From Scalable SAT to MaxSAT: Massively Parallel Solution Improving Search

Dominik Schreiber[1], **Christoph Jabs**[2], **Jeremias Berg**[2]

[1]KIT Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany
[2]Department of Computer Science, University of Helsinki, Helsinki, Finland
dominik.schreiber@kit.edu, christoph.jabs@helsinki.fi, jeremias.berg@helsinki.fi

Abstract

Maximum Satisfiability (MaxSAT) is an essential framework for combinatorial optimization at the core of automated reasoning. However, to date, no notable parallelizations with convincing scaling behaviour exist. We suggest to exploit and transfer recent advances in massively parallel SAT solving to perform scalable solution improving search (SIS) for MaxSAT solving. Building upon the distributed job scheduling and SAT solving platform MALLOB, we present the first MaxSAT solver that scales to hundreds of cores through a careful combination of parallel and distributed incremental SAT solving, task parallelism and flexible load balancing, and clause sharing within and across SAT solving tasks. Experiments on up to 768 cores (16 nodes) show that our approach clearly outscales state-of-the-art SIS-based MaxSAT solvers, marking a new baseline for parallel MaxSAT solving.

Introduction

Boolean satisfiability (SAT) solving is an essential tool at the core of automated reasoning (Fichte et al. 2023) where a propositional formula is searched for a consistent solution or shown to be *unsatisfiable*. Whenever an application features a notion of quality, i.e., some quantified *cost* or *merit* of solutions, *Maximum Satisfiability* (MaxSAT) is the problem of finding optimal or near-optimal solutions. MaxSAT solvers, which, in essence, search for a solution that minimizes a linear objective function, have seen many successful applications (Bacchus, Järvisalo, and Martins 2021; Li and Manyà 2021), such as automaton minimization (Heizmann, Schilling, and Tischner 2017), explainable AI (Ignatiev et al. 2022), or train scheduling (Lemos et al. 2024).

While there have been recent developments in MaxSAT via *Branch&Bound* (Li et al. 2021), the clear majority of the state-of-the-art MaxSAT solvers are SAT-based, i.e., reduce the optimization problem to a sequence of decision problems, which, in turn, are tackled with a CDCL SAT solver (Bacchus, Järvisalo, and Martins 2021; Berg et al. 2024; Marques-Silva, Lynce, and Malik 2021). Out of these, we focus on so-called solution-improving search (SIS) (Eén and Sörensson 2006; Paxian, Reimer, and Becker 2018), which repeatedly queries a SAT solver for solutions of increasing quality until an optimal one is found.

Exploiting parallel and distributed hardware for SAT solving has long been an active area of research, aiming to push the frontier of problems that are feasible to solve (Böhm and Speckenmeyer 1996; Balyo and Sinz 2018). Distributed SAT solvers based on careful *clause sharing* have recently reduced running times of diverse difficult instances from hours to seconds using thousands of cores at once, also being able to conquer instances that were previously infeasible (Schreiber and Sanders 2024). This scalable SAT solving is coupled with flexible *task parallelism*: Solving many SAT instances at once boosts resource efficiency and reduces mean response times (Sanders and Schreiber 2022). By contrast, there has been significantly less work on effective parallelization of MaxSAT (Lynce, Manquinho, and Martins 2018). Specifically, we are unaware of any published work describing parallel or distributed versions of the algorithms featured in the annual MaxSAT Evaluations of the last eight years, any solvers aimed at more than 32 threads, or any distributed solvers that would support *weighted* problems, i.e., the minimization of objectives with varying coefficients. Solvers in the current MaxSAT Evaluations only parallelize heuristics of the main algorithm or run an IP solver on a separate thread as a solver portfolio (Berg et al. 2024).

Regarding SIS in particular, prior approaches that test a fixed number of bounds in parallel (Terra-Neves, Lynce, and Manquinho 2016) achieve little speedup because they fail to accelerate individual bound tests. Conversely, testing a sequence of bounds with a *parallel* SAT solver was so far hindered by the parallel solvers' high upstart overhead per call, which proves wasteful for most calls. These two methods were never combined, and achieving good resource utilization has been an unsolved challenge so far. Our central finding is the observation that *flexible* scheduling of *incremental* SAT solving tasks can address all of these issues at once.

In this work, we propose the first architecture for general (i.e., partial weighted) MaxSAT solving that is suitable for massively parallel and distributed scales. Exploiting the distributed SAT solving and job scheduling framework MALLOB (Sanders and Schreiber 2022), our SIS approach makes several incremental SAT calls in parallel, and the available parallel resources are dynamically assigned to the currently active SAT calls. We generalize a prior parallel bound testing scheme for MaxSAT and enhance it based on empirical observations. We also present a two-level hierar-

chical clause-sharing approach within *and across* different SAT calls. In our experimental evaluation, our prototypical solver clearly outperforms the sequential state-of-the-art SIS MaxSAT solver PACOSE (Paxian, Reimer, and Becker 2018) and is able to scale up to 768 cores, marking a significant advancement for parallel and distributed MaxSAT solving.

The paper at hand is structured as follows. First, we provide crucial background knowledge for our work. We then present our distributed architecture and relate it to prior work. Lastly, we report on an experimental evaluation of our framework and present concluding remarks.

Background

In the following, we provide some important preliminaries.

Maximum Satisfiability

A literal ℓ is a $\{0,1\}$-valued variable x or its negation $\neg x$. A clause $C = \ell_1 \vee \ldots \vee \ell_k$ is a disjunction of literals. A formula in conjunctive normal form (CNF) $F = C_1 \wedge \ldots \wedge C_m$ is a conjunction of clauses. We can think of clauses (formulas) as sets of literals (clauses) to disregard ordering and repetitions.

A (truth) assignment α maps variables to 0 or 1. We can write truth assignments as the set of literals they assign to 1. The semantics of truth assignments are extended to negations of variables $\neg x$, clauses C, and formulas F in the standard way: $\alpha(\neg x) = 1 - \alpha(x)$, $\alpha(C) = \max\{\alpha(\ell) \mid \ell \in C\}$, and $\alpha(F) = \min\{\alpha(C) \mid C \in F\}$. A formula F is satisfied by an assignment α if $\alpha(F) = 1$ and is satisfiable if some assignment satisfies it. The *Boolean satisfiability (SAT)* problem asks to decide if a given formula is satisfiable.

We consider the general maximum satisfiability problem, i.e., *Weighted Partial MaxSAT*. The following *objective-based* definition is equivalent to the classical definition with hard and soft clauses via a straightforward conversion (Bacchus, Järvisalo, and Martins 2021).

An objective $O \equiv \sum_i w_i \ell_i$ is a *pseudo-Boolean* expression where each ℓ_i is a literal with a positive integer weight w_i. The value $\alpha(O)$ of O under assigment α is $\sum_i w_i \alpha(\ell_i)$. $|O|$ refers to the number of terms in O.

A MaxSAT instance (F, O) features a CNF formula F and an objective O. An assignment α that satisfies F is a solution of the instance and has cost $\alpha(O)$. α is optimal if it minimizes cost over all solutions. The goal is to compute an optimal solution, i.e., to minimize O subject to F. An instance's *optimal cost* is the cost of its optimal solution(s).

Example 1. *The MaxSAT instance (F, O) with $F \equiv \{(x_1 \vee x_2), (x_2 \vee \neg x_3)\}$ and $O \equiv 1x_1 + 4x_2 + 2\neg x_3$ has optimal solution $\{x_1, \neg x_2, \neg x_3\}$ with cost 3.*

Reified Encodings of Objectives. Given an objective O and a set of (positive) cost values B, $\text{CNF}(O, B)$ denotes an abstract pseudo-Boolean (PB) encoding (Warners 1998; Bailleux and Boufkhad 2003; Eén and Sörensson 2006; Joshi, Martins, and Manquinho 2015; Paxian, Reimer, and Becker 2018) of O for the values in B, i.e., a CNF formula that defines *output variables* o_k for each $k \in B$. The o_k are indicators for the value of O under satisfying assignments of $\text{CNF}(O, B)$; any assignment α that satisfies $\text{CNF}(O, B)$ sets $\alpha(o_k) = 0$ iff $\alpha(O) \leq k$. *Incremental PB-encodings* allow to add cost values to B and extend $\text{CNF}(O, B)$ lazily in each iteration just as needed (Martins et al. 2014; Paxian, Reimer, and Becker 2018).

Incremental SAT Solving. Incremental SAT solving under assumptions (Eén and Sörensson 2003) allows to efficiently solve a sequence of related SAT instances. Clauses can be supplied to the SAT solver in between solving calls and are permanent once added. Each solving call can be provided with a set of *assumption literals* $\mathcal{A}$, which are enforced for this call only. This interface allows solvers to retain valuable learned information across solving calls.

We write $\texttt{SolveInc}(F, \mathcal{A})$ for an (incremental) SAT solving call, which returns a tuple $(\textit{sat?}, \alpha)$. If the Boolean *sat?* is TRUE, α is a solution to F that assigns $\alpha(\ell) = 1$ for all $\ell \in \mathcal{A}$. If *sat?* is FALSE, no such solution exists.

Solution Improving MaxSAT

Algorithm 1 details a slight generalization of solution-improving search (SIS), also called SAT/UNSAT search, for MaxSAT. Invoked on an instance (F, O), the algorithm first calls a SAT solver on F without assumptions ($l.1$). If *sat?* is FALSE, no solutions to the instance exist and the algorithm stops ($l.2$). Otherwise, the call returns a solution $\alpha_{best} := \alpha$, which provides an upper bound $ub := \alpha(O)$ on the optimal cost. Line 3 also initializes the lower bound on the optimal cost to 0 and an initially empty set B of encoded cost values.

The main loop ($l.4$-9) iterates until an optimal solution is found. First, a query `NextBound` yields the next bound b to test ($l.5$). Then, a working formula F_W is constructed or updated to feature the clauses of the instance and a PB-encoding for testing bound b ($l.6$). The SAT solver is invoked on F_W while assuming the negation of the output variable o_b corresponding to b ($l.7$). If the SAT solver reports *sat?* = FALSE, no solutions of cost $\leq b$ exist, so the lower bound lb is increased to $b + 1$ ($l.8$). Otherwise (if the solver returns a solution), the best-known solution α_{best} and the upper bound ub are updated ($l.9$). Once $lb = ub$, we know that α_{best} is optimal and the algorithm terminates ($l.10$).

In sequential SIS, `NextBound` returns $ub - 1$, i.e., the highest yet untested value for the optimal cost. This results in a sequence of satisfiable queries, yielding new solutions; only the final query is unsatisfiable. The resulting number

Algorithm 1: MaxSAT Solution-improving search (SIS)

Input: MaxSAT instance (F, O)
Output: Optimal solution α_{best}

1: $(\alpha, \textit{sat?}) \leftarrow \texttt{SolveInc}(F, \emptyset)$
2: **if** *sat?* = FALSE **then return** "no solutions" **end if**
3: $lb \leftarrow 0$; $ub \leftarrow \alpha(O)$; $\alpha_{best} \leftarrow \alpha$; $B \leftarrow \emptyset$
4: **while** $ub \neq lb$ **do**
5: $b \leftarrow \texttt{NextBound}(ub, lb)$
6: $B \leftarrow B \cup \{b\}$; $F_W \leftarrow F \cup \text{CNF}(O, B)$
7: $(\alpha, \textit{sat?}) \leftarrow \texttt{SolveInc}(F_W, \{\neg o_b\})$
8: **if** *sat?* = FALSE **then** $lb \leftarrow b + 1$
9: **else** $ub \leftarrow \alpha(O)$; $\alpha_{best} \leftarrow \alpha$ **end if**
10: **return** α_{best}

of iterations can be linear in the sum of weights in O. In practice, many MaxSAT instances take significantly fewer iterations since the solution found by a solver is often lower than the enforced bound b, which shortcuts the search.

Example 2. *On the instance from Example 1, sequential SIS might find the initial solution $\{x_1, x_2, x_3\}$ with cost 5. Next, it will perform the SAT call* `SolveInc`$(F \cup \text{CNF}(O, \{4\}), \{\neg o_4\})$. *Assume that the optimal solution with cost 3 is found. Then the SAT call* `SolveInc`$(F \cup \text{CNF}(O, \{4, 2\}), \{\neg o_2\})$ *will terminate the algorithm by reporting unsatisfiability.*

Distributed SAT Solving

In distributed computing, several physical machines are used for a single computation. Since these machines usually have no shared main memory, their coordination commonly takes place via *message passing* over a network interface.

The earliest approaches to parallel SAT solving have been for distributed setups, achieving parallelism by splitting the input into disjoint sub-formulas and solving them in parallel (Böhm and Speckenmeyer 1996). While this paradigm is situationally powerful (Heisinger, Fleury, and Biere 2020), many real-world instances are difficult to split evenly, resulting in poor load balancing and in redundant work performed (Schulz and Blochinger 2010). Another approach, which gained traction in the early 2000s mostly for shared-memory parallelism, is *portfolio solving* (Hamadi, Jabbour, and Sais 2010): several solvers search the original formula with differing approaches, heuristics, or random seeding.

Many portfolio-style parallelizations exploit a crucial feature of their sequential SAT solvers, which mostly employ the so-called *Conflict-Driven Clause Learning* (CDCL) algorithm (Marques-Silva, Lynce, and Malik 2021). During their search, CDCL solvers learn redundant clauses from the conflicts they encounter. Intuitively, these *conflict clauses* represent a pruned sub-space of the search space, and their careful bookkeeping is crucial for performance. Parallel solver threads can share conflict clauses among one another, which results in effective cooperation (Balyo and Sinz 2018). This *clause-sharing portfolio* approach was adapted to massively parallel systems with HordeSat (Balyo, Sanders, and Sinz 2015)—running thousands of solvers and periodically exchanging selected conflict clauses in an *all-to-all* fashion. HordeSat's successor MallobSat (Schreiber and Sanders 2024) showed that careful clause sharing can be a main driver of scalability for distributed SAT even if all solver threads are initally identical, challenging the prevalent "*portfolio*" notion of such systems.

The Mallob System. Mallob is a distributed platform for scheduling and processing SAT instances (Schreiber and Sanders 2024) that also supports large-scale incremental SAT solving (Schreiber 2025). According to the International SAT Competition 2020–2024, Mallob's integrated SAT solving engine MallobSat represents the state of the art in distributed SAT solving. Mallob's defining feature in terms of job scheduling is *malleability*, i.e., the addition or removal of computational resources from a job *during its execution* (Sanders and Schreiber 2022).

Mallob runs m processes, each mapped to a fixed number c of *cores*. As such, $p = m \cdot c$ cores are available. Processes communicate via the standard *Message Passing Interface* (MPI). Mallob takes *jobs* from interfaces established at a subset of processes. An arriving job j first receives one initial process, following a decentralized protocol. This process, denoted the *root* of j, represents j in terms of external communication throughout the life time of j. Subsequent processes for j are allocated as follows: Whenever the system state changes, a *fair number of processes* $v_j \in \mathbb{N}^+$ is calculated for each active job j. The v_j are based on user-defined *job priorities* as well as each job's *maximum demand of resources*, which it can adjust dynamically. The assignment of jobs to processes is then updated to enforce that each job j has exactly v_j processes.[1] Each job j is organized as a *complete binary tree T_j of processes*, rooted at the root of j. At a re-scheduling, T_j expands or shrinks to feature exactly v_j nodes. A distributed protocol assigns each vacant position in a tree to an idle process. Mallob's SAT engine MallobSat uses T_j as a communication structure to periodically aggregate, filter, and broadcast the most valuable distinct produced clauses (Schreiber and Sanders 2024).

If a Mallob job is *incremental*, its associated resources (input data and solver objects) are preserved after a SAT solving call for subsequent calls (Schreiber 2025). Likewise, MallobSat's solver threads then use incremental SAT solving. This incrementality is crucial for performance when performing a sequence of mostly lightweight "queries" on a formula that may itself be large; distributing the full formula and initializing new solvers for each query can be prohibitively expensive (Schreiber 2025). Once an increment is solved, the job reduces its resource demand to a minimum and, consequently, its job tree shrinks to a single process. Once the next increment arrives, the job increases its demand again, causing its job tree to re-grow and, where possible, to re-adopt its former processes.

Distributed Solution Improving Search

We now describe our distributed MaxSAT solving approach.

Architecture

Fig. 1 illustrates the architecture of our approach. On an abstract level, we run a fixed number $x \geq 1$ of *searchers* in an interleaved manner. Each searcher performs SIS as described in Alg. 1 in some subset B_i of possible costs of solutions; `NextBound` returns different bounds for each searcher, as we will detail later. The best known upper and lower bound, ub and lb, as well as the best known solution α_{best} are shared across all searchers. Our procedure terminates once $lb = ub$.

On a technical level, each searcher submits and interacts with one *incremental SAT job* within the Mallob platform. This job is represented by a fixed process (the job's *root process*) and additionally features a fluctuating set of distributed processes. Mallob's scheduling ensures that at any point

[1] A process runs at most one job at a time. When a process leaves a job due to a re-scheduling, it may still keep the job's associated resources in order to possibly reuse them after a later re-scheduling.

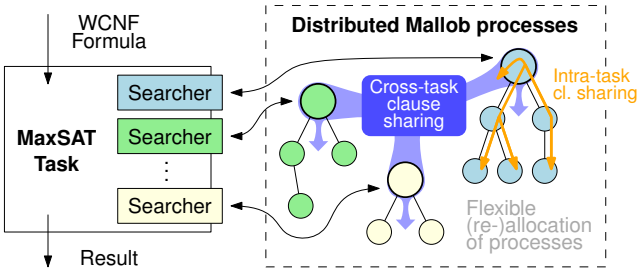

Figure 1: Overview on our approach's architecture. A MaxSAT task orchestrates a number of *searchers*, each of which runs an incremental SAT solving job in MALLOB.

in time, all available computational resources (except for the root processes of currently inactive jobs) are fairly distributed across all currently active SAT solving jobs.

Clause Sharing

Clause learning is one of the central techniques in modern CDCL SAT solvers. When invoked on a CNF formula F, a SAT solver will, during its search, learn so-called *conflict clauses* that are implied by F, i.e., satisfied by all solutions of F. Such clauses are added to the solver's clause database to prevent it from later reexploring the same parts of the search space. In distributed SAT solving, conflict clauses are also shared between threads to prevent different threads from exploring the same areas of the search space. Typically, the correctness of sharing a clause C between solver threads follows from all threads operating on the same input formula and only learning clauses that are entailed by—or, more precisely, that have *Reverse Unit Propagation* (Goldberg and Novikov 2003) w.r.t.—the clauses in their database.

In our distributed MaxSAT setup, establishing the correctness of clause sharing requires more care. Searchers S_1 and S_2 solving an instance (F, O) are operating on different formulas $F^1_W = F \cup \text{CNF}(O, B_1)$ and $F^2_W = F \cup \text{CNF}(O, B_2)$, where B_1 and B_2 are the respective sets of cost values searched over. As such, S_1 can learn a clause C that may not be entailed by F^2_W as it may contain variables not present in S_2. Our finding is that C can still be shared with S_2 as long as all searchers operate on some subset of a *common base formula* $F \cup \text{CNF}(O, B)$, where B contains any bounds that we might consider. In our setting, we obtain a global upper bound ub, set $B = \{0, \ldots, ub\}$, and pre-allocate all variables in the formula $\text{CNF}(O, B)$ in every searcher before encoding sub-formulas $\text{CNF}(O, B_1) \subseteq \text{CNF}(O, B)$ and $\text{CNF}(O, B_2) \subseteq \text{CNF}(O, B)$. Any clause C entailed by F^1_W will now be *satisfiability-* and *cost-preserving* for F^2_W; for any solution α of F^2_W that does *not* satisfy C, there is a solution α' of $F^2_W \cup \{C\}$ of equal cost. We provide a proof in our Appendix.[2] Intuitively, C is entailed by $F \cup \text{CNF}(O, B)$ and any solution of $F \cup \text{CNF}(O, B)$ restricted onto the variables of $F^2_W \cup \{C\}$ is also a solution of $F^2_W \cup \{C\}$ of equal cost. This argument also shows that C is *cost-propagation redundant* for F^2_W (Ihalainen, Berg, and Järvisalo 2022).

[2]Available online: https://doi.org/10.5281/zenodo.15463748

Clause sharing in practice. Our SAT solving tasks within MALLOB perform distributed clause sharing on two distinct levels. The first level is MALLOBSAT's existing clause sharing across the processes of a single job (Schreiber and Sanders 2024), which we denote as *intra-task clause sharing* (ITCS). The second level is *cross-task clause sharing* (XTCS), which we introduce as a new feature to MALLOB. We achieve XTCS by (i) defining *groups* of jobs among which XTCS should be performed, (ii) periodically collecting the addresses of all active root processes that belong to a common group, and (iii) using MALLOBSAT's clause sharing protocol on these addresses. More precisely, once a root process obtains clauses from ITCS, it contributes these clauses to the next XTCS operation. The clauses resulting from an XTCS operation, in turn, are broadcast to all (current) processes of all jobs involved in the XTCS.

MALLOBSAT's secondary clause selection metric after clause length, *Literal Block Distance* (Audemard and Simon 2009), has in fact no discernible merit for parallel clause sharing (Schreiber and Sanders 2024; Iida, Sonobe, and Inaba 2024). Moreover, the LBD of a clause found in a searcher S_1 is unlikely to be meaningful to a different searcher S_2. To still carry some meaning with each shared clause's LBD, we overwrite it with a custom heuristic at the XTCS level: We first assume the lowest possible (i.e., best) LBD value and then increment it for each *auxiliary* variable (i.e., variable not in F) in the clause. This prioritizes clauses that concern the problem's intrinsic logic rather than PB constraints, which may be irrelevant to other searchers.

Interval Search

In the following, we propose a strategy for assigning bounds to searchers during the MaxSAT solving procedure (`NextBound` in Alg. 1). This strategy generalizes an earlier *search space splitting* approach (Terra-Neves, Lynce, and Manquinho 2016), as we discuss in more detail later.

During our search, we maintain a sequence of intervals $\mathcal{I} = \langle [lb, ub_1], [ub_1 + 1, ub_2], \ldots, [ub_{k-1} + 1, ub_k] \rangle$. The i-th interval $I_i = [ub_{i-1} + 1, ub_i]$ represents an active SAT call of a searcher with bound ub_i, and $\mathcal{I}$ as a whole represents the range of admissible bounds which may still need to be tested before an optimal solution is identified. Initially, $\mathcal{I} = \emptyset$. The first searcher to test a bound is assigned the initial range $[0, ub - 1]$, which is inserted to $\mathcal{I}$ as a first interval. Subsequent queries for a bound to test are handled as follows: The rightmost interval $I_i = [l, r] \in \mathcal{I}$ that maximizes $r - l$ is identified and split into two equally sized halves. The left interval's upper bound, $\lfloor \frac{l+r}{2} \rfloor$, is then used as the next bound to test. Once a SAT call with bound ub_i returns, we distinguish three cases (Fig. 2):

1. If the SAT call reports satisfiability with found cost $ub'_i \leq ub_i$, all intervals I_j with $ub_j \geq ub'_i$ are removed and their corresponding SAT calls are interrupted. If the leftmost removed interval $[l, r]$ satisfies $l < ub'_i$, a truncated interval $I' = [l, ub'_i - 1]$ is appended to $\mathcal{I}$. Since I' does not (yet) represent any active SAT call, a special case applies: The next searcher to query a bound is immediately assigned I' without first splitting it.

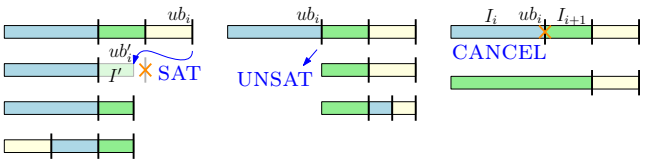

Figure 2: Illustration of progress in interval search, via an improved solution (left), unsatisfiability (center), and a searcher being cancelled (right). Black bars denote active SAT calls; colors / shades delineate the different intervals.

2. If the SAT call reports unsatisfiability, all intervals I_j with $ub_j \leq ub_i$ are removed and their corresponding SAT calls are interrupted.
3. If the SAT call stopped because its searcher was removed (see "Focusing the search" later), the corresponding interval I_i is merged with interval I_{i+1} to the right. (The "rightmost" searcher is never removed.)

As a slight refinement, note that a SAT call may find a solution of cost $ub' < ub$, thus improving upon the best known upper bound, even if the bound enforced was *larger than* ub. Therefore, we introduce a *leniency parameter* $\varepsilon = 0.01$. Upon an upper bound update ub', we cancel a removed interval's SAT call only if its bound exceeds $(1+\varepsilon) \cdot ub'$.

Our interval search has the following properties: If only a single searcher is used, its search behavior is equivalent to standard SIS (Alg. 1). With two searchers, the resulting search behavior corresponds to SIS interleaved with bisection over the admissible range of bounds. With 2^n searchers ($n \in \mathbb{N}$), the range of bounds to test is first split uniformly and then subdivided dynamically based on the further progression. A searcher *withdraws* from the procedure once no bound is left for it to test, which applies in particular if the overall range of $\mathcal{I}$ is smaller than the number of remaining searchers and, therefore, no interval can be split any further.

Example 3. *Let (F, O) be a MaxSAT instance with known upper bound $ub = 32$. A first searcher S_1 is assigned range $[0, 31]$ and thus searches for $\alpha(O) \leq 31$. After assigning bounds to two further searchers S_2 and S_3, we have $\mathcal{I} = \langle [0, 15]_{(S_2)}, [16, 23]_{(S_3)}, [24, 31]_{(S_1)} \rangle$. Searcher S_3 finds a solution with cost 20. This interrupts S_1, removes the two rightmost intervals and leaves a truncated interval $[16, 19]$. After assigning new bounds to S_3 and S_1, we have $\mathcal{I} = \langle [0, 7]_{(S_1)}, [8, 15]_{(S_2)}, [16, 19]_{(S_3)} \rangle$. Searcher S_2 now reports unsatisfiability; S_1 is interrupted, the two leftmost intervals are removed, and assigning new bounds to S_2 and S_1 yields $\mathcal{I} = \langle [16, 17]_{(S_2)}, [18, 18]_{(S_1)}, [19, 19]_{(S_3)} \rangle$.*

Skewed splitting. In SIS-based MaxSAT (Bacchus, Järvisalo, and Martins 2021), SAT calls resulting in unsatisfiability are often more expensive than SAT calls resulting in satisfiability. As such, we want to focus mainly on improving the best-known solution rather than incrementally updating the lower bound.

Our basic idea is to split intervals in $\mathcal{I}$ not in two equally sized halves but in a skewed manner, e.g., 75% vs. 25%. Integrating this change naïvely into interval search would result in the leftmost interval being the largest interval, and thus being the subject of splitting, for many iterations. Instead, we also need to refine the choice of which interval to split. To this end, we relate each interval I_i to a *mass* m_i, which is some arbitrary positive constant for the initial interval. If interval I_i is split, its arising sub-intervals both receive a mass of $m_i/2$, regardless of the skew applied during splitting. To identify an interval for splitting, we now select the rightmost interval of *maximum mass* rather than maximum size. This results in the same splitting decisions as with our original strategy while applying some skew $0 < \sigma < 1$ to every split. A remaining issue is which mass to attribute to a truncated interval I', which may cover any left-side subrange of its original parent interval. For any skew σ, note that the leftmost interval resulting from n splits receives 0.5^n of the original mass and σ^n of the original size. Generalizing this notion, if an interval I' covers a share α of its parent interval's range, we attribute a share $\sigma^{\log_{0.5}(\alpha)}$ of its parent's mass. $\sigma = 0.5$ results in the original interval search.

Example 4. *If we use $\sigma = 0.75$ in Example 3, starting all searchers yields $\mathcal{I} = \langle [0, 23]_{(S_2)}, [24, 29]_{(S_3)}, [30, 31]_{(S_1)} \rangle$ with mass ratios 2:1:1. A 4^{th} searcher would receive $[0, 17]$.*

Focusing the search. Early experiments indicated that tests at relatively low bounds can get stuck in their (UN)SAT call for an extended period while the upper bound converges to its optimum relatively quickly. Eventually, all active SAT calls test unsatisfiable bounds, and only one of them is relevant for progress. The resources spent by the remaining calls could instead be used to accelerate the final crucial call.

The simple measure we propose in this work is to reduce the number of active searchers as the procedure begins to stagnate. Whenever a certain amount of time t has passed *without any searcher making progress* (i.e., receiving a result from a SAT call), the searcher at the lowest bound is removed from the procedure and its SAT task is interrupted and cleaned up. At least one searcher is always kept alive.

Example 5. *Following Example 3, time t passes without any progress. S_2 is removed and its interval $[16, 17]$ is merged, leading to $\mathcal{I} = \langle [16, 18]_{(S_1)}, [19, 19]_{(S_3)} \rangle$. The freed resources are redistributed to the remaining two searchers.*

Pseudo Boolean Encodings

The specific PB-encoding used for realizing $\text{CNF}(O, B_i)$ is known to significantly affect the performance of SIS (Eén and Sörensson 2006; Paxian, Reimer, and Becker 2018). We pragmatically focus on three different PB encodings that are also used by PACOSE, the current state-of-the-art sequential SIS-based solver (Paxian, Reimer, and Becker 2018), and aim to balance between the number of clauses in $\text{CNF}(O, B)$ and the SAT solving performance in terms of *propagation*, i.e., how efficiently a solver can detect that the cost of a partial assignment exceeds the enforced bound. We consider the following three encodings:

(i) The (Warners) **Adder** (Warners 1998; Eén and Sörensson 2006) achieves a compact encoding (linear in the number of terms) by summing up the objective coefficients as binary numbers and restricting the individual bits of the sum. It results in the least amount of unit propagation.

(ii) The **Generalized Totalizer** (GTE) (Bailleux and Boufkhad 2003; Joshi, Martins, and Manquinho 2015) builds a binary tree in which the literal-coefficient pairs of the objective to be encoded form the leaves. Informally speaking, the internal nodes correspond to variables that count the sum of coefficients of the literals in different subsets of leaves. The outputs of the GTE are the variables associated with the root. The GTE propagates the best but requires a worst-case exponential number of clauses.

(iii) The **Dynamic Polynomial Watchdog** (DPW) (Paxian, Reimer, and Becker 2018) can be considered to strike a middle ground in terms of size and performance. It achieves a compact but well propagating encoding by using *totalizers* to sum up the individual bits of the different coefficients of the objective as a unary number, and then merging them.

Inspired by PACOSE (Paxian, Reimer, and Becker 2018), we heuristically pick an encoding based on O. We use a Warners adder if O is very large ($|O| > 10\,000$ or $\sum_i w_i > 10^{12}$), GTE if O is small ($|O| \leq 5$ or ($\sum_i w_i \leq 100$ with at most 20 unique weights)), and DPW otherwise. In addition, we added some memory-saving measures to prefer DPW over GTE (Adder over DPW) in case of large F ($> 10^7$ literals) whenever the more involved encoding is expected to be large ($|O| > 25$ for GTE, $|O| > 5000$ for DPW). The values were chosen based on PACOSE's decision heuristic, informed guesses, and preliminary tests to gauge memory requirements; a more thorough exploration of the parameter space may yield further improvements in future works.

Preprocessing

MaxSAT solvers commonly *preprocess* the input to reduce its size and/or complexity. The premier MaxSAT preprocessing solution is the standalone tool MAXPRE (Korhonen et al. 2017; Ihalainen, Berg, and Järvisalo 2022), which implements SAT-based simplifications (e.g., Bounded Variable Elimination) as well as MaxSAT-specific techniques such as subsumed label elimination and clause hardening.

In the spirit of devising a prototypical and yet powerful distributed MaxSAT solver, we integrated MaxPRE as follows: We first run MaxPRE with the minimum features needed to transform the input file into the objective-based form expected by our approach. Solving then commences on the resulting instance. Concurrently, we run MaxPRE's full-featured preprocessing, interrupting it every few seconds to assess the improvements made thus far. If the instance or objective function was reduced significantly ($\leq 90\%$ of the original number of total literals, variables, or objective terms), we restart the entire solving procedure while retaining the best bounds and best solution found so far. Improved bounds are always applied directly. Our extension of MaxPRE memorizes the performed transformations for each preprocessing iteration separately and can thus transform a solution to the problem at any preprocessing level into a solution to the original input problem. In general, the cost of a *non-optimal* solution to a preprocessed problem can deviate from the true cost of the according reconstructed solution to the original problem (Leivo, Berg, and Järvisalo 2020); as such, whenever our system is stopped before finding the optimal solution, it recomputes the cost of the best found reconstructed solution based on the original problem input.

Relation to Previous Work

We briefly set our approach in relation with prior work in the field of parallel MaxSAT. We refer to Lynce, Manquinho, and Martins (2018) for a more thorough literature review.

A two-thread MaxSAT algorithm was presented by Martins, Manquinho, and Lynce (2011b) and extended to support eight threads by Martins, Manquinho, and Lynce (2011a, 2012). The shared-memory approach runs solution-improving and core-guided algorithms on different threads. The papers also provide an alternative condition for safe clause sharing that requires analyzing the clause learning procedure to determine what types of clauses are used in the derivation of the learned clauses. Loosely speaking, only clauses that are derived from hard clauses can be shared. By contrast, our condition for sharing clauses can be checked without alterations to the clause learning procedure. Moreover, our approach is the first with hierarchical parallelism (running SAT calls in parallel while also parallelizing each SAT call) and featuring clause sharing at all levels.

The only other distributed MaxSAT solver we know is presented by Terra-Neves, Lynce, and Manquinho (2016). Their *search space splitting* approach at k searchers subdivides the range of admissible bounds *a priori* into k uniform intervals. Our approach can be considered a generalized and modernized version of this approach, introducing skewed and focused search, parallel solving within each SAT call, and information exchange across searchers. Terra-Neves et al.'s approaches are not massively parallel (having been run on ≤ 32 threads) and do not support weighted objectives.

Parallelizing core-guided approaches is a challenge outside the scope of this paper. Modern core-guided solvers apply different reformulations that change the objective, which complicates meaningful information exchange across core-guided threads (Morgado, Dodaro, and Marques-Silva 2014). Earlier papers parallelize core-guided search by forcing every core-guided thread to perform the exact same reformulation steps (Martins, Manquinho, and Lynce 2011a) or parallelize the minimization of the found cores (Berg et al. 2024). Moreover, lower bounds found by SIS threads cannot be used directly by core-guided threads because the SIS thread cannot provide a "witness" core for the lower bound.

Experimental Evaluation

We now turn to the experimental evaluation of our approach. Our software and data are available online (see footnote 2).

Implementation

We implemented our approach, which we refer to as MALLOBMAX, in C++ within the MALLOB framework. An entering MaxSAT task is launched as a local sub-program directly at the process where the task was introduced. As such, the MaxSAT task can use the job submission interface it arrived over to submit sub-jobs itself (SAT jobs in our case). Compared to MALLOB's prior, prototypical setup for incremental solving (Schreiber 2025), we have invested significant engineering effort to increase the practical performance

of incremental SAT tasks in MALLOB, including reworked inter-process communication, caching of formula data, and reducing and compressing the data to be communicated.

We implemented XTCS by reusing MALLOBSAT's clause sharing code, which lets us profit from non-trivial clause sharing features such as buffering, merging, and filtering of clauses (Schreiber and Sanders 2024). We limit the XTCS sharing volume to $2\times$ the ITCS sharing volume, which we expect to suffice for exchanging the most helpful clauses without overcrowding solver threads too much.

In terms of PB-encodings, we use OpenWBO's Warners Adder (Martins, Manquinho, and Lynce 2014) and RustSAT's (Jabs 2025) incremental GTE and DPW encoders. The latter two exploit the *Cone of Influence* technique (Paxian, Reimer, and Becker 2018), which allows to encode constraints just-in-time for each bound. We enhanced GTE and DPW to support pre-allocating auxiliary variables and adjusted the Adder encoder, which encodes all of $\text{CNF}(O, \{0, \ldots, ub\})$ at once, to support individual bound tests via assumptions *only*, i.e., without adding further clauses that would cause inconsistencies across the searchers' formulas (see Appendix – footnote 2).

Setup

We use the HPC cluster *SuperMUC-NG*, where each compute node features a two-socket Intel Skylake Xeon Platinum 8174 clocked at 2.7 GHz with $2 \times 24 = 48$ physical cores (96 hardware threads). Nodes have 96 GiB of RAM each and communicate via Intel OmniPath. We run MALLOBMAX on 1–16 nodes, i.e., 48–768 cores, with two processes per socket and hence 12 cores per process.

Experiments in distributed environments are resource-intensive and costly. The distributed SAT solving community has introduced measures to make responsible use of computational resources, which we follow in our experiments. In particular, we limit each parallel run to 300 s wall-clock time per instance, which is low in terms of sequential optimal MaxSAT literature but in line with the limits used in distributed SAT literature (Schreiber and Sanders 2024) and MaxSAT anytime solving (Berg et al. 2024). An underlying motivation is that the added cost of (massively) parallel runs should be traded off not only by more solved instances but also by much lower running times (Balyo, Sanders, and Sinz 2015). We also deliberately focus on a compact set of diverse and meaningful benchmarks—namely, 500 instances (226 unweighted, 274 weighted) randomly drawn from all 1902 instances of the MaxSAT Evaluation's 2023–2024 exact tracks (Berg et al. 2024). We configure MALLOBSAT to use the CADICAL SAT solver (Biere et al. 2024).

Rather than re-running small-scale parallel approaches with dated solver backends, we consider sequential solvers from MaxSAT Evaluation (MSE) 2024 (Berg et al. 2024) as more competitive baselines. We run PACOSE (Paxian, Reimer, and Becker 2018), the best available SIS MaxSAT solver; UWR-SCIP-MAXPRE (Piotrów 2020), which in 2024 performed the best on both weighted and unweighted instances at the low running times we are considering; and SPB-MaxSAT-c-Band (Zheng et al. 2024), the winning system of the MSE 2024 *anytime* tracks. To compute speedups, we run the sequential PACOSE for up to 6 h 20 min per input.

	exact		anytime	
Configuration	**#**	**PAR**	**MSE**	**TM**
m1/4 k1	290	263.5	.871	.838
m1 k1	299	252.4	.854	.832
m1 k2	301	251.2	.873	.846
m1 k4	301	253.4	.878	.850
m4 k1	315	235.1	.869	.846
m4 k2	315	235.8	.883	.859
m4 k2 no focus	311	238.9	.883	.859
m4 k2 σ0.75	316	234.5	.883	.856
m4 k4	315	237.5	.892	.867
m4 k4 no focus	313	238.5	.890	.864
m4 k4 σ0.75	314	237.3	.888	.860
m4 k4 no preprocessing	284	271.6	.879	.855
m4 k4 prior preprocessing 30 s	314	241.7	.896	.848
m4 k4 no XTCS	312	240.6	.889	.862
m16 k1	319	230.6	.878	.856
m16 k2	324	225.0	.898	.874
m16 k4	321	228.5	.908	.882
m16 k16	319	229.9	.910	.885
Pacose (MSE'24 exact)	279	284.6	.831	.800
UWr (MSE'24 exact)	328	222.9	.752	.717
SPB (MSE'24 anytime)	91	496.5	.932	.904

Table 1: Performance in terms of optimally solved instances (#), corresponding Penalized Average Runtime (each timeout counts as $2 \cdot 300$ s), MSE score (avg. ratio between optimal and best found cost at 300 s), and *Time-aware MSE* score (like MSE, but accounting for all improvements over time). m (k) denotes the number of nodes (searchers).

To assess anytime solving performance, we consider two metrics. First, the MSE metric attributes a score of $\frac{opt+1}{c+1}$ to each instance, where *opt* is the best *known* solution and c is the approach's best *found* solution after 300 s. Secondly, we generalize this metric into the *Time-aware MSE* (TM) metric to account for all x improvements an approach found over time. We apply the MSE score for each improved solution of cost c_i reported after time t_i ($1 \leq i \leq x$) and weigh it by the time span $t_{i+1} - t_i$ for which it was the best found cost (using $t_{x+1} := 300$ s for the final solution). The TM score is the sum of these scores, normalized to $[0, 1]$.

Results

We now discuss experimental results, as shown in Table 1.

Preprocessing via MaxPRE is highly beneficial to performance, with a preprocessing-free run performing the worst out of all tested MALLOBMAX configurations. Full *a-priori* preprocessing with a 30 s budget results in very high quality solutions and thus high MSE scores; however, TM scores reveal the high upfront cost to arrive at these solutions. Our incremental preprocessing allows to combine low overhead for easy instances with powerful simplifications for hard instances, thus performing the most convincingly. Employing **XTCS** at four searchers resulted in an overall 16.3% geometric mean speedup. Notably, we found that XTCS improves mean performance across all PB-encodings

(15/17/11 % speedup for Adder–/DPW–/GTE-encoded instances), which indicates that sharing clauses across threads with deviating working formulas is beneficial. The **number of searchers** employed makes a modest difference in terms of optimal solving performance. Since the costly unsatisfiability calls required for optimality profit from more resources per searcher, the single-searcher configurations of MALLOBMAX generally perform competitively. That being said, at 16 nodes, two or more searchers clearly outperform a single searcher. Running several searchers also benefits anytime performance; e.g., at 16 nodes, MALLOBMAX reaches an MSE score of 0.878 with one searcher and 0.908 with four searchers. This shows how parallel bound search can quickly yield good solutions without compromising on exact solving performance. Disabling **search focusing** ($t = 20\,s$) resulted in worse performance in terms of PAR-2 scores and solved instances, both with two and four searchers. Adding a **skew** to interval search ($\sigma = 0.75$) also resulted in better PAR-2 scores, however only by an insignificant margin.

Next, we assess the **scaling and competitiveness** of MALLOBMAX. We run two searchers with search focusing, XTCS, and no skew at 12, 48, 192, and 768 cores (= 1/4, 1, 4, 16 nodes). Fig. 3 (left) shows results in terms of optimal solving performance. MALLOBMAX drastically outperforms PACOSE already at 12 cores and continues to improve up to 16 nodes. At 16 nodes, MALLOBMAX performs similarly to UWR and even solves more instances at moderately low running times (≤ 100 s) and more unweighted instances (see Appendix – footnote 2). We deem these results very encouraging, considering that UWR is a highly tuned *core-guided* approach whereas pure SIS does not reflect the state of the art in (sequential) MaxSAT solving (Berg et al. 2024). We also show some *Virtual Best Solvers* (VBS), where the fastest approach is selected for each instance. A VBS of UWR and MALLOBMAX clearly outperforms a VBS of UWR and PACOSE, and MALLOBMAX's scaling from 1 to 16 nodes translates to a strong improvement of the respective VBS with UWR, showing that our approach provides a significant complement to UWR's performance.

Let us now consider anytime performance. SPB performs Stochastic Local Search (SLS) with no regard for optimality and achieves a higher MSE score than MALLOBMAX. The tested PACOSE and UWR configurations are unable to reach the anytime performance of MALLOBMAX. All in all, our approach appears to strike a good balance between optimal and anytime performance. Adding SLS techniques to our approach is likely to further boost its anytime performance – for instance, a VBS of SPB and 16-node 4-searcher MALLOBMAX leads to an MSE (TM) score of 0.971 (0.948).

Given a MaxSAT instance solved by PACOSE *and* by MALLOBMAX at some scale, we can compute the *speedup* by dividing PACOSE's running time by MALLOBMAX's. Given a set of instances, a conservative means of aggregating speedups (Schreiber and Sanders 2024) is the *geometric mean*, which, with two searchers, is 1.73 at 48 cores, 2.14 at 192 cores, and 2.52 at 768 cores. This metric can be misleading since most instances are solved within few seconds, where parallelization is rarely worthwhile. We can also assess how speedups develop if we increase the difficulty of instances, as measured via PACOSE's running time. Inspired by MALLOBSAT's evaluation (Schreiber and Sanders 2024), Fig. 3 (right) shows this *weak scaling* when fixing the number of searchers to two (continuous lines) and when fixing the searchers per node to one (dotted lines). Across all configurations, speedups improve as the difficulty of instances increases, and adding searchers and/or nodes likewise improves the scaling. For example, if we consider instances where PACOSE took at least $\tau = 500$ s, MALLOBMAX with one searcher per node reaches speedups of 19.2 (28.9, 33.6) at 48 (192, 768) cores. As a rough point of reference, MALLOBSAT previously achieved speedups of around 32 at 768 cores for pure SAT solving (Schreiber and Sanders 2024).

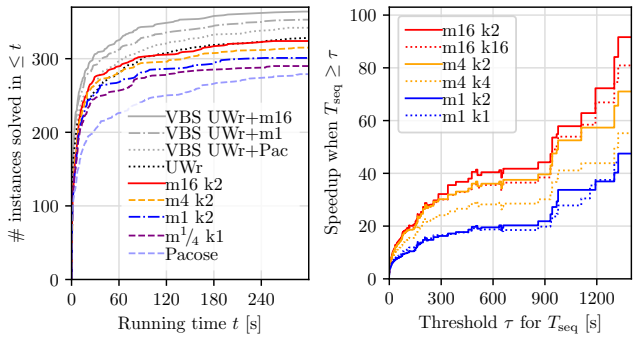

Figure 3: Left: Performance of exact MaxSAT approaches. Right: Weak scaling of MALLOBMAX: Given a lower bound τ on the running time T_{seq} of PACOSE, the y-coordinate denotes MALLOBMAX's geometric mean speedup over the instances where PACOSE took $\geq \tau$ seconds.

Conclusion

This paper introduces the first massively parallel approach to general (i.e., weighted partial) MaxSAT solving. Our architecture exploits the distributed SAT solving platform MALLOB and thus uses a flexible combination of task parallelism and parallel incremental SAT solving to search and iteratively restrict the space of admissible objective costs. Experiments on up to 768 cores confirm that our parallelization is effective and considerably outscales existing SIS solvers.

While the results obtained mark a notable step towards scalable MaxSAT solving, they also indicate clear directions going forward. In particular, we intend to combine SIS-based parallelization with SLS methods and with advanced lower bounding techniques, e.g., based on unsatisfiable cores, to further improve performance and scalability.

Acknowledgements

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). This work is also partially funded by Research Council of Finland (grants 356046 and 362987).

The authors wish to thank the authors of CaDiCaL for their kind help with debugging our cross-task clause sharing.

References

Audemard, G.; and Simon, L. 2009. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *IJCAI*, 399–404.

Bacchus, F.; Järvisalo, M.; and Martins, R. 2021. Maximum Satisfiability. In *Handbook of Satisfiability*. IOS Press.

Bailleux, O.; and Boufkhad, Y. 2003. Efficient CNF Encoding of Boolean Cardinality Constraints. In *CP*, 108–122.

Balyo, T.; Sanders, P.; and Sinz, C. 2015. Hordesat: A massively parallel portfolio SAT solver. In *Proc. SAT*, 156–172.

Balyo, T.; and Sinz, C. 2018. Parallel Satisfiability. In *Handbook of Parallel Constraint Reasoning*. Springer.

Berg, J.; Järvisalo, M.; Martins, R.; Niskanen, A.; and Paxian, T., eds. 2024. *MaxSAT Evaluation 2024: Solver and Benchmark Descriptions*. Finland: University of Helsinki.

Biere, A.; Faller, T.; Fazekas, K.; Fleury, M.; Froleyks, N.; and Pollitt, F. 2024. CaDiCaL 2.0. In *Proc. CAV*, 133–152.

Böhm, M.; and Speckenmeyer, E. 1996. A fast parallel SAT-solver – Efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17: 381–400.

Eén, N.; and Sörensson, N. 2003. Temporal induction by incremental SAT solving. In *Int. Workshop on Bounded Model Checking, BMC@CAV 2003*, 543–560.

Eén, N.; and Sörensson, N. 2006. Translating Pseudo-Boolean Constraints into SAT. *JSAT*, 2(1-4): 1–26.

Fichte, J. K.; Le Berre, D.; Hecher, M.; and Szeider, S. 2023. The Silent (R)evolution of SAT. *Comm. ACM*, 66(6): 64–72.

Goldberg, E. I.; and Novikov, Y. 2003. Verification of Proofs of Unsatisfiability for CNF Formulas. In *Proc. DATE*, 10886–10891. IEEE Computer Society.

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

Heisinger, M.; Fleury, M.; and Biere, A. 2020. Distributed Cube and Conquer with Paracooba. In *Proc. SAT*, 114–122.

Heizmann, M.; Schilling, C.; and Tischner, D. 2017. Minimization of visibly pushdown automata using partial MaxSAT. In *Proc TACAS*, 461–478. Springer.

Ignatiev, A.; Izza, Y.; Stuckey, P. J.; and Marques-Silva, J. 2022. Using MaxSAT for efficient explanations of tree ensembles. In *Proc. AAAI*, volume 36, 3776–3785.

Ihalainen, H.; Berg, J.; and Järvisalo, M. 2022. Clause Redundancy and Preprocessing in Maximum Satisfiability. In *Proc IJCAR*, volume 13385 of *LNCS*, 75–94. Springer.

Iida, Y.; Sonobe, T.; and Inaba, M. 2024. Parallel Clause Sharing Strategy Based on Graph Structure of SAT Problem. In *Proc. SAT*, 17:1–17:18.

Jabs, C. 2025. RustSAT: A Library For SAT Solving in Rust. arXiv:2505.15221.

Joshi, S.; Martins, R.; and Manquinho, V. 2015. Generalized Totalizer Encoding for Pseudo-Boolean Constraints. In *Proc. CP*, volume 9255 of *LNCS*, 200–209. Springer.

Korhonen, T.; Berg, J.; Saikko, P.; and Järvisalo, M. 2017. MaxPre: An Extended MaxSAT Preprocessor. In *Proc. SAT*, volume 10491 of *LNCS*, 449–456. Springer.

Leivo, M.; Berg, J.; and Järvisalo, M. 2020. Preprocessing in Incomplete MaxSAT Solving. In *ECAI*, 347–354.

Lemos, A.; Gouveia, F.; Monteiro, P. T.; and Lynce, I. 2024. Iterative Train Scheduling under Disruption with Maximum Satisfiability. *JAIR*, 79: 1047–1090.

Li, C.; Xu, Z.; Coll, J.; Manyà, F.; Habet, D.; and He, K. 2021. Combining Clause Learning and Branch and Bound for MaxSAT. In *Proc. CP*, 38:1–38:18.

Li, C. M.; and Manyà, F. 2021. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*. IOS Press.

Lynce, I.; Manquinho, V.; and Martins, R. 2018. Parallel Maximum Satisfiability. In *Handbook of Parallel Constraint Reasoning*, 61–99. Springer.

Marques-Silva, J.; Lynce, I.; and Malik, S. 2021. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, volume 336 of *FAIA*, 133–182. IOS Press.

Martins, R.; Joshi, S.; Manquinho, V.; and Lynce, I. 2014. Incremental Cardinality Constraints for MaxSAT. In *Proc. CP*, 531–548.

Martins, R.; Manquinho, V.; and Lynce, I. 2011a. Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In *ICTAI*, 313–320.

Martins, R.; Manquinho, V.; and Lynce, I. 2011b. Parallel Search for Boolean Optimization. In *RCRA Workshop*.

Martins, R.; Manquinho, V.; and Lynce, I. 2014. Open-WBO: A Modular MaxSAT Solver. In *Proc. SAT*, 438–445.

Martins, R.; Manquinho, V. M.; and Lynce, I. 2012. Clause Sharing in Parallel MaxSAT. In *Proc. LION*, 455–460.

Morgado, A.; Dodaro, C.; and Marques-Silva, J. 2014. Core-Guided MaxSAT with Soft Cardinality Constraints. In *Proc. CP*, volume 8656 of *LNCS*, 564–573. Springer.

Paxian, T.; Reimer, S.; and Becker, B. 2018. Dynamic Polynomial Watchdog Encoding for Solving Weighted MaxSAT. In *Proc. SAT*, volume 10929 of *LNCS*, 37–53. Springer.

Piotrów, M. 2020. UWrMaxSat: Efficient Solver for MaxSAT and Pseudo-Boolean Problems. In *ICTAI*, 132–136.

Sanders, P.; and Schreiber, D. 2022. Decentralized online scheduling of malleable NP-hard jobs. In *Euro-Par*, 119–135.

Schreiber, D. 2025. Distributed Incremental SAT Solving with Mallob: Report and Case Study with Hierarchical Planning. arXiv:2505.18836.

Schreiber, D.; and Sanders, P. 2024. MallobSat: Scalable SAT Solving by Clause Sharing. *JAIR*, 80: 1437–1495.

Schulz, S.; and 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*, 314–323. IEEE.

Terra-Neves, M.; Lynce, I.; and Manquinho, V. 2016. Non-Portfolio Approaches for Distributed Maximum Satisfiability. In *Proc. ICTAI*, 436–443.

Warners, J. P. 1998. A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form. *Inf. Process. Lett.*, 68(2): 63–69.

Zheng, J.; Chen, Z.; Li, C.; and He, K. 2024. Rethinking the Soft Conflict Pseudo Boolean Constraint on MaxSAT Local Search Solvers. In *IJCAI*, 1989–1997.