–35. Springer International Publishing, Cham (2019)
27. Kreinovich, V., Nguyen, H.T., Ouncharoen, R.: How to estimate forecasting quality: A system-motivated derivation of symmetric mean absolute percentage error (smape) and other similar characteristics (2014)
28. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: Painless: a framework for parallel SAT solving. In: Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 233–250. Springer (2017)
29. Liang, J.H., Oh, C., Ganesh, V., Czarnecki, K., Poupart, P.: Maplecomsps lrb vsids, and maplecomsps chb vsids. In: Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions. pp. 20–21. Department of Computer Science, University of Helsinki, Finland (2017)
30. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. Journal of Automated Reasoning **24**, 165–203 (2000)
31. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC). pp. 530–535. ACM (2001)
32. Qian, Y., Chen, Z., Zhang, X., Cai, S.: Prs-distributed in the SAT competition 2024. In: Proceedings of SAT Competition 2024: Solver and Benchmark Descriptions. Department of Computer Science Report Series B, Department of Computer Science, University of Helsinki, Finland (2024)
33. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Ph.D. thesis, Theses (School of Computing Science)/Simon Fraser University (2004)
34. Sami Cherif, M., Habet, D., Terrioux, C.: Un bandit manchot pour combiner CHB et VSIDS. In: Actes des 16èmes Journées Francophones de Programmation par Contraintes (JFPC). Nice, France (Jun 2021)
35. Schreiber, D.: Lilotane: A lifted SAT-based approach to hierarchical planning. J. Artif. Int. Res. **70**, 1117–1181 (may 2021)

36. Schreiber, D., Sanders, P.: Scalable SAT solving in the cloud. In: Li, C.M., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing – SAT 2021. pp. 518–534. Springer International Publishing, Cham (2021)
37. Schreiber, D., Sanders, P.: MallobSat: Scalable SAT solving by clause sharing. Journal of Artificial Intelligence Research (JAIR) (2024), presented at Pragmatics of SAT (PoS) 2024
38. Silva, J.M., Sakallah, K.A.: Grasp-a new search algorithm for satisfiability. In: Proceedings of International Conference on Computer Aided Design. pp. 220–227. IEEE (1996)
39. Vallade, V., Baarir, S., Sopena, J.: New concurrent painless solvers based on kissat-mab: P-kissat and p-kissat-str. In: SAT COMPETITION 2023 Proceedings. pp. 42,43 (2023)
40. Vallade, V., Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: On the usefulness of clause strengthening in parallel SAT solving. In: Proceedings of the 12th NASA Formal Methods Symposium (NFM). Springer (2020)
41. Vallade, V., Le Frioux, L., Oanea, R., Baarir, S., Sopena, J., Kordon, F., Nejati, S., Ganesh, V.: New concurrent and distributed painless solvers: P-mcomsps, p-mcomsps-com,p-mcomsps-mpi, and p-mcomsps-com-mpi. In: Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. p. 40. Department of Computer Science, University of Helsinki, Finland (2021)
42. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation **21**(4), 543–560 (1996)
43. Zhang, H., Stickel, M.: Implementing the davis–putnam method. Journal of Automated Reasoning **24**(1-2), 277–296 (2000)