

Parallel SMT Solving via Dynamic Partitioning, Core-Guided Pruning, and Backbone Detection

Ilana Shapiro¹ , Sorin Lerner^{2,1} , Nikolaj Bjørner³ 

¹UC San Diego, La Jolla, CA, USA

²Cornell University, Ithaca, NY, USA

³Microsoft Research, Redmond, WA, USA

ilshapiro@ucsd.edu, sorin.lerner@cornell.edu, nbjorner@microsoft.com



Abstract—Exploiting parallelism in modern CPU architectures remains a longstanding challenge in optimizing SMT solvers. We introduce a novel framework for parallel SMT solving that uses *feedback from active search* to steer solving. We dynamically build a *binary partition tree* of the search space by sampling from workers’ VSIDS statistics during solving. We introduce a novel search-space pruning mechanism that harnesses the full power of *core-based CDCL-style pruning* to continuously shrink the partition tree. We further optimize our architecture by incorporating *online backbone detection* into worker threads, as well as a *terminate-on-demand* mechanism to eagerly eliminate work on pruned subproblems. The resulting algorithm is highly generalizable and scales effectively with available resources. We implement our approach in the Z3 SMT solver and demonstrate that it outperforms both sequential Z3 and existing state-of-the-art parallel frameworks on challenging benchmarks from six logics in the SMT-COMP 2025 Parallel Track.

I. INTRODUCTION

Scaling distributed SMT to match rapid advances in multi-core CPUs, high-performance computing, and cloud platforms is a longstanding challenge. Ideally, solver performance should scale with available physical resources. However, most state-of-the-art SMT solvers, such as Z3 [1], CVC5 [2], OPENSMT [3], and YICES2 [4], remain largely single-threaded. Consequently, most advances in solver performance have focused on improving the techniques and heuristics of sequential SMT solvers, leaving substantial computational resources underexploited. This limitation constrains the effectiveness of SMT solvers in key domains such as verification [5], model checking [6], security [7], and program synthesis [8].

Existing parallel approaches broadly fall into two categories: portfolio and partitioning. In the portfolio method, multiple solvers are run in parallel under different configurations and/or random seeds and may exchange learned clauses during solving. The runtime of SMT solvers is highly variable and sensitive to seemingly trivial perturbations of the solver [9, 10]. Portfolio solving has led to substantial improvements over sequential approaches by exploiting this runtime instability through parallel sampling of the solver configuration space [11–13]. However, portfolio solving is hindered by limited scalability since it is bounded by the performance of the best sequential solver [9, 14]. The partitioning approach addresses scalability via a divide-and-conquer strategy: the input formula is decomposed into independent subproblems

whose disjunction is a tautology to the original formula. The goal is to shrink the search space of each subproblem such that solving them in parallel is faster than sequential. An optimal search space partitioning has the potential to vastly outperform portfolio solving. However, the partitioning strategy faces several inherent challenges: it is difficult to partition the search space into subproblems that are easier than the original, potentially wasting resources to solve infeasible sub-problems when the original formula is satisfiable.

A popular approach to partitioning is *cube-and-conquer*, in which n Boolean atoms are chosen from the input formula and assigned to both polarities. This creates 2^n independent cubes (partial assignments) for parallel solving. Selecting good splitting atoms is difficult, and badly selected atoms can produce a harder problem than the original, particularly when the partitions are static [13, 15]. Recent techniques have turned to *dynamic* partitioning [9, 14, 16], where split atoms are selected during search. A similarly challenging problem is cube scheduling across threads, as the order of subproblem exploration can substantially impact overall performance. The *partition tree* has shown success as a data structure for dynamic partitioning. Recent architectures have combined it with clause sharing and basic search-space pruning [9, 17]. However, none of these approaches fully leverage feedback from active search.

In this paper, we move toward a more robust dynamic partitioning framework by systematically *exploiting feedback from active search* to guide both partitioning and pruning decisions. We introduce a binary partition tree whose splitting atoms are selected on-the-fly from worker threads’ VSIDS statistics. By leveraging the conflict-driven reasoning of CDCL, we enable non-chronological backjumping across the partition tree, using per-thread unsatisfiable cores to actively prune the search space. Our partition tree also supports cross-thread information sharing and on-demand termination of workers on shared subproblems. We further augment our framework with core minimization and online backbone detection for additional search-space pruning. We implement our approach in Z3, and show that our approach outperforms both sequential Z3 and state-of-the-art parallel frameworks on challenging benchmarks from six logics in the SMT-COMP 2025 Parallel Track.

In summary, we make the following contributions:

- 1) We introduce a generalizable framework for parallel SMT

solving that *dynamically partitions* the input formula with Boolean atoms sampled on-the-fly from VSIDS statistics.

- 2) We introduce a *binary partition tree* for cube distribution with a novel pruning mechanism that harnesses the full power of *core-based CDCL-style pruning* to dynamically shrink the problem space.
- 3) We further optimize our architecture by incorporating *core minimization* and *online backbone detection* for more powerful search-space pruning.
- 4) We implement our tool in the SMT solver Z3 and show that our approach outperforms sequential Z3 and existing parallel frameworks on six logics from the SMT-COMP 2025 Parallel Track. Our implementation, data, and results are available at <https://zenodo.org/records/21755875>

II. PRELIMINARIES

A. Definitions and Notations

Satisfiability Modulo Theories (SMT) determine the satisfiability of a first-order logic formula with respect to a background theory. A *theory* is a pair $T = (\Sigma, I)$ where Σ is a signature and I is a class of Σ -interpretations. We assume a fixed background theory T with signature Σ that includes the Boolean sort `BOOL`, and assume that all terms are Σ -terms, entailment ($\models$) is entailment modulo T , equivalence is equivalence modulo T , and interpretations are T -interpretations. A *model* of a formula φ is a T -interpretation $\mathcal{M}$ such that $\mathcal{M} \models \varphi$. An *implicant* of φ is a conjunction of literals ν such that $\nu \models \varphi$, i.e. every model of ν is also a model of φ . An *atom* is a term of sort `BOOL` that contains no subterms of this sort. A *literal* is an atom or its negation. A *clause* is a disjunction of literals, and a *cube* is a conjunction of literals. A propositional formula in conjunctive normal form (CNF) is a conjunction of clauses. A formula φ is a term of sort `BOOL` and is satisfiable if it is satisfied by some interpretation in I , and unsatisfiable otherwise. A formula whose negation is unsatisfiable is *valid* [18].

Most SMT solvers follow the $\text{CDCL}(T)$ framework [19], which combines a CDCL SAT solver with one or more theory solvers. The input formula is first preprocessed into an equisatisfiable CNF formula ϕ preserving its T -semantics. The SAT solver then incrementally constructs an assignment α through alternating *propagation* (unit clauses force assignments to literals) and *decision* (solver chooses values for unassigned literals) phases. In the decision phase, the Variable State Independent Decaying Sum (VSIDS) heuristic typically guides branching decisions by prioritizing variables involved in recent conflicts. These assignments and their dependencies are tracked in an *implication graph* [20]. The SAT solver performs conflict analysis to learn a clause via resolution and adds it to its clause database to prune the search space. During conflict analysis, the solver may extract an *unsatisfiable core*: a subset of assigned literals inconsistent with respect to T . The process repeats until a T -consistent satisfying assignment is found (SAT) or an unrecoverable conflict is derived (UNSAT) [18].

B. Portfolio Solving

SMT solvers are extremely sensitive to small perturbations to the input formula or configuration heuristics [13]. The *portfolio* strategy leverages this intrinsic instability by running multiple solvers in parallel with slightly different configurations (e.g. parameter values, random seeds) or with permuted, but logically equivalent, inputs. Solvers exchange information learned during search. The portfolio approach has made substantial gains over sequential solving [11, 12, 21–23]. However, since the fastest solver determines the performance of the portfolio, increasing parallelism leads to diminishing returns.

C. Search-Space Partitioning

The *partitioning* approach divides φ into n independent subproblems $\varphi_1, \dots, \varphi_n$ such that $\varphi \equiv \bigvee_i \varphi_i$. Thus, if any φ_i is SAT, then φ is SAT, and if all φ_i are UNSAT, then φ is UNSAT. There are two main partitioning strategies: *cube-and-conquer* and *scattering*. In *cube-and-conquer*, a set of n splitting atoms $A = \{a_1, \dots, a_n\}$ is selected. Traditionally, heuristics such as lookahead (which ranks candidate literals by how much their tentative assignments simplify the current subproblem) and VSIDS (which prioritizes variables in recent conflict activity) are used for split atom selection [13, 24]. A *cube* is a conjunction of literals over these atoms, i.e. a formula of the form $C = \ell_1 \wedge \dots \wedge \ell_n$, where each $\ell_i \in \{a_j, \neg a_j\}$ for some $a_j \in A$. The 2^n resulting cubes define 2^n independent subproblems $\varphi \wedge C$ for parallel solving.

In contrast, *scattering* produces n partitioning formulae:

$$\begin{aligned} p_1 &= C_1, \\ p_2 &= \neg C_1 \wedge C_2, \\ p_3 &= \neg C_1 \wedge \neg C_2 \wedge C_3, \\ &\vdots \\ p_n &= \neg C_1 \wedge \dots \wedge \neg C_{n-1} \wedge C_n, \end{aligned}$$

where each C_i is a cube (not necessarily over the same set of atoms). Each partition induces a subproblem of the form $\varphi \wedge p_i$. By construction, all p_i are disjoint and their disjunction $\bigvee_{i=1}^n p_i$ covers the search space. Thus, $\varphi \equiv \bigvee_{i=1}^n (\varphi \wedge p_i)$.

A popular data structure for search-space partitioning is the *partition tree* [17], where the root node represents the input formula φ . Given a parent node associated with formula P , its i -th child represents a formula $P \wedge C_i$ produced by a *partitioning function* [25] such that for all i , the disjunction $\bigvee_{i=1}^n C_i$ holds, and for all $j \neq i$, $\neg(C_i \wedge C_j)$. Search-space partitioning can vastly outperform portfolio if good partitions are chosen. This is a challenging task, and poorly chosen partitions can result in harder problems than the original [24].

III. RELATED WORK

Portfolio SMT Solving. Portfolio SMT solving with clause sharing was first introduced in the Z3 SMT solver [12], which simply randomized 4 instances of Z3 and shared globally valid lemmas with at most 8 literals between them. Since Z3’s initial

attempt at portfolio parallelism, several portfolio architectures have been pursued [4, 26]. The most successful frameworks were developed in SMTS [17, 22, 23] and CVC5 [11]. SMTS adopts a unique client-server architecture that orchestrates an underlying solver. Globally valid lemmas are stored in an external database, scored by frequency, and randomly sampled. Lemmas with more than 3 literals are discarded [9]. CVC5 provides further optimizations such as delayed clause sharing and guided randomization [11]. Although portfolio solving has led to significant speedups (particularly in [11] and [22]), it still suffers from innate lack of scalability.

SMT Search-Space Partitioning. The state-of-the-art SMT search-space partitioning framework is SMTS. It is the first to combine partitioning with clause sharing and per-partition portfolios, and the first to revisit already attempted subproblems with a selection strategy rooted in the solver runtime random distribution [9, 17, 22, 23]. SMTS’s underlying solver is OPENSMT2, which uses the scattering strategy to construct partitions from CDCL search branch decision literals [3, 27]. Central to SMTS’s architecture is the *partitioning tree* ([17]), which is built dynamically during search [9].

In contrast to SMTS, CVC5-CLOUD does not propose a new framework; rather, it introduces techniques for combining multiple atom sources and partitioning with these atoms [14]. ARIPARTI maintains a dynamic partition tree similar to SMTS, but performs variable-level partitioning tailored to arithmetic theories, simplifying subproblems using Boolean and Interval Constraint Propagation [16]. Unlike SMTS, these works do not support clause sharing or revisit already attempted instances. Finally, other partitioning frameworks either rely on *static* partitioning trees [15, 22], which are less effective than a carefully constructed dynamic tree, or are tailored to specific logics (BITWUZLA [28] and PBOOLECTOR [29], which combine cube-and-conquer with bitblasting for QF-BV) and thus have limited generality. While SMTS and ARIPARTI support basic search-space pruning, more sophisticated strategies remain unexplored. Existing frameworks also do not fully incorporate feedback from search. Our framework addresses both of these gaps.

Parallel SAT Solving. The state-of-the-art parallel SAT framework MALLOB uses *malleable scheduling* to allocate progressively more resources to challenging tasks while preserving state across incremental SAT queries. The authors show that slight differences in learned-clause import timing cause initially identical solver threads to diverge, enabling effective search pruning without explicit portfolio diversification or search-space partitioning [30, 31]. MALLOB has recently been applied to parallel bit-vector SMT solving in BITWUZLLOB [32], which replaces BITWUZLA’s [28] sequential CDCL backend with MALLOB. This integration exploits BITWUZLA’s offline SAT architecture, where Boolean reasoning is delegated to complete incremental SAT calls. However, integrating MALLOB into a general CDCL(T)-based SMT solver is not straightforward, as CDCL(T) requires fine-grained interaction between the Boolean and theory solvers during search [32]. MALLOB’s IPASIR-based interface submits clauses and assumptions for a complete SAT solving call but provides no callbacks through

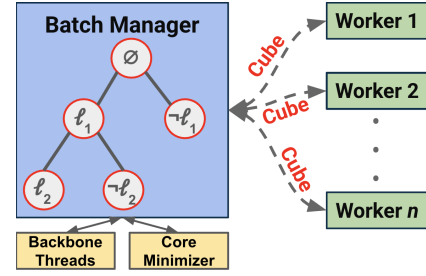


Fig. 1: Overview of Parallel Architecture

which a theory solver can inspect partial assignments or return propagations and conflicts during search [33]. IPASIR-UP [34] provides callbacks suitable for implementing this interaction but is not currently supported by MALLOB.

IV. PARALLEL ARCHITECTURE

In this section, we detail our novel parallel architecture and the strategies it employs to dynamically leverage information from search. Our motivating design principle is *leveraging feedback from active search*, with our final architecture refined through empirical evaluation (Figure 1). Our approach is based on cube-and-conquer, with cubes are created on-the-fly by sampling split atoms from solver threads’ VSIDS statistics¹. On the main thread, the *batch manager* coordinates thread synchronization. It maintains a dynamically evolving *binary partition tree* for cube storage, distributes cubes to n worker threads, and facilitates information exchange between workers. Each worker w runs a sequential Z3 instance with a progressively increasing conflict budget and repeatedly requests cubes from the batch manager. If the cube is SAT, the entire problem is SAT. If w exhausts its conflict budget without solving the cube, it returns control to the batch manager, which may perform subcubing, and selects a new cube for w (Section IV-A). If the cube is UNSAT, the batch manager prunes the partition tree using the solver’s UNSAT core (Section IV-B) and similarly assigns w a new cube. To mitigate redundant computation when multiple workers are assigned to the same cube, we employ a *terminate-on-demand* policy (Section IV-D) that aborts workers operating on now-stale cubes. Throughout execution, w accumulates learned lemmas that are reused across cubes. Finally, outside the partition tree, the batch manager coordinates two *backbone threads*, which prove backbone literals detected during search (Section IV-E) to shrink the search space of the worker threads, as well as a dedicated *core minimization thread* (Section IV-C), which asynchronously reduces workers’ UNSAT cores for more powerful pruning.

A. Binary Partition Tree

At the center of our algorithm is the *partition tree*. Unlike SMTS and ARIPARTI, which support n -ary partition trees,

¹VSIDS was selected over alternative heuristics, including lookahead, based on experimental signals

we adopt a binary tree to facilitate our core-guided pruning algorithm (Section IV-B). Each node n in our partition tree contains a *split atom* of positive or negative polarity (i.e. ℓ or $\neg\ell$), and the children of n are the polarities of the next split atom (i.e. ℓ_1 and $\neg\ell_1$) (Figure 1). Each node n encodes a cube C via the path of literals from n to root. When a worker is assigned node n , it solves the formula $\varphi \wedge C$, where φ is the input formula, under a designated conflict budget. The partition tree is initialized with the empty cube $\emptyset$ at the root (Figure 1). Thus, all workers begin with φ itself.

Nodes exist in one of three states: *open* (unsolved and has no assigned workers), *active* (has at least 1 assigned worker), or *closed* (was determined UNSAT). After solving their cube or hitting their conflict budget, workers report one of 3 statuses to the batch manager: SAT, UNSAT, and UNDEF. A SAT result terminates the search (φ is SAT). An UNSAT result marks the node as *closed*: the batch manager prunes the partition tree (Section IV-B), and shares the worker’s UNSAT core with the other workers. If the entire tree is closed, search is terminated with an UNSAT result.

When worker w reports UNDEF on node n , the batch manager decides (1) if n should be split, (governed by the *tree expansion policy*), and (2) w ’s next node assignment (given by the *node selection policy*). Globally valid unit clauses learned by w on n are shared with the other workers. We do not share larger lemmas at this time; this remains a direction for future work. w_i ’s conflict budget (initially 1000) increases dynamically by a factor of 1.5 after each UNDEF result. Finally, nodes accumulate *effort* from workers who report UNDEF. A worker w_i ’s effort is its current conflict budget; thus, effort is scaled dynamically. Multiple workers with distinct solver states (VSIDS scores, learned lemmas, etc) and random seeds may be assigned to the same (active) node n in *portfolio* if no open nodes remain. In this case, n only records the maximum effort among its workers to avoid disproportionately inflating the accumulated effort of nodes with higher parallelism.

Our tree expansion policy (Algorithm 1) draws from [9]. When w reports UNDEF for node n , the batch manager counts the active and unsolved (open or active) nodes in the tree. We expand n only if the number of unsolved nodes is less than twice the number of active nodes, all open nodes have been visited at least once, and the depth of n is equal to the depth of the shallowest timed-out leaf. Expansion is then randomly throttled (50% chance). As noted in [9], such heuristics ensure the tree remains largely balanced. If expansion proceeds, we split n on the highest scoring atom from w_i ’s VSIDS statistics. Our node selection policy (Algorithm 2) is also inspired by [9]. We first search for an open node (target status = open). If none remain, we select from active nodes (target status = active). Among nodes of the chosen status, we select the one with lowest accumulated effort, with greater depth as the tiebreaker. After this, order is random. Although inspired by [9], Algorithms 1 and 2 were empirically adapted to our framework, with some heuristics (e.g. large-tree expansion throttling) omitted due to negative experimental signals.

Algorithm 1: Tree Expansion (based on [9])

Input: Node n ; split literal ℓ ; effort $effort$

```

1  $n.update\_round\_max\_effort(effort)$ ;
2 if  $\neg n.is\_leaf()$  then
3   return;
4  $numActive \leftarrow count\_active\_nodes(root)$ ;
5  $numUnsolved \leftarrow count\_unsolved\_nodes(root)$ ;
6 if  $numUnsolved \geq numActive \cdot 2$  then
7   return;
8 if  $has\_unvisited\_open\_node(root)$  then
9   return;
10 if  $rand() \geq 0.5$  then
11   return; // 50% rejection
12  $s \leftarrow shallowest\_timedout\_leaf\_depth(root)$ ;
13 if  $depth(n) = s$  then
14    $n.split(\ell, \neg\ell)$ ;
```

Algorithm 2: Node Selection (based on [9])

Input: Root node $root$; status $target \in \{\text{open}, \text{active}\}$
Output: Best node $bestNode$

```

1  $bestNode \leftarrow \perp$ ;  $bestEffort \leftarrow \infty$ ;  $bestDepth \leftarrow -1$ ;
2  $DFS(root)$ 
3 return  $bestNode$ ;
4 Function  $DFS(cur)$ 
5   if  $cur = \perp \vee status(cur) = \text{closed}$  then
6     return;
7   if  $status(cur) = target$  then
8      $e \leftarrow effort(cur)$ ;  $d \leftarrow depth(cur)$ ;
9      $update \leftarrow false$ ;
10    if  $bestNode = \perp$  then
11       $update \leftarrow true$ ;
12    else if  $e \neq bestEffort$  then
13       $update \leftarrow (e < bestEffort)$ ;
14    else
15       $update \leftarrow (d > bestDepth)$ ;
16    if  $update$  then
17       $bestNode \leftarrow cur$ ;
18       $bestEffort \leftarrow e$ ;
19       $bestDepth \leftarrow d$ ;
20     $DFS(left(cur))$ ;
21     $DFS(right(cur))$ ;
```

B. Core-Guided Backjumping

In contrast to prior work [9, 16], when worker w reports UNSAT on node n , we do not simply close n and its subtree. Instead, we introduce a novel search-space pruning algorithm that leverages UNSAT cores to perform CDCL-style non-chronological backjumping over the partition tree (Algorithm

3). If n 's UNSAT core is empty, this proves global UNSAT (cubes are passed as assumptions in Z3, so an empty core means none were needed). Otherwise, all conflict literals lie on the path from the root to n (i.e. are a subset of w 's cube). If n 's literal is not in the core, it is irrelevant to the UNSAT result. Thus, we traverse upward from n to the nearest ancestor n_1 whose decision literal does appear in the core.² It is possible that n_1 was already closed by another thread with core C . In this case, we perform a *core strengthening* check. Let C' be the current core. If $|C'| < |C|$, we replace C with C' .

We then attempt to propagate this core upward via *sibling resolution*. Let ℓ and $\neg\ell$ be the complementary decision literals of n_1 and its sibling, respectively. If both siblings are closed with cores C_ℓ and $C_{\neg\ell}$, we compute the *sibling resolvent*:³

$$r = (C_\ell \cup C_{\neg\ell}) \setminus \{\ell, \neg\ell\}$$

If $r = \emptyset$, we conclude global UNSAT. Otherwise, we bubble up to the highest ancestor a such that all literals in r are contained in the path from a to root. We then close a and its subtree, and attach r to a and the newly closed nodes in its subtree. This process is applied recursively: if the sibling of a has already been closed by another thread, we compute a new resolvent and continue propagating upward. The result of this process is non-chronological propagation of cores across the partition tree, enabling integration of independently derived core from different workers and lifting CDCL-style clause resolution to search-space partitioning. We apply this process not only to n , but also to other nodes in the partition tree whose cube contains all literals in the UNSAT core.

C. Asynchronous Core Minimization

A core returned from the SMT engine is not necessarily minimal, and smaller cores enable more powerful pruning. To increase the power of Algorithm 3, we introduce a dedicated asynchronous *core minimization thread*. When worker w closes node n , it submits the pair (n, C) , where C is the unrefined UNSAT core, to the core minimization thread's pending queue. The minimizer attempts to refine each enqueued core C by iteratively removing literals ℓ : it flips ℓ to $\neg\ell$, and checks if this preserves UNSAT in under 5000 conflicts (Algorithm 4). If so, it discards ℓ , since UNSAT does not depend on ℓ . This is the standard deletion-based core (MUS/MUC) extraction paradigm [35], adapted to bounded SMT checks. If C is reduced, the pair (n, C) is sent to the batch manager for additional search tree pruning via core strengthening (Section IV-B).

Note that we must enqueue the pair (n, C) rather than C alone, since cores are minimized relative to the cube (node) under which they were derived. To ensure soundness, we store the associated node n with each core and use C to strengthen the core at n (Algorithm 3), which leads to more powerful pruning. Note that using a separate thread to minimize cores unblocks worker threads from exploring the search tree.

²It is certainly possible that $n_1 = n$.

³Both cores are nonempty by construction. An empty core would signal global UNSAT and thus is never attached to a node in an unsolved tree.

Algorithm 3: Core-Guided Backjumping

Input: Node n ; UNSAT core C

```

1 if  $C = \emptyset$  then
2    $\text{close}(\text{root}, C)$  ;           // global UNSAT
3   return;
4 while  $n \neq \perp$  do
5   if  $n.\text{literal} \in C$  then
6      $\text{CloseWithCore}(n, C)$ ;
7   return;
8    $n \leftarrow n.\text{parent}$ ;

9 Function  $\text{CloseWithCore}(n, C)$ 
10  if  $n.\text{closed}$  then
11    return;
12   $p \leftarrow n.\text{parent}$ ;
13   $\text{close}(n, C)$ ;
14   $l \leftarrow p.\text{left}$ ;  $r \leftarrow p.\text{right}$ ;
15  if  $l.\text{closed} \wedge r.\text{closed}$  then
16     $R \leftarrow (l.\text{core} \cup r.\text{core}) \setminus \{l.\text{lit}, r.\text{lit}\}$ ;
17    if  $R = \emptyset$  then
18       $\text{close}(\text{root}, R)$  ;       // global UNSAT
19      return;
20     $a \leftarrow \text{highest\_attach}(p, R)$ ;
21     $\text{close}(a, R)$ ;
22     $\text{PropagateUpward}(a)$ ;

23 Function  $\text{PropagateUpward}(cur)$ 
24  while  $cur.\text{parent} \neq \perp$  do
25     $p \leftarrow cur.\text{parent}$ ;  $l \leftarrow p.\text{left}$ ;  $r \leftarrow p.\text{right}$ ;
26    if  $\neg(l.\text{closed} \wedge r.\text{closed})$  then
27      return;
28     $R \leftarrow (l.\text{core} \cup r.\text{core}) \setminus \{l.\text{lit}, r.\text{lit}\}$ ;
29    if  $R = \emptyset$  then
30       $\text{close}(\text{root}, R)$  ;       // global UNSAT
31      return;
32     $\text{close}(p, R)$ ;
33     $cur \leftarrow p$ ;
```

D. Terminate on Demand

Recall that our node selection policy assigns workers to *active* nodes when no *open* nodes remain, resulting in a portfolio of m workers operating on the same node n . If a worker w_1 proves n UNSAT, the batch manager closes n and prunes the tree. Consequently, the remaining workers $w_2, \dots, w_m$, as well as any workers exploring pruned regions of the search space, may now be working on stale subproblems. To eliminate such redundant work, similar to [16], we include a *terminate-on-demand* mechanism based on *node leases*. When worker w is assigned node n , it acquires a lease that records the node's current *cancel epoch*. Each node maintains its own cancel epoch, which is incremented when the node is closed. After pruning, the batch manager terminates all workers

Algorithm 4: Partial Core Minimization

Input: UNSAT core C , SMT input formula φ
Output: Reduced core C or a satisfying model

```
1  $\Lambda \leftarrow C$ ; // literals to test
2  $\text{mus} \leftarrow \emptyset$ ; // min UNSAT core
  // Invariant:  $\varphi \wedge \text{mus} \wedge \Lambda$  is UNSAT
3 while  $\Lambda \neq \emptyset$  do
4    $\ell \leftarrow \text{last}(\Lambda)$ ;
5    $\Lambda \leftarrow \Lambda \setminus \{\ell\}$ ;
6    $\Gamma \leftarrow \text{mus} \cup \Lambda \cup \{\neg\ell\}$ ; // flip  $\ell$ 
7    $\Phi \leftarrow \varphi \cup \{\{\ell\} \mid \ell \in \Gamma\}$ ;
8    $r \leftarrow \text{check\_sat}(\Phi)$ ;
9   if  $r = \text{UNDEF}$  then
10     $\text{mus} \leftarrow \text{mus} \cup \{\ell\}$ ;
11   else if  $r = \text{SAT}$  then
12     $\text{set\_global\_sat}(\text{model}(\Phi))$ ;
13    return;
14   else if  $r = \text{UNSAT}$  then
15     $C \leftarrow \text{UnsatCore}(\Gamma)$ ;
16    if  $\neg\ell \notin C$  then
17       $\Lambda \leftarrow \emptyset$ ;  $\text{mus}' \leftarrow \emptyset$ ;
18      foreach  $c \in C$  do
19        if  $c \in \text{mus}$  then
20           $\text{mus}' \leftarrow \text{mus}' \cup \{c\}$ ;
21        else
22           $\Lambda \leftarrow \Lambda \cup \{c\}$ ;
23       $\text{mus} \leftarrow \text{mus}'$ ;
24  $C \leftarrow \text{mus} \cup \Lambda$ ;
```

with stale leases (i.e. whose recorded cancel epoch no longer matches that of its assigned node), eagerly aborting their current search and reassigning them to new nodes.

E. Online Backbone Detection

We further optimize our architecture with two dedicated *online backbone detection* threads as a lightweight search-space pruning mechanism. To our knowledge, this constitutes the first application of backbone detection to parallel SMT search. The *backbone* of a formula φ is the set of literals true in all models of φ [36]. In other words, a literal ℓ is in the backbone of φ iff $\varphi \models \ell$, i.e. iff $\varphi \wedge \neg\ell$ is UNSAT. Thus, $\neg\ell$ can prune the partition tree: if $\neg\ell$ matches node n 's literal, Algorithm 3 can close n with singleton core $\{\neg\ell\}$. Typical backbone detection algorithms assume φ is SAT and derive candidates from implicants [36]. In contrast, we extract backbone candidates dynamically based on phase age and evaluate them via two parallel threads operating in complementary modes. In *negative mode*, we negate candidates and attempt to quickly prove them as backbones by deriving a short UNSAT proof. If φ is UNSAT, every literal is vacuously in the backbone since φ has no models. Even so, proving an individual backbone literal under a small bounded conflict budget is often much easier than proving φ

UNSAT. In *positive mode*, we assume candidates directly and attempt to extend the assignment to a full model. Theoretically, the positive mode may accelerate search on SAT instances.

Throughout solving, the batch manager stores proven backbone literals and a dynamic ranking of the top 100 candidates from worker threads. Candidates are ranked based on *phase age* (number of assignments since their last phase change) and *cube hits* (number of times worker threads returned them):

$$\text{rank}(c) = \text{phase_age}(c) \cdot \log_2(2 + \text{cube_hits}(c))$$

Candidates are dispatched in batches Λ to the backbone threads, who attempt to prove them using a modified version of the chunked backbone algorithm [36] (Algorithm 5).⁴ Given Λ , the procedure repeatedly selects a chunk $\Gamma \subseteq \Lambda$ of backbone candidates of size at most K , and attempts to refute the simultaneous negation of all literals in Γ (i.e. it invokes the solver under the assumptions $\omega_N = \{\neg\ell \mid \ell \in \Gamma\}$). If the resulting formula is UNSAT, the UNSAT core C is intersected with the current assumptions ω_N . If this intersection is a singleton $\{\neg\ell\}$, then ℓ is a backbone literal. The batch manager collects ℓ for search-space pruning and shares it as a unit clause. Otherwise, all literals in C are removed from ω_N , and the refinement loop continues. In our setting, a satisfying assignment under ω_N signals immediate termination and global SAT; we do not further refine Λ . Our initial chunk size is 20, and our per-chunk `check_sat` conflict budget is 1000.

Algorithm 5 falls back to testing individual literals in Γ with *failed literal probing* when $\omega_N = \emptyset$ (i.e. when core refinement exhausts the current assumption set without producing a singleton core) or when the maximum-sized chunk ($k = |\Lambda|$) returns UNDEF. A literal ℓ is said to be *failed* in a CNF formula φ if unit propagation derives a contradiction from $\varphi \wedge \ell$ [37]. If ℓ is failed, then its complement $\neg\ell$ belongs to the backbone of φ . For each $\ell \in \Gamma$, our fallback algorithm checks if $\varphi \wedge \ell$ is UNSAT in under 10 conflicts; if so, $\neg\ell$ is a backbone literal.

V. EVALUATION

We implemented our approach inside Z3[1]. All experiments were conducted on a dual-socket server with two AMD EPYC 7763 processors (64 cores per socket, 128 cores, 2.45 GHz clock speed), and 512GB RAM, running Ubuntu 24.04.4 LTS. The International Satisfiability Modulo Theories Competition (SMT-COMP) is the standard platform in the SMT community to compare solver performance [38]. We show our approach outperforms several state-of-the-art parallel solvers on a diverse set of difficult benchmarks from the SMT-COMP 2025 Parallel Track. We then conduct ablations demonstrating that our approach scales with computational resources and benefits from all system components.

A. SMT-COMP 2025 Parallel Track Benchmarks

We evaluate our approach against the state-of-the-art parallel solvers SMTS [9], ARIPARTI [16], and CVC5 in both portfolio

⁴Algorithm 7 in [36]

Algorithm 5: Backbone Detection (based on [36])

Input: Backbone candidate literals Λ ; SMT input formula φ ; $K \in \mathbb{N}^+$ (chunk size)

```

1  $k \leftarrow \min(K, |\Lambda|)$ ; // Initial chunk size
2 while  $\Lambda \neq \emptyset$  do
3    $\Gamma \leftarrow$  pick  $k$  literals from  $\Lambda$ ;
4    $\omega_N \leftarrow \{\neg \ell \mid \ell \in \Gamma\}$ ;
5   while true do
6      $\Phi \leftarrow \varphi \cup \{\{\ell\} \mid \ell \in \omega_N\}$ ;
7      $r \leftarrow \text{check\_sat}(\Phi)$ ;
8     if  $r = \text{SAT}$  then
9        $\text{set\_global\_sat}(\text{model}(\Phi))$ ;
10      return;
11    else if  $r = \text{UNSAT}$  then
12       $C \leftarrow \text{UnsatCore}(\omega_N)$ ;
13      if  $C = \emptyset$  then
14         $\text{set\_global\_unsat}()$ ;
15        return;
16      else if  $C = \{\ell\}$  then
17         $\text{collect\_backbone}(\neg \ell)$ ;
18         $\Lambda \leftarrow \Lambda \setminus \{\neg \ell\}$ ;
19         $\varphi \leftarrow \varphi \cup \{\neg \ell\}$ ;
20         $\Gamma \leftarrow \Gamma \setminus \{\neg \ell\}$ ;
21      // Remove literals in the core
22       $\omega_N \leftarrow \omega_N \setminus C$ ;
23      if  $\omega_N = \emptyset$  then
24        test all  $\ell \in \Gamma$  with failed literal probing;
25         $\Lambda \leftarrow \Lambda \setminus \Gamma$ ;
26         $k \leftarrow \min(K, |\Lambda|)$ ; // reset chunk size
27        break; // Done with the chunk
28      else //  $r = \text{UNDEF}$ 
29        if  $k < |\Lambda|$  then
30          // retry with larger chunk
31           $k \leftarrow \min(2k, |\Lambda|)$ ;
32        else
33          // Done with the chunk
34          test all  $\ell \in \Gamma$  with failed literal probing;
35           $\Lambda \leftarrow \Lambda \setminus \Gamma$ ;
36        break;

```

(SMT-D [11]) and partitioning (CVC5-P [14]) modes on 6 logics from the SMT-COMP 2025 Parallel Track [39]: QF_NIA, QF_NRA, QF_LIA, QF_LRA, QF_IDL, and QF_RDL. The SMT-COMP Parallel Track curates benchmarks designed to be challenging for parallel solvers. We run each solver with 8 threads and a 20-minute wall-clock timeout per benchmark.

1) *Experimental Setup:* In line with Figure 1, in addition to the 8 partition tree threads that work on the input formula, our Z3 parallel setup adds a core minimizer thread (Algorithm 4) and 2 backbone detection threads (Algorithm 5) for a total of 11 threads. For the arithmetic

benchmarks, we set `tactic.default_tactic=smt`, `smt.auto_config=false`. For the difference logics, we set `smt.auto_config=true`, `smt.arith_solver=4`. The sequential backend solver for our Z3 parallel architecture is simply the current build of Z3 in single-threaded mode.

We configure each external solver with the parameters specified in the associated paper. Our SMTS setup uses the iterative tree partitioning algorithm from [9]. OPENSMT is the base solver with $T = 8$ parallel solver instances. We enable partitioning and lemma sharing. The solver timeout T_S starts at 32 seconds and doubles when elapsed time reaches $4 \cdot T_S$. We use the current version of OPENSMT as the sequential SMTS baseline. We cannot report nonlinear arithmetic results for SMTS as OPENSMT does not support these logics.

Our ARIPARTI setup runs the dynamic variable-level partitioning framework from [16] with a maximum of $T = 8$ concurrent worker tasks. ARIPARTI supports three different backend solvers (Z3, CVC5, OPENSMT); for comparison, we select Z3 to best match our own architecture. ARIPARTI uses a pre-compiled Z3 binary from 2023 (v4.12.1); we thus run this baseline separately from our own current sequential Z3 build.

SMT-D and CVC5-P both use CVC5[2] as their sequential backend solvers. We build and run the current version of CVC5 as the sequential baseline for SMT-D and CVC5-P. Our SMT-D (CVC5 portfolio mode) setup uses the worker diversity strategy from [11] using a portfolio of $T = 8$ independent CVC5 instances. Clause sharing (which requires the full SMT-D gRPC broker) is omitted. Worker diversity follows the guided randomization strategy of [11]: workers are divided into a *standard cluster* (75%, $\lfloor \frac{3T}{4} \rfloor$ workers) using default randomness, and a *noisy cluster* (25%, $\lfloor \frac{T}{4} \rfloor$ workers) using high randomness to explore parts of the search space that standard workers ignore. Standard workers are populated by cycling through logic-specific base option sets (Table II of [11]) in three passes: (1) base options as-is, (2) base options with flipped decision heuristic (`--decision=justification` if the logic default is `internal`, else `--decision=internal`), and (3) base options with `--decision=internal` and distinct random seeds. Noisy workers use the same logic-specific base options with `--decision=internal`, `--random-freq=0.75`, and distinct random seeds. For logics not specified in Table II (nonlinear arithmetic), we fall back to CVC5 defaults with `--decision=internal`.

Our CVC5-P (partition mode) setup uses the graduated portfolio approach from [14]. Given $T = 8$ threads, we use a ranked list of partitioning strategies (default: `decision-scatter`, `decision-cube`, following the recommended $m = 2$ portfolio of [14]) to construct a graduated portfolio: partition batches are allocated greedily in order of increasing size (2, 4, 8, ...), cycling through the strategy ranking at each size level until we have T partitions. For each batch, CVC5 is invoked sequentially with delays of $t_1 = 3s$ and $t_2 = 0.1s$ (following [14]) to generate the partitions. Generated partitions across all batches are then solved concurrently using T CVC5 solver instances (multijob scheduling).

2) *Cross-Solver Results*: The sequential backend solvers of each parallel architecture vary significantly in performance. For a fair evaluation, we thus examine the performance delta between the sequential backend solver and 8-thread parallel mode for each architecture. Specifically, we examine the delta in number of solved examples and in PAR-2 scores (the sum of the runtimes for all solved instances, plus twice the timeout value multiplied by the number of unsolved instances [40]). A lower PAR-2 score indicates better performance. Of note, performance profiles naturally vary across SMT-LIB logics because theory solvers employ different reasoning procedures that interact differently with search-space partitioning. In Z3, the NIA and NRA solvers perform much of their theory reasoning during final checks, while the RDL and IDL solvers perform most theory reasoning while propagating assignments to atomic formulas. The LIA solver uses a combination of propagation-based inference and final checks. Consequently, the cost and timing of theory reasoning (and hence the benefit of search-space partitioning) vary by logic. Our evaluation is thus designed to evaluate our framework across these different solving paradigms.

Table I summarizes our results. In QF_LRA, we achieve the largest increase in solved instances and the largest PAR-2 reduction by a substantial margin over all competing solvers. In QF_NIA, we again achieve the largest PAR-2 reduction and tie with ARIPARTI for the largest gain in solved instances. In QF_LIA, we again achieve the largest PAR-2 reduction while remaining competitive in solved-instance gains, trailing ARIPARTI by only one benchmark and outperforming all other solvers. On QF_IDL, our solved-instance delta is likewise only one lower than SMTS, and our PAR-2 reduction is only slightly smaller. Our architecture substantially improves over sequential performance on QF_IDL; all other architectures besides SMTS *regress* relative to their sequential baselines on this logic. QF_NRA is our weakest logic. ARIPARTI is the clear winner on QF_NRA, but our architecture remains competitive with the remaining solvers. Results on QF_RDL are inconclusive, as all architectures show negligible changes. Finally, recall that ARIPARTI is specifically designed for arithmetic benchmarks by performing arithmetic-theoretic variable-level partitioning[16]. Our architecture is highly generalized and not tailored to arithmetic; we nonetheless remain competitive with ARIPARTI on arithmetic benchmarks and also deliver strong results on difference logic (QF_IDL).

Table II confirms our system delivers the strongest overall performance across the benchmark suite. We report overall SAT, UNSAT, and total solved-instance deltas between sequential and 8-thread parallel mode for each architecture. Since SMTS does not support nonlinear arithmetic (2 of the 6 benchmark logics), we additionally report average deltas to enable a fair comparison across all solvers. We also report the average PAR-2 reduction (in thousands) relative to sequential performance. Our architecture achieves both the **largest average solved-instance gain** and the **largest average PAR-2 reduction** overall. In particular, we have the **largest gain in UNSAT examples** (SMTS wins for SAT examples by a small margin; we tie

closely behind with ARIPARTI). The strong gains in both our raw and average UNSAT deltas speak to the effectiveness of our core-guided search-space pruning procedure (Algorithm 3).

B. Ablations

The SMT-COMP Parallel benchmark set is relatively small and designed for cross-solver comparison, making it less suitable for isolating the effects of individual algorithmic components. Larger, balanced, and more diverse benchmark suites yield more robust estimates of each component's contribution. We thus run ablations of our system on randomly selected subsets from the 2024 and 2025 SMT-LIB QF_LIA and QF_NIA nonincremental benchmark suites [41]. Each experiment uses a 10-minute (600-second) per-benchmark timeout. We demonstrate (1) how our system scales with computational resources and (2) the importance of the core-guided pruning mechanism, core minimizer thread, and backbone threads.

1) *Scalability*: To show how our system scales with available resources, we run it with 1 (vanilla sequential Z3), 1 (using our parallel framework), 2, 4, 8, and 16 worker threads on two randomly selected subsets from the complete 2024 SMT-LIB nonincremental benchmark suite [42] (consisting of 5000 and 2313 benchmarks, respectively, for QF_LIA and QF_NIA). Figure 2 summarizes our results via cactus plots. We use a logarithmic x-axis for runtime in seconds (1 to 600), and a y-axis representing the cumulative number of benchmarks each configuration solved under the 600-second timeout. On QF_LIA, easy benchmarks do not benefit from parallelization, likely due to concurrency overhead on otherwise simple problems. Parallelism becomes beneficial on harder instances, with 16-threads taking the lead after 73 seconds. The benefit is even more pronounced for QF_NIA: sequential execution performs substantially worse than every parallel configuration, and performance scales smoothly with thread count, with 16-threads taking the lead after only 10 seconds. The increased benefit of our framework on QF_NIA is consistent with its greater practical and theoretical difficulty: Z3 performance on QF_NIA is unstable in practice, and while QF_LIA satisfiability is NP-complete [43], QF_NIA satisfiability is undecidable [44]. Search-space partitioning is thus beneficial when concurrency overhead, which increases with the thread count, is amortized by the reduced complexity of the resulting subproblems.

2) *Solver Ablations*: The remaining ablations use a carefully balanced subset of the 2025 SMT-LIB nonincremental QF_LIA benchmark suite [45], consisting of 800 benchmarks (400 SAT / 400 UNSAT) up to 1.2MB in size each. Trivial instances solved by sequential Z3 in under 10 seconds are excluded. To account for inherent variability in runtimes, we repeat each ablation five times, and report the mean differences in both PAR-2 score (in thousands) and number of solved instances relative to the full configuration. We further visualize the pooled results across all five repetitions of each ablation using scatter plots, where $\times$ denotes a SAT benchmark solved by either configuration, $\square$ denotes an UNSAT benchmark, and $*$ denotes a timed-out benchmark. The axes use a logarithmic scale for per-benchmark

TABLE I: SMT-COMP 2025 Benchmark Results

		z3 (ours)	SMTS	ARIPARTI	SMT-D	CVC5-P
QF_LRA (38)	Solved (Seq)	1	22	6	3	3
	PAR-2 (Seq)	89.37	46.10	80.70	85.80	85.80
	Solved (8T)	4	22	7	5	4
	PAR-2 (8T)	83.69	45.11	77.82	82.55	84.15
	Δ Solved	+3	+0	+1	+2	+1
	Δ PAR-2	-5.69	-0.99	-2.88	-3.25	-1.66
QF_LIA (44)	Solved (Seq)	2	10	2	1	1
	PAR-2 (Seq)	102.82	86.89	101.19	103.59	103.59
	Solved (8T)	8	14	9	3	2
	PAR-2 (8T)	89.25	75.32	87.83	100.14	102.19
	Δ Solved	+6	+4	+7	+2	+1
	Δ PAR-2	-13.56	-11.57	-13.37	-3.45	-1.40
QF_NRA (44)	Solved (Seq)	2	†	5	1	1
	PAR-2 (Seq)	100.81	†	94.71	104.38	104.38
	Solved (8T)	3	†	8	2	2
	PAR-2 (8T)	98.40	†	86.98	101.71	101.26
	Δ Solved	+1	†	+3	+1	+1
	Δ PAR-2	-2.41	†	-7.73	-2.68	-3.13
		z3 (ours)	SMTS	ARIPARTI	SMT-D	CVC5-P
QF_NIA (44)	Solved (Seq)	13	†	8	4	4
	PAR-2 (Seq)	76.41	†	89.65	98.46	98.46
	Solved (8T)	16	†	11	4	3
	PAR-2 (8T)	69.12	†	82.58	98.02	99.58
	Δ Solved	+3	†	+3	+0	-1
	Δ PAR-2	-7.29	†	-7.08	-0.44	+1.12
QF_RDL (24)	Solved (Seq)	1	0	1	0	0
	PAR-2 (Seq)	55.57	57.60	55.72	57.60	57.60
	Solved (8T)	1	0	1	0	0
	PAR-2 (8T)	55.64	57.60	56.10	57.60	57.60
	Δ Solved	+0	+0	+0	+0	+0
	Δ PAR-2	+0.07	+0.00	+0.38	+0.00	+0.00
QF_IDL (45)	Solved (Seq)	14	3	5	2	2
	PAR-2 (Seq)	82.21	101.80	98.98	104.97	104.97
	Solved (8T)	18	8	4	1	2
	PAR-2 (8T)	75.15	92.78	100.26	106.40	105.51
	Δ Solved	+4	+5	-1	-1	+0
	Δ PAR-2	-7.06	-9.02	+1.28	+1.43	+0.54

Legend: Solved: number of solved benchmarks (sequential and 8-thread parallel); PAR-2: penalized average runtime (in thousands); Δ : difference between 8-thread and sequential runs; †: solver does not support this logic.

TABLE II: Aggregate SMT-COMP 2025 Deltas

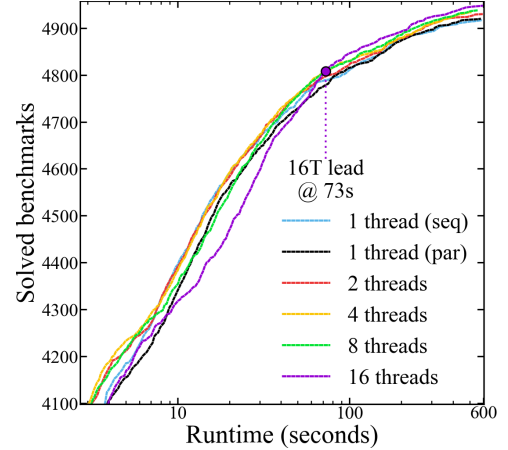
	z3 (ours)	SMTS	ARIPARTI	SMT-D	CVC5-P
Δ SAT	+12	+9*	+12	+3	+3
Δ UNSAT	+5	+0*	+1	+1	-1
Δ All	+17	+9*	+13	+4	+2
Avg. Δ SAT	+2.00	+2.25	+2.00	+0.50	+0.50
Avg. Δ UNSAT	+0.83	+0.00	+0.17	+0.17	-0.17
Avg. Δ All	+2.83	+2.25	+2.17	+0.67	+0.33
Avg. Δ PAR-2	-5.99	-5.39	-4.90	-1.40	-0.76

*SMTS does not support NIA, NRA; the rest handle all 6 logics. PAR-2 score is in thousands.

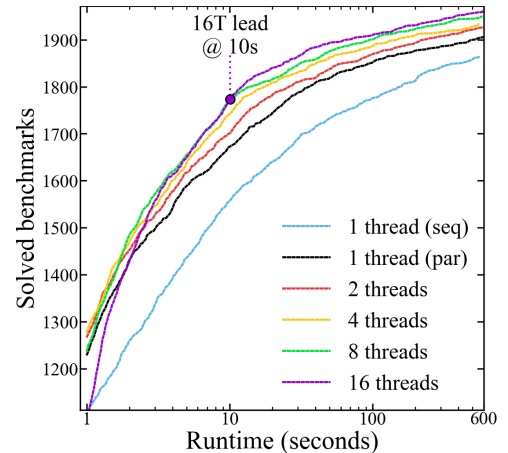
runtimes in the range $[0, 600]$ seconds; the diagonal denotes equal runtimes for both configurations.

Core-Guided Pruning. To ablate core-guided pruning (Algorithm 3), when closing node n , we ignore n 's derived core and instead use n 's full cube (path from root to n) as its core. This prunes only n and its subtree, similar to SMTS and ARIPARTI. Removing core-guided pruning also renders the core minimizer thread obsolete, so we disable it. This ablation addresses the only fully synchronous component of our architecture: core-guided pruning modifies shared tree state and must therefore execute under a global lock, temporarily blocking the batch manager. Our results show that the benefit of pruning outweighs this synchronization overhead (Table III), with the results pooled across all five runs visualized in Figure 3. Enabling core-guided pruning solves an average of four additional benchmarks and decreases the mean PAR-2 score by 3.52 points. UNSAT benchmarks account for an average of 3.60 (90%) of the additional solves and 2.74 points (78%) of the PAR-2 reduction, supporting the hypothesis that core-guided pruning particularly benefits UNSAT problems by dynamically shrinking the search space.

Core Minimizer Thread. We next ablate the core minimizer thread. Across five runs, enabling core minimization solves an average of 7.4 additional benchmarks and decreases the mean



(a) QF_LIA



(b) QF_NIA

Fig. 2: Scaling of Parallel Z3 Using 1 to 16 Solvers

PAR-2 score by 6.24 points (Table IV). Of these additional solves, an average of 5.8 (78%) are UNSAT, supporting the

TABLE III: Core-Guided Pruning Ablation Aggregates

Metric	Full system (with core-guided pruning)	No core- guided pruning
Avg. SAT	389.0	388.6
Avg. UNSAT	376.2	372.6
Avg. total solved	765.2	761.2
Avg. PAR-2	123.83	127.35
Avg. Δ SAT	-0.00	-0.40
Avg. Δ UNSAT	-0.00	-3.60
Avg. Δ solved	-0.00	-4.00
Avg. Δ PAR-2	+0.00	+3.52

PAR-2 score is in thousands. Aggregates are over 5 runs for each mode.

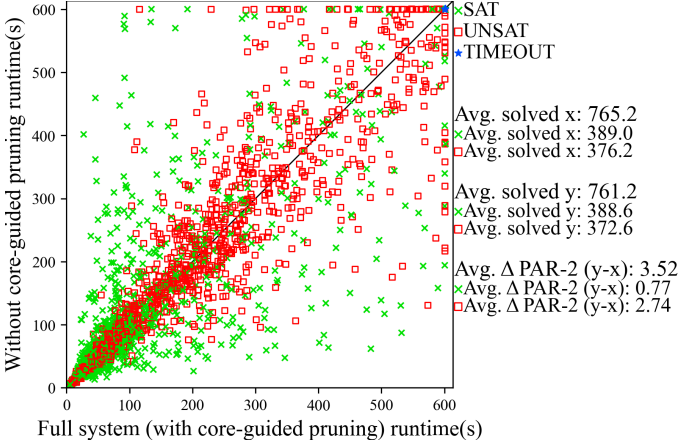


Fig. 3: Core-Guided Pruning Ablation (5 Runs, Pooled)

hypothesis that core minimization particularly benefits UNSAT problems by enabling more aggressive search-space pruning. Figure 4 visualizes the results pooled across all five runs. Despite per-instance variability across runs, reflected by the dispersion around the diagonal, core minimization reduces overall PAR-2 by 5%, with reductions of 4% for SAT and 5% for UNSAT benchmarks.

Core Minimizer Thread. We next ablate the core minimizer thread. The aggregate results are reported in Table IV, with the results pooled across all five runs visualized in Figure 4. Enabling core minimization solves an average of 7.4 additional benchmarks and decreases the mean PAR-2 score by 6.24 points. UNSAT benchmarks account for an average of 5.8 (78%) of the additional solves and 4.53 points (73%) of the PAR-2 reduction, supporting the hypothesis that core minimization particularly benefits UNSAT problems by strengthening the cores used by core-guided pruning, thereby enabling more aggressive search-space pruning.

Backbone Threads. Recall that backbone detection proves that a literal ℓ belongs to the backbone by showing that $\varphi \wedge \neg \ell$ is UNSAT; we call this *negative mode* [36]. Our bidirectional online backbone detection architecture complements this with a *positive mode*, which assumes candidate literals directly and attempts to extend the resulting assignment into a full model. We evaluate the full system with 2 backbone threads (negative and positive modes) against configurations with 1

TABLE IV: Core Min Ablation Aggregates

Metric	Full system (with core min)	No core minimization
Avg. SAT	389.0	387.4
Avg. UNSAT	376.2	370.4
Avg. total solved	765.2	757.8
Avg. PAR-2	123.83	130.07
Avg. Δ SAT	-0.00	-1.60
Avg. Δ UNSAT	-0.00	-5.80
Avg. Δ solved	-0.00	-7.40
Avg. Δ PAR-2	+0.00	+6.24

PAR-2 score is in thousands. Aggregates are over 5 runs for each mode.

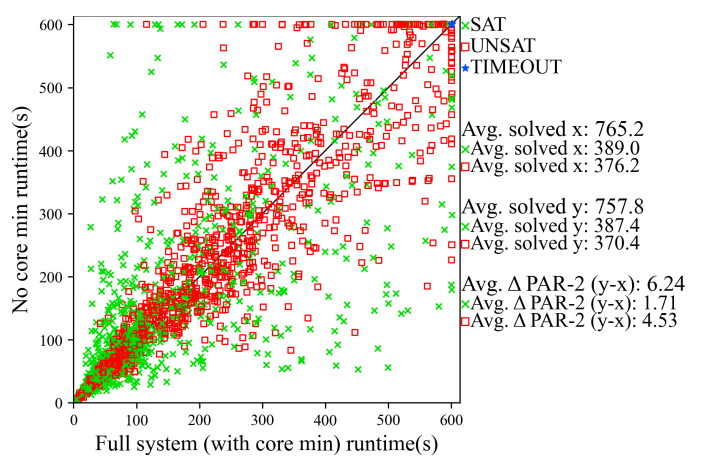


Fig. 4: Core Minimization Ablation (5 Runs, Pooled)

thread (negative mode only) and 0 backbone threads. The aggregate results are reported in Table V, with the results pooled across all five runs visualized in Figure 5.

Performance improves with each added backbone thread. Moving from 0 to 1 backbone thread solves an average of 3.8 additional benchmarks and decreases mean PAR-2 by 3.75 points; UNSAT benchmarks account for 3.4 (89%) of these additional solves and 3.64 points (97%) of the PAR-2 reduction. Moving from 1 to 2 backbone threads solves another 3.6 benchmarks and decreases mean PAR-2 by a further 2.00 points; UNSAT benchmarks account for 2.6 (72%) of these additional solves and 1.30 points (65%) of the additional PAR-2 reduction. Although both transitions favor UNSAT, positive mode yields larger incremental SAT gains than negative mode: +1.0 vs +0.4 average solved instances, and -0.70 vs -0.10 PAR-2 points. This is consistent with our hypothesis that positive mode facilitates SAT search. Overall, the full setup solves 7.4 more benchmarks and decreases mean PAR-2 by 5.75 points relative to 0 backbone threads. UNSAT benchmarks account for 6.0 (81%) of these total additional solves and 4.94 points (86%) of the total PAR-2 reduction. This strong UNSAT skew is consistent with proven backbone literals shrinking the search space and accelerating exhaustive UNSAT proofs, while the incremental SAT gains from positive mode demonstrate the complementary benefit of the full dual-mode architecture.

TABLE V: Backbone Ablation Aggregates

Metric	Full system (2 backbone threads)	1 backbone thread	0 backbone threads
Avg. SAT	389.0	388.0	387.6
Avg. UNSAT	376.2	373.6	370.2
Avg. total solved	765.2	761.6	757.8
Avg. PAR-2	123.83	125.83	129.58
Avg. Δ SAT	-0.00	-1.00	-1.40
Avg. Δ UNSAT	-0.00	-2.60	-6.00
Avg. Δ solved	-0.00	-3.60	-7.40
Avg. Δ PAR-2	+0.00	+2.00	+5.75

PAR-2 score is in thousands. Aggregates are over 5 runs for each mode.

VI. CONCLUSION

We introduced a novel framework for parallel SMT solving that *learns from active search* by integrating dynamic VSIDS-based partitioning, core-guided search-space pruning, and online backbone detection into a feedback-driven architecture that uses the evolving search state to steer solving. Our system achieves the strongest overall performance across six challenging SMT-COMP 2025 benchmark suites against four state-of-the-art parallel SMT frameworks, and scales effectively with computational resources. Future work will investigate additional mechanisms for learning from active search, such as online parameter tuning, improved heuristics for split atom and backbone candidate selection, and tailoring proof strategies to individual subproblems. Our approach naturally extends to quantifiers within Z3; we plan to conduct further evaluations on quantified benchmarks. Finally, we intend to incorporate non-unit clause sharing, particularly subtree-guarded approaches that exploit search-space locality.

VII. ACKNOWLEDGEMENTS

We thank Diego Oliver Fernandez Pons for suggesting the idea of using backbone candidates. We also thank the anonymous reviewers for ample constructive feedback.

DATA AVAILABILITY

The artifact(s) associated with this paper is/are available at Zenodo under the following DOI: 10.5281/zenodo.21755875.

REFERENCES

- [1] L. M. de Moura and N. S. Bjørner, “Z3: an efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24
- [2] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, “CVC5: A versatile and

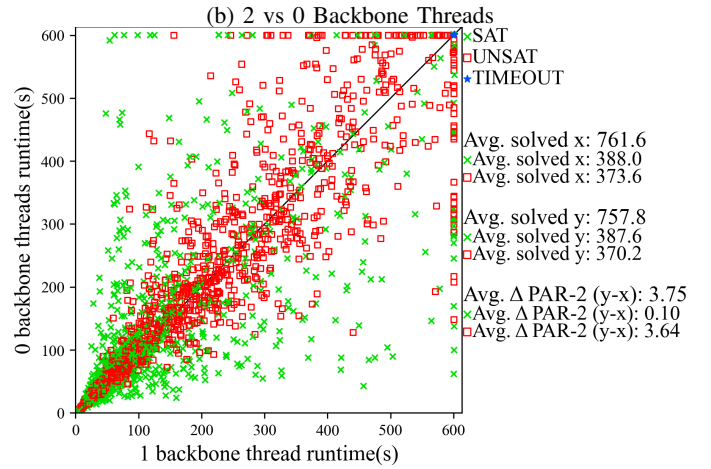
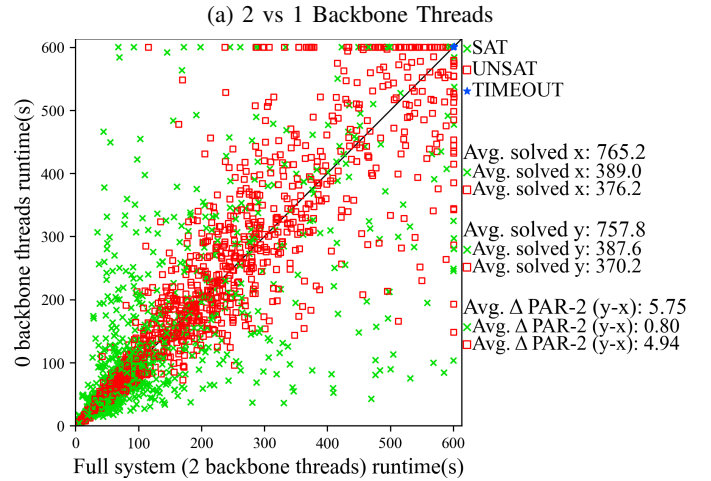
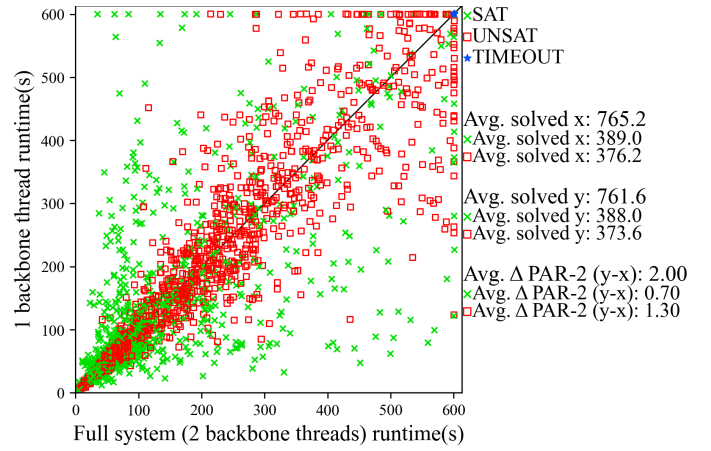


Fig. 5: Backbone Thread Ablations (5 Runs, Pooled)

industrial-strength SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 415–442.

- [3] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, “Opensmt2: An SMT solver for multi-core and cloud computing,” in *Theory and*

- Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds. Springer, 2016, pp. 547–553. [Online]. Available: https://doi.org/10.1007/978-3-319-40970-2_35
- [4] B. Dutertre, “Yices 2.2,” in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds. Springer, 2014, pp. 737–744. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_49
- [5] L. C. Cordeiro and B. Fischer, “Verifying multi-threaded software using SMT-based context-bounded model checking,” in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 331–340. [Online]. Available: <https://doi.org/10.1145/1985793.1985839>
- [6] A. Komuravelli, A. Gurfinkel, and S. Chaki, “SMT-based model checking for recursive programs,” in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds. Springer, 2014, pp. 17–34. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_2
- [7] C. Skalka and J. P. Near, “SMT-boosted security types for low-level MPC,” in *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*, ser. Lecture Notes in Computer Science, V. Vafeiadis, Ed. Springer, 2025, pp. 258–285. [Online]. Available: https://doi.org/10.1007/978-3-031-91121-7_11
- [8] P. Hozzová and N. Bjørner, “Synthesiz3 this: an SMT-based approach for synthesis with uncomputable symbols,” in *Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design, FMCAD 2025, Menlo Park, CA, USA, October 6-10, 2025*, A. Irfan and D. Kaufmann, Eds. TU Wien Academic Press, 2025. [Online]. Available: https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_28
- [9] T. Kolárik, A. E. J. Hyvärinen, S. Asadzadeh, and N. Sharygina, “Parallel SMT solving via iterative tree partitioning,” in *TACAS 2026*, 2026.
- [10] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno, “Mariposa: Measuring SMT instability in automated program verification,” in *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, A. Nadel and K. Y. Rozier, Eds. IEEE, 2023, pp. 178–188. [Online]. Available: https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_26
- [11] C. W. Barrett, P. Chen, B. Cook, B. Dutertre, R. B. Jones, N. Le, A. Reynolds, K. Sheth, C. Stephens, and M. W. Whalen, “SMT-D: new strategies for portfolio-based SMT solving,” in *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, N. Narodytska and P. Rümmer, Eds. IEEE, 2024, pp. 1–10. [Online]. Available: https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_10
- [12] C. M. Wintersteiger, Y. Hamadi, and L. de Moura, “A concurrent portfolio approach to SMT solving,” in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 715–720.
- [13] A. E. J. Hyvärinen and C. M. Wintersteiger, *Parallel Satisfiability Modulo Theories*. Cham: Springer International Publishing, 2018, pp. 141–178. [Online]. Available: https://doi.org/10.1007/978-3-319-63516-3_5
- [14] A. Wilson, A. Nötzli, A. Reynolds, B. Cook, C. Tinelli, and C. W. Barrett, “Partitioning strategies for distributed SMT solving,” in *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, A. Nadel and K. Y. Rozier, Eds. IEEE, 2023, pp. 199–208. [Online]. Available: https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_28
- [15] Z. Battleman, J. E. Reeves, and M. J. H. Heule, “Problem partitioning via proof prefixes,” in *28th International Conference on Theory and Applications of Satisfiability Testing, SAT 2025, Glasgow, Scotland, August 12-15, 2025*, ser. LIPIcs, J. Berg and J. Nordström, Eds. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 3:1–3:18. [Online]. Available: <https://doi.org/10.4230/LIPIcs.SAT.2025.3>
- [16] M. Zhao, S. Cai, and Y. Qian, “Distributed SMT solving based on dynamic variable-level partitioning,” in *Computer Aided Verification*, A. Gurfinkel and V. Ganesh, Eds. Cham: Springer Nature Switzerland, 2024, pp. 68–88.
- [17] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina, “Search-space partitioning for parallelizing SMT solvers,” in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Heule and S. A. Weaver, Eds. Springer, 2015, pp. 369–386. [Online]. Available: https://doi.org/10.1007/978-3-319-24318-4_27
- [18] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of Satisfiability*, 2nd ed., ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, Feb. 2021, vol. 336, ch. 33, pp. 825–885.
- [19] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “Solving SAT and SAT modulo theories: From an abstract davis-putnam-logemann-loveland procedure to dpll(T),” *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006. [Online]. Available: <https://doi.org/10.1145/1217856.1217859>
- [20] J. P. M. Silva and K. A. Sakallah, “GRASP: A search

- algorithm for propositional satisfiability,” *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506–521, 1999. [Online]. Available: <https://doi.org/10.1109/12.769433>
- [21] T. Balyo, P. Sanders, and C. Sinz, “Hordesat: A massively parallel portfolio SAT solver,” in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Heule and S. A. Weaver, Eds., vol. 9340. Springer, 2015, pp. 156–172. [Online]. Available: https://doi.org/10.1007/978-3-319-24318-4_12
- [22] M. Marescotti, A. E. J. Hyvärinen, and N. Sharygina, “Clause sharing and partitioning for cloud-based SMT solving,” in *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, ser. Lecture Notes in Computer Science, C. Artho, A. Legay, and D. Peled, Eds., 2016, pp. 428–443. [Online]. Available: https://doi.org/10.1007/978-3-319-46520-3_27
- [23] M. Marescotti, A. E. J. Hyvärinen, and N. Sharygina, “SMTs: distributed, visualized constraint solving,” in *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, 2018.
- [24] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina, “Lookahead in partitioning SMT,” in *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, pp. 271–279. [Online]. Available: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_37
- [25] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä, “Partitioning search spaces of a randomized search,” *Fundam. Informaticae*, vol. 107, no. 2-3, pp. 289–311, 2011. [Online]. Available: <https://doi.org/10.3233/FI-2011-404>
- [26] T. Weber, “Par4,” Uppsala University, Tech. Rep., 2019. [Online]. Available: http://smt2019.galois.com/papers/tool_paper_9.pdf
- [27] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä, “A distribution method for solving SAT in grids,” in *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and C. P. Gomes, Eds. Springer, 2006, pp. 430–435. [Online]. Available: https://doi.org/10.1007/11814948_39
- [28] D. Schreiber, A. Niemetz, and M. Preiner, “Massively parallel bit-precise verification with bitwuzla and mallob,” in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Junges and G. Katz, Eds. Cham: Springer Nature Switzerland, 2026, pp. 170–191.
- [29] C. Reisenberger, “PBoolector: A Parallel SMT Solver for QF BV by Combining Bit-Blasting with Look-Ahead,” Master’s thesis, Johannes Kepler University Linz, 2014. [Online]. Available: <https://fmv.jku.at/master/Reisenberger-MasterThesis-2014.pdf>
- [30] D. Schreiber, N. Rigi-Luperti, and P. Sanders, “Mallob: Scalable automated reasoning on demand,” in *Computer Aided Verification (CAV)*, 2026, to appear.
- [31] D. Schreiber and P. Sanders, “Scalable SAT solving in the cloud,” in *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, ser. Lecture Notes in Computer Science, C. Li and F. Manyà, Eds., vol. 12831. Springer, 2021, pp. 518–534. [Online]. Available: https://doi.org/10.1007/978-3-030-80223-3_35
- [32] D. Schreiber, A. Niemetz, and M. Preiner, “Massively parallel bit-precise verification with bitwuzla and mallob,” in *Tools and Algorithms for the Construction and Analysis of Systems: 32nd International Conference, TACAS 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11–16, 2026, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2026, p. 170–191. [Online]. Available: https://doi.org/10.1007/978-3-032-22752-2_9
- [33] D. Schreiber, “Mallob: Ipasir bridge,” gitHub repository. [Online]. Available: <https://github.com/domschrei/mallob-ipasir-bridge>
- [34] K. Fazekas, A. Niemetz, M. Preiner, M. Kirchweger, S. Szeider, and A. Biere, “IPASIR-UP: User propagators for CDCL,” in *The 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), July 04-08, 2023, Alghero, Italy*, ser. LIPIcs, M. Mahajan and F. Slivovsky, Eds., vol. 271. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 8:1–8:13.
- [35] O. Guthmann, O. Strichman, and A. Trostanetski, “Minimal unsatisfiable core extraction for SMT,” in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 57–64. [Online]. Available: <https://doi.org/10.1109/FMCAD.2016.7886661>
- [36] M. Janota, I. Lynce, and J. Marques-Silva, “Algorithms for computing backbones of propositional formulae,” *AI Commun.*, vol. 28, no. 2, pp. 161–177, 2015. [Online]. Available: <https://doi.org/10.3233/AIC-140640>
- [37] J. W. Freeman, “Improvements to propositional satisfiability search algorithms,” Ph.D. dissertation, University of Pennsylvania, 1995.
- [38] C. Barrett, L. de Moura, and A. Stump, “SMT-COMP: satisfiability modulo theories competition,” in *Proceedings of the 17th International Conference on Computer Aided Verification*, ser. CAV’05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 20–23. [Online]. Available: https://doi.org/10.1007/11513988_4
- [39] M. Jonáš, F. Bobot, D. Déharbe, and D. Winterer, “SMT-COMP 2025,” 2025, chair: Martin Jonáš, Masaryk University, Czechia.
- [40] N. Froleyks, M. Heule, A. Iser, M. Järvisalo, and M. Suda, “SAT competition 2020,” *Artif. Intell.*, vol. 301, p. 103572, 2021. [Online]. Available:

<https://doi.org/10.1016/j.artint.2021.103572>

- [41] C. Barrett, P. Fontaine, and C. Tinelli, “The SMT-LIB standard: Version 2.6,” Department of Computer Science, The University of Iowa, Tech. Rep., 2017. [Online]. Available: <https://smt-lib.org>
- [42] M. Preiner, H. Schurr, C. W. Barrett, P. Fontaine, A. Niemetz, and C. Tinelli, “SMT-LIB release 2025 (non-incremental benchmarks) (version 2025.08.04),” <https://doi.org/10.5281/zenodo.16740866>, Aug. 2025, accessed on YYYY-MM-DD. [Online]. Available: <https://doi.org/10.5281/zenodo.16740866>
- [43] D. Chistikov, “An introduction to the theory of linear integer arithmetic (invited paper),” in *44th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2024, Gandhinagar, Gujarat, India, December 16-18, 2024*, ser. LIPIcs, S. Barman and S. Lasota, Eds., vol. 323. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 1:1–1:36. [Online]. Available: <https://doi.org/10.4230/LIPIcs.FSTTCS.2024.1>
- [44] S. Zhu and Z. Kincaid, “Breaking the mold: Nonlinear ranking function synthesis without templates,” in *Computer Aided Verification*, A. Gurfinkel and V. Ganesh, Eds. Cham: Springer Nature Switzerland, 2024, pp. 431–452.
- [45] M. Preiner, H. Schurr, C. W. Barrett, P. Fontaine, A. Niemetz, and C. Tinelli, “SMT-LIB release 2024 (non-incremental benchmarks),” <https://doi.org/10.5281/zenodo.11061097>, Apr. 2024, accessed on YYYY-MM-DD. [Online]. Available: <https://doi.org/10.5281/zenodo.11061097>