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

Ilana Shapiro[1], Sorin Lerner[2,1], Nikolaj Bjørner[3]
[1]*UC San Diego, La Jolla, CA, USA*
[2]*Cornell University, Ithaca, NY, USA*
[3]*Microsoft 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 parallel framework that dynamically builds a *binary partition tree* of the search space by sampling from workers' VSIDS statistics during solving. We leverage 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. Portfolio solving has led to substantial improvements over sequential approaches by exploiting the instability of SMT solvers via parallel sampling of the solver configuration space [9–11]. However, because portfolio solving is bounded by the performance of the best sequential solver, the approach is hindered by limited scalability [12, 13].

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. In addition, the runtime of SMT solvers is very unstable and sensitive to seemingly trivial changes, such as randomization during search [13, 14].

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. It is difficult to select good splitting atoms, and badly selected atoms can produce a harder problem than the original. Though techniques such as lookahead have been developed to find good split atoms, cube-and-conquer has had limited success, particularly when the partitions are static [11, 15]. Recent techniques have turned to *dynamic* partitioning [12, 13, 16], where split atoms are selected during search. This enables more sophisticated heuristics for choosing partitioning variables. 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 [13, 17]. However, none of these approaches fully leverage the feedback generated during 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

https://doi.org/

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 flexible, 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 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 demonstrate 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/20129487

II. Preliminaries

A. Definitions and Notations

Satisfiability Modulo Theories (SMT) seeks to 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 $\nu \models_T \varphi$, 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 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 that preserves its T-semantics. The SAT solver then incrementally constructs an assignment α through alternating *propagation* (unit clauses force assignments to literals) and *decision* (the solver chooses values for unassigned literals) phases. 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. Theory solvers may also contribute lemmas. 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 [11]. 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 [9, 10, 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 $\varphi_1, \ldots, \varphi_n$ for parallel solving such that $\bigvee_i \equiv \varphi_i$. Thus, if any one 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 atoms $A = \{a_1, \ldots, a_n\}$ is selected. A *cube* is a conjunction of literals over these atoms, that is, a formula of the form $C = \ell_1 \wedge \cdots \wedge \ell_n$, where each $\ell_i \in \{a_j, \neg a_j\}$ for some $a_j \in A$. The 2^n resulting cubes over these atoms yield 2^n independent subproblems of the form $\varphi \wedge C$, which can be solved in parallel. In contrast, scattering produces a sequence of n partitioning formulae:

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

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* [24] such that for all i, the disjunction $\bigvee_i C_i$ holds, and for all $j \neq i$, $\neg(C_i \wedge C_j)$. Our approach is based on this data structure as well.

Search-space partitioning has the potential to vastly outperform portfolio if good partitions are chosen. However, identifying good partitions is challenging, and poorly chosen partitions can result in harder problems than the original [15].

III. Related Work

Portfolio Solving. Portfolio solving with clause sharing was first introduced in the Z3 SMT solver [10]. Their strategy was simple: randomize 4 instances of Z3, and share 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, 25]. The most successful frameworks were developed in SMTS [17, 22, 23] and CVC5 [9]. SMTS adopts a unique client-server architecture that orchestrates an underlying solver. Globally valid lemmas are stored in an external database; these lemmas are scored by frequency and randomly sampled, and lemmas with more than 3 literals are discarded [13]. CVC5 provides further optimizations such as delayed clause sharing and guided randomization [9]. Although portfolio solving has led to significant speedups (particularly in [9] and [22]), it still suffers from innate lack of scalability.

Search-Space Partitioning. The state-of-the-art 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 [13, 17, 22, 23]. SMTS's underlying solver is OPENSMT2, which uses the scattering strategy to construct partitions from CDCL search branch decision literals [3, 26]. Central to SMTS's architecture is the *partitioning tree* ([17]), which is built dynamically during search [13].

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 [12]. 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 [22, 27], 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 largely unexplored. Existing frameworks also do not fully incorporate feedback from search. Our addresses both of these gaps.

IV. PARALLEL ARCHITECTURE

In this section, we detail our novel parallel architecture and the strategies it employs to dynamically leverage information from search. We use the cube-and-conquer approach: cubes are created on-the-fly by sampling split atoms from solver threads' VISIDS statistics. Figure 1 outlines our high-level design.

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 the now-stale cube. 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 partition tree pruning.

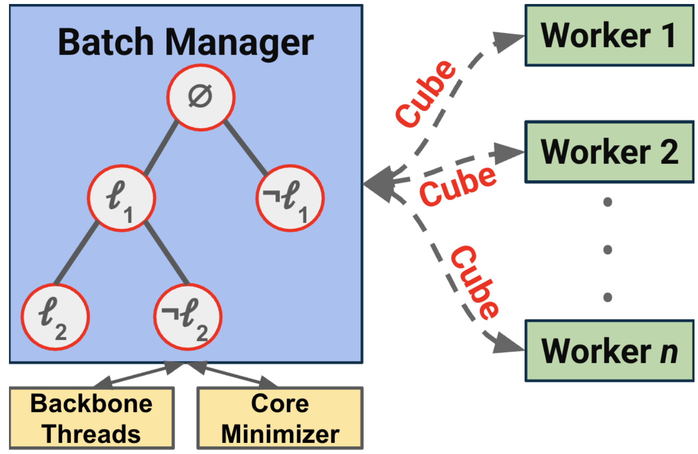


Fig. 1: Overview of Parallel Architecture

A. Binary Partition Tree

At the center of our algorithm is the *partitioning tree*. Unlike SMTS and ARIPARTI, who support n-ary partition trees, we find a binary tree gives the best result, likely due to our 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

Algorithm 1: Tree Expansion (based on [13])

```
Input: Node n; split literal ℓ; effort effort
n.update_round_max_effort(effort);
if ¬n.is_leaf() then
    return;
numActive ← count_active_nodes(root);
numUnsolved ← count_unsolved_nodes(root);
if numUnsolved ≥ numActive · 2 then
    return;
if has_unvisited_open_node(root) then
    return;
if rand() ≥ 0.5 then
    return ;                          // 50% rejection
s ← shallowest_timedout_leaf_depth(root);
if depth(n) = s then
    n.split(ℓ, ¬ℓ);
```

Algorithm 2: Node Selection (based on [13])

```
Input: Root node root; target status target
Output: Best node bestNode
bestNode ← ⊥; bestEffort ← ∞; bestDepth ← −1;
DFS(root)
return bestNode;
Function DFS(cur)
    if cur = ⊥ ∨ status(cur) = closed then
        return;
    if status(cur) = target then
        e ← effort(cur); d ← depth(cur);
        update ← false;
        if bestNode = ⊥ then
            update ← true;
        else if e ≠ bestEffort then
            update ← (e < bestEffort);
        else
            update ← (d > bestDepth);
        if update then
            bestNode ← cur;
            bestEffort ← e;
            bestDepth ← d;
    DFS(left(cur));
    DFS(right(cur));
```

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*). Unit clauses learned by w on n are shared with the other workers. We do not share larger lemmas at this time; exploring 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 may act on a node n in portfolio; in this case, n only records the maximum effort among its workers. This avoids disproportionately inflating the accumulated effort of nodes with higher parallelism. Note that such portfolio workers are not redundant as each operates with a distinct random seed and solver state shaped by prior cubes. Each node tracks how many workers have attempted it.

Our tree expansion policy (Algorithm 1) draws from [13]. When w reports UNDEF for node n, the batch manager counts the number of `active` and unsolved (`open` or `active`) nodes in the partition tree. We expand n only if there are at least twice as many `active` nodes as unsolved nodes, if all `open` nodes have been visited at least once, and if the depth of n is equal to the depth of the shallowest timed-out leaf. Furthermore, we abort expansion based on random throttling (50% chance). As noted in [13], 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 [13]. We prioritize `open` nodes; if none are available, we select from `active` nodes. Among nodes of the chosen status, we select the one with the lowest accumulated effort, with greater depth as the tiebreaker. After this, the order is random.

B. Core-Guided Backjumping

In contrast to prior work [13, 16], when worker w reports UNSAT on node n, we do not simply close n and its subtree. Instead, we perform CDCL-style non-chronological backjumping over the partition tree using UNSAT cores and recursive sibling resolution (Algorithm 3). If n's UNSAT core is empty, this proves global UNSAT. Otherwise, we maintain the invariant that all conflict literals lie on the path from the root to n (i.e., a subset of w's cube).[1] 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]

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

[1] This follows from Z3's assumption-based UNSAT core extraction. Here, the assumptions correspond to the cube.

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

[3] Both 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
if C = ∅ then
    close(root, C) ;                    // global UNSAT
    return;
while n ≠ ⊥ do
    if n.literal ∈ C then
        CloseWithCore(n, C);
        return;
    n ← n.parent;
Function CloseWithCore(n, C)
    if n.closed then
        return;
    p ← n.parent;
    close(n, C);
    l ← p.left;   r ← p.right;
    if l.closed ∧ r.closed then
        R ← (l.core ∪ r.core) \ {l.lit, r.lit};
        if R = ∅ then
            close(root, R) ;            // global UNSAT
            return;
        a ← highest_attach(p, R);
        close(a, R);
        PropagateUpward(a);
Function PropagateUpward(cur)
    while cur.parent ≠ ⊥ do
        p ← cur.parent;   l ← p.left;   r ← p.right;
        if ¬(l.closed ∧ r.closed) then
            return;
        R ← (l.core ∪ r.core) \ {l.lit, r.lit};
        if R = ∅ then
            close(root, R) ;            // global UNSAT
            return;
        close(p, R);
        cur ← p;
```

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.

Algorithm 4: Partial Core Minimization

```
Input: UNSAT core C, SMT input formula φ
Output: Reduced core C or a satisfying model
Λ ← C;                          // literals to test
mus ← ∅;                        // min UNSAT core
// Invariant: φ ∧ mus ∧ Λ is UNSAT
while Λ ≠ ∅ do
    ℓ ← last(Λ);
    Λ ← Λ \ {ℓ};
    Γ ← mus ∪ Λ ∪ {¬ℓ} ;                    // flip ℓ
    Φ ← φ ∪ {{ℓ} | ℓ ∈ Γ};
    r ← check_sat(Φ);
    if r = UNDEF then
        mus ← mus ∪ {ℓ};
    else if r = SAT then
        set_global_sat(model(Φ));
        return;
    else if r = UNSAT then
        C ← UnsatCore(Γ);
        if ¬ℓ ∉ C then
            Λ ← ∅;   mus′ ← ∅;
            foreach c ∈ C do
                if c ∈ mus then
                    mus′ ← mus′ ∪ {c};
                else
                    Λ ← Λ ∪ {c};
            mus ← mus′;
C ← mus ∪ Λ;
```

C. Asynchronous Core Minimization

A core returned from the SMT engine is not necessarily minimal, and smaller cores enable more powerful pruning. To that end, 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 [30], 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.

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 $w_2, \ldots, w_m$, 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 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. The *backbone* of a formula φ is the set of literals true in all models of φ [31]. If ℓ is a backbone literal, it is a consequence of φ, and 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 [31]. 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. 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):

$$\mathrm{rank}(c) = \mathrm{phase\_age}(c) \cdot \log_2\big(2 + \mathrm{cube\_hits}(c)\big)$$

Candidates are dispatched in batches Λ to the backbone threads, who attempt to prove them using a modified version of the chunked backbone algorithm [31] (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 $\varphi \wedge \ell$ [32]. 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.

[4] Algorithm 7 in [31]

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. 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 [13], AriParti [16], and CVC5 in both portfolio (SMT-D [9]) and partitioning (CVC5-P [12]) modes on 6 logics from the SMT-COMP 2025 Parallel Track [33]: 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.

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`. We configure each external solver with the parameters specified in the associated paper. AriParti supports three different backend solvers (Z3, CVC5, OpenSMT); for comparison, we select Z3 to best match our own architecture. Precise details of external solver configurations are specified the appendix.

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 examples solved and the delta in PAR-2 scores. PAR-2 is a composite metric that combines runtime and number of solved benchmarks: the sum

Algorithm 5: Backbone Detection (based on [31])

```
Input: Backbone candidate literals Λ; SMT input
       formula φ; K ∈ ℕ⁺ (chunk size)
k ← min(K, |Λ|);        // Initial chunk size
while Λ ≠ ∅ do
    Γ ← pick k literals from Λ;
    ω_N ← {¬ℓ | ℓ ∈ Γ};
    while true do
        Φ ← φ ∪ {{ℓ} | ℓ ∈ ω_N};
        r ← check_sat(Φ);
        if r = SAT then
            set_global_sat(model(Φ));
            return;
        else if r = UNSAT then
            C ← UnsatCore(ω_N);
            if C = ∅ then
                set_global_unsat();
                return;
            else if C = {ℓ} then
                collect_backbone(¬ℓ);
                Λ ← Λ \ {¬ℓ};
                φ ← φ ∪ {¬ℓ};
                Γ ← Γ \ {¬ℓ};
            // Remove literals in the core
            ω_N ← ω_N \ C;
            if ω_N = ∅ then
                test all ℓ ∈ Γ with failed literal probing;
                Λ ← Λ \ Γ;
                k ← min(K, |Λ|);  // reset chunk size
                break ; // Done with the chunk
        else // r = UNDEF
            if k < |Λ| then
                // retry with larger chunk
                k ← min(2k, |Λ|);
            else
                // Done with the chunk
                test all ℓ ∈ Γ with failed literal probing;
                Λ ← Λ \ Γ;
            break;
```

of the runtimes for all solved instances, plus twice the timeout value multiplied by the number of unsolved instances [34]. A lower PAR-2 score indicates better performance.

The sequential backend solver for our Z3 parallel architecture is simply the current build of Z3 in single-threaded mode. SMTS uses OPENSMT[3] as its backend solver; we build and run the current version of OPENSMT as the sequential SMTS baseline. OPENSMT (and thus SMTS) does not support nonlinear arithmetic, and so we cannot report results for these benchmarks with SMTS. Though ARIPARTI also uses Z3 as its backend solver, it uses a pre-compiled binary from 2023 (v4.12.1); we thus run this baseline separately from our own current Z3 build. Finally, SMT-D and CVC5-P both use CVC5[2] as their backend. We build and run the current version of CVC5 as the sequential baseline for SMT-D and CVC5-P.

Table I summarizes our results. Note that we cannot report nonlinear arithmetic results for SMTS as OPENSMT does not support these logics. 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, our method similarly yields 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. Notably, our architecture substantially improves over sequential performance on QF_IDL; however, all other architectures besides SMTS *regress* relative to their sequential baselines. 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.

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

We run ablations of our system on randomly selected subsets from the 2024 and 2025 SMT-LIB QF_LIA and QF_NIA non-incremental benchmark suites [35]. Each experiment uses a 10-minute (600-second) per-benchmark timeout. We demonstrate how our system scales with computational resources and show the importance of the backbone threads, core minimizer thread, and core-guided search-space pruning procedure.

1) Scalability: To show how our system scales with available resources, we run it with 1, 2, 4, and 8 worker threads on two randomly selected subsets from the complete 2024 SMT-LIB

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

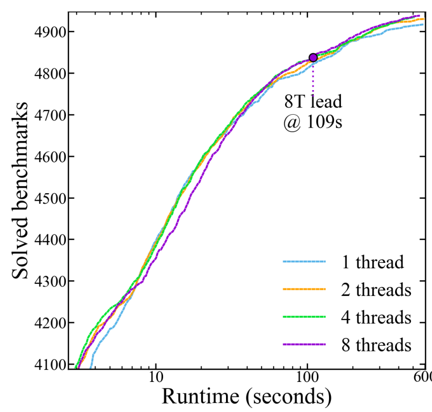
		Z3 (ours)	SMTS	ARIPARTI	SMT-D	CVC5-P
QF_NIA (44)	Solved (Seq)	13	†	8	4	4
	PAR-2 (Seq)	76.41	†	89.65	98.46	98.46
	Solved (8T)	16	†	11	4	3
	PAR-2 (8T)	69.12	†	82.58	98.02	99.58
	Δ Solved	**+3**	†	**+3**	+0	-1
	Δ PAR-2	**-7.29**	†	-7.08	-0.44	+1.12
QF_RDL (24)	Solved (Seq)	1	0	1	0	0
	PAR-2 (Seq)	55.57	57.60	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.

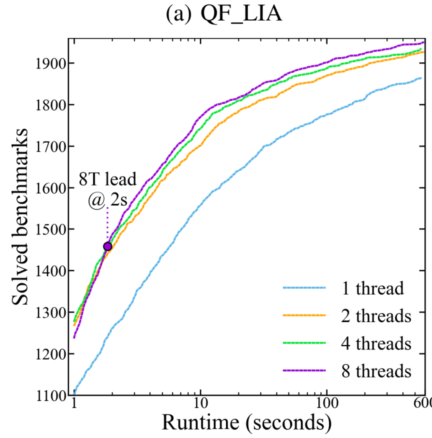
TABLE II: Aggregate SMT-COMP 2025 Deltas

	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.



(a) QF_LIA



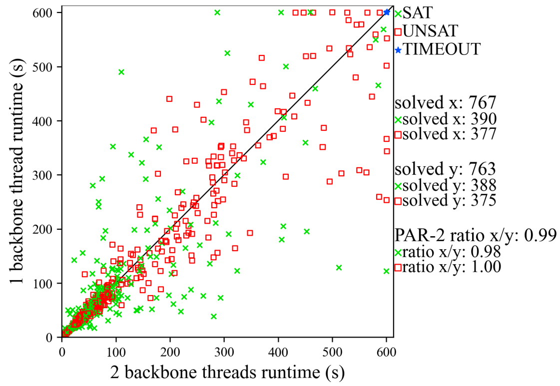
(b) QF_NIA

Fig. 2: Scaling of Parallel Z3 Using 1 to 8 Solvers

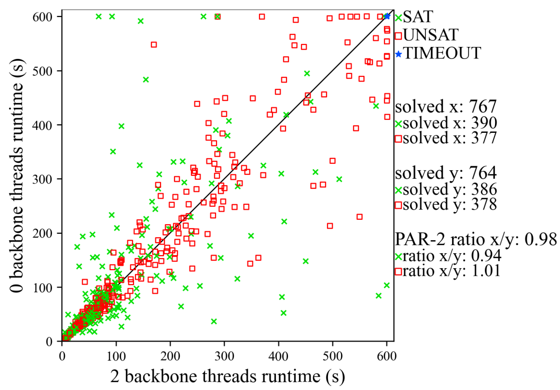
nonincremental benchmark suite [36] (consisting of 5000 and 2313 benchmarks, respectively, for QF_LIA and QF_NIA). Figure 2 summarizes our results via cactus plots. We use a logarithmic x-axis for runtime in seconds (1 to 600), and a y-axis representing the cumulative number of benchmarks each configuration 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. However, parallelism quickly becomes beneficial on harder instances: the 8-thread configuration consistently leads at 109 seconds and beyond. The advantages of parallelism are even more pronounced in QF_NIA. Sequential mode performs substantially worse than all parallel configurations, and performance scales smoothly with thread count: the 8-thread configuration takes the lead after only 2 seconds.

2) Solver Ablations: The remaining ablations use a carefully balanced subset of the 2025 SMT-LIB nonincremental QF_LIA benchmark suite [37], consisting of 800 benchmarks (400 SAT / 400 UNSAT) up to 1.2MB in size each. We exclude trivial instance solved by sequential Z3 in under 10 seconds. Results are summarized using scatter plots, where × denotes a SAT benchmark solved by either configuration, □ denotes an UNSAT benchmark, and ∗ denotes a timed-out benchmark. The axes use a logarithmic scale over per-benchmark runtimes in the range $[0, 600]$ seconds, where the diagonal denotes equal runtimes between the two configurations.

Backbone Threads. We run our full system with 2 backbone threads (negative+positive mode) and compare to ablations with 1 (negative mode only) and 0 backbone threads. Performance decreases incrementally with the removal of each backbone



(a) 2 vs 1 Backbone Threads



(b) 2 vs 0 Backbone Threads

Fig. 3: Backbone Thread Ablations

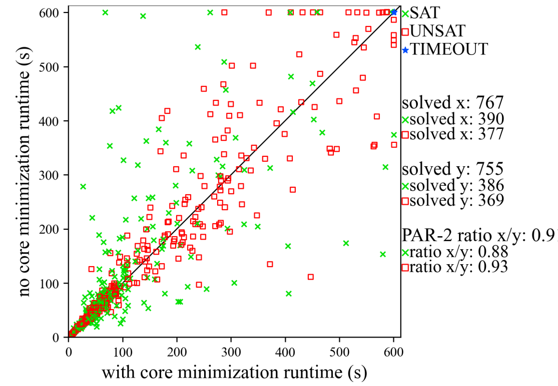


Fig. 4: Core Minimization Ablation

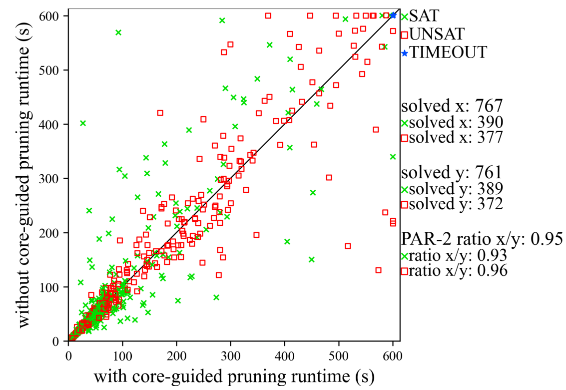


Fig. 5: Core-Guided Pruning Ablation

thread (Figure 3). Compared to 1 backbone thread, the full setup solves 4 more examples and achieves a 2% reduction in the SAT PAR-2 ratio for an overall reduction of 1%. Compared to 0 backbone threads, the full setup solves 3 more examples, but achieves a greater SAT PAR-2 ratio reduction of 6% for an overall reduction of 2%. The backbone threads appear to particularly benefit performance on SAT instances.

Core Minimizer Thread. We next ablate the core minimizer thread and compare against the full system (Figure 4). Enabling core minimization solves 12 additional benchmarks (8 of which are UNSAT) and reduces PAR-2 by 9%. Interestingly, the SAT PAR-2 reduction (12%) exceeds the UNSAT reduction (7%), despite the larger UNSAT gains, because many SAT benchmarks solved by *both* configurations become faster with core minimization enabled (×'s are far above the diagonal), while the additional UNSAT solves tend to occur near the timeout boundary. These UNSAT gains solidify the hypothesis that the core minimizer particularly benefits UNSAT instances by enabling more aggressive search-space pruning.

Core-Guided Pruning. To ablate core-guided pruning (Algorithm 3), when closing node n, we ignore the 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. This ablation also renders the core minimizer thread obsolete, so we disable it. We then compare our full system with the ablated configuration (Figure 5). Enabling core-guided pruning solves 6 more benchmarks (5 of which are UNSAT) for an overall PAR-2 ratio reduction of 5%. Like the core minimization ablation, the SAT PAR-2 reduction (7%) slightly exceeds the UNSAT reduction (4%) since many SAT benchmarks solved by both configurations run faster with pruning enabled, while most unique UNSAT solves occur near the timeout boundary. Nevertheless, 80% of the additional solved benchmarks are UNSAT, which strongly supports the hypothesis that core-guided pruning particularly benefits UNSAT problems by dynamically shrinking the search space.

VI. Conclusion

We introduced a novel framework for parallel SMT solving that integrates dynamic VSIDS-based partitioning, core-guided search-space pruning, and online backbone detection for a feedback-driven architecture that uses the evolving search state to steer solving. We showed that 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 continue to integrate feedback from search, such as online parameter tuning and improved heuristics for split atom and backbone candidate selection. We also intend to incorporate non-unit clause sharing, particularly subtree-guarded approaches that exploit search-space locality.

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 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] 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

[10] 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.

[11] 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

[12] 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

[13] T. Kolárik, A. E. J. Hyvärinen, S. Asadzadeh, and N. Sharygina, "Parallel smt solving via iterative tree partitioning," in *TACAS 2026*, 2026.

[14] 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

[15] 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

[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. 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, 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

[25] T. Weber, “Par4,” Uppsala University, Tech. Rep., 2019. [Online]. Available: http://smt2019.galois.com/papers/tool_paper_9.pdf

[26] 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

[27] 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

[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] 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

[31] 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

[32] J. W. Freeman, “Improvements to propositional satisfiability search algorithms,” Ph.D. dissertation, University of Pennsylvania, 1995.

[33] M. Jonáš, F. Bobot, D. Déharbe, and D. Winterer, “SMT-COMP 2025,” 2025, chair: Martin Jonáš, Masaryk University, Czechia.

[34] 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

[35] 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

[36] 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

[37] ——, “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

APPENDIX

Our SMTS setup uses the iterative tree partitioning algorithm from [13] using OPENSMT as the base solver with $T = 8$ parallel solver instances, and with partitioning and lemma sharing both enabled. The solver timeout T_S starts at 32 seconds and doubles whenever the elapsed time reaches $4 \cdot T_S$.

Our ARIPARTI setup runs the dynamic variable-level partitioning framework from [16] using Z3 (v4.12.1) as the backend solver, with a maximum of $T = 8$ concurrent worker tasks.

Our SMT-D (CVC5 portfolio mode) setup uses the worker diversity strategy from [9] 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 [9]: 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 [9]) 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 covered by Table II (nonlinear arithmetic), our setup falls back to CVC5 defaults with `--decision=internal`.

Our CVC5-P (partition mode) setup uses the graduated portfolio approach from [12]. Given $T = 8$ threads, we use a ranked list of partitioning strategies (default: `decision-scatter`, `decision-cube`, following the recommended $m = 2$ portfolio of [12]) to construct a graduated portfolio: partition batches are allocated greedily in order of increasing size $(2, 4, 8, \ldots)$, cycling through the strategy ranking at each size level until we have T partitions. For each batch, CVC5 is invoked sequentially with delays of $t_1 = 3$s and $t_2 = 0.1$s (following [12]) to generate the partitions. Generated partitions across all batches are then solved concurrently using T CVC5 solver instances (multijob scheduling).