Journal of Artificial Intelligence Research 80 (2024) 1437-1495 Submitted 12/2023; published 08/2024

MallobSat: Scalable SAT Solving by Clause Sharing

Dominik Schreiber dominik.schreiber@kit.edu
Peter Sanders sanders@kit.edu
Karlsruhe Institute of Technology,
Kaiserstr. 12, 76131 Karlsruhe, Germany

Abstract

SAT solving in large distributed environments has previously led to some famous results and to impressive speedups for selected inputs. However, in terms of general-purpose SAT solving, prior approaches still cannot make efficient use of a large number of processors. We aim to address this issue with a complete and systematic overhaul of the distributed solver HordeSat with a focus on its algorithmic building blocks. In particular, we present a communication-efficient approach to clause sharing, careful buffering and filtering of produced clauses, and effective orchestration of state-of-the-art solver backends. In extensive evaluations, our approach named MallobSat significantly outperforms an updated HordeSat, doubling its mean speedup. Our clause sharing results in effective parallelization even if all threads execute *identical* solver programs that only differ based on which clauses they import at which times. We thus argue that MallobSat is not a portfolio solver with the added bonus of clause sharing but rather a *clause-sharing solver* where adding some explicit diversification is useful but not essential. We also discuss the last four iterations of the International SAT Competition (2020–2023), where our system ranked very favorably, and identify several previously unsolved competition problems that MallobSat solved successfully. Last but not least, our approach is *malleable*, i.e., supports running on a fluctuating set of resources, which allows us to combine parallel job processing and parallel SAT solving in a flexible manner for best resource efficiency.

1. Introduction

The *Propositional Satisfiability* (SAT) problem, i.e., to satisfy a given Boolean expression or to report its unsatisfiability, is an essential building block at the core of automated reasoning, symbolic AI, and formal verification (Fichte et al., 2023). In today's applied SAT solving, researchers and industrial users increasingly strive to exploit modern *distributed* environments with hundreds to thousands of cores (Heisinger et al., 2020; Froleyks et al., 2021; Cook, 2021; Burgess et al., 2022) in order to reduce processing times and to tackle difficult problems that are infeasible to solve with sequential algorithms. Prior massively parallel approaches that successfully solved open mathematical problems (Heule et al., 2016; Subercaseaux & Heule, 2023) are usually fine-tuned to the particular problem at hand. In terms of SAT solving that is *general purpose*, i.e., that works efficiently on application problems never seen before, existing distributed solvers still leave much to be desired. Linear or even "*superlinear speedups*" (Balyo et al., 2015) achieved for few individual instances must be set in relation with the total work invested in every single formula to achieve such peak speedups. Specifically, the prior state of the art in distributed SAT solving, HordeSat, achieved a median speedup of 13 on 2048 cores on industrial benchmarks (Balyo et al., 2015), implying a median efficiency of only 0.6%.

We argue that two distinct challenges must be met to improve on the practical merit of distributed SAT solving. First and foremost, distributed SAT solvers themselves need to become more efficient and more scalable for realistic and diverse inputs. Secondly, the deployment of SAT solving tasks in distributed environments must be performed in a more careful manner—ideally processing several tasks at a time and allotting a large amount of computational resources only to sufficiently difficult problems.

In this work, we aim to address both challenges with a new distributed SAT solver, denoted MallobSat[1] throughout this work. We revisit the popular massively parallel SAT solver HordeSat (Balyo et al., 2015) and carefully re-engineer its algorithmic building blocks. Most notably, we rethink all-to-all clause sharing. MallobSat aggregates the globally most promising distinct clauses and reliably targets a certain sharing volume that is sublinear in the number of participants. This ensures an effective and scalable clause sharing operation even with thousands of cores. We also introduce distributed filtering of recently shared clauses. To further improve the practicality of our approach, we propose measures to make more careful use of main memory, such as the manipulation of shared clauses' LBD values, and we extend and update the used sequential SAT solver backends. Last but not least, MallobSat is designed to be *malleable*, i.e., it supports a fluctuating number of workers during its execution. This allows us to integrate MallobSat in the decentralized online job scheduling platform Mallob (Sanders & Schreiber, 2022).

In comprehensive empirical analyses on HPC systems, we investigate the benefits of our techniques. An especially intriguing result is that our system remains scalable *even if solver threads are not diversified at all*, i.e., if hundreds of *identical* CaDiCaL programs run in parallel and solely differ based on the points in time at which shared clauses arrive. Following this observation, we suggest that the term "*portfolio solver*" (Hamadi et al., 2010; Biere, 2013; Ehlers et al., 2014; Balyo et al., 2015; Le Frioux et al., 2017a) does not adequately capture the main essence of approaches such as ours and endorse the term "*clause-sharing solver*" (*cf.* Manthey et al., 2013; Michaelson et al., 2023) instead.

Our scaling results show that our solver doubles the speedups of an updated version of HordeSat and scales up to 3072 cores (64 nodes). To our knowledge, this is the largest scale yet at which an integrated distributed SAT solver has been evaluated. Furthermore, we analyze the results of four iterations of the International SAT Competition (ISC), which indicate that our system represents the state of the art in (massively) parallel SAT solving. We identify several instances that presumably have never been solved before but that MallobSat at 3072 cores solves successfully. We also demonstrate that MallobSat's malleable deployment can result in an appealing combination of "embarrassingly parallel" job processing and the speedups obtained by parallel SAT solving.

Put together, our contributions are as follows:

- We provide an overview of parallel and distributed SAT solving research and specifically discuss experimental methodology to evaluate parallel solvers.
- We present a scalable distributed clause-sharing solver with compact clause exchange, exact distributed clause filtering, modern solver interfaces, memory awareness, and malleable deployment capabilities.

1. Mallob is an acronym for **Mal**leable **Lo**ad **B**alancer as well as (in the context of MallobSat) **Ma**ssively P**a**ra**ll**el **Lo**gic **B**ackend. In prior competitive events, MallobSat participated under the name of its surrounding execution environment Mallob (with varying suffixes).

- We provide extensive evaluations and detailed empirical insights regarding our solver's performance, scaling, and clause sharing behavior on up to 3072 cores.
- We compare our approach to existing approaches (a) directly via an experimental comparison to HordeSat and (b) indirectly via a discussion and analysis of the results of the International SAT Competition 2020–2023. Follow-up experiments of ours confirm that MallobSat can indeed push the frontier of solvable problems.
- We show how malleability helps to solve a record number of SAT instances in a massively parallel environment (6400 cores) in a highly resource-efficient manner.

This article is structured as follows. We introduce the SAT problem in Section 2 and then discuss parallel and distributed SAT solving in Section 3. In Section 4 we provide an overview of our design decisions. We then present our approaches on clause sharing in Section 5 and on diversification in Section 6. We describe further technical contributions, such as memory awareness, in Section 7. Section 8 features our empirical analyses and experimental evaluations. We conclude our work in Section 9.

Context. This article is largely based on Chapters 2, 4, and 8 of Schreiber's (2023b) dissertation. Those chapters, in turn, include some content from a publication by Schreiber and Sanders (2021) and from four competition submission descriptions (Schreiber, 2020, 2021b, 2022, 2023a). Compared to the dissertation, some sections of this article are entirely original (8.6, 8.9) or have been revised based on new insights (8.3, 8.5).

2. SAT Fundamentals

In the following, we provide a brief overview of (sequential) SAT solving.

A *Boolean variable* is a variable that can only be `true` or `false`. A *literal* is a Boolean variable or its negation. A *clause* is a disjunction of literals, i.e., an expression that evaluates to `true` if and only if at least one of the literals in the clause is `true`. A *unit clause* is a clause with one literal, a *binary clause* has two literals, and a clause of *length* l has l literals. A *Conjunctive Normal Form* (CNF) formula is a conjunction of clauses, i.e., an expression that evaluates to `true` if and only if each of the clauses evaluates to `true`. An *assignment* $\mathcal{A}$ for a logical expression F assigns values to the variables occurring in F. $\mathcal{A}$ is a *satisfying assignment (for F)* if and only if F evaluates to `true` under $\mathcal{A}$. A CNF formula F is *satisfiable* if and only if a satisfying assignment to F exists. Otherwise, F is *unsatisfiable.*

An instance of the *SAT decision problem* is given as a CNF formula F.[2] The task is to decide whether F is satisfiable. In practical SAT solving, we usually consider the *constructive SAT problem* that requires to output a satisfying assignment $\mathcal{A}$ if F was found to be satisfiable. An extension of the constructive SAT problem, in turn, is the *certified SAT problem* that additionally requires a *proof of unsatisfiability* if F was found to be unsatisfiable. While we do not consider the certified SAT problem in this article, note that our system recently received support for this extension (Michaelson et al., 2023).

The SAT decision problem is the original NP-complete problem (Cook, 1971). As such, no polynomial-time algorithm for SAT solving is known. Today's most efficient SAT solvers build upon the DPLL algorithm (Davis et al., 1962), which performs a *backtracking search*

2. Any Boolean expression with common junctors ($\wedge$, $\vee$, implication, equivalence, XOR, etc.) of size n can be transformed into a CNF formula of size $\mathcal{O}(n)$ by introducing helper variables (Tseitin, 1983).

over the space of partial variable assignments, and its successor CDCL (*Conflict-Driven Clause Learning*; see Marques-Silva et al., 2021), which derives and maintains redundant *conflict clauses* during its search. These solvers feature a plethora of techniques in order to perform well on diverse problems. Some important cornerstones include
– highly efficient data structures (Moskewicz et al., 2001; Biere et al., 2020);
– heuristics deciding which variable to next assign a value (Biere & Fröhlich, 2015)
and which value (*phase*) to assign (Pipatsrisawat & Darwiche, 2007; Cai et al., 2022);
– managing and garbage-collecting redundant clauses (Audemard & Simon, 2009);
– frequent *restarts* of the search (Gomes et al., 2000; Oh, 2015);
– pre– and *inprocessing* techniques to simplify the problem (Biere et al., 2021); and
– interacting with incomplete *local search* solving approaches (Cai et al., 2022).

3. Parallel SAT Solving

In the following, we discuss how parallel and distributed environments are exploited for SAT solving. We focus on three mostly orthogonal paradigms: explicit search space partitioning, parallel portfolios, and solving independent SAT problems in parallel.

3.1 Explicit Search Space Partitioning

Since DPLL and CDCL are fundamentally based on the notion of searching the space of (partial) variable assignments, it appears natural to parallelize this search itself. For many years, most parallel SAT solvers followed this paradigm (Balyo & Sinz, 2018). We now highlight some important works on this subject.

Böhm and Speckenmeyer (1996) introduced one of the first parallel SAT solvers. Their solver assigns a certain subspace of variable assignments to each worker. A rebalancing of work is performed whenever the estimated workload of a worker goes below a certain limit. This solver achieved near-linear speedups on particular formulas (random k-SAT and Tseitin graph instances) for up to 256 processors. Another early parallel SAT solver, named PSATO (Zhang et al., 1996), introduced the notion of *guiding paths*. A guiding path represents a path of decision variables along the decision tree of a parallel DPLL procedure and tracks which nodes along the path still need to be explored in the opposite direction. Workers can process different guiding path prefixes independently, which is equivalent to assigning a subspace of variable assignments to each worker. Interestingly, this very early work on parallel SAT solving considered not only distributed computation but also preemption and fault tolerance: Guiding paths are distributed by a leader node to all workers, and bookkeeping the guiding paths the workers followed allows to resume an interrupted or cancelled solving procedure later on (Zhang et al., 1996).

Guiding path based solver PaSAT (Sinz et al., 2001) introduced *clause sharing* (then called *lemma exchange*) to parallel SAT solving. Intuitively, if a solver finds a conflict clause, then it may share this clause with the other solvers in order to reduce redundant work. We discuss clause sharing in more detail in Section 3.2.1.

The basic ideas of dynamic load balancing and clause sharing were adopted and refined in subsequent works. Chrabakh and Wolski (2003) proposed to recursively split a subproblem once solving time surpasses a certain threshold, and Blochinger et al. (2003) introduced load balancing via *randomized work stealing* (Sanders, 1994; Blumofe & Leiserson, 1999) to parallel SAT solving: If a worker runs out of work, it will attempt to "steal" a subproblem from a different worker. A more recent distributed search-space partitioning solver is satUZK-ddc (Grinten, 2017), which was evaluated on 160 cores and achieved a median speedup of 5.7 over sequential solver satUZK-seq (Grinten, 2017).

Heule et al. (2011) coined the term *Cube&Conquer* for an extreme variant of search space partitioning where a large number of partial assignments—named *cubes*—are generated using *look-ahead solvers* (see Heule & van Maaren, 2021) and then distributed over all available workers. By careful partitioning, generating many cubes and distributing them randomly, the expectation is that good load balancing is achieved in practice. C&C solving in large distributed systems managed to solve long-standing open problems of mathematics, such as the Pythagorean Triples problem (Heule et al., 2016) or Schur number five (Heule, 2018). On shared-memory hardware, the Lingeling-based C&C solver Treengeling (Biere, 2012, 2014) was among the top performing parallel solvers in some iterations of the International SAT Competition (Belov et al., 2014). Some works pursue a middle ground between C&C and conventional partitioning, for instance letting C&C solvers split cubes dynamically based on perceived difficulty (Audemard et al., 2016), not unlike the much older GrADSAT (Chrabakh & Wolski, 2003). Integrated distributed C&C solvers include Dolius (Audemard et al., 2014) and, more recently, Paracooba (Heisinger et al., 2020).

The main drawback of search space partitioning approaches is that the heuristics used to split problems are crucial for overall performance and ideally fine-tuned to the application at hand (Heule et al., 2016). A bad branching choice may not divide the work at hand evenly but rather yield two problems that are as hard as the original problem—a phenomenom Schulz and Blochinger (2010) referred to as "*bogus splits*". Another possible problem is that some of the split problems may turn out to be trivial ("*oblique splits*", Schulz & Blochinger, 2010). If such splits happen repeatedly, a *ping-pong phenomenom* occurs where more time is spent on communication and waiting than on solving (Jurkowiak et al., 2001). In terms of SAT solving that is *general purpose*, i.e., practically efficient for diverse application instances, today's search space partitioning solvers tend to be outperformed (Balyo & Sinz, 2018) by another parallelization paradigm which we describe in the following.

3.2 Solver Portfolios

Consider a large assembly of puzzle experts who are given a single difficult puzzle, e.g., a Sudoku problem (Weber, 2005). Their task is to cooperatively solve the puzzle as quickly as possible. Let us also assume that our experts are not particularly social and achieve the highest output if left undisturbed. Since our experts have different mindsets and strategies, it can be an effective approach to have all experts work independently on the entire puzzle. Only one of the experts needs to arrive at a solution in order to accomplish the task.

The SAT solving approach that corresponds to the described "assembly of experts" is termed the *(pure) portfolio* approach (Gomes & Selman, 2001): A number of sufficiently diverse solvers are executed in parallel and "compete" for solving the same problem. The optimistic view on this simple parallelization is that it emulates the *Virtual Best Solver*, VBS in short (Xu et al., 2012)—a theoretical device that has access to a set of approaches and, given a particular problem, always selects the approach that solves this problem the fastest.[3] A more pessimistic (or perhaps realistic) view is that a pure portfolio, while it may be effective for many problems, is not *efficient*: Only a single solver contributes to the final solution regardless of the total number of solvers. If we consider instances in an isolated manner, it can be argued that a pure portfolio solver does not achieve *any* speedup, since there is, by definition, a sequential algorithm that results in the same running time.

A crucial aspect of portfolios is the *diversification* of solvers. Diversification is any kind of mechanism that lets the participants in a portfolio behave differently and thus explore different parts of search space (Hamadi et al., 2010; Balyo et al., 2015). In the following, we mention some common diversification strategies.

- *Supplying different random seeds to solvers.* This simple measure lets solvers take different branches when taking random decisions (Balyo et al., 2015).
- *Setting initial variable phases.* When a SAT solver selects a variable to assign, its *phase* decides *which* value to assign. If initial variable phases are set differently, solvers may explore search space in a different manner (Hamadi et al., 2010).
- *Using different solver parameters.* Solvers commonly have a large set of configuration options, e.g., restart intervals (Hamadi et al., 2010), pre- and inprocessing options (Biere, 2010) or clause database management (Audemard & Simon, 2017).
- *Using different SAT solvers.* Perhaps the strongest diversification technique, employing wholly different SAT solving algorithms and/or implementations tends to lead to very different search behavior across the portfolio members (Xu et al., 2012). However, this diversification technique is limited with respect to the portfolio size, since there is only a limited number of SAT solvers that benefit a portfolio.[4]

3.2.1 Clause Sharing

Going back to our puzzle analogy, consider now that our experts occasionally hold brief meetings, where each expert has the opportunity to share the insights they gained since the last meeting, e.g., a partial solution to the puzzle. All experts then continue to work independently, free to include the other experts' insights into their own reasoning.

The described "meetings" in our assembly of experts correspond to what we know as *clause sharing* in parallel SAT solving. While clause sharing has been introduced in the context of search space partitioning solvers (see Section 3.1), it is clause-sharing *portfolio* solvers which tend to perform the best in terms of general-purpose parallel SAT solving (Balyo et al., 2016, 2017; Froleyks et al., 2021). Intuitively, each shared clause has the potential to prune search space, and sharing promising pruning opportunities can reduce redundant work and lead to significant acceleration (Hamadi et al., 2010; Balyo et al., 2015). That being said, clause sharing also adds overhead in terms of computation, synchronization and/or communication. Since the number of produced conflict clauses is linear in the number of solvers (disregarding duplicates), parallel solvers need to prioritize which clauses to share (Hamadi et al., 2010; Audemard et al., 2012; Ehlers et al., 2014). Most commonly, the following two metrics are considered:

3. This neglects slowdowns from running solvers in parallel due to resource contention (Aigner et al., 2013).
4. Bach et al. (2022) computed optimal pure portfolios of k solvers based on data from the 2020–2021 competitions. Their results indicate stagnating performance beyond $k \approx 20$.

- *Clause length.* This metric follows a simple intuition: The fewer variables are part of a conflict, the larger the subspace of assignments that the conflict constrains. Shorter clauses are thus more likely to be useful to another solver. Using clause length to prioritize clauses to share ranges back to PaSAT (Sinz et al., 2001). The first portfolio solver ManySAT (Hamadi et al., 2010) shared clauses of length ≤ 8.
- *Literal block distance* (Audemard & Simon, 2009). This metric, also called LBD or *glue* value, indicates how many distinct decisions had to be made to result in this conflict. Low-LBD clauses are more likely to be useful again in the search procedure (Audemard & Simon, 2009). In contrast to clause length, LBD is a metric that depends on the particular solver state in which the clause was derived and may also be updated during subsequent search (Audemard & Simon, 2009).

Beyond these quality metrics, Audemard et al. (2012) proposed to have solvers assess incoming clauses based on how well they fit to the solver's current variable phases, and Vallade et al. (2020) proposed a metric for clauses based on the formula's community structure. Since the distribution over produced clauses quality may vary over time and across inputs, imposing fixed quality thresholds (e.g., Hamadi et al., 2010; Biere, 2010) can result in too many or too few shared clauses (Hamadi et al., 2012). Therefore, adaptive quality limits have been proposed. For instance, Hamadi et al. (2012) introduced an adaptive control scheme where the threshold for clause sharing evolves over time between each pair of solvers, and HordeSat (Balyo et al., 2015) features an initially very strict LBD limit for each solver that is lifted successively until the targeted output volume is reached.

Katsirelos et al. (2013) argue that the scalability of clause sharing is intrinsically limited. Intuitively, proving a formula's unsatisfiability requires a *network of clausal dependencies*. All clauses along a critical path through this network need to be derived sequentially. Katsirelos et al. (2013) show that typical proofs output by sequential solvers feature clauses that are part of *all* critical paths and therefore present bottlenecks for a parallel derivation. While this study does indicate that some problems are intrinsically hard to parallelize with clause sharing, it is difficult to estimate its practical consequences for today's solvers. Most importantly, Katsirelos et al. (2013) assume that a parallel solver needs to reproduce the proof that the sequential solver found. In reality, many application instances may allow parallel solvers to find another proof of similar volume but with fewer (perceived) bottlenecks (*cf.* Fossé & Simon, 2018). It is also worth considering that the majority of clauses learned by a sequential solver do not actively contribute to its final proof (Simon, 2014), indicating that the resolution steps taken by sequential CDCL solvers are suboptimal as well. Ehlers and Nowotka (2019) argue that scalability limits for individual instances are of limited concern if the main objective is to achieve good performance in terms of *weak scaling*, where the difficulty of problems increases together with the degree of parallelism.

3.2.2 An Overview of Clause-Sharing Portfolio Systems

Hamadi et al. (2010) introduced parallel portfolio SAT solving with ManySAT. It featured two crucial ingredients adopted by most later portfolio solvers: *Diversification* of individual solvers and *clause sharing* across solvers. While ManySAT was fine-tuned to four cores only, it initiated what can be considered a paradigm shift in parallel SAT solving. Since 2010, portfolio solvers such as Plingeling (Biere, 2010), SArTagnan (Kottler & Kaufmann, 2011), and PeneLoPe (Audemard et al., 2012) began to dominate competitions (Balyo & Sinz, 2018). An extreme case worth mentioning is ppfolio (Roussel, 2012)—a script that just ran several winners from a past competition in parallel. For many years Plingeling was considered the best performing parallel SAT solver (Balyo & Sinz, 2018). It runs configurations of the sequential solver Lingeling (Biere, 2010) and, in later versions, the local search solver YalSAT (Biere, 2014). Lingeling instances are diversified in terms of restart scheduling, inprocessing options, and initial variable phases. Recent Plingeling versions share learned clauses up to length 40 and LBD 8 (Biere, 2013).

Audemard and Simon (2014) proposed portfolio solver Syrup based on the sequential solver Glucose (Audemard & Simon, 2009). Syrup introduced (a) cautious clause exchange that only considers a clause for sharing after it has been locally encountered twice, and (b) putting each incoming clause in a *probation* where only one of its literals is watched. Only if this literal is falsified, the clause is *promoted* to be handled normally. These techniques reduce the volume of exchanged clauses substantially (Audemard & Simon, 2014) and were (partially) adopted by later systems (Audemard et al., 2016; Ehlers & Nowotka, 2019). With the distributed version of Syrup, named D-Syrup, Audemard et al. (2017) emphasized the benefit of grouping multiple solver threads in a single process. In addition, they observed degrading performance the more often all-to-all clause sharing is performed, thus replacing it with more frequent point-to-point clause exchange.

Ehlers et al. (2014) explored massively parallel SAT solving with TopoSAT, which also orchestrates modified Glucose instances. Solver threads export clauses after some time since the last sharing passed *or* when their internal export buffer runs full. TopoSAT can also follow a lazy clause exchange policy that only exports clauses after at least four process-local solvers deemed this clause relevant (Ehlers & Nowotka, 2019). Clauses are exchanged via point-to-point messaging along communication graphs (Ehlers et al., 2014). At import, each incoming clause is attributed an LBD value equal to the clause's length—a conservative upper bound. To our understanding, TopoSAT 2 in the 2020 competition (Ehlers et al., 2020) has each solver process send each batch of clauses to every other solver process, which implies a quadratic number of messages.

Balyo et al. (2015) introduced a generic framework for clause-sharing portfolios with HordeSat. Its modular solver interface allows to orchestrate different SAT solvers without changing their internal workings. Each process runs multiple solver threads (four in the original evaluation). Dedicated threads are used for communication across processes—solvers never need to be synchronized. HordeSat performs periodic all-to-all clause exchange via a collective operation named *all-gather*: Each process writes its best locally produced clauses into a fixed-size buffer, and all such buffers are concatenated to a single buffer that is then sent to all processes. Initially only clauses with LBD score ≤ 2 are considered for export. A process lifts this restriction successively whenever it is not able to contribute the expected volume of clauses. HordeSat performs approximate filtering of previously seen incoming clauses using Bloom (1970) filters. Balyo et al. (2015) evaluated HordeSat on up to 2048 cores, which to our knowledge is the largest evaluated scale of a portfolio solver prior to our work. HordeSat reached a median speedup of 13.8 at 1024 cores (13.1 at 2048 cores) on ISC application benchmarks. Balyo et al.'s (2015) claim that HordeSat achieves "*superlinear average speedup for difficult instances*" is problematic since the underlying speedup metric is statistically not meaningful—see Section 3.5.2.

Partially inspired by HordeSat, the PaInLeSS framework (Le Frioux et al., 2017a) offers an even more modular architecture to "painlessly" develop and explore parallel SAT solvers. It offers parallelization via portfolios, clause sharing, and search space partitioning. PaInLeSS adopted many of HordeSat's features, such as diversification based on sparsely and randomly setting variable phases, periodic all-to-all clause exchange of 1500 literals per solver unit per sharing, and an adaptive LBD limit for exporting clauses based on the volume of produced clauses. PaInLeSS portfolios using MCOMSPS[5] as a backend solver (Le Frioux et al., 2017b; Vallade et al., 2020, 2021) were some of the best performing parallel approaches in recent competitions (Froleyks et al., 2021) and also feature concurrent clause strengthening in dedicated threads (Vallade et al., 2020, 2021). There have also been efforts to extend PaInLeSS to distributed solving (Vallade et al., 2021).

A shared-memory solver performing very well in 2022, ParkissatRS (Zhang et al., 2022), also builds on top of the PaInLeSS framework but uses a variant of Kissat (Biere et al., 2020) as a solver backend. Diversification is achieved by randomly shuffling the branching order of decision variables. The successor to this system, named PRS (Zhang et al., 2023), enhances this diversification and also features a distributed setup, where clause sharing across processes is performed via unidirectional ring communication.

Fleury and Biere (2022) recently presented Gimsatul, a system with a remarkably different architecture compared to most portfolio solvers. Clauses are not copied across different solver threads but truly shared physically. Consequently, Gimsatul is written from scratch instead of reusing existing solvers. This approach led to a decreased memory footprint as well as to near-linear self-speedups for up to 16 cores. While Gimsatul's architecture is restricted to shared memory parallelism, linking multiple Gimsatul instances with distributed clause sharing may be a possibility.

3.3 Processing Multiple SAT Tasks in Parallel

In general, processing several SAT formulas in parallel constitutes a more efficient use of computational resources than using all resources for a single formula at a time. The argument supporting this claim is simple: Since mean speedups reported by general-purpose distributed SAT solvers are strongly sublinear, solving k formulas with p/k processing elements each leads to higher efficiencies than solving one formula at a time with p processing elements. The most basic way to achieve this is by running p sequential solvers concurrently to process p formulas (*cf.* Aigner et al., 2013). Recently, Biere et al. (2022) proposed a mechanism to migrate the internal state of a (sequential) SAT solver from one machine to another, which can allow the preemption and/or rescheduling of such independent tasks.

Ngoko et al. (2017) presented a distributed system for cloud environments: A centralized scheduler uses running time predictions to compute an offline schedule that features stages of portfolio solving. No clause sharing is performed, with the reasoning that "*such solutions [for exchange of knowledge] are not necessarily suitable for distributed clouds in which the communication time could be important*" (Ngoko et al., 2017).

The C&C solver Paracooba (Heisinger et al., 2020) supports parallel processing of multiple jobs and *malleable load balancing*: The partitioning of a problem into many cubes

5. MapleCOMSPS (Liang et al., 2016b), or MCOMSPS, is a solver based on CoMinisat-PS (Oh, 2016) with the LRB heuristic from MapleSAT (Liang et al., 2016a) and some further changes.

allows to dynamically resize ongoing SAT tasks if compute nodes register or unregister from a computation. Ozdemir et al. (2021) proposed a C&C platform for "serverless cloud" setups. In terms of general purpose SAT solving, these approaches can suffer from uneven work subdivision just like other C&C approaches (Section 3.1), which we circumvent by designing an approach based on clause-sharing portfolio solving.

3.4 Other Massively Parallel Approaches

We briefly refer to some other massively parallel approaches to SAT solving which are outside of the focus of this work. Arbelaez and Codognet (2012) presented a parallelization of local search SAT solving (see Kautz et al., 2021) on up to 256 cores. In recent years, graphical processing units (GPUs) have become increasingly prevalent in HPC environments and, as of now, contribute significant amounts of today's top supercomputers' performance (Khan et al., 2021). Recent works aim to exploit GPUs for selected SAT solving tasks, ranging from concurrent clause strengthening (Prevot et al., 2021) over parallel inprocessing (Osama et al., 2023) to massively parallel local search (Cen et al., 2023).

3.5 On Evaluating SAT Solver Performance

We now discuss how the performance of (sequential and) parallel SAT solvers is commonly assessed and add some of our own considerations.

3.5.1 Performance Metrics

The performance of a SAT solver is examined by running it on a fixed set of benchmark problems at a fixed timeout T per instance. One of the most popular metrics to analyze the resulting data is the number of solved instances, sometimes referred to as *solved count*. This metric is often reported graphically for *all* timeouts in the interval $[0, T]$ (e.g., see Fig. 1). Since solvers behave differently on satisfiable vs. unsatisfiable problems (Oh, 2015), it is useful to separate the number of solved satisfiable and unsatisfiable instances.

Another commonly used metric is called *Penalized Average Runtime* (PAR). For any integer X, the PAR-X score of a run is defined as its arithmetic mean running time over all considered instances, where each timeout is attributed a running time of X times the time limit (Froleyks et al., 2021). The value of X weighs the solved count versus the average running time on solved instances. PAR-1 pretends that all unsolved instances are solved at the time limit, and PAR-X for $X \to \infty$ ranks solvers according to their solved count. Other common PAR metrics are PAR-10 (e.g., Arbelaez & Codognet, 2012; KhudaBukhsh et al., 2016) and PAR-2 (e.g., Froleyks et al., 2021; Bach et al., 2022).

PAR scores are suited to aggregate a solver's performance on many instances to a single value and thus to compare different solver systems. However, PAR scores can be vulnerable to noise introduced by running time variances. The underlying issue is that PAR-X for $X > 1$ features discontinuities: A minor variation in the running time on some instance can result in hitting the time limit, which incurs a penalty and therefore leads to a jump in the score. This effect is amplified if a small time limit is used. In our research, we noticed this effect mostly for satisfiable instances, which are well-known to result in higher running time variance compared to unsatisfiable instances (Ohmura & Ueda, 2009; Simon, 2014;

Oh, 2015). Averaging multiple runs of each configuration can be useful to reduce variances but can also be costly depending on the setup.

In cases where PAR scores and the sets of solved instances are sufficiently similar, we will consider an additional metric that we refer to as *Commonly Solved Average Runtime* (CSAR). Given a set of competitors, we identify the set of instances that *all* competitors were able to solve and then compute each competitors's arithmetic mean running time on those instances. While CSAR neglects additional solved instances, it still complements PAR in a useful manner since it features significantly less noise. In contrast to PAR scores, CSAR scores cannot be compared across different sets of experiments.

3.5.2 Speedup Metrics

In parallel computing, we aim to quantify the benefit of a parallelization over a sequential baseline. Comparing (i.e., dividing) the achieved PAR scores can be an option but is rather unintuitive since it intermingles running times and unsolved instances. The preferable option is to make use of different *speedup metrics*, which we discuss in the following.

For a single input $\mathcal{I}$ and p cores, the *speedup* of a parallel algorithm A_{par} with running time $T_{\text{par}}(\mathcal{I}, p)$ over a sequential algorithm A_{seq} with running time $T_{\text{seq}}(\mathcal{I})$ is defined as $s(\mathcal{I}, p) = T_{\text{seq}}(\mathcal{I})/T_{\text{par}}(\mathcal{I}, p)$. The *efficiency* of A_{par} is defined as $E(\mathcal{I}, p) := s(\mathcal{I}, p)/p$. A parallel algorithm has *linear scaling* if $s(\mathcal{I}, p) \in \Theta(p)$ and scales *perfectly* if $s(\mathcal{I}, p) = p$, i.e., $E(\mathcal{I}, p) = 1$ (Sanders et al., 2019). *Superlinear speedups* $s(\mathcal{I}, p) > p$ are possible if the parallel algorithm profits from additional resources, e.g., having access to more memory in total (Sanders et al., 2019, p. 62) or if a sequential scheduling of the parallel execution threads of A_{par} constitutes a better sequential algorithm than A_{seq} itself. In the context of SAT solving, speedups that are achieved on isolated instances should be treated with caution: Even minor changes of a solver or an input (or merely the non-determinism of parallel execution) can sometimes lead to running time variations by several orders of magnitude. This effect is likely to yield some very large speedups, including superlinear speedups.

In general, a meaningful number of speedups across a diverse benchmark set B should be observed and accumulated. An initial idea for such an accumulation might be to compute the *arithmetic mean of speedups* $\frac{1}{|B|}\sum_{\mathcal{I}\in B} s(\mathcal{I}, p)$ (Balyo et al., 2015; Balyo & Sinz, 2018). This, however, is not adequate since arithmetic means over normalized values or ratios are meaningless (Fleming & Wallace, 1986). For example, if algorithm A compared to algorithm B is 10× faster on instance $\mathcal{I}_1$ but 10× slower on instance $\mathcal{I}_2$, the arithmetic mean speedup of A over B and of B over A are *both* $(10 + 0.1)/2 = 5.05$. The statistically sound way to compute a mean speedup is the *geometric mean* $s_{geom} := (\prod_{\mathcal{I}\in B} s(\mathcal{I}, p))^{1/n}$ (Fleming & Wallace, 1986), which is used rarely in SAT research.

The *median speedup* s_{med} is defined as the $\lfloor \frac{|B|}{2} \rfloor$-th speedup from a sorted list of speedups $s(\mathcal{I}, p), \mathcal{I} \in B$. While this value can be unsatisfactory if B contains many easy instances where parallelization does not pay off (Balyo & Sinz, 2018), median speedups on such benchmark sets can also indicate the amount of overhead a parallel solver incurs.

The *total speedup* $s_{tot} := \sum_{\mathcal{I}\in B} T_{\text{seq}}(\mathcal{I}) / \sum_{\mathcal{I}\in B} T_{\text{par}}(\mathcal{I}, p)$ is the speedup with respect to the total time spent by A_{par} vs. A_{seq} on the entire benchmark set (Balyo et al., 2015). While this is a sensible metric with a direct and intuitive meaning, total speedups put a large emphasis on the instances that took a long time. Therefore, it should not be misinterpreted as a kind of "average speedup" or expected speedup for a single instance.

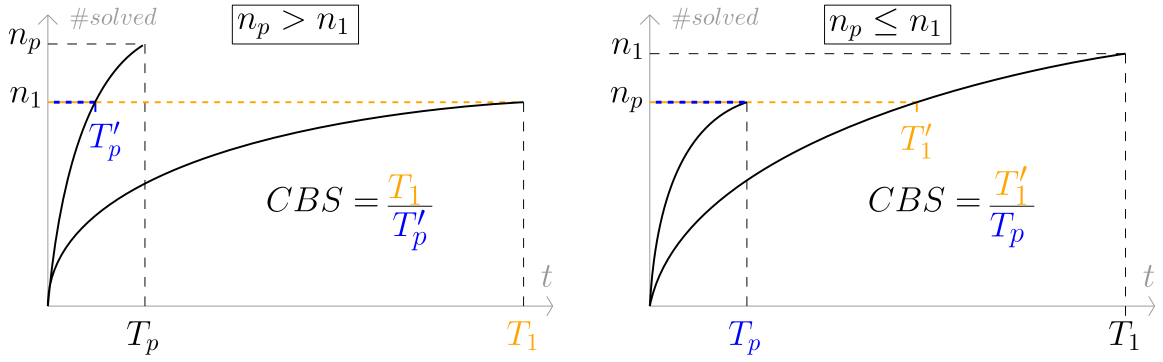


Figure 1: Illustration of Count-Based Speedup (CBS, Balyo et al., 2015). Each x coordinate denotes a per-instance time limit and each y coordinate denotes the solved count with this limit. Assume that a parallel solver solved n_p instances within T_p seconds and a sequential baseline solved n_1 instances within T_1 seconds. The definition is split into two cases, $n_p > n_1$ (left) and $n_p \leq n_1$ (right).

The last speedup metric we discuss is the *Count Based Speedup* (CBS) by Balyo et al. (2015). As illustrated in Fig. 1, we compute the ratio between the time limit of the approach that solved fewer instances and the time limit required by the other approach to solve equally many instances. CBS does not require to handle one-sided timeouts (see Section 3.5.3) and can be used to estimate the acceleration *any* improved approach brings over a baseline, regardless of the degree of parallelism involved. Since the set of solved instances may differ between the solvers, CBS can indirectly account for additional solved instances. A drawback of CBS is its sensitivity to outliers: If solver A solves 100 instances within 1 s and solver B solves 99 instances within 1 s but one instance in 1000 s, then the CBS of A over B is 1000 even though A only performs better on a single instance.

3.5.3 Handling One-sided Timeouts

A parallel solver often solves more instances than a sequential solver, leading to incomplete data when comparing per-instance performance. This raises the question of how to calculate speedups in such cases. One option is to only consider the instances that both the sequential solver and the parallel solver were able to solve. This neglects parts of the merit of the parallel solver but results in truthful speedup measures for the considered set. Another option is to assume each of the sequential solver's timeouts to be solved exactly at the time limit (Balyo et al., 2015) like PAR-1 scores. This can be combined with running the sequential solver for a longer time than the parallel solver to avoid under-approximating the sequential running times by too much. In particular, setting the sequential time limit to p times the time limit of the parallel solver at p cores results in equal CPU time budgets for both approaches. That being said, we assume that there are instances that can be solved realistically only by a parallel solver but not by a sequential solver, e.g., because a huge set of learned clauses is required that a single solver cannot realistically maintain. Reporting such instances as solved after an impractically high time limit may inflate the reported speedups or render them difficult to interpret. For this reason, our preferred approach is to keep speedups and additional solved instances separate.

3.5.4 Weak Scaling

Weak scaling (Gustafson, 1988) denotes a parallel algorithm's scaling behavior for cases where the work at hand increases together with the degree of parallelism. It accounts for the fact that investing large amounts of resources is commonly tied to accordingly large inputs. Balyo and Sinz (2018) suggest to transfer weak scaling to parallel SAT solving as follows: If A_{par} was run with c cores, then only consider the instances that took A_{seq} at least $k \cdot c$ seconds to solve for some constant k. Evaluating the resulting speedups for different scales can give an impression on how well a solver scales to larger inputs. Large k can result in a very small number of considered instances, especially if only commonly solved instances are considered, which can add noise and amplify few large speedups (*cf.* Balyo et al., 2015). We thus suggest to consider multiple values of k for more robust results. As a variant, we can consider all instances with a minimum sequential running time x and then graph the mean/median/total speedup for each such $x \in [0, T_{\text{seq}}]$ (see Fig. 11).

4. Overview

In the following, we provide a high-level introduction to our approach.

Some of our design decisions are directly adopted from HordeSat (Balyo et al., 2015). First, we follow a modular portfolio solver design with a generic and compact interface to which different sequential solver backends can be connected. This "blackbox approach" allows a distributed solver to profit from a large pool of diversification and can also help to transfer progress in sequential SAT solving to distributed solving with little effort, in contrast to systems that have a tighter integration with a specific sequential solver—e.g., TopoSAT (Ehlers et al., 2014) or D-Syrup (Audemard et al., 2017) with Glucose.

Secondly, we follow a two-level parallelization model where multiple (distributed) *processes* communicate via message passing and each process features multiple *solver threads*. In this point we will go further than HordeSat, which was tuned to spawn a process for each set of four cores, by targeting process layouts that are faithful to the hardware at hand. In other words, we aim to group all cores of a socket into a single process, which requires careful use of concurrent data structures given that a socket can feature dozens of cores.

Thirdly, we perform all communication in dedicated threads so that solvers do not need to be interrupted. Again, we will go further than HordeSat by performing *asynchronous communication*. As a consequence, even the communication thread of a process can perform other tasks while a collective operation is being performed.

In contrast to HordeSat, we do not assume any kind of shared memory (disk or RAM) across processes. For this reason, we let a particular process parse the input formula and then broadcast the data in a serialized form to all involved processes.

Furthermore, we require our computation to be *malleable*. Malleability is the ability of a parallel task to handle a fluctuating amount of processing elements *during* its execution (Feitelson, 1997). In particular, we use the job model of Mallob (Sanders & Schreiber, 2022). Mallob represents each task j as a *binary tree* T_j of $v_j \geq 1$ processes that, at any time, may be directed to expand or shrink to an updated size v'_j. A process joining j receives the input formula from its parent node in T_j and then initializes and runs our multi-threaded solver engine. A process leaving j suspends the engine but does not necessarily terminate it, since it might be able to re-join j at a later point in time. Communication across a task's processes can be performed along the binary tree structure of T_j.

5. Clause Sharing

In the following, we describe our approach to clause sharing, including central design decisions, the exchange operation itself, and the buffering and filtering of clauses.

5.1 Design Decisions

We now present our view on clause sharing and subsequent design decisions.

We see a distributed solving procedure as a collaborative effort of many different experts. In this collaboration, clause sharing acts as a kind of *search space pruning*: If a single solver is able to find a crucial conflict c that has not yet been found by any other solver, broadcasting c prevents other solvers from exploring the subspace pruned by c. Following this intuition, we explore *all-to-all* clause sharing rather than limiting the receivers of a given clause to a subset of solvers (*cf.* Ehlers et al., 2014). Prior all-to-all clause sharing (Balyo et al., 2015; Audemard et al., 2017; Ehlers & Nowotka, 2019) does suffer from problems in terms of scalability and/or quality, which we intend to overcome with algorithmic improvements. Furthermore, in line with prior observations,[6] we assume that clause sharing works best if the mean *clause turnaround time*, i.e., the time it takes for a produced clause to be imported by another solver, is as low as possible. Intuitively, the longer it takes for a clause to be shared and imported, the higher the probability that other solvers redundantly performed the same work. We thus aim to reduce clause turnaround times.

On a technical level, in large parts we adopt HordeSat's information flow (Fig. 2). Solver threads write produced clauses into an *export buffer*. Periodically, each process *flushes* clauses from its export buffer and contributes them to a *collective sharing operation*. Each process then receives the aggregated *sharing buffer* and forwards the featured clauses to its solver threads. Clauses may be filtered or selected at several points of this procedure.

A central design consideration of clause sharing is which and how many clauses to share (Section 3.2.1). When sharing more and more clauses in a distributed solver, we assume that the merit of clause sharing eventually levels off whereas communication and computational overhead continue to increase (Ehlers et al., 2014; Balyo et al., 2015; Ehlers & Nowotka, 2019). For this reason, we aim to limit clause sharing to a point where its merit is nearly maximized while avoiding any unnecessary overhead. We believe that fixing the volume of the *globally best distinct* clauses is a robust and effective means to control the degree of clause sharing. In addition, this global view allows clause sharing to seamlessly adapt to the instance at hand and to the current state of solvers—effectively enforcing dynamic quality criteria (*cf.* Hamadi et al., 2012) rather than fixed clause length or LBD thresholds.

6. "*sharing clauses, as soon as possible, among search units provides better results*" (Audemard et al., 2017); "*it appears [...] important to exchange learnt clauses fast*" (Ehlers & Nowotka, 2019)

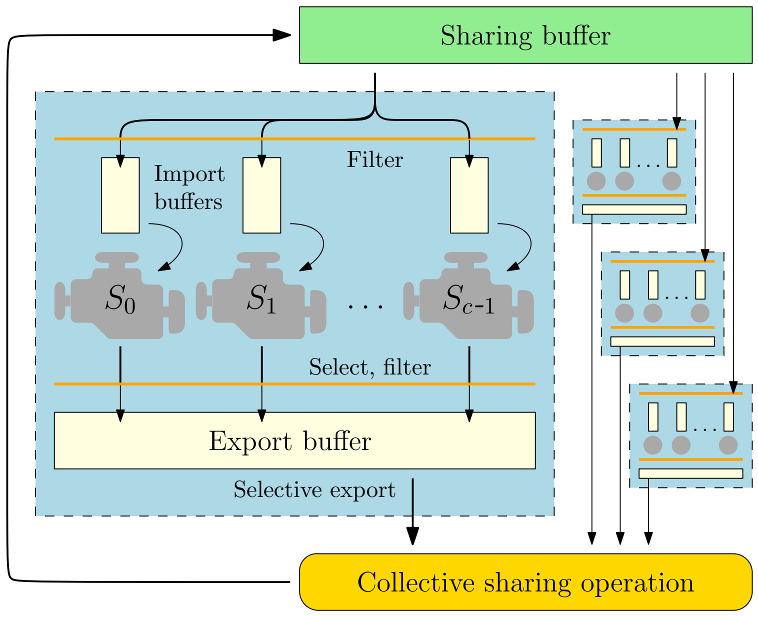


Figure 2: Information flow in the clause sharing of HordeSat and also our system. Each dashed rectangle corresponds to a solver process. One of these processes is depicted in more detail with c solvers, clause buffering and filtering.

Under the view of clause sharing as search space pruning, we believe that clause length is a sensible metric for clause quality. Without any further assumptions, shorter clauses in general prune larger parts of search space and therefore provide the most valuable and most densely encoded information to other solvers (see Section 3.2.1). LBD values, which are a crucial clause quality metric in sequential SAT solving, depend on the producing solver's internal state and are thus not necessarily as meaningful on a global scale, i.e., for all receiving solvers with varying internal states. As such, we design our clause sharing to prefer short clauses first and to merely break ties using LBD values.

5.2 Clause Exchange Operation

We now discuss the collective (communication) operation that forms the basis for clause sharing, beginning with HordeSat and then presenting our own approach.

5.2.1 HordeSat

HordeSat periodically performs an all-to-all clause exchange using a collective operation named *all-gather*: Each process i writes a selection of locally produced clauses into a buffer b_i of fixed size $\beta_0 := |b_i|$. The concatenation of all b_i, the *sharing buffer* B, is broadcast to all processes. Then each process can forward the shared clauses in B to its solvers.

The above clause exchange mechanism has various shortcomings. First, whenever a process i does not completely fill b_i, parts of the sharing buffer B remain unused and carry no information (Audemard et al., 2017). HordeSat attempts to remedy this by initially only considering very low-LBD clauses and then successively lifting this restriction for under-producing processes. This, however, does not work reliably in all cases and also implies an initial "warmup phase" where many decent clauses may be discarded. A second problem is that B can contain a significant portion of duplicate clauses. In particular, we noticed that the clause sets exported by our solvers in the first few sharings are often very similar. This effect is especially pronounced for unit clauses, for which HordeSat never filters duplicates. Thirdly, the targeted sharing volume is proportional to the number of involved processes. For sufficiently large setups, this may constitute a bottleneck both in communication volume and in the local work necessary to handle every clause received.

5.2.2 Compact Clause Exchange

Fig. 3 illustrates our compact clause exchange operation. We assume a distributed binary tree T of processes as a communication structure (see Section 4). At each clause exchange, each leaf node i in T exports clause buffer b_i of size $|b_i| \leq \beta_0$ and sends b_i to its parent node in T. As soon as an inner node received as many buffers as it has children, it exports its own clauses and then performs a two- or three-way *merge* of the present buffers: All input buffers are read simultaneously from left to right and aggregated into a single sorted output buffer, similar to merging sorted sequences in textbook Mergesort (Sanders et al., 2019, p. 160). We keep the clauses in each buffer ordered by length, then by LBD and finally in lexicographic order. We sort the literals within each exported clause, which brings them into a canonical form and helps to recognize and filter duplicates (e.g., clause "c" in Fig. 3).

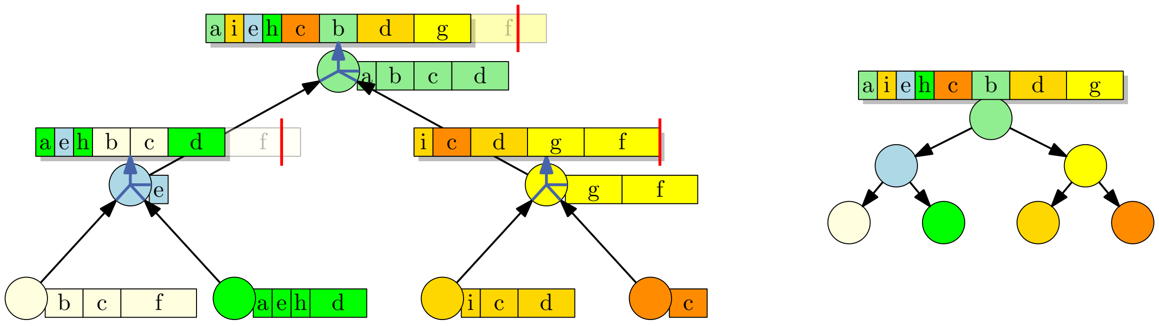


Figure 3: Example for our clause exchange with seven processes. Each circle denotes a process. Sharing buffer B is aggregated (left) and then broadcast (right). Each boxed letter denotes a clause; the size and color of a box indicate the clause's length and origin respectively. Each vertical red bar denotes the limit on a merged buffer's size, which results in dismissing a clause ("f") in two cases.

In the merge operation at a node of T that is the root of a subtree with u nodes, we *limit* the size of the output buffer with a function $b(u)$. Any remaining clauses in the input buffers that exceed the limit $b(u)$ are inserted into the local export buffer $\mathcal{B}$ (Section 5.3.2) where they may be re-exported at a later point in time. As our clauses are always ordered by quality, we aggregate some of the globally most valuable information while imposing a strict limit on the sharing volume. We also ensure high density of useful information in the transferred data because each sent buffer is of compact shape and free of duplicates.

Malleable implementation. Our compact clause exchange is straight forward to make malleable as described in Section 4. In our implementation, a clause exchange is initiated by the root of T broadcasting a signal through T. Even if T is modified while this broadcast takes place, the set of processes $\tilde{T}$ receiving the signal always constitutes a valid binary tree of processes. This "snapshot" $\tilde{T}$ of T defines the participants of our aggregation operation. Inner nodes in $\tilde{T}$ expect a contribution of clauses from each child in $\tilde{T}$, and leaves in $\tilde{T}$ prepare to export and send clauses to their parent. If a process leaves the computation before it can send clauses, it sends an empty buffer to their parent in $\tilde{T}$ to ensure that the aggregation progresses. The broadcast of the sharing buffer, by design, then reaches all processes that belong to T at *that* point in time.

Buffer limit scaling. For our current setup,[7] we modeled the *buffer limit function* $b(u)$ based on three simple constraints. For small setups, the buffer size should be proportional to the number u of processes. Therefore, the value and the derivative of $b(u)$ at $u = 1$ should both be equal to β_0: (i) $b(1) = \beta_0$ and (ii) $b'(1) = \beta_0$. For sufficiently large setups, the buffer limit should converge to a certain upper bound β_∞. Therefore, (iii) $b(u) \to \beta_\infty$ for $u \to \infty$. A simple function that satisfies constraints (i) and (iii) is

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

for $k \in \mathbb{R}^+$. To satisfy constraint (ii), we set $k := \frac{\beta_0}{\beta_\infty - \beta_0}$ and therefore[8] arrive at

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

At large scales, where $b(u)$ approaches β_∞ and stops increasing, note that we may still profit from adding solvers to the computation—not in the sense of sharing more clauses, but in the sense of sharing *better* clauses.

5.3 Clause Buffering

In distributed clause sharing, the journey of a produced clause from its original solver to the clause database of another solver features several steps of buffering where the clause may be deferred or discarded for various reasons. We describe this journey for HordeSat and then propose our improvements. Our aim is to discard a clause c only if the volume of shareable clauses of equal or better quality than c already exhausts the sharing budget.

5.3.1 HordeSat

In HordeSat, clauses of sufficient quality exported by local solvers are written into an *export buffer* structure $\mathcal{B}$ that features *buckets*, one for each admissible clause length. Each such bucket is a stack of fixed capacity that stores each clause together with its individual

7. Note that earlier versions of our system used a deviating buffer limit function based on applying a certain *discount* $\alpha < 1$ at each further level of the reduction tree (Schreiber & Sanders, 2021). The function described here proved to be more ergonomic. We refer to Schreiber (2023b) for a discussion.
8. Since $b'(u) = k(\beta_\infty - \beta_0)e^{-k(u-1)}$, $b'(1) = \beta_0$ yields $k(\beta_\infty - \beta_0) = \beta_0$ and thus $k = \beta_0/(\beta_\infty - \beta_0)$.

LBD value. If a bucket is full, any further clauses of this length will be discarded until the next export of clauses. At each export, the buffers are flushed by decreasing clause quality until the local export volume is met or no more clauses remain. A consequence of this simple strategy is that some buckets running full may result in losing potentially useful clauses. Another consequence is that completely filled buckets preserve the first (oldest) clauses that have been produced while more recent clauses are discarded.

Regarding the import of incoming clauses, the main thread of a HordeSat process copies all admitted clauses from clause sharing into an *import buffer* $\mathcal{B}_S$ for each solver S and increases its size as necessary. S can then import the clauses in $\mathcal{B}_S$ at its own discretion. $\mathcal{B}_S$ is guarded by a mutex that is locked by the solver thread before reading clauses and by the main thread before writing clauses. If S cannot acquire this lock, it retries at a later point in time. If S does not import clauses sufficiently fast, $\mathcal{B}_S$ may increase in size indefinitely. We noticed that certain solvers can spend minutes in expensive preprocessing routines without retrieving shared clauses, which leads to very large import buffers at times.

5.3.2 Adaptive Clause Buffering

We observed that the statistical distribution over the length of clauses exported by a solver depends on many variables, such as the input formula, the type and configuration of the solver, and the point in time during solving. A clause buffering structure with fixed-size buckets for each clause length such as HordeSat's may thus be suboptimal.

Following this intuition, we aimed at a more dynamic allocation of space across the buckets in the export buffer $\mathcal{B}$. We again implement $\mathcal{B}$ as an array of buckets, one bucket for each clause length $l \geq 1$. Each bucket is implemented as a stack of clause literals. A single *budget* integer shared by all buckets represents the remaining number of literals that can still be inserted until $\mathcal{B}$ is full. If this budget is insufficient for inserting a given clause c of length l, an attempt is made to discard clauses from a bucket $l' > l$ in order to "steal" space for c. If this is unsuccessful for all admissible l', c is discarded. At each *flush* operation, all stacks that carry less than half their capacity are shrunk. With this flexible buffering structure, we balance the available space dynamically among the different clause quality levels and discard any clauses below a certain quality threshold. This threshold is determined indirectly by the distribution over the local solvers' produced clause lengths.

We do not insert produced clauses beyond length 60 in $\mathcal{B}$. Note that long clauses with tens of literals are only admitted if the mean produced clause length is accordingly high, which we did observe on some instances. In terms of LBD scores, we dropped HordeSat's method to only admit low-LBD clauses to $\mathcal{B}$ and to successively lift this limit for underproducing solvers. Steering the export volume per process is not as crucial for our approach, and in fact, MallobSat achieved better performance without this method in early tests (Schreiber & Sanders, 2021). We do use separate buckets for each length-LBD combination up to a certain clause length, allowing us to use LBD scores as tie breakers during export.

We noticed that solvers occasionally produce tens of thousands of unit clauses in a single burst, which overburdens $\mathcal{B}$ and results in losing most produced clauses. For this reason, we allow the buffers to store indefinitely many unit clauses while keeping the shared budget for all other slots. Even keeping all derivable unit clauses of the problem in main memory is not a problem since the representation of F itself is strictly larger.

We use the same data structure as $\mathcal{B}$ for each import buffer $\mathcal{B}_S$. This way, the buffering of incoming clauses is robust towards solvers that may not import clauses for a long period of time: $\mathcal{B}_S$ drops the clauses of worst quality if it runs full in such cases.

5.4 Clause Filtering

Tied to clause sharing, the *clause filtering problem* is to decide for a shared clause c and a solver S whether S has received or produced c before and should therefore not receive c (*cf.* Balyo et al., 2015). Since solvers forget most redundant clauses over time and may in some cases benefit from re-learning crucial clauses (Audemard & Simon, 2014; Balyo et al., 2015), a possible variant is to filter such clauses not indefinitely but to rather re-allow their sharing after some amount of time or some number of sharing operations have passed.

5.4.1 HordeSat

HordeSat's clause filtering is realized with approximate membership query (AMQ) data structures (Balyo et al., 2015), specifically *Bloom filters* (Bloom, 1970). Each process employs one *node filter* and t *solver filters* (one for each solver thread). At clause export, each clause is registered in its solver filter and then tested against the node filter. At clause import, each clause is tested against the node filter and then against each solver filter. The usage of AMQs implies that false positives may occur, leading to the rejection of some potentially useful clauses that have in fact not been shared before. This risk of false positives motivated Balyo et al. (2015) to skip filtering for unit clauses due to their importance. This can be problematic because some unit clauses are produced redundantly by many solvers and therefore waste considerable amounts of space in the sharing buffers.

5.4.2 Base Approach

In our first approach, we adjusted HordeSat's clause filtering to align it with our new clause sharing operation. We omitted node filters because their main use is to check for duplicate clauses *across* processes, what is already done during the aggregation of buffers in our case. We complemented the solver filters with an additional filtering of unit clauses, using an exact set instead of an AMQ data structure. This way we do not get any false positives for unit clauses and make sure that each such clause is being shared once.

We also implemented a mechanism where every X seconds, half of all clauses (chosen randomly) in each clause filter are forgotten and thus can be shared again. However, this probabilistic forgetting can result in a "degenerating" AMQ and empirically did not perform convincingly compared to keeping filter information indefinitely (Schreiber & Sanders, 2021). For this reason, we omit this mechanism from our evaluations in this article.

5.4.3 Distributed Filter

Our base approach still features Bloom filters, which may wrongly reject unseen clauses. The probability for such false positives grows with the number of clauses registered in the filters, which may become noticeable if millions of clauses are being shared.

For any $e \geq 0$, we define *epoch* e as the time interval that begins with the e-th sharing operation (or, if $e = 0$, with the start of solving) and ends with the $e + 1$-th sharing operation. The next filtering mechanism we describe is exact in the sense that a shared clause c is *admitted for import* in epoch e if and only if c has *not* been shared and admitted for import in epochs $e-z,\dots,e-1$, where $z \geq 0$ is the user-defined *resharing period*. We can keep track of all (successfully) shared clauses within this period by distributing the required bookkeeping across all processes. Specifically, each process only remembers the clauses it contributed *itself*. We can use a periodic garbage collection to remove clauses older than z epochs from the filter. As such, the memory requirements at each process are limited to the volume of locally produced clauses from the last $\mathcal{O}(z)$ epochs.

On each process, we use a hash table H that maps produced clause c to $H[c] := (p(c), e_{\text{prod}}(c), e_{\text{sh}}(c))$, where bitset $p(c)$ encodes which local solvers produced c, $e_{\text{prod}}(c) \in \mathbb{N}$ is the last epoch where a local solver produced c, and $e_{\text{sh}}(c) \in \mathbb{N} \cup \{-z\}$ is the last epoch where c was shared and admitted for import. If c was not shared before, then $e_{\text{sh}}(c) = -z$.

If solver S in epoch e produces a clause c that meets basic quality criteria (see Section 5.3.2), S looks up c in H. If $c \notin H$, S tries to insert c into export buffer $\mathcal{B}$. On success, S inserts c into H as well, setting $p(c) = \{S\}$ and $e_{\text{prod}}(c) = e$. Note that the successful insertion in $\mathcal{B}$ presents an additional quality criterion for sharing c and may, in turn, delete "worse" clauses in $\mathcal{B}$. If $c \in H$, S still updates $p(c) := p(c) \cup \{S\}$ and $e_{\text{prod}}(c) := e$.

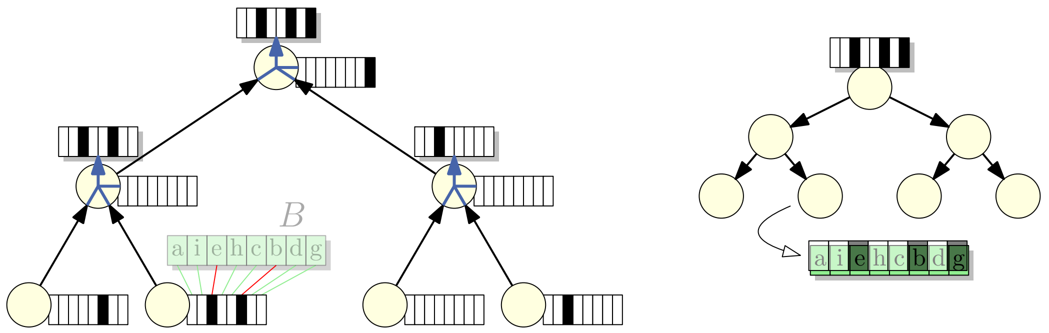


Figure 4: Distributed clause filtering, continuing the example from Fig. 3. Based on sharing buffer B, filter vector v is aggregated (left) and then broadcast (right). For one particular process the construction of $\tilde{v}$ is shown: The process remembers two clauses, "e" and "b", as recently shared and sets the according two bits.

Fig. 4 illustrates our distributed clause filtering. Upon obtaining the sharing buffer B from our clause exchange operation (see Section 5.2.2), the following steps are performed:

- Each process P constructs a bit vector $\tilde{v}$ whose i-th bit is set if and only if P recognizes that the i-th clause in B, c_i, was shared and admitted for import within the last z epochs. Specifically, P computes $\tilde{v}[i] = q_{c_i} := [c_i \in H \wedge e \leq e_{\text{sh}}(c_i) + z] \in \{0,1\}$.
- All local bit vectors $\tilde{v}$ are reduced to a single *filter vector* v via bitwise OR operations. v is aggregated and then broadcast to all processes just like B.
- **Import:** Each process iterates over B and v and only admits clause c_i for import if $v[i] = 0$. Each admitted clause c is inserted into the import buffer $\mathcal{B}_{S_j}$ of each local solver S_j if $(c \notin H \vee j \notin p(c))$. If $c \in H$ for an admitted clause c, then c is *marked as shared*: $e_{\text{sh}}(c) := e$ and we reset $p(c) := 0$ after its use.

Our filter has two purposes: filtering clauses shared recently (*distributed filtering*) and preventing clauses from being mirrored back to their producers (*by-solver filtering*). For the former purpose, we establish our filter's correctness with the following theorem:

Theorem 5.1 (Correctness of distributed filtering)
The described approach admits a shared clause c for import in epoch e if and only if c has not been admitted for import in epochs $e-z, \dots, e-1$.

Proof. Assume that some process shares c in epoch e. The following chain of reasoning holds in both directions:

c is admitted for import in epoch e
$\Leftrightarrow$ at epoch e, $q_c = 0$ on all processes
$\Leftrightarrow$ at epoch e, $c \notin H \vee e > e_{\text{sh}}(c) + z$ on all processes
$\Leftrightarrow$ at epoch e, $e_{\text{sh}}(c) < e - z$ on each process with $c \in H$
$\Leftrightarrow \forall\, e' \in \{e-z, \dots, e-1\}$, no process has marked c as shared
$\Leftrightarrow \forall\, e' \in \{e-z, \dots, e-1\}$, c has not been admitted for import.

The last equivalence holds due to the following argument: A shared clause c always has at least one process of origin where $c \in H$. Since each such process with $c \in H$ marks c as shared if and only if c is admitted for import, it follows that a clause c is admitted for import if and only if there is a process that marks c as shared. $\square$

In terms of by-solver filtering, an exact approach may guarantee the following: If a solver S produces clause c, then c will not be handed to S for exactly z epochs. We relax two aspects of this guarantee to allow for a more efficient implementation.

First, if z epochs expire without c being shared and admitted, then c may still be blocked *once* from being handed to S in a later epoch. This is because the epoch of production $e_{\text{prod}}(c)$ is a field shared between all local solvers. Other solvers producing c update $e_{\text{prod}}(c)$ as well, which can prolong the filtering status of c. This inaccuracy can be eliminated by replacing $p(c)$ and $e_{\text{prod}}(c)$ with a vector of production epochs, one for each local solver, at the cost of storing additional data for each clause in H.

Secondly, we only employ by-solver filtering for clauses that are successfully inserted or updated in H. A clause c that does not fit in $\mathcal{B}$ is not inserted in H but may still be shared successfully by another process at a later point in time. This is possible if, at the time of production of c, $\mathcal{B}$ is full and some of the later incoming clauses are of equal or worse quality than *all* clauses in $\mathcal{B}$. Since we configure $\mathcal{B}$ to hold $x = 10$ times the export volume β_0 per epoch, this scenario is unlikely except if the distribution over produced clause quality changes suddenly, e.g., if solvers produce batches of particularly short clauses during some pre– or inprocessing. We could eliminate this inaccuracy by inserting every produced clause in H no matter its quality, which would increase the filter's memory footprint.

The communication cost of our clause filtering approach is a second all-reduction whose work and communication volume are dominated by the corresponding clause exchange. Each process now needs to iterate over the sharing buffer twice—once for filtering and once for importing clauses. On each process, H requires memory linear in the volume of locally produced clauses that were inserted in $\mathcal{B}$ in the last $\mathcal{O}(z)$ epochs. Such an insertion is done only if the clauses in $\mathcal{B}$ of equal or higher quality do not exceed the capacity of $\mathcal{B}$.

5.5 Compensating for Unused Sharing Volume

The volume of clauses successfully shared across processes can often stay behind the sharing volume that we target with our scaled sharing buffer limit (Section 5.2.2). There are three causes for this effect: (i) the processes do not produce enough clauses that are admissible for sharing; (ii) duplicate clauses are detected and consequently eliminated during aggregation; (iii) our filtering mechanism leads to the rejection of some transmitted clauses. While (i) is a normal occurrence, e.g., at the beginning of a large formula's processing, we wish to compensate for the sharing volume that remained unused for algorithmic reasons, i.e., due to causes (ii) and (iii). For this purpose, we multiply all buffer limits in epoch e with a *compensation factor* $\kappa^{(e)}$ that is calculated based on statistics of recent sharings.[9]

Let X_{target} be the targeted number and X_{actual} the actual number of successfully shared literals so far. Let $\tilde{x}_{\text{in}}$ be an estimate for the number x_{in} of *incoming literals* next sharing (i.e., all literals contributed by all processes *before* aggregation and filtering) and let $\tilde{x}_{\text{out}}$ be an estimate for the number x_{out} of successfully shared literals. We keep $\tilde{x}_{\text{in}}$ and $\tilde{x}_{\text{out}}$ normalized by κ. Since we want the next sharing of expected effective size $\kappa\tilde{x}_{\text{out}}$ to meet the anticipated sharing volume $\tilde{x}_{\text{in}}$ and to also compensate for the discrepancy $X_{\text{target}} - X_{\text{actual}}$, we target $\kappa\tilde{x}_{\text{out}} = X_{\text{target}} - X_{\text{actual}} + \tilde{x}_{\text{in}}$. We thus set $\kappa = \min\{\kappa_{\max}, (X_{\text{target}} - X_{\text{actual}} + \tilde{x}_{\text{in}})/\tilde{x}_{\text{out}}\}$, where $\kappa_{\max}$ is a small constant that presents an upper bound for κ ($\kappa_{\max} = 5$ in our setup). We estimate $\tilde{x}_{\text{in}}$ and $\tilde{x}_{\text{out}}$ using "elastic" updates with update factor δ:

$$\tilde{x}_{\text{in}}^{(e+1)} := \delta\tilde{x}_{\text{in}}^{(e)} + (1-\delta)\frac{{x_{\text{in}}}^{(e)}}{\kappa^{(e)}}$$

$$\tilde{x}_{\text{out}}^{(e+1)} := \delta\tilde{x}_{\text{out}}^{(e)} + (1-\delta)\frac{{x_{\text{out}}}^{(e)}}{\kappa^{(e)}}$$

These estimates allow our sharing to react to changes in the distribution over produced, filtered, and duplicate clauses. Instead of aggregating X_{target} and X_{actual} exactly, we decay these values over time. As such, missed out sharing volume is either compensated for in a timely fashion or otherwise forgotten over time. Limiting κ to $\kappa_{\max}$ helps to keep communication manageable and to distribute larger amounts of compensation over multiple epochs rather than performing a single huge sharing with comparably bad clauses.

5.6 Handling LBD Values

Each clause c a solver produces is associated with a particular LBD value (see Section 3.2.1). In HordeSat, each solver imports each clause together with its original LBD value. Since LBD is an essential metric for clause quality in sequential SAT solving, many solvers keep clauses with LBD ≤ 2 *indefinitely* (Audemard & Simon, 2009; Biere et al., 2020). These solvers, however, are usually tuned to expect a single solver's worth of clauses. In large setups, a statistical argument can be made that the volume of "very good" clauses can become very large compared to a sequential execution (Ehlers & Nowotka, 2019)—especially if we prefer sharing such clauses. As such, the solvers' clause databases may grow in size significantly, leading to computational overhead and increased memory usage.

9. We write $x^{(e)}$ to denote the value of a variable x at the time of sharing epoch e.

By contrast, in the system TopoSAT 2 each LBD value is reset to $|c|$, i.e., an upper bound on possible LBD values, before c is imported (Ehlers & Nowotka, 2019). This lets a solver prioritize the clauses it produced itself over external clauses and presumably leads to many shared clauses being deleted quickly (Audemard & Simon, 2014).

We have implemented both LBD handling approaches and propose a third approach: At the import of a clause c, we *increment* its LBD value. As such, we preserve the LBD-based prioritization of clauses while also ensuring that a solver prefers local clauses and retains authority over which of its clauses are kept indefinitely. We expect that this strategy keeps clause databases more manageable than with HordeSat-style LBD handling and still allows the solvers to make good use of the shared clauses' original LBD values.

6. Achieving Diversity

Intuitively, diversification of solvers has two important merits. First, if different solvers explore different subspaces, the probability is higher for some solver to arrive at a satisfying assignment. Secondly, starting off the solvers along different directions leads to different learned clauses, which results in more diverse sets of shared clauses.

Our approach features three levels of diversification. First, our system takes a *portfolio policy* π as an input, e.g., $\pi = \langle k, k, c, l, g \rangle$ for Kissat-Kissat-CaDiCaL-Lingeling-Glucose, and then maps the i-th solver thread in the computation ($i \geq 0$) to the solver backend $\pi[i \mod |\pi|]$. Specifically, if each process runs l solvers, then the j-th process of MallobSat ($j \geq 0$) launches solver threads with $i \in \{jl, \ldots, (j+1)l - 1\}$. Secondly, each solver thread has a *diversification index* x, which indicates that it is the x-th thread running this particular solver backend. This index x is used to cycle through solver-specific diversification options. Lastly, a *diversification seed* is computed for each thread based on i, which is used for random decisions in the solver and additional diversification (Section 6.2).

6.1 Solver Portfolio

We have integrated four sequential solvers in our system. In addition to an updated version of HordeSat's Lingeling (Biere, 2010) backend, we support the popular solver Glucose by Audemard and Simon (2009) (a fork of Minisat by Eén & Sörensson, 2004) as well as the state-of-the-art solvers Kissat and CaDiCaL (Biere, 2018; Biere et al., 2020).

We provide all used configurations in Tab. 10 (Appendix A). We picked most configurations and their ordering following a *greedy k-portfolio* (Bach et al., 2022): We first measured the 32-core performance of individual configurations on the ISC 2020 benchmark set. We picked the best performing configuration C_1, then picked the configuration C_2 that maximizes the performance of the virtual portfolio $\{C_1, C_2\}$, then picked C_3 that maximizes the performance of $\{C_1, C_2, C_3\}$, and so on until improvements become negligible. While we found this approach to result in good performance, dedicated machine learning approaches may be able to further improve our setup (Xu et al., 2012; Biere, 2015; Bach et al., 2022).

For the initial version of MallobSat (Schreiber, 2020; Schreiber & Sanders, 2021), we focused on **Lingeling** as an efficient and reliable SAT solver with well-performing diversification options from Plingeling (Biere, 2014). Note that Lingeling features some non-standard reasoning, such as Gaussian elimination and cardinality constraint reasoning (Biere, 2012, 2013). Due to these techniques, Lingeling has a crucial advantage on some unsatisfiable instances compared to many other solvers (Ignatiev et al., 2017). While these techniques cannot be expressed in terms of general resolution and therefore have to be disabled for certified SAT solving (Biere, 2013), we are able to exploit them for (uncertified) parallel SAT solving. We use the most recent version of Lingeling in terms of sequential solving features (Biere, 2018). Similarly, we use most CDCL diversification options from the latest Plingeling (Biere, 2018). Every eleventh solver thread uses local search solver **YalSAT** (integrated in Lingeling), alternatingly with and without preprocessing.

For the **Glucose** (Audemard & Simon, 2009) interface, we used some of the sources of Syrup (Audemard & Simon, 2014) to import and export clauses asynchronously. This includes the technique of exporting clauses only after they have been encountered for the second time. We adopted and adjusted the diversification of Syrup, which includes different scheduling strategies for clause deletion and restarts, toggling simplification techniques, and a dynamic adaption of some parameters after the first x conflicts. In addition, the first search descent and initial variable activities are randomized.

For **CaDiCaL** (Biere, 2017), we used the existing clause export interface and implemented a clause import interface. A first version of this interface was provided by Maximilian Schick (2021). We diversify CaDiCaL instances via its `sat` and `unsat` presets, randomizing restart intervals, and toggling individual options such as random walks, bounded variable elimination, and inprocessing in general. For **Kissat** (Biere et al., 2020), we implemented our own clause import and export mechanism based on our CaDiCaL variant and employ diversification similar to CaDiCaL. In our current setup, we let the solvers try to import arrived clauses whenever at decision level 0—in earlier versions we had Kissat find a minimum of 500 conflicts between attempted clause imports, which can increase clause turnaround times. Moreover, we do not limit clause export to clauses found during conflict analysis but rather export *any* redundant clause that CaDiCaL and Kissat derive, including those found during inprocessing (Biere et al., 2021).

Note that certain inprocessing, such as *bounded variable elimination* (Eén & Biere, 2005), has a peculiar impact on clause sharing: Incoming clauses that feature a literal that has been eliminated in the importing solver cannot be imported but must be discarded (Biere, 2013). We performed measurements with Kissat and observed that this can, in some cases, invalidate around 90% of incoming clauses. We experimented with disabling such inprocessing for many or even all solvers but could not observe improved performance. We believe that overcoming these negative interactions between clause sharing and inprocessing is an important line of future work, especially considering the recent emergence of new powerful preprocessing techniques (Haberlandt et al., 2023; Reeves & Bryant, 2023).

6.2 Diversification Techniques

In the following, we present the additional diversification techniques we employ.

Sparse random variable phases. A *variable phase* in a solver decides which value to assign to the variable if it is chosen as a decision variable. HordeSat features a diversification technique where in a run with p solvers, each variable's initial phase in a solver is overridden with probability $1/p$ (Balyo et al., 2015). The variable phase is then determined by a coin flip. We adopted this technique in MallobSat; it is enabled after configuration-based diversification is exhausted. For Kissat, we implemented an option to provide a vector of initial variable phases since this was not part of its original interface.

Input permutation. There is no formal notion of order among the clauses in a CNF formula F—any permutation of a given sequence of clauses is logically the same input to a SAT solver. In practice however, the order in which clauses are given to a SAT solver can lead to considerable running time variations (Biere & Heule, 2019). Permuting the input before handing it to a solver can thus be used for diversification. In our implementation, all but the first ten solver threads have a 50% chance to perform input permutation.

A formula arrives at each of our SAT solving processes in the form of a flat array of integers. For very large formulas,[10] permuting this input by explicitly reordering the data for each solver is unacceptable in terms of running time and/or additional memory usage. Creating a pointer to each clause and then permuting the pointers is more viable but leads to irregular and thus cache-unfriendly memory accesses while importing the permuted formula.

We select up to $k = 128$ clauses to which we store a pointer. The first clause in the input is always selected while the remaining $k - 1$ clauses are selected at random. As such, each of the k pointers represents a chunk of the input beginning at the referenced clause and ending at the next pointer's address or at the end of the input data. These k pointers are then permuted and the input chunks are read in the corresponding order. This procedure remains cache-friendly for large inputs, and for any pair of clauses (c_1, c_2) in the input there is a non-zero probability that the order of c_1 and c_2 is reversed.

Noisy numerical parameters. To further diversify solvers, we suggest to add a small amount of random noise to certain numerical parameters. For each such parameter, we sample a number from a Gaussian distribution centered at zero and then add the number to the parameter's default. We have chosen restart intervals and variable score decays as promising candidates for this kind of randomization.

7. Technical Improvements

In addition to the described main ingredients to our system, we now mention some pragmatic improvements that contribute to the efficiency and viability of our system.

7.1 Preemption of Solvers

To support malleability, each process must be able to suspend, resume, and terminate its local solver threads at will. We noticed that we cannot rely on solver threads to stop on their own because a solver can sometimes get stuck in expensive preprocessing and inprocessing (Biere, 2016) for several minutes. Instead, we bundle the set of local solver threads in a separate *subprocess*. This incurs some overhead as a new process is forked, a shared memory segment for efficient inter-process communication (IPC) is set up, and the subprocess runs an additional management thread. However, suspension and termination of a process is supported on the OS level in a safe manner through *signals*. As solver threads may be unresponsive when the subprocess catches a termination signal, they are interrupted and

10. The largest instance of ISC 2022, `SAT_MS_sat_nurikabe_p16.pddl_166.cnf`, has 213 million clauses. It is serialized to more than 712 million integers (including separators) and thus around 2.65 GB of raw data.

cleaned up forcefully. We can also adjust the `oom_score` of each subprocess such that the OS kills it first if a machine runs out of main memory (Schreiber, 2023a). In both cases, this leaves the main process and the distributed computation in a valid state. Last but not least, our subprocessing allows for a kind of fault tolerance: If a solver crashes,[11] the concerned subprocess can be restarted without disrupting the distributed solving effort.

7.2 Memory Awareness

The high memory consumption of parallel portfolios is a known issue (Iser et al., 2019; Fleury & Biere, 2022). Executed on large formulas, our system can cause nodes to run out of main memory. To counteract this issue, we first introduce a simple step of precaution: For a given threshold $\hat{s}$, if a given serialized formula description has size $s > \hat{s}$, then only $t' = \max\{1, \lfloor t \cdot \hat{s}/s \rfloor\}$ threads will be spawned for each process. The choice of $\hat{s}$ depends on the amount of available main memory per process. The system we used only features 2 GB of RAM per solver if all physical cores are used. Based on monitoring the memory usage for different large formulas in our system, we use $\hat{s} := 50 \cdot 10^6$. As t' only depends on s, the t' threads can be started immediately without any further inspection of the formula.

While the above step of precaution can be effective for some inputs, it does not address memory usage that is initially acceptable but then grows to unsustainable levels. We thus introduce an additional, reactive measure: All local processes on a particular machine periodically check memory consumption and exchange certain diagnostics. If a memory threshold is exceeded ($>$ 90% of RAM used), one or multiple processes are chosen to trigger a *memory panic* (*cf.* Audemard & Simon, 2014). The heuristic that decides on the particular process(es) considers the memory used by each process as well as the importance of its role in the portfolio—processes closer to the job tree's root are considered more important. A process experiencing a memory panic asks some of its solver threads to terminate. If this does not lead to the desired reduction in memory usage (e.g., because the solvers do not react sufficiently fast), the entire subprocess may be killed and restarted with fewer solvers.

8. Evaluation

We now evaluate our work. After describing our implementation and explaining our setup, we first evaluate MallobSat on individual inputs at fixed scales. Then we discuss and analyze the performance of our system in the International SAT Competition. Lastly, we evaluate MallobSat within our malleable job scheduler Mallob (Sanders & Schreiber, 2022), solving several inputs in parallel. Our software and data are available online.[12]

8.1 Implementation

We implemented MallobSat in C++17 as an application engine within Mallob, a decentralized job scheduling platform (Sanders & Schreiber, 2022). Using the Message Passing Interface (MPI; Gropp et al., 1999), Mallob can be deployed on a single machine or on many interconnected machines at once. It features an API that applications can be con-

11. For example, Lingeling can crash on an instance named `lang28.cnf.gz.CP3-cnfmiter.cnf` because of its *simple probing* mechanism (Biere, 2012) resulting in vast amounts of irredundant clauses.
12. `https://zenodo.org/doi/10.5281/zenodo.10184679`

nected to and then allows to schedule and process tasks from these applications on demand. There are no dedicated processes for the scheduling of jobs—every process can be configured to take job submissions from an external source and to participate in decentralized scheduling negotiations. All protocols are designed without explicit synchronization (barriers or similar) and only use asynchronous point-to-point messages with few exceptions. We refer to Schreiber's (2023b) dissertation for further details on Mallob's design.

In order to run MallobSat in an isolated manner on a single instance, we added the special mode of operation "*mono*" to Mallob. This mode of operation introduces a single job to Mallob's decentralized scheduler, which immediately scales up the job to span the entire system. The job's workers are mapped to processes in a straight forward manner, only exchanging messages along the emerging job tree T. After the job description (i.e., the formula) has reached all processes in T, Mallob incurs virtually no overhead compared to an isolated distributed solver apart from a few periodic checks.

8.2 Experimental Setup

We performed our experiments on **SuperMUC-NG**. This system at the *Leibniz Rechenzentrum* features a total of "*311 040 compute cores with a main memory of 719 TB and a peak performance of 26.9 PetaFlop/s*".[13] In particular, SuperMUC-NG features 6 336 "thin" compute nodes each with a two-socket Intel Skylake Xeon Platinum 8174 processor clocked at 2.7 GHz with 48 physical cores (96 hardware threads) and 96 GB of main memory. Nodes are interconnected via OmniPath (Birrittella et al., 2015) and run SUSE Linux Enterprise Service (SLES). We allocated up to 134 machines ($= 134 \times 2 \times 24 = 6432$ cores) at a time, mapping each MPI process either to four cores as in HordeSat or to an entire socket of 24 cores. We compiled our software with GCC 11.2 and Intel MPI 2019.12.

We updated HordeSat to also use the latest Lingeling+YalSAT (Biere, 2018) backend. Furthermore, we fixed a performance bug in HordeSat due to Lingeling missing a certain time measurement callback, which resulted in a fallback to expensive system calls and sometimes caused solvers to spend more than 10% of their time in kernel mode.

We tested the parallel solvers with a comparably low wallclock time limit of 300 s since we aim to solve SAT instances as rapidly as possible in a costly large-scale distributed environment, where we invest substantially more resources than with sequential solving.[14]

Performance metrics. As described in Section 3.5.1, we rate solver runs based on solved count, PAR-2 score, and, in cases where solved counts and PAR-2 scores are similar, CSAR score. To compute PAR-2 scores restricted to satisfiable and to unsatisfiable instances, we assume that there are equally many satisfiable and unsatisfiable instances.

Selection of benchmarks. For the evaluation of the configurations of our system we use the 349 benchmark instances from ISC 2022 (Balyo et al., 2022a) that *some* solver was able to solve (across all tracks). This may lead to slight underestimation of a run's performance but drastically reduces running times and, consequently, resource usage. For instance, our baseline configuration spent 4.9 h on the 349 instances whereas it would have spent up

13. `https://doku.lrz.de/supermuc-ng-10745965.html`
14. In the ISC, the by-instance CPU timeout in the cloud track (1000 s $\times$ 1600 hardware threads) exceeds that of the sequential main track (5000 s $\times$ 1 hardware thread) by a factor of 320.

to 9.2 h on all 400 instances. To subsequently evaluate the behavior and speedups of our system (Section 8.4 and later), we use a different benchmark set to reduce overfitting effects, namely the ISC 2021 benchmark set (Balyo et al., 2021) consisting of 400 instances.

8.3 SAT Solving Configuration

We begin our experiments with a pretuned configuration of MallobSat obtained by a series of explorative tests on a random selection of 125 instances from the ISC 2022 Anniversary benchmark set. This configuration features our latest techniques and data structures; two sharings per second with compensation for unused sharing volume; distributed filtering with a resharing period of 30 epochs (15 s); incrementing each LBD score before import; a clause buffer limit of $\beta_\infty = 100\,000$; and a process setup with one MPI process for each socket of 24 cores. In the following, we adjust individual components and/or parameters of our system and analyze the respective differences. If not indicated otherwise, we use a setup with 768 cores (16 machines). We chose this scale as a compromise between making responsible use of resources and still being able to observe effects that are specific to distributed solving (such as fully exhausted diversification in terms of distinct solver configurations).

Process layout. We begin with the observation that our setup faithful to the hardware at hand is indeed beneficial for performance and memory usage (Tab. 1). In a direct comparison of our most recent "$16 \times 2 \times 24$" setup with the earlier HordeSat-style "$16 \times 12 \times 4$" setup, we arrived at a PAR-2 score of 142.3 for the former and 147.1 for the latter, although the latter setup was barely able to solve three additional instances close to the time limit. Our new setup's advantage in terms of CSAR scores is even larger. More importantly, our new setup reduced mean RAM usage[15] by more than 16% (286 GB down from 342 GB), mostly due to the fact that fewer copies of a formula are present on each machine.

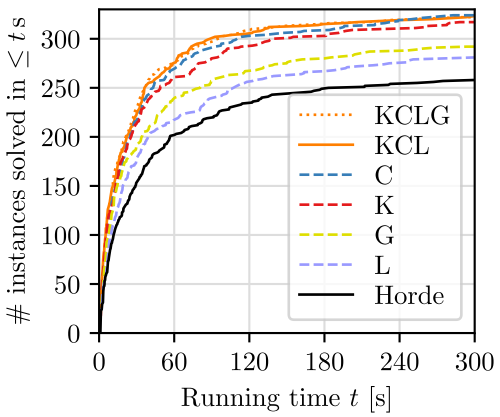
	Mean RAM	#	PAR-2	CSAR
$16 \times 12 \times 4$	342.0	**323**	147.1	36.1
$16 \times 2 \times 24$	**286.0**	320	**142.3**	**27.9**

Table 1: Impact of process layout, written as (# machines) $\times$ (# processes per machine) $\times$ (# solvers per process). The better result per column is highlighted.

Solver Backends. To assess the impact of individual solver backends, we first consider single-solver portfolios with each of our solver backends. As Fig. 5 shows, MallobSat outperforms HordeSat if both use Lingeling. Using more recent solvers—CaDiCaL and Kissat in particular—further improves performance. Interestingly, our CaDiCaL portfolio outperforms our Kissat portfolio although Kissat is the more recent solver (Biere et al., 2020). It is also worth noting that our CaDiCaL portfolio alone performs almost equally well as a balanced mix of three (KCL) or all four (KCLG) solvers, indicating that our system does not rely on a mixed portfolio in order to perform well. Including Glucose

15. RAM usage is measured by aggregating the global Resident Set Size (RSS) main memory usage of all Mallob processes, including SAT subprocesses, every second.

in our mixed portfolio (KCLG) results in similar performance as omitting it (KCL), and Glucose also tends to use the highest amount of memory. In the following, if not indicated otherwise, we use our KCL portfolio since it achieves good performance and also allows to observe our system's behavior if multiple different solver backends are used in parallel.

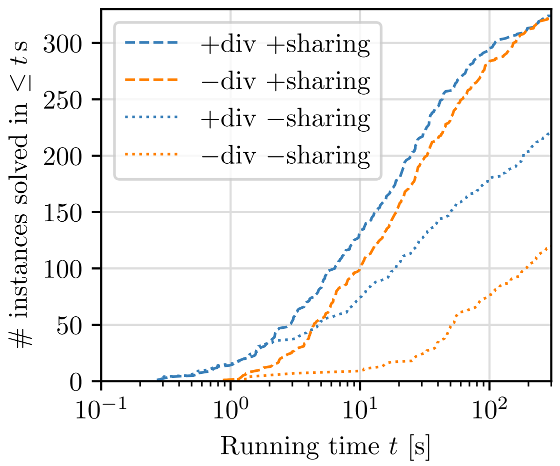


	overall		SAT		UNSAT	
Portfolio	#	PAR	#	PAR	#	PAR
KCLG	322	140.3	156	145.4	166	135.2
KCL	323	139.8	158	141.6	165	138.0
C	324	142.1	159	140.3	165	143.9
K	318	152.4	160	144.7	158	160.1
G	292	189.3	133	216.4	159	162.1
L	284	205.6	128	235.1	156	176.0
Horde (L)	253	243.8	121	254.3	132	233.2

Figure 5: Performance of MallobSat—with Kissat (K), CaDiCaL (C), Glucose (G), Lingeling (L), and two combinations (KCLG, KCL)—and of HordeSat.

Diversification. Since we were not able to isolate meaningful performance differences between enabling and disabling individual diversification techniques, we show results for all of our diversification put together—random seeds, initial phases, input permutation, noisy parameters, and our handcrafted set of solver configurations—and for all of it disabled. We ran these experiments with CaDiCaL only for a cleaner picture. The results, shown in Fig. 6, were startling to us: While diversification benefits performance initially, this advantage is largely lost the more time is available. In other words, omitting *all* diversification, which essentially leaves 768 *identical* CaDiCaL programs running in parallel, still results in rather competitive performance. Our explanation is as follows: The nondeterminism introduced by parallel and distributed execution causes the solvers to vary in terms of the exact points in time at which they import clauses. After a few seconds, the solvers have diverged sufficiently for our clause sharing to act as a kind of distributed search space pruning. We repeated the same experiments with clause sharing disabled, where "–div –sharing" does in fact run 768 identical programs, and found that diversification does have a substantial merit in that case. Still, the impact of clause sharing is far greater. We further discuss the implications of these results in Section 8.5.

Handling LBD values. We compare our strategy of incrementing each incoming LBD value with the established approaches of using each LBD as is (Balyo et al., 2015) and resetting each LBD at import (Ehlers & Nowotka, 2019). As Tab. 2 shows, editing LBD value appears to make a difference regarding (median) memory usage. While the differences in PAR-2 performance are insignificant, we found our strategy to result in a slightly lower



div.	sh.	overall #	overall PAR	SAT #	SAT PAR	UNSAT #	UNSAT PAR
+	+	324	142.9	158	145.4	166	140.5
−	+	323	151.7	157	153.0	166	150.5
+	−	223	297.9	153	163.3	70	432.6
−	−	120	448.8	77	400.9	43	496.8

Figure 6: Impact of diversification (configurations, seeds, phases, permutation, noise) and clause sharing, one by one and together, in MallobSat with CaDiCaL only. Note the logarithmic scale along the x axis.

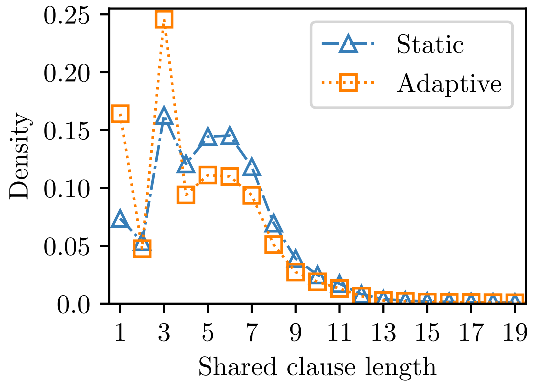
CSAR score than both other strategies. As such, we consider our strategy an appealing combination of good performance with reduced memory requirements.

	Mean RAM	Median RAM	#	PAR-2	CSAR
Original LBD	269.0	99.1	321	143.6	31.3
Reset LBD	269.0	**92.5**	**322**	**143.2**	32.2
Increment LBD	**267.1**	93.5	321	143.4	**30.0**

Table 2: Impact of LBD handling in terms of RAM usage (in GB) and scores.

Clause buffering. Next we evaluate our clause buffering techniques. Compared to export buffers à la HordeSat (with bucket sizes adjusted to our export buffer size), our adaptive export buffers reduce the mean shared clause length from 6.5 to 5.7. As shown in Fig. 7 (left), unit and ternary clauses are much more likely to be shared whereas longer clauses become less likely. Binary clauses are produced less frequently than unit or ternary clauses, hence both buffer types can handle their modest volume well. Despite sharing shorter clauses, adaptive export buffers resulted in no improvement but rather a moderate decline in performance (Fig. 7 right). A possible explanation is that the unlimited buffering of unit clauses may in fact be detrimental for some instances where huge amounts of unit clauses arise and prevent other clauses from being shared. Our adaptive import buffering, on the other hand, outperformed lock-free ring buffers (Schreiber & Sanders, 2021).

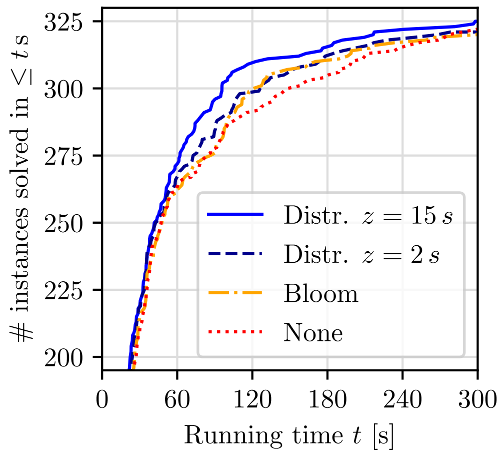
Clause filtering. In terms of clause filtering, we compared our distributed filter at different resharing periods z with HordeSat-style filtering (Section 5.4.2) and no filtering at all. MallobSat counts the resharing period in terms of sharing epochs; for the sake



	#	PAR-2	CSAR
Both adaptive	**321**	143.4	27.9
Static export	**321**	**141.9**	**26.5**
Static import	320	146.5	31.9

Figure 7: Impact of different clause buffers. The left figure shows a histogram over successfully shared clauses' lengths for adaptive and static export buffers.

of simplicity we display z in terms of seconds. As MallobSat performs two sharings per second, $z = 7.5\,\text{s}$ is equivalent to filtering clauses for 15 epochs.



	#	PAR	CSAR
No filtering	322	148.4	—
Bloom filters	321	147.5	—
Distr., $z = 2\,s$	321	145.6	31.78
Distr., $z = 4\,s$	324	141.4	30.49
Distr., $z = 7.5\,s$	**325**	139.6	29.38
Distr., $z = 15\,s$	**325**	**138.8**	**28.47**
Distr., $z = 30\,s$	322	141.9	28.48
Distr., $z = \infty$	322	141.8	28.55
Distr., $z = 7.5\,s$, NC	322	145.5	—

Figure 8: Performance of distributed filter configurations, Bloom filtering, and no filtering (note the offset of the y axis). The table also includes a run with no compensation ("NC") for unused sharing volume.

We provide results in Fig. 8. Disabling clause filtering completely resulted in the worst performance. All distributed filtering runs resulted in improved PAR-2 scores compared to Bloom filtering. Distributed filtering at blocking period $z = 15\,\text{s}$ performed best in terms of PAR-2 scores and performed similar to higher z in terms of CSAR scores. It appears to be particularly beneficial to filter clauses that are re-shared after a brief period. Clauses that re-occur after an extended period are less important to filter and might even be useful to admit for sharing again. Removing clauses from the filter after some time also keeps the memory footprint more manageable for higher time limits. The total ratio of filtered clauses ranges from 24.9% ($z = 2\,\text{s}$) to 56.2% ($z = \infty$). A resharing period of $z = 15\,\text{s}$ is sufficiently high to block more than half of all clauses (52.2%) at the tested sharing volume.

Upon closer inspection, we found that the instances that profit the most from our clause filtering (speedup ≥ 2) are unsatisfiable model checking tasks generated by CBMC (Clarke et al., 2004). These include nine of "*some of the hardest SAT problems [...] when verifying open-source code at AWS*" (Fofaliya et al., 2022) and two string safety problems (Osama & Wijs, 2021). We suspect that different solvers explore different components of these highly structured instances in different orders, thus arriving at similar sets of learned clauses at deviating points in time. Our distributed filtering recognizes these temporally shifted similarities and filters 65–85% of all shared clauses on these instances (at $z = 15\,\text{s}$).

Unused sharing volume compensation. To assess how well our compensation technique (Section 5.5) makes up for the filtered clauses, we also performed a run at resharing period $z = 7.5\,\text{s}$ without such compensation—neither for filtered clauses nor for clauses identified as duplicates during aggregation. We observed worse results than all other distributed filtering runs except for $z = 2\,\text{s}$ (see Fig. 8). In total, the run without compensation exchanged 80.5% and admitted for sharing 87.5% of the literals that the corresponding run with compensation exchanged/admitted. MallobSat reaches 88.6% of its total targeted sharing volume with compensation and 72.3% without compensation. The sharing volume that remains unused despite our compensation is, in large parts, due to points in time where insufficient amounts of distinct clauses are available for sharing. Overall, our compensation mechanism proved to be beneficial for the performance of our sharing approach.

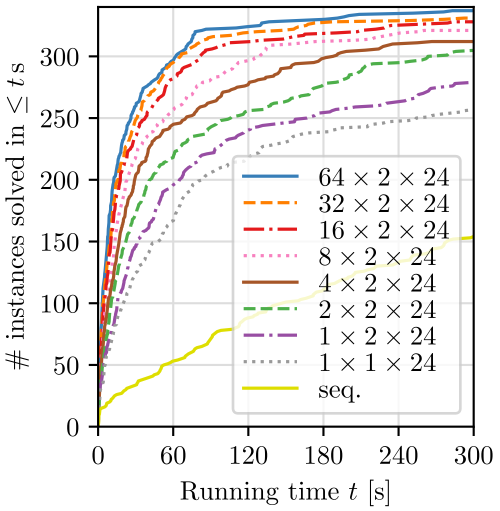
Frequency of clause sharing. Next, we observe the impact of the frequency at which sharing is performed. We tried frequencies of 1/s, 2/s, and 3/s and resized the base buffer size per process by a factor of 1, 1/2, and 1/3 respectively to keep the overall sharing volume fixed. The PAR-2 and CSAR scores given in Table 3 indicate that performing clause sharing more than once a second is indeed beneficial for performance. Note that this result contrasts earlier work (Audemard et al., 2017) where more frequent all-to-all sharing resulted in worse performance. Clearly, our clause sharing implementation and its embedding in the solver processes is sufficiently scalable to profit from reduced clause turnaround times while the added overhead is negligible. Two sharings per second performed best in terms of PAR-2 scores while three sharings per second performed similarly well in terms of CSAR scores.

	#	PAR-2	CSAR
3/s	323	140.9	**30.0**
2/s	**325**	**138.8**	30.1
1/s	321	144.5	32.4

Table 3: Impact of clause sharing frequency.

8.4 Scaling and Speedups

In order to assess the scalability of our tuned system, we use the set of 400 benchmarks from ISC 2021. We run MallobSat on 1, 2, 4, ..., 128 24-core sockets (i.e., up to 3072 cores on 64 machines) with one solver per physical core. As a sequential baseline we run KissatMABHyWalk (Zheng et al., 2022), the winning sequential solver from ISC 2022, without proof logging. We set the sequential solver's time limit per instance to 115 200 s (32 hours), which is equivalent to the maximum CPU time per instance for our run with 384 cores. We limited the main memory for each sequential run to 12 GB. After some further tests on ISC 2022 instances, we adjusted MallobSat's sharing buffer limit to $\beta_\infty = 250\,000$ literals for the following experiments (see Schreiber, 2023b for more details).



		overall		SAT		UNSAT	
m	cores	#	PAR	#	PAR	#	PAR
1	24	257	251.3	126	256.1	131	246.5
1	48	279	221.2	139	222.6	140	219.7
2	96	305	184.0	150	187.3	155	180.7
4	192	312	164.2	153	169.3	159	159.1
8	384	321	147.2	156	159.1	165	135.3
16	768	328	132.4	156	151.1	172	113.8
32	1536	331	124.4	157	145.3	174	103.5
64	3072	337	115.1	159	137.2	178	92.9
	seq.	154	414.0	87	389.1	67	438.9

Figure 9: MallobSat scaling results for an increasing number of machines m. We also include KissatMABHyWalk ("seq.") with the same timeout of 300 s.

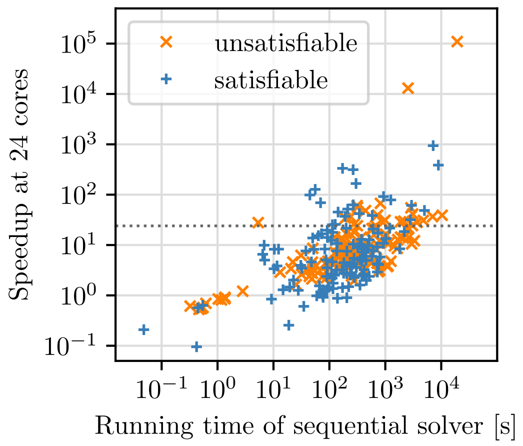
Fig. 9 shows the scaling behavior of our MallobSat configuration. Performance increases consistently up to 3072 cores. Improvements do diminish noticeably beyond 16 nodes (768 cores). On satisfiable instances, the number of solved instances only increases marginally from 96 cores onward. On unsatisfiable instances, we observe more pronounced scaling up to 3072 cores both in terms of instances solved and in terms of PAR-2 scores. At the largest scale tested, our system solves 19 more unsatisfiable instances than satisfiable instances. While this may partly be a result of the considered benchmark set, we later show that clause sharing especially benefits unsatisfiable instances (Section 8.5), which may indicate that more work is required to achieve similar benefits for satisfiable instances.

Tab. 4 lists the speedups of our system compared to the state-of-the-art sequential solver KissatMABHyWalk. At each scale we only consider the set of instances that both the sequential approach and the parallel approach were able to solve (see Section 3.5.3). We provide median and geometric mean speedups as well as total speedups (see Section 3.5.2).

		overall				SAT				UNSAT			
m	cores	#	med.	geom.	total	#	med.	geom.	total	#	med.	geom.	total
1	24	254	6.51	7.20	12.13	126	5.12	6.30	8.91	128	7.48	8.21	14.92
1	48	275	8.32	9.42	17.46	138	6.73	7.92	11.18	137	9.76	11.21	23.52
2	96	302	10.88	12.62	21.54	150	9.36	10.87	15.70	152	12.77	14.63	26.29
4	192	309	15.52	17.71	37.09	153	13.17	16.13	30.16	156	17.87	19.41	42.52
8	384	316	19.00	22.63	63.20	154	14.96	18.80	35.06	162	25.33	27.00	86.15
16	768	322	25.14	31.75	105.01	154	19.86	28.27	51.05	168	35.86	35.32	137.13
32	1536	323	32.92	37.40	138.70	154	30.76	33.30	63.58	169	38.02	41.57	182.29
64	3072	324	40.99	43.67	159.00	154	42.09	41.93	78.29	170	38.90	45.30	200.54

Table 4: Speedups of MALLOBSAT over KISSATMABHYWALK on 24 to 3072 cores. Each section shows the number of commonly solved instances and the median, geometric mean, and total speedup for these instances.

At 3072 cores, we observed a median speedup of 41 and a total speedup of 159. As a rough point of reference, prior literature reported median speedups of up to 13.8 (at 1024 cores) and total speedups of up to 109 (at 2048 cores)—both with HORDESAT (Balyo et al., 2015).[16] At 384 cores, where the parallel and sequential approach received equal resources, MALLOBSAT achieves an efficiency of 63.20/384 $\approx$ 16.5% in terms of total speedup. Our system at 1536 cores is able to solve the same number of instances (331) within five minutes per instance as the sequential baseline within 32 hours per instance. That being said, MALLOBSAT at this scale was allowed to use 4× the CPU time of the sequential baseline.



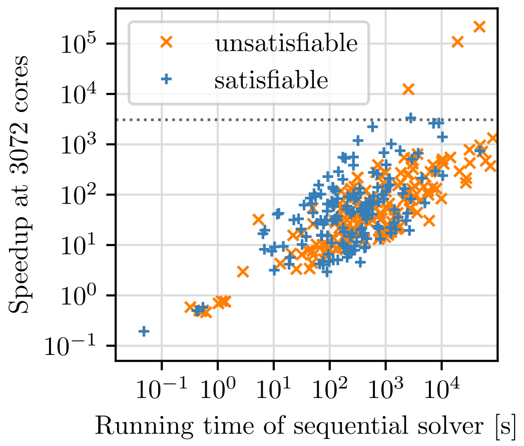


Figure 10: Per-instance log-scale speedups of MALLOBSAT relative to the sequential solver's running time, on 24 cores (left) and on 3072 cores (right). The horizontal dotted lines separate sublinear from superlinear speedups.

16. Note that we explicitly disregard the "arithmetic mean speedups" reported by Balyo et al. (2015) since these measures are statistically not meaningful (see Section 3.5.2 for a discussion).

Fig. 10 shows per-instance speedups of MallobSat at the smallest and largest tested scale. Speedups correlate with sequential running times, which confirms the increasing merit of parallel solving as instances become more difficult. At 3072 cores, all 14 instances that resulted in a slowdown (speedup < 1) took the sequential solver less than 1.4 seconds. On the more difficult half of instances, which took the sequential solver more than 349 s to solve, MallobSat at 3072 cores achieves a mean speedup of 104.5 (median 81.9). On the 40 instances that took the sequential solver more than an hour to solve, the mean speedup reaches 418.7 (median 370.5) and thus an efficiency of $418.7/3072 = 13.6\%$. Increasing the sequential and parallel running times may further extend the observed speedups.

While our 24-core run achieved superlinear speedups on 45 instances, the 3072-core run achieved only four such speedups. Our solver executes many solver configurations that excel on different instances, whereas the sequential baseline only runs a single configuration. We believe this discrepancy to be the main reason for the observed superlinear speedups. At 3072 cores, this effect is watered down by using far more cores than there are configurations. The 3072-core run did solve 31% more instances than the 24-core run, which clearly indicates *weak scaling*. Fig. 11 provides a full graphical overview of the scaling of MallobSat.

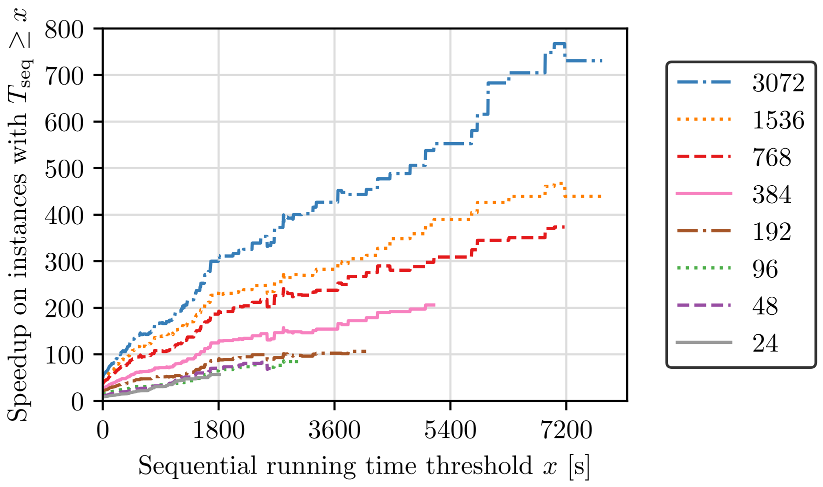


Figure 11: Scaling of MallobSat. For sequential running time threshold x, we consider all solved instances that took KissatMABHyWalk $T_{\text{seq}} \geq x$ seconds to solve. We graphed the resulting speedups until < 25 instances are considered.

Using the SAT benchmark database GBD (Iser & Sinz, 2019), we traced the observed speedups to specific benchmark families. Among the instances at the lower edge of the cone of speedups in Fig. 10, where MallobSat at 3072 cores scales the worst, are unsatisfiable string safety verification tasks (Osama & Wijs, 2021) (speedups of 13–25 at sequential running times of 800–1800 s) and satisfiable Hamiltonian Cycle encodings (Heule, 2021) (speedups of 17–36 for seven instances with sequential times of 700–2300 s).

Conversely, Tab. 5 shows some of the *best* consistent speedups MallobSat achieves. All of the displayed instances are unsatisfiable since speedups on satisfiable instances are

Instance	Seq. (s)	Speedup on x cores							
		24	48	92	192	384	768	1536	3072
ctl_4201_555_unsat.cnf	1873.9	14.4	20.1	32.8	49.0	75.3	105.7	138.7	171.2
edit_distance031_283.cnf	2465.9	20.3	36.3	59.8	85.9	122.7	184.2	230.8	247.1
edit_distance031_284.cnf	2561.6	22.0	34.3	59.4	87.7	130.4	185.7	230.3	256.8
mp1-bsat201-707.cnf	3087.9	41.4	67.7	125.3	165.5	234.1	260.7	287.5	321.3
mp1-bsat210-739.cnf	7011.8	36.6	60.3	109.1	150.8	253.0	294.4	324.3	387.5
mp1-klieber2017s-2000-022-eq.cnf	908.0	28.1	49.0	65.8	81.6	92.7	110.6	149.7	172.7
randomG-B-Mix-n16-d05.cnf	3053.2	24.4	41.1	63.5	106.5	163.4	238.5	332.6	413.3
rphp4_065_shuffled.cnf	671.5	24.1	42.3	66.0	94.0	129.0	179.4	231.5	276.8
rphp4_080_shuffled.cnf	2145.0	29.7	53.1	85.3	161.1	229.6	295.1	416.3	542.1
rphp4_090_shuffled.cnf	3709.9	28.2	51.3	91.7	138.9	261.7	379.4	503.2	638.1
satch2ways14u.cnf	1868.9	29.0	45.9	60.8	87.7	120.7	183.0	218.1	270.9
satch2ways15.cnf	10383.0	39.0	59.6	78.5	106.8	180.0	265.9	369.4	506.1
sp4-33-one-stri-tree-noid.cnf	727.5	39.4	55.6	95.6	133.4	198.0	266.6	320.0	390.0

Table 5: Instances where the speedup of MallobSat increases consistently (by $\geq 5\%$ per step) and exceeds 5% efficiency (i.e., speedup ≥ 153.6) at 3072 cores.

more erratic. The benchmark families featured more than once encode cluster graph editing distance (`edit_distance*`) (Mengel, 2021), balanced random SAT problems (`mp1-bsat*`) (Spence, 2017), relativized Pidgeon Hole Principle (PHP) problems (`rphp4*`) (Elffers & Nordström, 2016), and automated test configuration (`satch*`) (Biere, 2021).

8.5 Insights on Clause Sharing and Diversification

Next, we analyze the impact of MallobSat's clause sharing at the largest tested scale of 3072 cores. We re-ran MallobSat at this scale *without* clause sharing, rendering it a *pure* portfolio solver. Fig. 12 (left) shows the results. Considering satisfiable and unsatisfiable instances separately, we confirm Balyo et al.'s (2015) findings that clause sharing is the most useful for unsatisfiable instances. While Balyo et al. (2015) did not present evidence that clause sharing benefits satisfiable instances, we do observe improved performance on both kinds of instances. The CBS (Count Based Speedup, Section 3.5.2) of the clause sharing run over the sharing-less run is 4.08 for satisfiable and 15.59 for unsatisfiable instances.

Similarly, we ran MallobSat with and without diverse solver configurations. Fig. 12 (right) shows that a diverse portfolio featuring dozens of solver configurations ("KCL + conf.") only moderately outperforms a CaDiCaL-based portfolio with very light diversification (seeds and random phases only, "C, no conf."). The latter only solves eight instances less and results in competitive performance especially on unsatisfiable instances. Specifically, the CBS is 2.23 for satisfiable and 1.38 for unsatisfiable instances. This paints a similar picture as our earlier 768-core experiments (Section 8.3, "Diversification") where we arrived at a well-performing CaDiCaL-only solver with no explicit diversification *at all*.

Since clause sharing is demonstrably the main driver of our solver's scalability, we no longer believe that the term *portfolio solver* is an accurate label to characterize such approaches. Going forward, we refer to MallobSat as a *clause-sharing solver* (Manthey et al., 2013; Michaelson et al., 2023)—hoping to counteract the widespread perception of

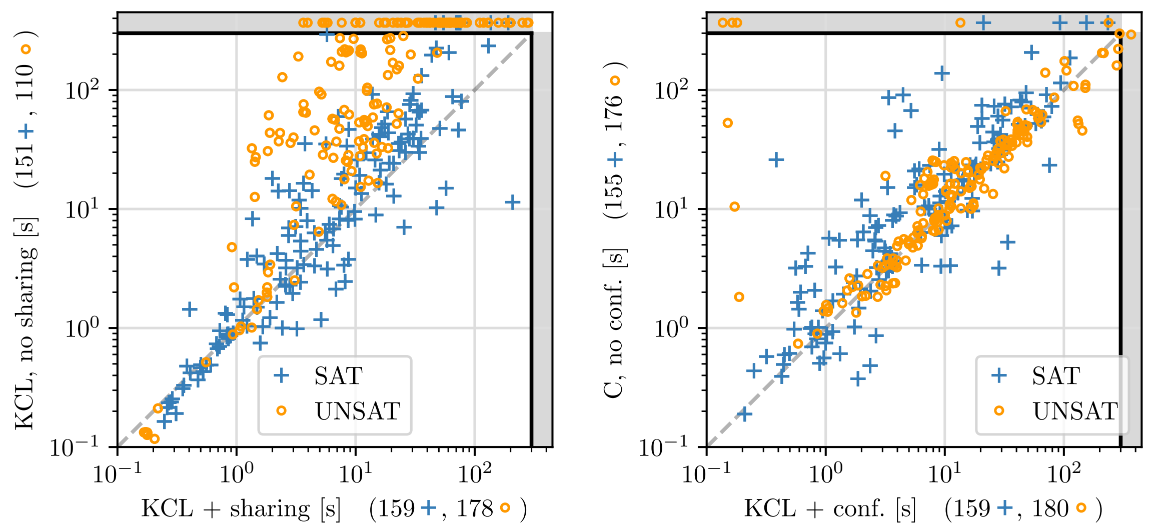


Figure 12: Impact of MallobSat's clause sharing (left) and solver/configuration-based diversification (right) at 3072 cores.

such solvers as systems where "*a problem instance is independently given to a collection of solvers competing for a solution in parallel*" (Fichte et al., 2023, emphasis ours) or "*where each thread runs a different SAT solver on the same instance[, which] in combination with clause-sharing leads to surprisingly good performance for small portfolio sizes*" (Ozdemir et al., 2021, emphasis ours). Quite the opposite, our system bases its scalability on careful clause sharing whereas adding some explicit diversity across solvers is beneficial but not essential. We are unsure whether similar observations can be made for other portfolio solvers with deviating approaches to clause sharing (Vallade et al., 2021; Zhang et al., 2023).

We now examine some detailed clause sharing statistics. Fig. 13 (top) illustrates the impact of the sharing buffer's sublinear scaling: The number of successfully shared literals per solver and per sharing decreases as the number of solvers increases. In these measures, the clauses blocked by distributed filtering (see Fig. 13 bottom right) have already been compensated for by correspondingly larger sharing buffers.

The average length of successfully shared clauses (Fig. 13 bottom left) mostly increases with the number of involved solvers as more and more literals are shared. Since the set of distinct short clauses that the solvers produce is limited, this leads to an increase in the mean length of successfully shared clauses. This is no longer the case at the largest scale of 128 workers (3072 solvers), where the total sharing volume increases only by about 31% compared to 64 workers. We rather see a slight reduction in the mean shared clause length (7.5 to 7.3) at this scale—showing that more selective sharing relative to the global volume of produced clauses can indeed improve shared clause quality. We conjecture that the mean clause quality may continue to improve when further increasing the number of solvers.

The ratio of clauses admitted by the distributed filter (Fig. 13 bottom right) indicates the increasing relevance of filtering the more workers are involved. Adding workers increases the probability that a clause is exported redundantly at several workers. Furthermore, as the

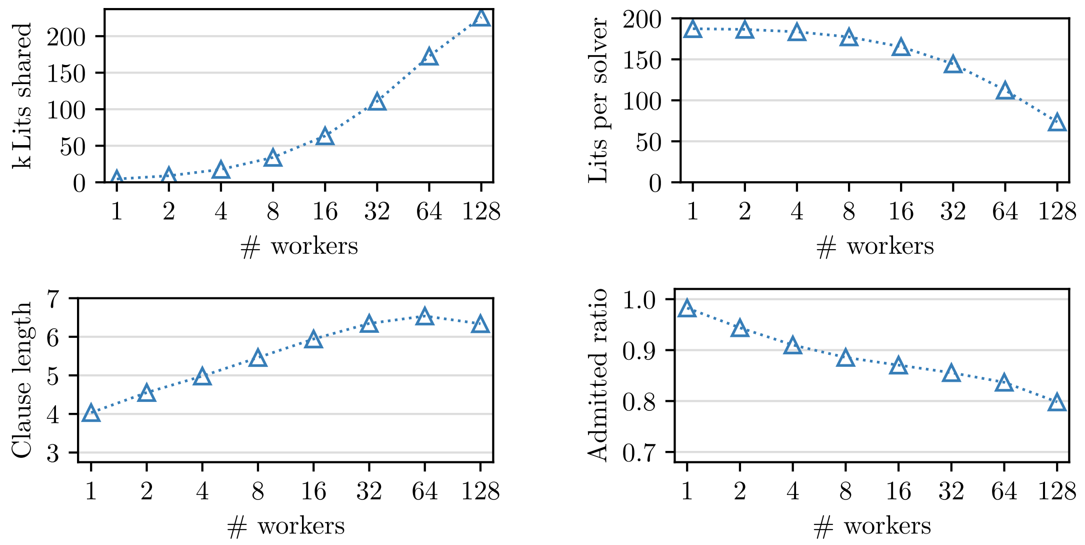


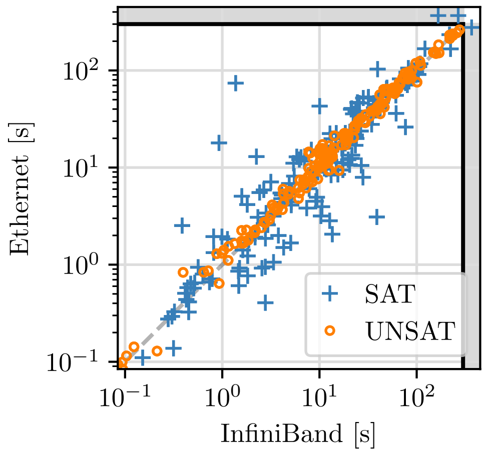
Figure 13: Clause sharing behavior for 1–128 workers (1–3072 cores) in terms of the median successfully shared literals (top left), the same measure divided by the number of solvers (top right), the mean length of successfully shared clauses (bottom left), and the mean ratio of exchanged clauses admitted by the distributed filter (bottom right). Note the y axis offsets in the bottom figures.

sharing budget per solver decreases, stricter quality criteria are enforced, which results in a smaller interval of admissible clause lengths and, therefore, an increased ratio of duplicates.

8.6 Communication Hardware

All experiments so far have been run on a large cluster with expensive, rapid interconnects across nodes. In order to see how MallobSat behaves on less expensive communication hardware, we ran an additional test with limited resources. We used **bwUniCluster2.0**, an HPC system that features both an InfiniBand interconnect (comparable to SuperMUC-NG's network) as well as less expensive, user-accessible Gigabit Ethernet. Message passing over the latter is realized via TCP/IP, which incurs relatively high overhead and latencies (Graham et al., 2006). Each HPC compute node of bwUniCluster2.0 features 96 GB of RAM and two Intel Xeon Gold 6230 sockets each with 20 cores (40 hardware threads). For each interconnect we ran MallobSat on 16 nodes, i.e., 640 cores, on 400 instances from ISC 2021 for up to 300 s per instance. Since we observed a few hang-ups with the current setup on this cluster, we retroactively removed twelve affected instances from both runs.

As Fig. 14 shows, both runs performed very similarly. On the 323 commonly solved instances, the Ethernet setup only performed slightly worse with a 1% increase in CSAR score. As such, the test supports our assumption that MallobSat does not crucially rely on high-end communication hardware. Since our system performs careful communication with only a few collective operations per second and performs this communication strictly



	#	PAR-2	CSAR
InfiniBand	325	137.6	139.5
Ethernet	324	140.1	140.9

Figure 14: MallobSat performance with Ethernet vs. InfiniBand interconnect.

asynchronously, it is able to handle slower interconnects well. This property also proved relevant in the International SAT Competition, as we elaborate in Section 8.8.

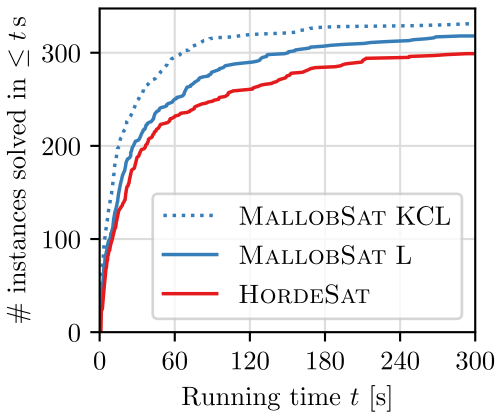
8.7 Comparison to HordeSat

We now compare our system to the prior state of the art. We ran updated HordeSat on up to 1536 cores and computed speedups in the same manner as for MallobSat (configured as in Section 8.4). As Tab. 6 shows, speedups are larger than reported for original HordeSat—now achieving a median speedup of 17.5 on 1536 cores—although we use an up-to-date sequential baseline and disregard sequential timeouts. Likely causes for this improvement are our updates to HordeSat, differing benchmark sets, and deviating time limits. Tab. 6 also includes a modified run of our system with a Lingeling-only portfolio as HordeSat's. With the same solver backend, MallobSat improves on HordeSat's

			overall				SAT				UNSAT			
Sys.	m	cores	#	med.	geom.	total	#	med.	geom.	total	#	med.	geom.	total
H	1	24	184	3.80	3.26	7.95	82	3.03	2.56	5.62	102	4.74	3.95	10.17
H	1	48	214	3.67	3.87	8.02	98	2.96	2.94	5.01	116	5.27	4.89	11.51
H	2	96	241	6.00	6.00	19.35	105	5.35	5.10	15.29	136	7.34	6.80	22.11
H	4	192	267	8.54	8.13	26.72	117	7.54	6.86	17.14	150	9.77	9.27	32.70
H	8	384	284	10.93	11.23	34.35	127	9.61	10.21	17.75	157	11.52	12.14	43.69
H	16	768	293	13.72	13.83	43.31	133	12.16	12.46	26.50	160	15.06	15.08	54.95
H	32	1536	294	17.52	15.92	49.73	132	16.26	14.63	31.65	162	18.11	17.07	60.97
M'L	32	1536	311	20.37	21.14	59.51	147	17.46	17.81	31.71	164	25.33	24.63	83.24

Table 6: Speedups of HordeSat (H) and of MallobSat with a Lingeling-only portfolio (M'L), with the same metrics as in Tab. 4.

mean speedup by 33% while also solving 17 more instances. With our mixed portfolio, MallobSat more than doubles the speedups of HordeSat at all scales (see Tab. 4).



	overall		SAT		UNSAT	
	#	PAR	#	PAR	#	PAR
Mal' KCL	331	122.8	158	142.4	173	103.2
Mal' L	318	153.8	149	183.9	169	125.1
Horde	299	184.8	133	226.6	166	143.1

Figure 15: Updated HordeSat vs. MallobSat on ISC 2021 benchmarks with 1536 cores.

Fig. 15 provides a direct comparison of HordeSat vs. MallobSat on the ISC 2021 benchmarks at 1536 cores. We again show the performance of MallobSat both with Lingeling only as well as with our final mixed portfolio consisting of equal parts of Kissat, CaDiCaL, and Lingeling. Clearly, MallobSat only reaches its full potential with our new solver backends. Since we designed, tested, and tuned our techniques based on these solver backends, this does not imply that HordeSat would improve by the same margin if it employed this portfolio as well. The CBS of MallobSat over HordeSat is 1.98 with equal solvers and 4.54 if MallobSat uses our mixed portfolio.

Note that the ISC 2021 benchmark set also features 20 instances from the ISC 2022 benchmark set, which we used for tuning. To investigate possible overfitting effects, let us also briefly discuss the approaches' performance when excluding these 20 instances (leaving 95% of ISC 2021 instances). On this restricted set, MallobSat L solves 16 instances more than 1536-core HordeSat—three less than when considering all 400 instances. In comparison, when instead excluding 20 *random* instances, the expected advantage of MallobSat L over HordeSat amounts to 18.05 instances. Similarly, the difference in solved instances between MallobSat KCL and HordeSat on our restricted set is below the expected value by 1–3 instances, depending on the scale. The mean speedups of HordeSat and MallobSat (L and KCL) all increase slightly and the relative advantage of MallobSat over HordeSat remains stable or increases; e.g., the speedup of MallobSat L compared to HordeSat's is 33% higher on all 400 instances and 35% higher on the 380 instances.

8.8 MallobSat in the International SAT Competition

The *International SAT Competition* (ISC) is a research-oriented competition (Froleyks et al., 2021) whose first iteration took place in 1992 (Buro & Kleine-Büning, 1992). Since then, over its 25 iterations at the time of writing, the ISC tracked the progress made in SAT solving through new algorithms, techniques, and engineering (Biere et al., 2023).

Our system participated in four iterations of the ISC (2020–2023). We now discuss our submissions of MallobSat and their performance in these four years. In short, our submissions competed with a total of thirteen other submissions in four iterations of the cloud track, winning each time, and competed with a total of 30 other submissions in three iterations of the (shared-memory) parallel track, achieving a top-three rank each time. These results confirm that MallobSat (i) performs well as a general-purpose SAT solver on unseen inputs, (ii) can be efficiently deployed in cloud environments, and (iii) compares favorably to other parallel SAT solvers, from shared-memory to distributed scales. We provide the following discussion as an alternative to an explicit, "fresh" experimental comparison of our system to other existing solvers (beyond HordeSat).

8.8.1 Setup

Since 2020, the parallel and cloud tracks of the ISC are evaluated in Amazon Web Services (AWS) (Balyo et al., 2020). In this cloud environment, solvers are deployed in Docker containers. In the parallel track, one machine of type `m4.16xlarge` is used. This instance type features *Intel Broadwell Xeon E5-2686 v4* processors and is advertised to feature 64 logical cores and 256 GB of RAM.[17] We assume that the machine is in fact two-socket hardware with 18 cores (36 hardware threads) each, totalling 72 hardware threads 64 of which are accessible for users. In the cloud track, 100 machines of type `m4.4xlarge` are used in parallel. This type is advertised to feature 16 virtual CPUs and 64 GB of RAM.[18]

The organizers suggested the following setup for MPI-based systems: Each worker node reports its IP to a single leader node (via `SSH` and the leader node's file system). The leader node then executes the solver system via MPI on the assembled list of IPs. The TCP/IP based message passing resulting from this setup may disadvantage systems with high communication bandwidth. As discussed in Section 8.6, this does not affect our system in any significant manner. As such, apart from the above initial setup and the use of OpenMPI instead of Intel MPI, the configuration of MallobSat in our ISC submissions is in essence the same as in our experiments on HPC clusters.

8.8.2 By-year Discussion

We now discuss each iteration of the ISC separately. Table 7 accompanies this discussion with an overview. For each iteration, we identified all instances that *only* (some configuration of) our system was able to solve among all main, parallel, and cloud track solvers. We provide these lists online. We believe that many of these *exclusively solved instances* are likely to have never been solved before (see Section 8.9 for related follow-up experiments).

ISC 2020. The first time MallobSat participated in the ISC (Schreiber, 2020), our work was focused on preparing a HordeSat-based malleable SAT solving engine with reasonable efficiency, with the aim of deploying it within Mallob. Our configuration performed very cautious clause sharing (up to clause length five) and ran an updated Lingeling portfolio also including YalSAT. We did not consider a submission to the parallel track that year.

17. `https://aws.amazon.com/ec2/dedicated-hosts/pricing`
18. `https://aws.amazon.com/de/ec2/instance-types/`

Year	Track	Submission name	#	PAR	Rank
2020	Cloud	mallob-mono	299	583	1.
		TopoSAT 2	278	706	2.
2021	Cloud	Mallob-HC*	337	373	—
		Mallob	316	480	1.
		MergeHordeSat	260	858	2.
	Parallel	P-MCOMSPS	320	2386	1.
		Mallob-parallel	318	2411	2.
2022	Cloud	Mallob-KiCaLiGlu	341	345	1.
		Paracooba	221	795	2.
	Parallel	ParkissatRS	326	2105	1.
		Mallob-Ki	292	2988	3.
	Anniv. Cloud	Mallob-KiCaLiGlu	4687	279	1.
		Paracooba	3619	725	2.
	Anniv. Parallel	Mallob-Ki	4400	1992	1.
		MergeSAT-AWS	4076	2690	2.
2023	Cloud	Mallob1600	328	426	1.
		PRS-distributed	305	531	2.
	Parallel	PRS-parallel	320	2272	1.
		Mallob64	301	2746	2.
		Mallob32	293	2941	—

Table 7: Overview of MallobSat's performance in ISC 2020–2023, including all of our submissions as well as the best ranking competitor for each track. The 2022 Anniversary tracks featured 5355 benchmark instances; all other tracks featured 400 instances. *Mallob-HC was not ranked officially since the mixed portfolio it employed was disallowed in the competition.

Our SAT solving engine proved to be competitive even though its main feature, malleability, remained unused. MallobSat outperformed the second cloud solver TopoSAT 2 by a decent margin (Balyo et al., 2020) and was able to solve two (“`Steiner`”) instances exclusively. However, since the organizers did not provide performance data of HordeSat, the degree to which MallobSat advanced the state of the art remained unclear.

ISC 2021. Following the success of MallobSat in 2020, we submitted a configuration both to the parallel and the cloud track (Schreiber, 2021b). Among that year's improvements, MallobSat now shared clauses up to length 30 and also targeted a significantly higher sharing volume, from around 66 000 literals in 2020 up to 268 000 (cloud track) and 24 000 (parallel track) literals per sharing respectively. We also introduced a rudimentary and probabilistic clearing of our (Bloom) clause filters at a half life of 300 s. Motivated by huge formulas in the prior year's benchmark set (Froleyks, 2020), we added basic memory awareness to MallobSat, spawning fewer solver threads based on formula size.

MallobSat in 2021 dominated the cloud track, solving 56 instances more than the submission that placed second. MallobSat also proved competitive in the parallel track, scoring the second place overall as the only solver with a top-3 rank on both satisfiable and unsatisfiable instances. While mixing multiple solver backends was disallowed, the organizers kindly ran such a cloud track version of MallobSat *hors concours*. This version, Mallob-HC, outperformed all other solvers and was able to solve 42 out of 400 instances exclusively. Following this competition, the unprecedented performance of our system was acknowledged in an Amazon Science blog post on automated reasoning, commenting that it "*is now, by a **wide** margin, the most powerful SAT solver on the planet*" (Cook, 2021).

ISC and FLoC Olympic Games 2022. 2022 marked the first iteration of the ISC where authors were allowed to submit *mixed portfolios* with different sequential solvers to the cloud track (Balyo et al., 2022b). We configured MallobSat to use Kissat, CaDiCaL, Lingeling, and Glucose. For the first time, MallobSat used subprocessing and an early version of distributed filtering. In the parallel track (where mixed portfolios remain forbidden) we intended to submit a Kissat portfolio (Schreiber, 2022) but in fact submitted a Lingeling portfolio due to a misconfiguration. The same mistake also led to a deviating sharing volume of 17k literals instead of the intended 24k literals in the parallel track.

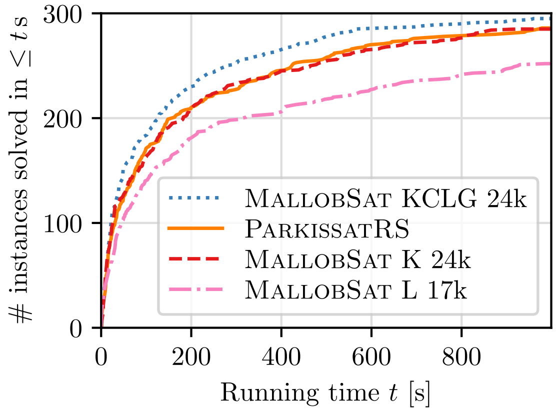
The cloud track, unfortunately, only saw two qualified participants (with a third one disqualified)—MallobSat and Paracooba (Heisinger, 2022). In the parallel track, our misconfigured submission actually scored the top rank in all unsatisfiability sub-tracks. Overall our parallel solver scored the 3rd place. We identified 16 exclusively solved instances for our system among the 400 benchmark instances. We ran a follow-up evaluation on 64 hardware threads to assess the impact of our misconfiguration in the parallel track. Fig. 16 shows that our intended configuration outperforms our submitted configuration and is on par with the winning parallel solver ParkissatRS (Zhang et al., 2022) at a timeout of 1000 s. Our mixed portfolio would further improve performance by a considerable margin.

Since 2022 marked the 25th iteration of the ISC, it featured three further tracks on the *Anniversary benchmark set*—the largest set of SAT instances yet, with a total of 5355 formulas. MallobSat won both the cloud and the parallel Anniversary track. Since this benchmark set features many instances on which solvers have been tuned for years, these results are not necessarily as meaningful as those in the usual tracks.[19] We did identify 266 exclusively solved instances for our system among these instances.

ISC 2023. In the most recent iteration of the ISC at the time of writing, we submitted our system in a configuration similar to our final tuned version in this chapter with some notable differences (Schreiber, 2023a). Our submission shares clauses *three* times a second and still uses an older buffer limit function. It also contains a bug in the clause filtering that leads to suboptimal by-solver filtering and only features a preliminary compensation technique for unused sharing volume. We submitted two variants to the parallel track—one that employs 32 solver threads and one that employs 64 solver threads, in both cases within a single process—in order to assess whether using all logical cores is in fact beneficial. In the cloud track, MallobSat now spawns a single process for each physical machine.

The 2023 iteration of the ISC marks the first year where competitors in the sequential main track were allowed to specify the proof validation tool that should be used for their

19. Biere et al. (2020) suggest to use a benchmark set for up to three years after its publication.



	#	PAR-2
MallobSat KCLG 24k	295	623.4
ParkissatRS	286	687.0
MallobSat K 24k	285	692.5
MallobSat L 17k	252	858.6

Figure 16: Performance at 64 hardware threads of ParkissatRS (2022) and of MallobSat with the submitted 2022 configuration ("L 17k"), the intended configuration ("K 24k"), and a mix of all supported solvers ("KCLG 24k", a configuration that would be disallowed in the parallel track).

submission (Balyo et al., 2023). Some of these proof systems allow for more powerful reasoning than the classical resolution calculus most conventional CDCL solvers still use. Even though parallel solvers are not required to emit proofs, this change indirectly affected all tracks: Since the ISC prompts authors to "*Bring Your Own Benchmarks*", the submissions using other proof systems were accompanied by benchmarks that are known to be difficult to solve with pure resolution (Bogaerts et al., 2023; Codel et al., 2023).

The 2023 cloud track featured a small but strong set of competitors, with our submission of MallobSat winning by the smallest margin so far (yet still decidedly). On satisfiable instances the new system PRS-distributed (Zhang et al., 2023) achieved performance close to MallobSat whereas on unsatisfiable instances the system Malloblin (Chowdhury, 2023) performed similarly. The latter system is a fork of MallobSat that replaced Lingeling's local search solver YalSAT with an alternative named yallin. Since this local search solver cannot find unsatisfiability, we attribute Malloblin's success on unsatisfiable instances to the choice of Lingeling vs. our mixed portfolio. In the parallel track, Malloblin achieved the second place on unsatisfiable instances, similar to our Lingeling-based submission in 2022. On satisfiable instances and in the overall rating, our own submission of MallobSat with 64 solvers scored the second place after PRS-parallel (Zhang et al., 2023). Our 32-solver variant performed worse than the 64-solver variant, indicating that using all advertised logical cores is beneficial on the used hardware. Despite the stronger competition, our system exclusively solved four instances.

8.9 Conquering Hard Instances

To reinforce our claim that MallobSat pushes the frontier of solvable problems, we identified instances from the Anniversary 2022 benchmark set that remained unsolved in the 2022 competition and that, based on the available metadata (Iser & Sinz, 2019), have presumably *never* been claimed as solved before. We grouped these exceptionally hard problems by instance family and author, discarded all instances missing this data, and picked the smallest instance (w.r.t. file size) from each group. This resulted in a selection of 100 instances. We then ran MallobSat on 3072 cores for up to 20 minutes per instance—notably, still a much shorter time span than the sequential time limit in the ISC (1 h 23 min).·

	Instance	Family	Time [s]
Satisfiable	`dislog_a18_x18_n28.cnf`	Discrete logarithm	532
	`gss-28-s100.cnf`	Cryptography	343
	`MD5-28-3.cnf`	Cryptography	427
	`mp1-23.7.cnf`	Cryptography	880
	`quad_res_r29_m32.cnf`	Prime testing	111
	`sum_of_3_cubes_52_bits_39.cnf`	Sum of 3 cubes	1055
Unsatisfiable	`16pipe_16_ooo.cnf`	Hardware verification	859
	`battleship-15-15-unsat.cnf`	Battleship	845
	`connm-ue-csp*sat05-545.cnf`	Generic CSP	658
	`DLTM_twitter690_74_15.cnf`	Influence maximization	647
	`gus-md5-14-sc2009.cnf`	Cryptography	798
	`hid-uns-enc-7-1-0-0-0-0-11545.cnf`	Hidoku	1151
	`mp1-Nb7T08.cnf`	Polynomial multiplication	502
	`mp1-rubikcube120.cnf`	Rubik's Cube	445
	`Mycielski-10-hints-5.cnf`	Mycielski graph	315
	`newpol6-8.cnf`	Polynomial multiplication	698
	`puzzle51_unsat.cnf`	Sliding puzzle	1102
	`rphp5_090_shuffled.cnf`	Pigeon hole	380
	`satch2ways17w.cnf`	Test configuration	850
	`Schur_161_5_d34.cnf`	Schur coloring	871
	`slp-synthesis-aes-bottom17.cnf`	Cryptography	590
	`sokoban-p20.sas.cr.35-sc2016.cnf`	Planning	307

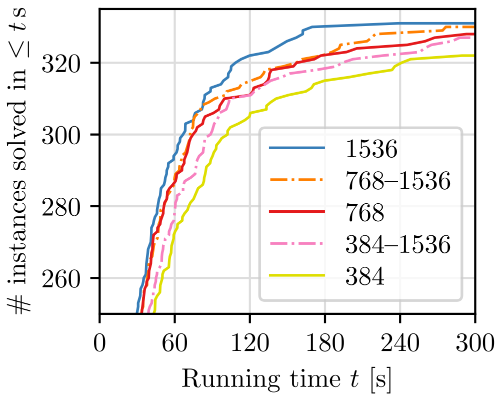
Table 8: Successfully solved hard instances at 3072 cores.

Indeed, MallobSat was successful for 22 instances, given in Tab. 8. Many of the concerned instances originate from diverse application domains such as cryptography, number theory, verification, and puzzle solving. As an example, `16pipe_16_ooo.cnf` is a hardware verification task with "*a superscalar model that can issue up to* [x] *instructions out of program order on every cycle*" (Velev, 2003) for $x = 16$. It was part of the 2010, 2016, and 2022 competitions and remained unsolved in all of them.

Based on the shown results, we can confirm that distributed solving with MallobSat can tackle problems that have been out of reach for prior solvers and at smaller scales.

8.10 Malleable SAT Solving

In the following, we evaluate how the performance of MallobSat is impacted by malleability, i.e., by a fluctuating set of associated workers. We use a 32-machine setup (1536 cores) of our scheduler Mallob with two streams of jobs: a *benchmark stream* and a *disturbance stream*. The benchmark stream sequentially introduces 400 jobs corresponding to the ISC 2021 benchmarks at a time limit of 300 s per instance. The disturbance stream introduces an infeasible[20] formula with a timeout of 10 s every 20 s. We assign a priority p to each disturbance job and keep default priority 1 for each benchmark job. Since Mallob assigns resources proportional to job priority (Sanders & Schreiber, 2022), each benchmark job is forced to yield $\alpha = \frac{p}{p+1}$ of its resources 50% of the time. We run the experiment once with $p = 1$ ($\alpha = 1/2$) and once with $p = 3$ ($\alpha = 3/4$). We compare the performance of our disturbed benchmark stream with the performance observed in our scaling results.



	overall		SAT		UNSAT	
cores	#	PAR	#	PAR	#	PAR
384	322	141.7	157	150.0	165	133.3
384–1536	327	133.9	156	150.5	171	117.4
768	328	130.1	157	146.5	171	113.6
768–1536	330	127.4	157	146.4	173	108.4
1536	331	122.8	158	142.4	173	103.2

Figure 17: SAT performance with(out) fluctuating resources—note the y axis offset.

As Fig. 17 shows, the disturbed run oscillating between 1536 and 384 cores is not able to reach the performance of an undisturbed 768-core run. It does surpass the performance of a 384-core run, which confirms that the fluctuating resources have some merit. Disturbed performance is quite close to 768-core performance for unsatisfiable instances but drops to the 384-core performance for satisfiable instances. Similarly, the run oscillating between 1536 and 768 cores performs well on unsatisfiable instances but drops to the 768-core performance on satisfiable instances. As such, while MallobSat's malleability preserves the performance achieved by the (undisturbed) base resources, the utility of fluctuating resources appears to be limited to unsatisfiable instances. A possible reason is that unsatisfiable instances profit the most from clause sharing. As such, even solvers that are suspended frequently can contribute to the solving effort in a meaningful way by exporting clauses whenever they are active. On satisfiable instances, which benefit less from clause sharing, a solver that is only active half of the time is unlikely to contribute in a truly meaningful manner, i.e., by being the first to find a solution. As such, further work is required to achieve similar positive effects when solving satisfiable instances.

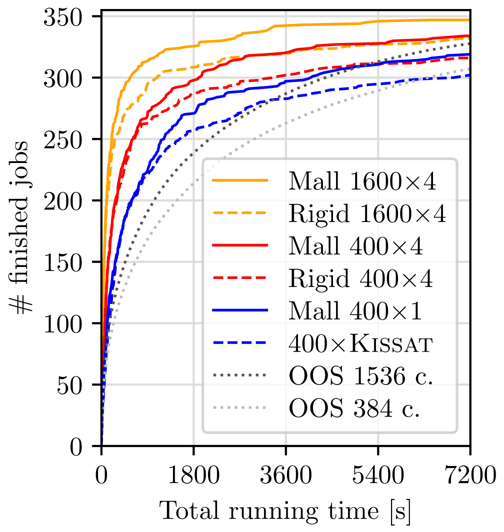
In the last set of experiments, we evaluate MallobSat with malleable scheduling if many concurrent SAT tasks are present. The objective we consider is to complete as many problems as possible among the benchmark instances of ISC 2021 within two hours. We randomly partition the 400 formulas into 16 equally sized groups and configure 16 of our processes to additionally parse and introduce all jobs within a certain group at system start. We compare this malleable scheduling with two simpler scheduling strategies. First,

20. Since the formula encodes the existence of a period-19 pattern in *Conway's Game of Life* (Gardner, 1970) in a 20×20 grid, we would welcome to be proven wrong.

we use the same setup while capping the maximum demand of each job to 1/400 of the total resources. This approach thus runs 400 parallel solvers, all with fixed and equal amounts of resources, in parallel. We refer to this approach as *rigid scheduling*. Secondly, we consider the isolated processing times of MallobSat at a comparable scale and sort the 400 instances by processing time in ascending order. This yields what we refer to as an *Optimal Offline Schedule* (OOS)—the best possible way to *sequentially* schedule runs of our distributed solver if we had perfect knowledge on the processing time of each job.

Setup	m	Cores	Processes	Cores/job
1600×4	136	6400	1600	16
400×4	36	1600	400	4
400×1	9	400	400	1

Table 9: Scales used in experiments on massively parallel job processing.



	Cores	#	PAR
Mall 1600×4	6400	347	2220.8
Rigid 1600×4	6400	332	2825.9
Mall 400×4	1600	334	2909.9
Rigid 400×4	1600	316	3508.3
Mall 400×1	400	319	3635.3
400×Kissat	400	303	5204.5
OOS	1536	328	3695.0
OOS	384	308	4467.4

Figure 18: Performance of different (real-world or virtual) scheduling strategies for the ISC 2021 benchmark set on 384–6400 cores.

As detailed in Tab. 9, we run our scheduling at three scales that initially run each job on 16, four, and a single core respectively. At the lowest scale of 400 cores, instead of running Mallob with rigid scheduling we provide the results of KissatMABHyWalk, pretending that all 400 runs are done in parallel on 400 cores and without any scheduling overhead.

Fig. 18 shows results in terms of finished jobs over time. Note that we computed the OOS at slightly different scales (384 and 1536 cores) compared to the other schedulers (400, 1600, 6400 cores). They are still suited to illustrate the following: Running each job at the maximum available scale may eventually solve the most instances but is clearly not resource efficient. Over long time periods, the OOS stay behind the response times achieved by just running 400×Kissat. All rigid strategies achieve better resource efficiency than the OOS since each job is executed at a comparably low degree of parallelism and in turn all jobs can be processed in parallel. We observed significant speedups for increasing the resources per job from one to four cores and from four to 16 cores respectively.

Enabling our scheduling's full malleability further improves performance. The improvement of our 400-core malleable scheduler over 400×Kissat shows how our system is able to seamlessly shift from a multi-instance sequential solver to a (small-scale) parallel solver, both with state-of-the-art performance. This is achieved while incurring low overhead, even though we run an MPI process for every single solver thread, and without requiring any additional resources except for using both hardware threads on each core. Our 400-core malleable scheduler even surpasses the performance of the 1600-core rigid scheduler after around 1.5 hours. At this point in time, less than 100 jobs remain for the malleable approach and each job thus runs on at least four cores—the resources per job of the rigid approach. The same applies for the 1600-core malleable scheduler, which begins to outperform the 6400-core rigid scheduler after around 40 minutes since each job then receives at least 16 cores. Evidently, MallobSat is able to make good use of these added resources, confirming that our malleable solving approach is useful and effective in such scenarios. At the largest scale, our malleable approach is able to solve 347 instances—far more than any system in the ISC 2021 or in our prior evaluations—while only spending 12 800 ch (core hours).[21]

9. Conclusion

In order to improve the scalability and resource-efficiency of general-purpose SAT solving in large distributed environments, we presented a novel system MallobSat—a successor to HordeSat that features compact clause sharing, state-of-the-art solver backends, and malleability. We showed that this solver advances the state of the art and consistently leads to improved speedups. We also confirmed that most of our approach's scalability is due to our clause sharing approach. Using results from the International SAT Competition as well as our own follow-up experiments, we demonstrated that MallobSat pushes the frontier of problems that are feasible to solve. Last but not least, we showed that Mallob's combination of parallel job processing and flexible parallel SAT solving is able to reduce scheduling and response times and to improve resource efficiency in a distributed environment.

In the scope of this article, we omitted discussing two crucial features of our solving system. First, we have added support for *incremental SAT solving* to MallobSat (Schreiber, 2023b), which will boost its viability for a plethora of applications (Axelsson et al., 2008; Liu et al., 2016; Kleine-Büning et al., 2019; Schreiber, 2021a). Secondly, MallobSat is the first solver of its kind to feature feasible and scalable production of *proofs of unsatisfiability* (Michaelson et al., 2023). We aim to further advance these lines of work in the future and to gain deeper insights from the produced proofs (*cf.* Katsirelos et al., 2013).

In terms of our algorithmic and heuristic methods, possible future work may include more sophisticated clause quality metrics (beyond clause length and LBD; e.g., Vallade et al., 2020) and more informed and automated approaches to constructing diverse solver portfolios. With the increasing attention towards more powerful proof systems (Balyo

21. In our latest scaling experiments MallobSat solved 331 instances using 12 400 ch (at 1536 cores) or 337 instances using 23 100 ch (at 3072 cores).

et al., 2023), another important question is how to extend clause sharing to support this more sophisticated reasoning. For instance, a preprocessing technique called *Structured Bounded Variable Addition* (SBVA; Haberlandt et al., 2023) led to impressive results in the most recent SAT Competition (Balyo et al., 2023). It appears natural to investigate how to enable sound clause sharing in conjunction with such techniques and also how to parallelize pre– and inprocessing such as SBVA at distributed scales in order to reduce the redundant work performed across solvers (*cf.* Gebhardt & Manthey, 2013).

It remains to be seen whether tightly integrated parallel solvers such as Gimsatul (Fleury & Biere, 2022) continue to gain traction and eventually outperform conventional, modular clause-sharing solvers due to higher efficiency. In this context, a natural approach would be to orchestrate such integrated solvers, one per machine or per socket, with our distributed clause sharing techniques. Conversely, there are recent developments of powerful application interfaces emerging for SAT solvers, such as IPASIR-UP (Fazekas et al., 2023) and IPASIR 2,[22] which offer sophisticated ways to orchestrate solvers and may hence allow to develop efficient clause-sharing solvers without any kind of solver-specific code.

Acknowledgments

The authors thank the anonymous reviewers for their valuable feedback. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`www.lrz.de`). Some of this work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. Some of this work was performed on the computational resource bwUniCluster funded by the Ministry of Science, Research and the Arts Baden-Württemberg and the Universities of the State of Baden-Württemberg, Germany, within the framework program bwHPC. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 882500).

22. Under active development, see `https://github.com/ipasir2`

Appendix A. Solver Configuration

	Idx.	Configuration
Lingeling	A	classify=0
	0	gluescale=5
	1	plain=1 decompose=1
	2	plain=0|1 locs=-1 locsrtc=1 locswait=0 locsclim=16777216
	3	restartint=100
	4	sweeprtc=1
	5	restartint=1000
	6	scincinc=50
	7	restartint=4
	8	phase=1
	9	phase=-1
	10	block=0 cce=0
Glucose	A	!adaptStrats simplify [>2]randomizeFirstDescent [>2]rndInitAct
	0	adaptStrats simplify
	1	lubyRestart lubyRestartFactor=100 (max)VarDecay=.999
	2	chanseokStrat coLBDBound=4 glureduce 1stReduceDB=2k clsBeforeReduce=2k !incReduceDb
	3	(max)VarDecay=.95 firstReduceDB=4k lbdQueueSize=100 K=0.7 incReduceDB=500
	4	!adaptStrats !simplify
	5	chanseokStrat
	6	adaptStrats !simplify
	7	chanseokStrat coLBDBound=3 glureduce 1stReduceDB=30k (max)VarDecay=.99 randomizeOnRestarts
CaDiCaL	0	phase=0
	1	config=sat
	2	elim=0
	3	config=unsat
	4	condition=1
	5	walk=0
	6	restartint=100
	7	cover=1
	8	shuffle=1 shufflerandom=1
	9	inprocessing=0
Kissat	A	quiet=1 check=0
	0	eliminate=0
	1	delay=10
	2	restartint=100
	3	walkinitially=1
	4	restartint=1000
	5	sweep=0
	6	config=unsat
	7	config=sat
	8	probe=0
	9	failedcont=50 failedrounds=10
	10	minimizedepth=10^4
	11	modeconflicts=10^5 modeticks=10^9
	12	reducefraction=90
	13	vivifyeffort=1000
	14	xorsclslim=8

Table 10: Solver configuration of MallobSat (Chapter 6.1). "A" denotes the default for all configurations; "[>2]" denotes configuring all but the first three solvers.

References

Aigner, M., Biere, A., Kirsch, C. M., Niemetz, A., & Preiner, M. (2013). Analysis of portfolio-style parallel SAT solving on current multi-core architectures. In *Pragmatics of SAT*, Vol. 29, pp. 28–40.

Arbelaez, A., & Codognet, P. (2012). Massively parallel local search for SAT. In *Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, Vol. 1, pp. 57–64. IEEE.

Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.-M., & Piette, C. (2012). Revisiting clause exchange in parallel SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 200–213. Springer.

Audemard, G., Hoessen, B., Jabbour, S., & Piette, C. (2014). Dolius: A distributed parallel SAT solving framework. In *Pragmatics of SAT*, pp. 1–11. Citeseer.

Audemard, G., Lagniez, J.-M., Szczepanski, N., & Tabary, S. (2016). An adaptive parallel SAT solver. In *Principles and Practice of Constraint Programming (CP)*, pp. 30–48. Springer.

Audemard, G., Lagniez, J.-M., Szczepanski, N., & Tabary, S. (2017). A distributed version of syrup. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 215–232. Springer.

Audemard, G., & Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. In *Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pp. 399–404.

Audemard, G., & Simon, L. (2014). Lazy clause exchange policy for parallel SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 197–205. Springer.

Audemard, G., & Simon, L. (2017). Glucose and syrup in the SAT'17. In *Proc. SAT Competition*, pp. 16–17.

Axelsson, R., Heljanko, K., & Lange, M. (2008). Analyzing context-free grammars using an incremental SAT solver. In *Int. Colloquium on Automata, Languages, and Programming (ICALP)*, pp. 410–422. Springer.

Bach, J., Iser, M., & Böhm, K. (2022). A comprehensive study of k-portfolios of recent SAT solvers. In *Pragmatics of SAT*, pp. 2:1–2:18.

Balyo, T., Biere, A., Iser, M., & Sinz, C. (2016). SAT race 2015. *Artificial Intelligence*, *241*, 45–65.

Balyo, T., Froleyks, N., Heule, M. J. H., Iser, M., Järvisalo, M., & Suda, M. (2020). The results of SAT competition 2020. `https://satcompetition.github.io/2020/downloads/satcomp20slides.pdf`.

Balyo, T., Froleyks, N., Heule, M. J. H., Iser, M., Järvisalo, M., & Suda, M. (Eds.). (2021). *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki.

Balyo, T., Heule, M. J. H., Iser, M., Järvisalo, M., & Suda, M. (Eds.). (2022a). *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki.

Balyo, T., Heule, M. J. H., Iser, M., Järvisalo, M., & Suda, M. (2022b). The results of SAT competition 2022. `https://satcompetition.github.io/2022/slides/satcomp22slides.pdf`.

Balyo, T., Heule, M. J. H., Iser, M., Järvisalo, M., & Suda, M. (2023). The results of SAT competition 2023. `https://satcompetition.github.io/2023/downloads/satcomp23slides.pdf`.

Balyo, T., Heule, M. J. H., & Jarvisalo, M. (2017). SAT competition 2016: Recent developments. In *AAAI Conference on Artificial Intelligence*, Vol. 31, pp. 5061–5063.

Balyo, T., Sanders, P., & Sinz, C. (2015). Hordesat: A massively parallel portfolio SAT solver. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 156–172. Springer.

Balyo, T., & Sinz, C. (2018). Parallel satisfiability. In Hamadi, Y., & Sais, L. (Eds.), *Handbook of Parallel Constraint Reasoning*. Springer.

Belov, A., Diepold, D., Heule, M. J. H., & Järvisalo, M. (2014). SAT competition. `http://satcompetition.org/2014/index.shtml`.

Biere, A. (2010). Lingeling, Plingeling, Picosat and Precosat at SAT race 2010. In *Proc. SAT Competition*.

Biere, A. (2012). Lingeling and friends entering the SAT challenge 2012. In *Proc. SAT Challenge*, pp. 33–34.

Biere, A. (2013). Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In *Proc. SAT Competition*, Vol. 2013, p. 1.

Biere, A. (2014). Yet another local search solver and lingeling and friends entering the SAT competition 2014. In *Proc. SAT Competition*, No. 2, p. 65.

Biere, A. (2015). Lingeling and friends entering the SAT race 2015. In *FMV Reports Series*.

Biere, A. (2016). Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2016. In *Proc. SAT Competition*, pp. 44–45.

Biere, A. (2017). CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT competition 2017. In *Proc. SAT Competition*, pp. 14–15.

Biere, A. (2018). CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT competition 2018. In *Proc. SAT Competition*, pp. 14–15.

Biere, A. (2021). CNF encodings of complete pairwise combinatorial testing of our SAT solver SATCH. In *Proc. SAT Competition*, p. 46.

Biere, A., Chowdhury, M. S., Heule, M. J. H., Kiesl, B., & Whalen, M. W. (2022). Migrating solver state. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 27:1–27:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Biere, A., Fazekas, K., Fleury, M., & Heisinger, M. (2020). CaDiCaL, kissat, paracooba, plingeling and treengeling entering the SAT competition 2020. In *Proc. SAT Competition*, p. 50.

Biere, A., Fleury, M., Froleyks, N., & Heule, M. J. H. (2023). The SAT museum. In *Pragmatics of SAT*, pp. 72–87.

Biere, A., & Fröhlich, A. (2015). Evaluating CDCL variable scoring schemes. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 405–422. Springer.

Biere, A., & Heule, M. J. H. (2019). The effect of scrambling CNFs. In *Pragmatics of SAT*, pp. 111–126.

Biere, A., Järvisalo, M., & Kiesl, B. (2021). Preprocessing in SAT solving. In *Handbook of Satisfiability*, pp. 391–435. IOS Press.

Biere, A., Järvisalo, M., Le Berre, D., Meel, K. S., & Mengel, S. (2020). The SAT practitioner's manifesto. `https://doi.org/10.5281/zenodo.4500928`.

Birrittella, M. S., Debbage, M., Huggahalli, R., Kunz, J., Lovett, T., Rimmer, T., Underwood, K. D., & Zak, R. C. (2015). Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *IEEE Symp. High-Performance Interconnects*, pp. 1–9.

Blochinger, W., Sinz, C., & Küchlin, W. (2003). Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, *29*(7), 969–994.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Comm. ACM*, *13*(7), 422–426.

Blumofe, R. D., & Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *J. ACM*, *46*(5), 720–748.

Bogaerts, B., Nordström, J., Oertel, A., & Yıldırımoglu, C. U. (2023). Crafted benchmark formulas requiring symmetry breaking and/or parity reasoning. In *Proc. SAT Competition*, p. 67.

Böhm, M., & Speckenmeyer, E. (1996). A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, *17*, 381–400.

Burgess, M. A., Gretton, C., Milthorpe, J., Croak, L., Willingham, T., & Tiu, A. (2022). Dagster: Parallel structured search with case studies. In *Pacific Rim Int. Conf. Artificial Intelligence*, pp. 75–89. Springer.

Buro, M., & Kleine-Büning, H. (1992). *Report on a SAT competition.* Fachbereich Math.-Informatik, Univ. Gesamthochschule Paderborn, Germany.

Cai, S., Zhang, X., Fleury, M., & Biere, A. (2022). Better decision heuristics in CDCL through local search and target phases. *JAIR*, *74*, 1515–1563.

Cen, Y., Zhang, Z., & Fong, X. (2023). Massively parallel continuous local search for hybrid SAT solving on GPUs. `https://arxiv.org/abs/2308.15020`.

Chowdhury, M. S. (2023). kissat-hywalk-gb, kissat-hywalk-exp, kissat-hywalk-exp-gb, and malloblin entering the SAT competition-2023. In *Proc. SAT Competition*, p. 28.

Chrabakh, W., & Wolski, R. (2003). GrADSAT: A parallel SAT solver for the grid. In *IEEE SC03*, Vol. 53.

Clarke, E., Kroening, D., & Lerda, F. (2004). A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 168–176. Springer.

Codel, C. R., Reeves, J. E., & Bryant, R. E. (2023). Pigeon hole and mutilated chessboard with mixed constraint encodings and symmetry-breaking. In *Proc. SAT Competition*, p. 72.

Cook, B. (2021). Automated reasoning's scientific frontiers. `https://www.amazon.science/blog/automated-reasonings-scientific-frontiers`. Amazon Science.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proc. ACM Symp. on Theory of computing*, pp. 151–158.

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Comm. ACM*, *5*(7), 394–397.

Eén, N., & Biere, A. (2005). Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 61–75. Springer.

Eén, N., & Sörensson, N. (2004). An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 502–518. Springer.

Ehlers, T., Kulczynski, M., Nowotka, D., & Sieweck, P. (2020). TopoSAT 2. In *Proc. SAT Competition*, p. 60.

Ehlers, T., & Nowotka, D. (2019). Tuning parallel SAT solvers. In *Pragmatics of SAT*, Vol. 59, pp. 127–143.

Ehlers, T., Nowotka, D., & Sieweck, P. (2014). Communication in massively-parallel SAT solving. In *Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, pp. 709–716. IEEE.

Elffers, J., & Nordström, J. (2016). Documentation of some combinatorial benchmarks. In *Proc. SAT Competition*, pp. 67–69.

Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., & Biere, A. (2023). IPASIR-UP: User propagators for CDCL. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 8:1–8:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Feitelson, D. G. (1997). Job scheduling in multiprogrammed parallel systems. IBM Research Report, Citeseer.

Fichte, J. K., Le Berre, D., Hecher, M., & Szeider, S. (2023). The silent (r)evolution of SAT. *Comm. ACM*, *66*(6), 64–72.

Fleming, P. J., & Wallace, J. J. (1986). How not to lie with statistics: the correct way to summarize benchmark results. *Comm. ACM*, *29*(3), 218–221.

Fleury, M., & Biere, A. (2022). Scalable proof producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses. In *Pragmatics of SAT*.

Fofaliya, R., Grundy, J., Jones, R., Khazem, K., Kiesl, B., Nakos, A., Tautschnig, M., & Whalen, M. W. (2022). AWS CBMC benchmarks. In *Proc. SAT Competition*, p. 54.

Fossé, R., & Simon, L. (2018). On the non-degeneracy of unsatisfiability proof graphs produced by SAT solvers. In *Principles and Practice of Constraint Programming (CP)*, pp. 128–143. Springer.

Froleyks, N. (2020). Planning track benchmarks. In *Proc. SAT Competition*, p. 64.

Froleyks, N., Heule, M. J. H., Iser, M., Järvisalo, M., & Suda, M. (2021). SAT competition 2020. *Artificial Intelligence*, *301*, 103572.

Gardner, M. (1970). The fantastic combinations of John Conway's new solitaire game "Life". *Sc. Am.*, *223*, 20–123.

Gebhardt, K., & Manthey, N. (2013). Parallel variable elimination on cnf formulas. In *Annual Conference on Artificial Intelligence*, pp. 61–73. Springer.

Gomes, C. P., & Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, *126*(1-2), 43–62.

Gomes, C. P., Selman, B., Crato, N., & Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reason.*, *24*(1), 67–100.

Graham, R. L., Shipman, G. M., Barrett, B. W., Castain, R. H., Bosilca, G., & Lumsdaine, A. (2006). Open mpi: A high-performance, heterogeneous mpi. In *IEEE Int. Conf. Cluster Computing*, pp. 1–9. IEEE.

Grinten, A. v. d. (2017). *Design, implementation and evaluation of a distributed CDCL framework*. Ph.D. thesis, Universität zu Köln.

Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI: portable parallel programming with the message-passing interface*, Vol. 1. MIT press.

Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Comm. ACM*, *31*(5), 532–533.

Haberlandt, A., Green, H., & Heule, M. J. H. (2023). Effective auxiliary variables via structured reencoding. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 11:1–11:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Hamadi, Y., Jabbour, S., & Sais, J. (2012). Control-based clause sharing in parallel SAT solving. In *Autonomous Search*, pp. 245–267. Springer.

Hamadi, Y., Jabbour, S., & Sais, L. (2010). ManySAT: a parallel SAT solver. *JSAT*, *6*(4), 245–262.

Heisinger, M. (2022). Paracooba enters SAT competition 2022. In *Proc. SAT Competition*, p. 42.

Heisinger, M., Fleury, M., & Biere, A. (2020). Distributed cube and conquer with paracooba. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 114–122. Springer.

Heule, M. J. H. (2018). Schur number five. In *AAAI Conference on Artificial Intelligence*, Vol. 32, pp. 6598–6606.

Heule, M. J. H. (2021). Hamiltonian cycle instances using the chinese remainder encoding. In *Proc. SAT Competition*, p. 54.

Heule, M. J. H., Kullmann, O., & Marek, V. (2016). Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 228–245. Springer.

Heule, M. J. H., Kullmann, O., Wieringa, S., & Biere, A. (2011). Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference*, pp. 50–65. Springer.

Heule, M. J. H., & van Maaren, H. (2021). Look-ahead based SAT solvers. In *Handbook of Satisfiability*, pp. 155–184. IOS Press.

Ignatiev, A., Morgado, A., & Marques-Silva, J. (2017). On tackling the limits of resolution in SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 164–183. Springer.

Iser, M., Balyo, T., & Sinz, C. (2019). Memory efficient parallel SAT solving with inprocessing. In *Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, pp. 64–70. IEEE.

Iser, M., & Sinz, C. (2019). A problem meta-data library for research in SAT. In *Pragmatics of SAT*, Vol. 59, pp. 144–152.

Jurkowiak, B., Li, C. M., & Utard, G. (2001). Parallelizing satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, *9*, 174–189.

Katsirelos, G., Sabharwal, A., Samulowitz, H., & Simon, L. (2013). Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *AAAI Conference on Artificial Intelligence*, Vol. 27, pp. 481–488.

Kautz, H. A., Sabharwal, A., & Selman, B. (2021). Incomplete algorithms. In *Handbook of Satisfiability*, pp. 185–203. IOS Press.

Khan, A., Sim, H., Vazhkudai, S. S., Butt, A. R., & Kim, Y. (2021). An analysis of system balance and architectural trends based on TOP500 supercomputers. In *Proc. High Performance Computing in Asia-Pacific Region*, pp. 11–22.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., & Leyton-Brown, K. (2016). SATenstein: Automatically building local search SAT solvers from components. *Artificial Intelligence*, *232*, 20–42.

Kleine-Büning, M., Balyo, T., & Sinz, C. (2019). Using DimSpec for bounded and unbounded software model checking. In *Int. Conf. on Formal Engineering Methods (ICFEM)*, pp. 19–35. Springer.

Kottler, S., & Kaufmann, M. (2011). SArTagnan – a parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*.

Le Frioux, L., Baarir, S., Sopena, J., & Kordon, F. (2017a). PaInleSS: a framework for parallel SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 233–250. Springer.

Le Frioux, L., Baarir, S., Sopena, J., & Kordon, F. (2017b). painless-maplecomsps. In *Proc. SAT Competition*, p. 26.

Liang, J. H., Ganesh, V., Poupart, P., & Czarnecki, K. (2016a). Learning rate based branching heuristic for SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 123–140. Springer.

Liang, J. H., Oh, C., Ganesh, V., Czarnecki, K., & Poupart, P. (2016b). MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB. In *Proc. SAT Competition*, pp. 52–53.

Liu, D., Yu, C., Zhang, X., & Holcomb, D. (2016). Oracle-guided incremental SAT solving to reverse engineer camouflaged logic circuits. In *Design, Automation & Test in Europe (DATE)*, pp. 433–438. IEEE.

Manthey, N., Philipp, T., & Wernhard, C. (2013). Soundness of inprocessing in clause sharing SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 22–39. Springer.

Marques-Silva, J., Lynce, I., & Malik, S. (2021). CDCL SAT solving. In *Handbook of Satisfiability*, pp. 131–153. IOS Press.

Mengel, S. (2021). A naive SAT-encoding of cluster editing. In *Proc. SAT Competition*, p. 62.

Michaelson, D., Schreiber, D., Heule, M. J. H., Kiesl-Reiter, B., & Whalen, M. W. (2023). Unsatisfiability proofs for distributed clause-sharing sat solvers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 348–366. Springer.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pp. 530–535.

Ngoko, Y., Trystram, D., & Cérin, C. (2017). A distributed cloud service for the resolution of SAT. In *Proc. IEEE SC2*, pp. 1–8.

Oh, C. (2015). Between sat and unsat: the fundamental difference in cdcl sat. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 307–323. Springer.

Oh, C. (2016). *Improving SAT solvers by exploiting empirical characteristics of CDCL*. Ph.D. thesis, New York University.

Ohmura, K., & Ueda, K. (2009). c-sat: A parallel SAT solver for clusters. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 524–537. Springer.

Osama, M., & Wijs, A. (2021). Verifying string safety properties in AWS C99 package with CBMC. In *Proc. SAT Competition*, p. 64.

Osama, M., Wijs, A., & Biere, A. (2023). Certified SAT solving with GPU accelerated inprocessing. *Formal Methods in System Design*, *62*, 79–118.

Ozdemir, A., Wu, H., & Barrett, C. (2021). SAT solving in the serverless cloud. In *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 241–245. IEEE.

Pipatsrisawat, K., & Darwiche, A. (2007). A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 294–299. Springer.

Prevot, N., Soos, M., & Meel, K. S. (2021). Leveraging GPUs for effective clause sharing in parallel SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 471–487. Springer.

Reeves, J. E., & Bryant, R. E. (2023). Preprocessors PReLearn and ReEncode entering the SAT competition 2023. In *Proc. SAT Competition*, p. 23.

Roussel, O. (2012). Description of ppfolio (2011). In *Proc. SAT Challenge*, p. 46.

Sanders, P. (1994). A detailed analysis of random polling dynamic load balancing. In *International Symposium on Parallel Architectures, Algorithms and Networks*, pp. 382–389, Kanazawa, Japan. IEEE, Los Alamitos, CA.

Sanders, P., Mehlhorn, K., Dietzfelbinger, M., & Dementiev, R. (2019). *Sequential and Parallel Algorithms and Data Structures*. Springer.

Sanders, P., & Schreiber, D. (2022). Decentralized online scheduling of malleable NP-hard jobs. In *Proc. Euro-Par*, pp. 119–135. Springer.

Schick, M. (2021). Cube&conquer-inspired malleable distributed SAT solving. Master's thesis, Karlsruhe Institute of Technology (KIT).

Schreiber, D. (2020). Engineering HordeSat towards malleability: mallob-mono in the SAT 2020 cloud track. In *Proc. SAT Competition*, pp. 45–46.

Schreiber, D. (2021a). Lilotane: A lifted SAT-based approach to hierarchical planning. *JAIR*, *70*, 1117–1181.

Schreiber, D. (2021b). Mallob in the SAT competition 2021. In *Proc. SAT Competition*, pp. 38–39.

Schreiber, D. (2022). Mallob in the SAT competition 2022. In *Proc. SAT Competition*, pp. 46–47.

Schreiber, D. (2023a). Mallob{32,64,1600} in the SAT competition 2023. In *Proc. SAT Competition*, pp. 46–47.

Schreiber, D. (2023b). *Scalable SAT Solving and its Application.* Ph.D. thesis, Karlsruhe Institute of Technology. `https://doi.org/10.5445/IR/1000165224`.

Schreiber, D., & Sanders, P. (2021). Scalable SAT solving in the cloud. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 518–534. Springer.

Schulz, S., & Blochinger, W. (2010). Cooperate and compete! a hybrid solving strategy for task-parallel SAT solving on peer-to-peer desktop grids. In *Proc. Int. Conf. HPC & Simulation*, pp. 314–323. IEEE.

Simon, L. (2014). Post mortem analysis of SAT solver proofs. In *Pragmatics of SAT*, pp. 26–40.

Sinz, C., Blochinger, W., & Küchlin, W. (2001). PaSAT – parallel SAT-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, *9*, 205–216.

Spence, I. (2017). Balanced random SAT benchmarks. In *Proc. SAT Competition*, p. 53.

Subercaseaux, B., & Heule, M. J. H. (2023). The packing chromatic number of the infinite square grid is 15. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 389–406. Springer.

Tseitin, G. S. (1983). On the complexity of derivation in propositional calculus. In *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pp. 466–483. Springer.

Vallade, V., Le Frioux, L., Baarir, S., Sopena, J., Ganesh, V., & Kordon, F. (2020). Community and LBD-based clause sharing policy for parallel SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 11–27. Springer.

Vallade, V., Le Frioux, L., Oanea, R., Baarir, S., Sopena, J., Kordon, F., Nejati, S., & Ganesh, V. (2021). New concurrent and distributed painless solvers: p-mcomsps, p-mcomsps-com, p-mcomsps-mpi, and p-mcomsps-com-mpi. In *Proc. SAT Competition*, p. 40.

Velev, M. N. (2003). Automatic abstraction of equations in a logic of equality. In *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, pp. 196–213. Springer.

Weber, T. (2005). A SAT-based sudoku solver. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pp. 11–15.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2012). Evaluating component solver contributions to portfolio-based algorithm selectors. In *Theory and Applications of Satisfiability Testing (SAT)*, pp. 228–241. Springer.

Zhang, H., Bonacina, M. P., & Hsiang, J. (1996). PSATO: a distributed propositional prover and its application to quasigroup problems. *J. Symbolic Computation*, *21*(4-6), 543–560.

Zhang, X., Chen, Z., & Cai, S. (2022). Parkissat: Random shuffle based and pre-processing extended parallel solvers with clause sharing. In *Proc. SAT Competition*, p. 51.

Zhang, X., Chen, Z., & Cai, S. (2023). PRS: A new parallel/distributed framework for SAT. In *Proc. SAT Competition*, pp. 39–40.

Zheng, J., He, K., Chen, Z., Zhou, J., & Li, C.-M. (2022). Combining hybrid walking strategy with kissat mab, cadical, and lstech-maple. In *Proc. SAT Competition*, p. 20.