SMT-D: New Strategies for Portfolio-Based SMT Solving

Clark Barrett[1,5], Pei-Wei Chen[3,*], Byron Cook[1], Bruno Dutertre[1], Robert B. Jones[1], Nham Le[1,2,*], Andrew Reynolds[1,6], Kunal Sheth[4,*], Christopher Stephens[1], and Michael W. Whalen[1]

[1]Amazon Web Services, Seattle, USA, {byron, dutebrun, rbtjones, chrisss, mww}@amazon.com
[2]University of Waterloo, Warterloo, Canada, nham.van.le@uwaterloo.ca
[3]University of California, Berkeley, USA, pwchen@berkeley.edu
[4]University of Illinois, Urbana-Champaign, USA, kunal@kunalsheth.info
[5]Stanford University, Stanford, USA, barrett@cs.stanford.edu
[6]University of Iowa, Iowa City, USA, andrew-reynolds@uiowa.edu

***Abstract*—We introduce SMT-D, a tool for portfolio-based distributed SMT solving. We propose a general architecture consisting of two main components: (i) solvers extended with the capability of sharing and importing information on the fly while solving; and (ii) a central manager that orchestrates and monitors solvers while also deciding which information to share with which solvers. We introduce new information-sharing strategies based on the idea of maximizing the amount of useful diversity in the system. We show that on hard benchmarks from recent related work, SMT-D instantiated with the cvc5 SMT solver achieves significant speed-up over sequential performance, is competitive with existing portfolio approaches, and contributes a number of unique solutions.**

I. Introduction

Solvers for satisfiability modulo theories (SMT) are used as general-purpose constraint solvers in a wide variety of applications, including those arising in computer science [6], [10], mathematics [12], [21], operations research [20], and more. Unsurprisingly, as users push SMT solvers to solve more diverse and challenging problems, solver performance becomes the limiting factor in many applications.

Today, state-of-the-art SMT solvers like cvc5 [2], Yices [8], and Z3 [7] do not benefit from additional cores, and if the solving job times out or crashes, any work done during the solving attempt is lost. An effective strategy for distributed SMT solving could address both issues: it can help scale SMT solving across multiple threads and machines, and by sharing information among solver instances, any progress made can be retained and used by others, even if one of the instances crashes or fails.

Two main approaches to distributed SMT solving have been explored: portfolio solving and divide-and-conquer. Portfolio solving is essentially a race between multiple independent SMT solver instances. Each solver is different in some way: either it is a completely different solver, or it is configured differently, or it is provided with a different (but logically equivalent) input. Portfolio solving aims to leverage the well-known high variance that often exists when solving equivalent SMT problems: the hope is that one of the solvers in the portfolio finishes quickly. Portfolio solving can be enhanced by sharing information among the solver instances. Typically, this information consists of formulas that the SMT solvers have learned that can be used to prune the search space. In divide-and-conquer solving, a single problem is partitioned in such a way that if each partition is solved, this provides a solution to the original problem. The main challenge is finding a way to divide the problem that actually improves performance.

In this paper, we introduce SMT-D, a new tool for portfolio-based distributed SMT solving. SMT-D's architecture consists of two main components: (i) solvers extended with the capability to share and import information on the fly while solving; and (ii) a central manager that orchestrates and monitors solvers while also deciding which information to share with which solvers. We also introduce a new information-sharing strategy based on the idea of maximizing the amount of "good" diversity in the system. On hard benchmarks from recent work [22], SMT-D instantiated with the cvc5 SMT solver achieves significant speed-ups over sequential performance, is competitive with existing portfolio approaches, and contributes a number of unique solutions.

In summary, our contributions include:

- a flexible and general architecture for portfolio-based SMT solving with information sharing;
- new portfolio strategies including *delayed sharing* and *guided randomization*;
- an implementation in SMT-D; and
- an evaluation of SMT-D and existing systems on several sets of challenging benchmarks.

The rest of the paper is organized as follows. Section II covers background and related work. Section III describes the architecture of SMT-D. Section IV explains our novel portfolio strategies, and Section V provides additional implementation details. Experimental results are reported in Section VI, and

*These authors did much of the work on this project and did so during internships at Amazon Web Services.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_10

Algorithm 1: The CDCL(T) loop

```
Input  : an SMT formula F
Output: SAT or UNSAT
clauseDB ← toCNF(F);
while True do
    do
        conflict ← BooleanPropagate(clauseDB);
        changed ← False;
        if conflict = ∅ then
            conflict, changed ← theoryCheck() ;
    while changed ∧ conflict = ∅;
    if conflict ≠ ∅ then
        level, lemma ← resolveConflict(conflict);
        clauseDB ← clauseDB ∪ lemma;
        if level < 0 then
            return UNSAT;
        backtrack(level) ;
    else
        if nextLiteral() = NULL then
            return SAT ;
```

Section VII concludes.

II. Preliminaries

We assume the standard logical setting for SMT with the usual notions of terms, interpretations, and theories (see, e.g., [5]). We assume a fixed background theory $\mathcal{T}$ (which could be a composition of one or more individual theories). A $\mathcal{T}$-*interpretation* is an interpretation that interprets symbols in $\mathcal{T}$ as expected. An *atom* is a term of sort BOOL that does not contain any proper sub-terms of sort BOOL. A *literal* is either an atom or the negation of an atom. A clause is a disjunction of literals, and a *cube* is a conjunction of literals. A formula is a term of sort BOOL and is *satisfiable* (resp., *unsatisfiable*) if it is satisfied by some (resp., no) $\mathcal{T}$-interpretation. A formula whose negation is unsatisfiable is *valid*.

A. CDCL(T)-Based SMT Solvers

Most modern SMT solvers are based on the CDCL(T) framework [17], in which a SAT solver and one or more theory solvers cooperate. The SAT solver incrementally builds a truth assignment for the *Boolean skeleton* of the formula, obtained by replacing each unique atom by a Boolean variable. It does this using a standard CDCL loop that is modified to also take into account theory reasoning. The modified CDCL($\mathcal{T}$) approach is shown in Algorithm 1. Initially, an input formula F is converted to conjunctive normal form (CNF), and each clause is stored in a clause database. The main loop first calls Boolean propagation, which may assign some atoms to true or false. If Boolean propagation produces no conflicts, then the theory solvers are called to check for theory conflicts. These two steps repeat until a fixed point is reached. If there is a conflict, it is resolved by learning a conflict lemma and backtracking to an earlier level in which there is no conflict. Otherwise, the $nextLiteral$ function is used to make a case split on a new literal. More details can be found in [5].

B. Portfolio Solving with Lemma Sharing

SMT solvers are highly sensitive. Small changes to the input formula or solver heuristics can result in orders of magnitude difference in solving time [11]. While a cause of frustration for users, this phenomenon can be leveraged to create an effective *portfolio* solving strategy: multiple solvers (each configured differently or with permuted, but logically equivalent, inputs) are run in a "racing" mode and the result of the fastest one is returned. This approach has been explored extensively for both SAT and SMT solving [1], [15], [16], [24] and produces reliable speed-ups [23]. Still, portfolio solving is limited by the performance of the best and luckiest individual solver, leading to diminishing returns with increasing parallelism. Additional performance can be obtained with *information sharing*. Each solver in the portfolio shares its learned conflict lemmas with the others, with the hope that this exchange of information will help find the solution faster.

Implementing a lemma-sharing portfolio in practice is highly non-trivial. System-wise, one must provide scalability, fault tolerance, and low overhead; algorithmic-wise, one must find a good balance between sharing useful information and overloading the system with too many lemmas. Moreover, a well-designed distributed solver should be modular and general, leaving room for future extensions. Ideally, it should also accommodate a wide range of different solvers, support new sharing strategies, and be compatible with other parallel strategies such as partitioning. After a review of related work, we discuss our design and implementation, including design decisions that aim to meet the criteria mentioned above.

C. Related Work

Parallel strategies for SAT solving have been explored extensively [1], [13], [14], [24]. SMT solvers must take into account the more sophisticated CDCL(T) architecture and the different performance profiles of SMT applications. However, the two main approaches for parallel SAT solving are also found in the existing research literature on parallel SMT solving, namely *portfolio solving* and *partitioning*.

Portfolio solving for SMT. Z3 was the first SMT solver to implement portfolio solving with information sharing [23]. The Z3 implementation focuses on a shared-memory implementation and achieves a speed-up of 3.5x on average for moderately difficult integer difference logic benchmarks using a portfolio of four copies of Z3. The sharing strategy used is simple: lemmas with eight literals or fewer are shared, and others are not. Shared lemmas are put into a queue, and each solver in the portfolio checks its queue whenever it backtracks to decision level 0. Unfortunately, portfolio solving is no longer supported in recent versions of Z3.

SMTS [15] is another system implementing portfolio solving with information sharing. As with the Z3 approach, lemmas to be shared are loaded into queues that are accessed when the solvers backtrack to decision level 0. SMTS uses a

central database to store shared lemmas. A filtering heuristic is used to decide which lemmas to add to the database, and a selection heuristic is used to decide which lemmas to share from the database. SMTS obtains its best results using a filter that discards lemmas with more than four literals and a selection heuristic that randomly samples from the database. The SMTS authors specifically flag the need for better filtering and selection techniques in their discussion of future work. Our work builds on and extends these previous approaches in several ways, as we discuss in the next section.

Partitioning in SMT. SMTS [15] implements several partitioning strategies that outperform sequential solving. Relatedly, Wilson et al. [22] implement a partitioning-based parallel solver using cvc5 (which we will refer to as CVC5-P going forward) and show that it outperforms traditional portfolio solving on a set of challenging benchmarks. CVC5-P does not use any information sharing, leaving the integration of sharing to future work. SMTS does explore a limited form of sharing mixed with partitioning: each partition can be solved using a portfolio with lemma sharing, which yields even better performance. The focus of this paper is on portfolio solving with sharing but without partitioning. We aim to build a robust and high-performance solution that could be expanded to include partitioning strategies in future work.

III. An Architecture for Portfolio-Based SMT Solving

In this section, we describe a general architecture for portfolio-based SMT solving and contrast it with prior approaches. Figure 1 depicts our architecture. It is designed to run on either a cluster of computing nodes or a multicore machine. Multiple solver instances (called *workers*) work on the same problem and share information through a central *broker*. The workers are SMT solvers instrumented to be able to export and import learned lemmas on the fly. Workers also track local statistics about lemma imports, exports, and filtering.

The central broker plays two roles. First, in the *control plane* (Fig. 1a), it manages the system by starting, configuring, monitoring, and terminating workers, and by monitoring the overall system and network health (through periodically transmitted ping/pong messages). Second, in the *data plane* (Fig. 1b), it controls system data flow by managing lemma exchange between workers and by tracking and monitoring solver and system-level lemma statistics. In particular, the data-plane broker (i) tracks which lemmas arrive from which individual workers and (ii) decides which lemmas to forward to which workers. This already enables a finer level of control than in previous approaches, where lemma sources are not tracked and static selection criteria are used to decide which lemmas to share. The broker tracks both control and data, including statistics such as the number of lemmas exported or imported so far, time spent in various phases of processing those lemmas, whether a worker has solved its copy of the problem, and so forth.

We advocate a simple hub-and-spoke architecture, similar to that used in SMTS [16]. Using a central broker simplifies coordination and does not require workers to synchronize with each other. We have also observed empirically that in our implementation, the broker is not a communication bottleneck (see Sec. VI). Our hub-and-spoke architecture tolerates worker failure and communication lag or failure. The design makes progress as long as the central broker and some workers are active. The broker is a single point of failure, but can be engineered to be robust.

A. Workers

As mentioned above, the workers are SMT solvers modified to support importing and exporting of learned lemmas during search. This allows for more fine-grained information sharing than prior approaches, where lemmas are only imported at decision level 0, and requires modifying the CDCL loop as shown in Algorithm 2. The loop now calls an export procedure whenever a new lemma is learned as a result of conflict analysis (Line 14). Additionally, during the propagation phase, the worker adds lemmas received from the broker to its database by invoking an import procedure (Line 7). While these changes to CDCL are non-trivial, we can often leverage extensions already present in CDCL SAT solvers to support SMT functionality such as theory propagation. These mechanisms can typically be repurposed for lemma sharing.

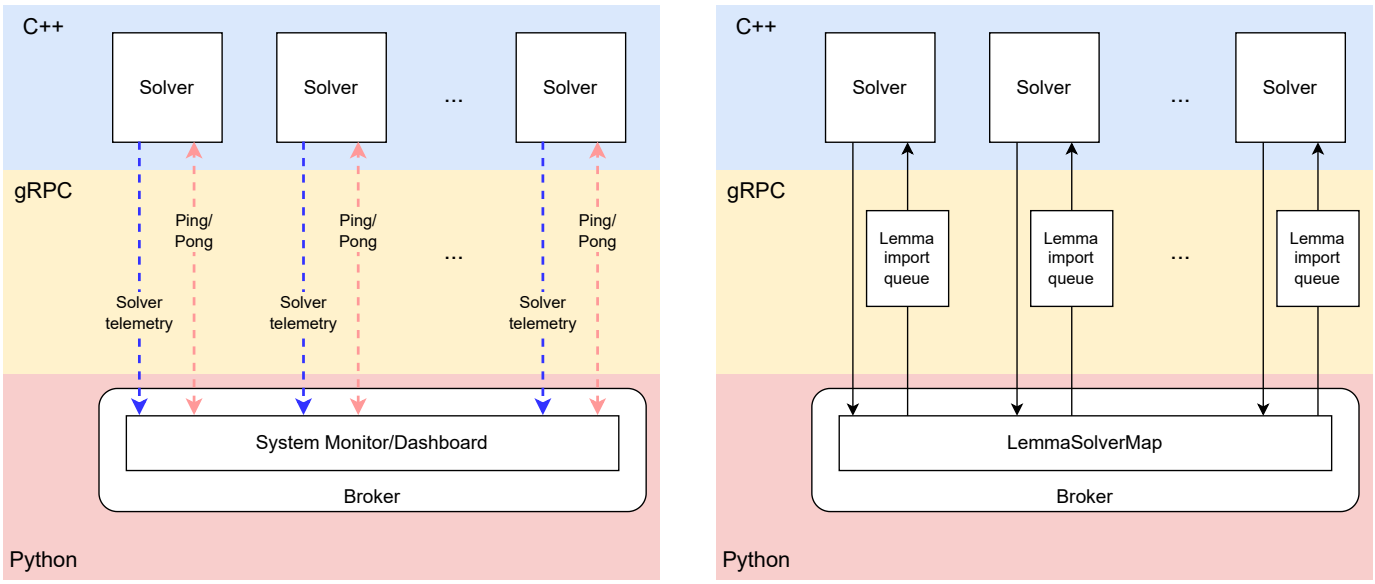
The worker sends telemetry to the broker whenever lemmas are exported or imported (Line 9 and Line 15). Each solver has a mechanism for locally filtering lemmas. The goal is to import and export only *useful* lemmas. We discuss various considerations for local filtering in Section V.

B. Central Broker

The central broker configures both the workers and network communication channels and manages both the control and data planes. During solving, it coordinates the exchange of information between workers and detects termination.

A major role of the central broker is to distribute lemmas learned by one worker to the other workers, while discarding duplicates and managing additional filters. Because multiple workers can learn and export identical lemmas, the broker ensures that each unique lemma is only forwarded (at most) once to each worker. Again, this offers a more fine-grained control mechanism than prior work, in which all lemmas up to a certain size are always shared (Z3) or lemmas are sampled randomly (SMTS) from the database of all shared lemmas.

The core broker algorithm is shown in Algorithm 3. The broker maintains two global variables: $archivedLemmas$ is the set of all lemmas it has received; and $lemmaSolverMap$ is a map from lemmas to worker ids that keeps track of the origin(s) of each lemma. When the broker receives a lemma, the lemma is canonicalized by sorting the set of its literals (Line 5). This ensures that one source of lemma redundancy is eliminated. The broker then uses this canonical form to detect whether the lemma is new (i.e., not in $archivedLemmas$) and to update $lemmaSolverMap$. Function $shouldSend$ controls



(a) Control Plane

(b) Data Plane

Figure 1: Architecture of SMT-D

Algorithm 2: Modified CDCL(T) loop with sharing

Input : an SMT formula F
Output: SAT or UNSAT

$clauseDB \leftarrow toCNF(F)$;
while *True* **do**
  do
    $conflict \leftarrow BooleanPropagate(clauseDB)$;
    $changed \leftarrow$ False;
    if $conflict = \emptyset$ **then**
      $newLemmas \leftarrow importLemmas()$;
      $clauseDB \leftarrow clauseDB \cup newLemmas$;
      $sendtelemetry()$;
      $conflict, changed \leftarrow theoryCheck()$;
  while $(newLemmas \neq \emptyset \vee changed) \wedge conflict = \emptyset$;
  if $conflict \neq \emptyset$ **then**
    $level, lemma \leftarrow resolveConflict(conflict)$;
    $exportLemma(lemma)$;
    $sendtelemetry()$;
    $clauseDB \leftarrow clauseDB \cup lemma$;
    if $level < 0$ **then**
      return UNSAT;
    backtrack($level$);
  else
    if $nextLiteral() = NULL$ **then**
      return SAT;

Algorithm 3: The broker's core lemma exchange routine

$archivedLemmas \leftarrow \emptyset$;
$lemmaSolverMap \leftarrow \emptyset$;
while $True$ **do**
  $\ell, w \leftarrow readMessage()$;
  $\ell \leftarrow canonicalize(\ell)$;
  if $\ell \in archivedLemmas$ **then**
    continue;
  $lemmaSolverMap[\ell].add(w)$;
  if $shouldSend()$ **then**
    for $\ell \in lemmaSolverMap$ **do**
      $send(\ell, allWorkers \setminus lemmaSolverMap[\ell])$;
      $lemmaSolverMap.pop(\ell)$;
      $archivedLemmas.add(\ell)$;

the timing of when lemmas are transmitted to the workers. When *shouldSend* is true, the broker sends each lemma l stored in *lemmaSolverMap* to the workers that did not export it. We discuss implementation choices for *shouldSend* in Section V.

IV. Portfolio Strategies

Constructing effective strategies for portfolio solving with information sharing requires balancing trade-offs from a number of different goals:

- *Maximize diversity:* workers should work on different parts of the search space to avoid redundant work.
- *Share useful lemmas:* ideally, workers should export lemmas that are useful to all instances. A common heuristic for evaluating the value of a lemma is its size (i.e., number of literals in the clause). Smaller clauses are more likely to be useful, as they prune a larger portion of the search space.
- *Avoid overwhelming solvers:* each solver maintains a database containing both locally-learned lemmas and lemmas imported from the broker. Core solver perfor-

mance degrades as the size of the database grows. Sharing too many lemmas can thus be detrimental to overall system performance.
- *Manage communication overhead:* we do not want to overload the communication network with too much data, as this also slows down the system.

Our proposed architecture supports a wide variety of strategy options. We mention two general strategies here, and then discuss specific parameter settings used in our implementation in Section V. The first strategy is *delayed sharing*, which avoids sharing a large set of lemmas that all solvers discover locally. The second strategy is a novel approach to diversity that we call *guided randomization*.

A. Delayed Sharing

In initial experiments with an early prototype, we observed that for some large problems, workers initially export a large number of lemmas and delay calling the $importLemmas$ procedure. Later, when they do try to import the lemmas, the system stalls due to the large amount of communication traffic. Telemetry revealed that this was caused by the initial preprocessing and theory reasoning performed by the solvers.

Before entering the CDCL loop proper, SMT solvers perform formula simplification, conversion to clausal form, and some eager theory reasoning. It is possible for solvers to produce many lemmas during this phase; if each worker is an instance of the same SMT solver, such lemmas are likely to be learned by all solvers working on the problem.To address this issue, we added a delayed sharing mechanism, which ensures that only lemmas learned *after* the preprocessing phase are exported. Enabling this mechanism boosts performance on all of our benchmarks.

B. Guided Randomization

Baseline mechanisms for diversifying solver behavior include selecting different random seeds and modifying solver configurations to ensure that different instances use different search parameters. However, these basic mechanisms have diminishing benefit as we increase portfolio size, as we show in Section VI-B. Using the telemetry collected by the broker, we can observe the number of uniquely learned lemmas (i.e., those learned by a single worker). This metric is a reasonable proxy for system diversity, and indeed, in early experiments, we observed that this number plateaus as we scale the number of workers.

We address this problem by dividing the pool of workers into two clusters, a standard cluster and a *noisy* cluster. Each cluster uses different levels of randomness and different scoring and filtering heuristics. Scoring and filtering can also treat lemmas local to the cluster differently than clauses from other clusters. The noisy cluster uses a high degree of randomness. Intuitively, we expect that solvers in this cluster will learn mostly useless clauses, because they are using heuristics that are far away from the default configurations which have been tuned to be effective. They are also likely to end up exploring parts of the search space that low-randomness solvers ignore. But once in a while, noisy solvers may get lucky and learn clauses that can be useful to solvers in the other cluster.

To maintain diversity in the noisy cluster, we keep the clause databases for solvers in the cluster somewhat isolated. We do this by configuring solvers in noisy clusters to ignore each other and only import lemmas that the central manager determines are highly likely to be useful, (*e.g.,* unit clauses). We discuss a concrete instantiation of this strategy in the next section.

V. Implementation

SMT-D is a distributed SMT solver that implements our proposed architecture and strategies.For the worker instances, we use a version of cvc5 with the main loop modified to support importing and exporting clauses, as discussed in Section III-A. Workers run in separate processes, and each worker process has a separate *wrapper* thread that manages the control plane interface and networking details.

The central broker is written in Python. Communication between broker and workers is implemented with gRPC [9]. We chose gRPC instead of lower-level mechanisms like sockets, because gRPC's high-level API provides better monitoring capabilities and has sufficient performance for (at least) 64 solvers. gRPC also allows us to abstract the parallel and distributed aspects of the system. Thus, SMT-D can be deployed either on a single multicore machine or on a cluster of machines in the cloud.

To export lemmas, we serialize them as strings in the SMT-LIB format [4]. Correspondingly, lemma import requires parsing SMT-LIB strings. This adds some overhead[1] but provides a significant interoperability advantage, as all SMT solvers can parse and print terms in SMT-LIB format. More compact formats could be used at the cost of increased implementation effort and reduced interoperability. For example, SMTS uses a dedicated binary format, but this limits the choice of solvers to those that support this format. Choosing SMT-LIB reduces the cost of adding additional solvers beyond cvc5 to SMT-D.

As explained previously, SMT-D implements comprehensive telemetry for both the control and data planes. We found this real-time information about the solving process at both the local and global levels to be crucial when debugging the system, evaluating different portfolio configurations, and evaluating lemma scoring and filtering strategies. The implementation is heavily parameterized, so that whenever possible, users can choose configuration options at runtime, rather than having to change hard-coded configuration settings.

A. Local Filtering

Several considerations must be taken into account at the worker level. SMT solvers can dynamically create new atoms and new symbols during search. This poses a soundness problem in a distributed setting as one must ensure that new symbols created by a solver instance are interpreted

[1]So far, this has not been a performance limiter, as analysis shows that individual cvc5 workers can import at least 1,000 lemmas/second with <5% parsing overhead. None of the benchmarks reach that level.

consistently by other instances. We currently avoid this issue at the export stage by filtering out lemmas that contain symbols not present in the original formula.[2] New theory atoms are fine as long as they do not introduce new symbols. More sophisticated approaches are possible, but require a mechanism for exporting the definitions of new symbols in a canonical way. Implementing such a mechanism requires extending the baseline SMT solver in a non-trivial way, and we leave it for future work.

As mentioned, our primary goal when filtering is to only export useful lemmas. As in prior work, we use the number of literals in the lemma as our main export filter.

Importing lemmas has a cost. The central broker aims to limit redundancy by only sending a given lemma once to each worker. It is still possible for a worker to produce a lemma internally before learning that another worker has produced the same lemma. Thus, we check in the import procedure whether an imported lemma has already been discovered locally. If so, we drop it. This can be implemented efficiently using mechanisms such as hashing and Bloom filters.

B. Sending Lemmas from the Broker

Our broker uses two indicators to determine when to send lemmas. The first is the wall clock time elapsed since the last lemma transmission. The other is the number of unsent lemmas for a particular worker in the *lemmaSolverMap* map. Function *shouldSend* returns true if the elapsed time is greater than a parameter *delay* or if the number of unsent lemmas is larger than a threshold *maxQueueSize*. By setting these two parameters, the broker can implement different communication policies. It can send lemmas in size-driven batches (like SMTS [15]), in time-driven epochs (like Mallob [19]), or both. We found empirically that so far, the best results come from sharing lemmas individually as soon as they are received. The current sharing limiter is cvc5 parser performance, which supports importing at least 1,000 lemmas per worker per second. With clause sharing filtered by size ≤ 8, only one of the benchmarks approaches that limit, even with 64 workers. If we encounter network bandwidth limitations at some point, we expect that time-driven epochs will provide the best efficiency.

C. Monitoring

SMT-D monitors the number of lemmas imported and exported by each worker. Information from solver wrappers is used to monitor message latency and broker/solver roundtrip times. The *lemmaSolverMap* map also tracks how many solvers independently learned each lemma, that is, the number of lemmas learned by exactly one solver, two solvers, and so forth. This helps dynamically measure diversity in the system, including the amount of redundant work being performed by different solvers. The broker also maintains its own counts of the number of exported and imported lemmas for each worker. Mismatches between the numbers stored in the broker and the numbers reported by the workers mean that the system is overloaded (thus messages are late or dropped) or that there is a bug. During the development of SMT-D, the monitor helped detect multiple bugs and helped inform the design of our lemma-sharing heuristics.

[2]This problem does not occur in the problems in our evaluation, as problems in these logics do not introduce new symbols.

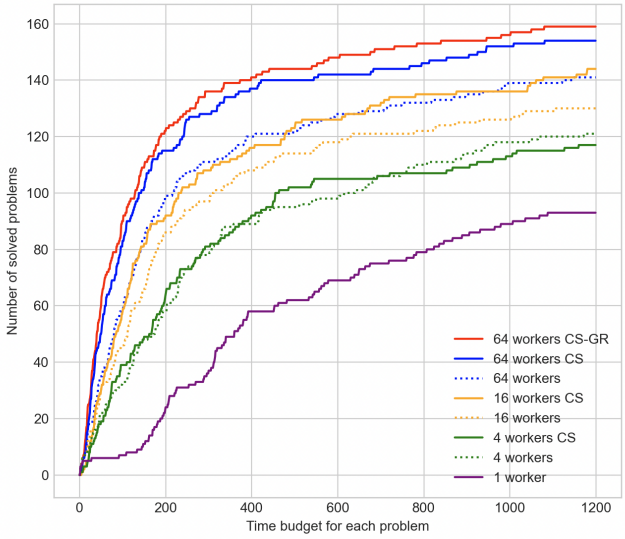


Figure 2: Scalability of SMT-D.

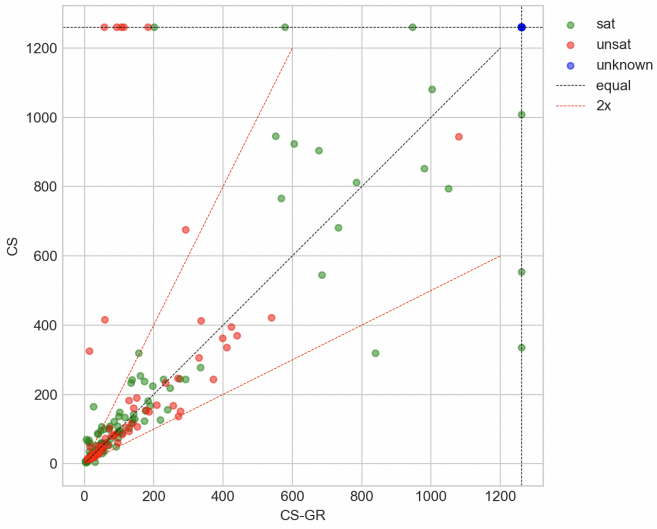


Figure 3: Guided Randomization (CS-GR) vs naive Clause Sharing (CS). Dots on the upper and right-most edges are problems that time out with CS and CS-GR, respectively.

VI. Evaluation

We measure SMT-D performance on the set of benchmarks used in [22], which consists of 214 challenging benchmarks taken from the Cloud track of SMTCOMP22 [18] and other problems from the SMT-LIB benchmark library [3]. The benchmarks come from five SMT-LIB logics: QF_LRA (139), QF_IDL(48), QF_LIA (16), QF_UF (7), and QF_RDL (4).

Benchmarks		SMT-D baseline		SMT-D 64x CS		SMT-D 64x CS-GR		SMTS baseline		SMTS 64x CS		CVC5-P 64x	
Category	Count	Solved	PAR-2	Solved	PAR-2	Solved	PAR-2	Solved	PAR-2	Solved	PAR-2	Solved	PAR-2
QF_LRA	139	90	154	121	61 (↓60%)	120	60 (↓**61%**)	117	69	**127**	**41** (↓41%)	99	130 (↓16%)
QF_IDL	48	1	114	20	72 (↓37%)	**21**	**70** (↓**39%**)	8	99	15	82 (↓17%)	5	107 (↓6%)
QF_LIA	16	0	38	8	22 (↓42%)	9	20 (↓**47%**)	11	13	**14**	**11** (↓15%)	1	36 (↓5%)
QF_UF	7	2	14	3	11 (↓21%)	**7**	**2** (↓**86%**)	6	5	6	3 (↓40%)	4	9 (↓36%)
QF_RDL	4	0	10	2	6 (↓40%)	**2**	**6** (↓**40%**)	0	10	0	10 (0%)	0	10 (0%)
SAT	115	52	172	86	83 (↓52%)	86	82 (↓**52%**)	83	87	**99**	**44** (↓49%)	59	151 (↓12%)
UNSAT	85	41	124	68	55 (↓56%)	**73**	**43** (↓**65%**)	59	75	63	63 (↓16%)	50	106 (↓15%)
UNKNOWN	14	0	34	0	34 (0%)	0	34 (0%)	0	34	0	34 (0%)	0	34 (0%)
ALL	214	93	330	154	171 (↓48%)	159	159 (↓**52%**)	142	196	**162**	**141** (↓28%)	109	291 (↓12%)

Table I: Results comparing SMT-D with other distributed solving tools. PAR-2 scores in thousands.

The goal of our evaluation is to understand the value and potential of our clause-sharing mechanism. Our first set of experiments evaluates different options and configurations of SMT-D (see Section VI-B). This experiment shows the effectiveness of clause sharing over no sharing and the value of guided randomization. Our second set of experiments compares SMT-D with other tools.

A. Configuration

We use a competition build of cvc5 with the elective CLN and GLPK build options enabled. For each logic, we configure cvc5 workers with different sets of options to enhance diversity. These option sets are listed in Table II and are based on the authors' knowledge of the tool and the configurations typically used in the SMT competition. We populate the portfolio by first instantiating a cvc5 instance for each set of options. If we have more workers available in the portfolio, we cycle through the different option sets again, but this time using a different decision engine from the default one for that logic (–decision=justification if the default is –decision=internal and –decision=internal otherwise). After this, we continue to cycle through the different sets of options, this time using only –decision=internal and using a different random seed for each instance. When using noisy solvers, only solvers with –decision=internal are used for the noisy partition.

Table II lists the different sets of options used for each logic. The first set of options listed for each logic is the one used when running a single instance of cvc5 for that logic.

In all experiments, we set the timeout for solving each query to be 1200 seconds, the same timeout used in the parallel and cloud tracks of the SMT competition (in both 2022 and 2023) [18]. Experiments were performed on Amazon EC2 c6a.48xlarge instances, with 96 physical cores and 384 GB of RAM.

Our main metric used for comparison is the *PAR-2 score* used in [22] and the annual SAT competition. PAR-2 is the sum of run times for all instances, but where unsolved instances receive a score of twice the timeout value ($1200 \times 2 = 2400$). This provides a single metric that takes into account both runtime and number of benchmarks solved. The lower the PAR-2 score, the better. We also use *cactus plots* to show the number of solved instances (y-axis) within a limit of s seconds per instance (x-axis). We are primarily interested in the effectiveness of different parallelization strategies and implementations.

Logic	Options
QF_LRA, QF_RDL (option set 1)	–miplib-trick true –miplib-trick-subs 4 –use-approx true –lemmas-on-replay-failure true –replay-early-close-depth 4 –replay-lemma-reject-cut 128 –replay-reject-cut 512 –unconstrained-simp true –use-soi true
(option set 2)	–restrict-pivots false –use-soi true –new-prop true –unconstrained-simp true
(option set 3)	(defaults only)
QF_LIA, QF_IDL (option set 1)	–miplib-trick true –miplib-trick-subs 4 –use-approx true –lemmas-on-replay-failure true –replay-early-close-depth 4 –replay-lemma-reject-cut 128 –replay-reject-cut 512 –unconstrained-simp true –pb-rewrites true –ite-simp true –simp-ite-compress true –use-soi false
(option set 2)	–miplib-trick true –miplib-trick-subs 16 –use-approx true –lemmas-on-replay-failure true –replay-early-close-depth 4 –replay-lemma-reject-cut 16 –replay-reject-cut 64 –unconstrained-simp true –pb-rewrites true –ite-simp true –simp-ite-compress true –use-soi true
(option set 3)	(defaults only)
QF_UF	(defaults only)

Table II: Options used in cvc5 portfolios

B. Scalability and Effectiveness of Guided Randomization

We first report on scalability experiments of SMT-D, both with and without sharing. We also show the effect of adding guided randomization. When using guided randomization, we divide the portfolio into two clusters: a *standard* cluster, which uses default cvc5 randomness settings, and a *noisy* cluster, which assigns the cvc5 `rnd_freq` option to 75%. This option controls how often the SAT decision tries to pick a random variable instead of a heuristically-driven choice. We assign

25% of the workers to the noisy cluster and 75% to the standard cluster.[3] Solvers in the standard cluster import and export clauses of length ≤ 8. In the noisy cluster, clauses of length ≤ 4 are exported, but only unit clauses are imported.

To distinguish the different configurations of SMT-D, we use *CS* for configurations with clause sharing and *CS-GR* for configurations with clause sharing and guided randomization. Fig. 2 shows how different configurations of SMT-D scale with the number of workers. The figure includes results for baseline cvc5, portfolios of 4, 16, and 64 workers, with and without sharing, and a run of 64 workers with guided randomization. Specific numbers for three of the configurations (baseline, 64x CS, and 64x CS-GR) can be found in Table I.

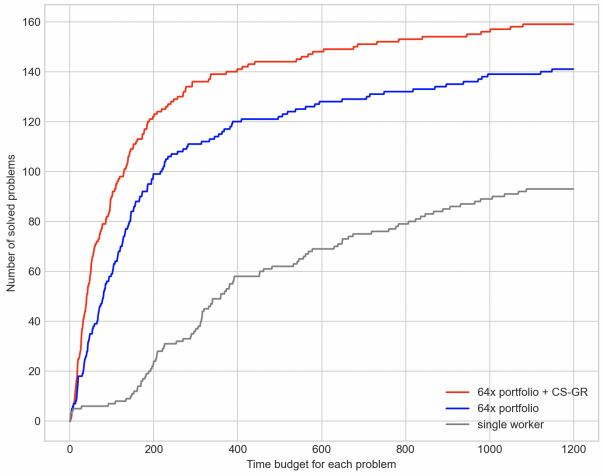
We observe that SMT-D scales nicely when going from 1 to 64 solvers. In addition, clause sharing improves performance for all portfolio sizes greater than four, and guided randomization provides an additional boost. Our experiments showed that guided randomization does not help much until we reach portfolio sizes of more than 32. We suspect additional portfolio members add diversity until a point of diminishing returns where the guided randomization helps. This is why we only include results for CS-GR for a portfolio of 64. A comparison of the 64x CS configuration with and without guided randomization is shown in Fig. 3. While there is orthogonality, overall CS-GR improves performance, including by more than 2x for a significant number of problems (dots to the left of the top "2x" line). As a whole, among all instances solved by both CS and CS-GR, there are 24 instances where CS-GR is more than 2x faster than CS, and only 5 instances where CS-GR is 2x slower. CS-GR solves 5 more problems, and improves the PAR-2 score by 12k (7%) over CS.

Though we did not measure it precisely, total memory consumption per solver is relatively stable, which is good news for the distributed case where cores do not share memory. Broker memory was not a significant issue in these experiments. Detailed studies of memory usage will be an important part of ongoing development of the tool.

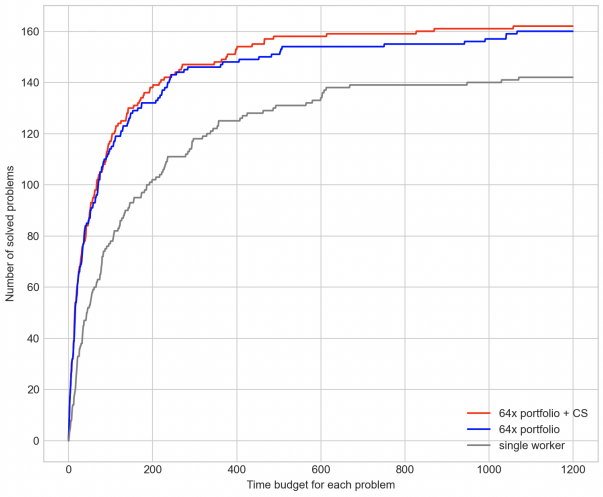
C. Comparison with State-Of-The-Art Tools

We next compare SMT-D with SMTS [16],[4] the strongest solver in quantifier-free divisions of SMTCOMP22's cloud track,[5] and CVC5-P, the partitioning solver from [22].

We use SMTS with sharing on and partitioning off. The reason for not enabling the partitioning capability is simple and deliberate: our goal is to understand and compare only the clause-sharing capabilities of the two frameworks. Results of SMTS with both sharing and partitioning enabled would be inconclusive, as it would be difficult to figure out which technique contributed what. And it would not be an apples-to-apples comparison, as our approach does not yet integrate



(a) SMT-D



(b) SMTS

Figure 4: Comparing SMT-D's and SMTS' improvement over a single base solver.

partitioning. We anticipate integrating clause-sharing with partitioning in future work. In contrast, in our comparison with CVC5-P, we do use the partitioning capabilities of CVC5-P. But this is again deliberate as our goal with that comparison is different, namely to explore how clause sharing compares to partitioning when using the *same* underlying solver (SMTS uses a *different* underlying solver, namely OpenSMT2).

a) Comparison to SMTS*:* It is important to note that on this benchmark set, OpenSMT2, the baseline solver for SMTS, is stronger than cvc5.[6] However, the best configuration of SMT-D (64 CS-GR) improves this situation significantly,

[3]These percentages were chosen based on an empirical analysis of a small sample of possible values. We plan to do a more extensive evaluation of these parameters in the future.

[4]We used commit 29d51340 from the cube-and-conquer branch, as recommended to us by the SMTS authors.

[5]SMT-COMP 2023's cloud track omitted all quantifer-free divisions.

[6]One reason for this is that the benchmarks we are using, from [22], were selected specifically because they are challenging for cvc5.

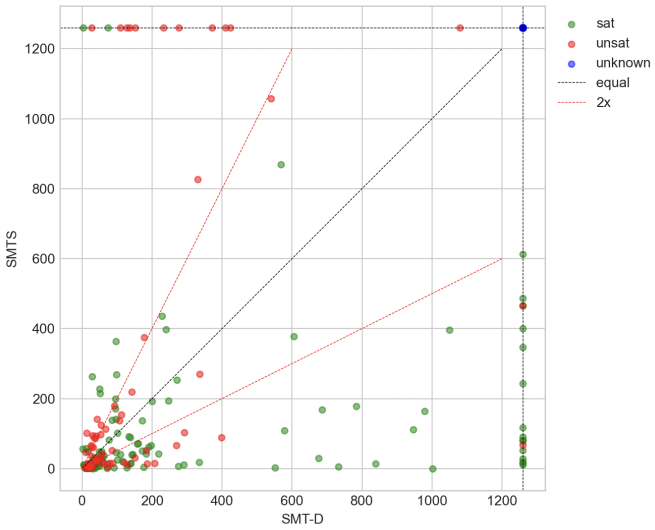


Figure 5: SMT-D 64x CS-GR vs SMTS 64x. Dots on the upper and right-most edges are problems that time out for SMTS and SMT-D, respectively.

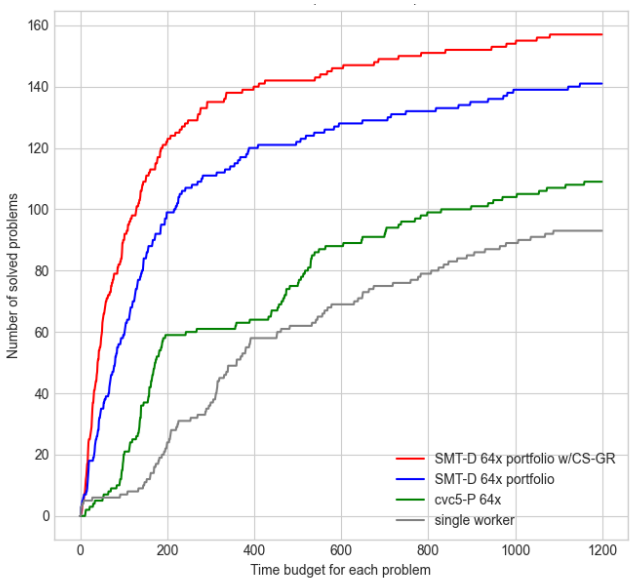


Figure 6: SMT-D and CVC5-P, 64 workers vs 1 worker.

	single worker	8x portfolio + CS
Z3	190	205
SMT-D	219	174

Table III: Comparison between SMT-D and Z3 on 129 benchmarks. Entries show PAR-2 scores in thousands.

as can be seen by the relatively larger gap between the best configuration and the "single worker" configuration in Fig. 4. Table I shows that overall, in terms of benchmarks solved, the best configuration of SMT-D (64 CS-GR) solves almost the same number of problems as the best configuration of SMTS, despite the large difference in their base solvers. Compared to the baseline, the best configuration of SMT-D improves the overall PAR-2 score by 52% (for SMTS, this number is 28%) and solves 66 more problems (compared to 20 more problems solved by SMTS). Moreover, for the 48 QF_IDL benchmarks and for the UNSAT benchmarks as a whole, cvc5 goes from performing worse than SMTS when comparing baselines to performing better when comparing the best version of each.

Although one might hope for even better scaling as the level of parallelism increases, it is important to keep in mind that SMT is a hard problem and is not easily parallelizable. Thus, we don't expect to be able to achieve linear speed-up. Rather, we hope to solve problems beyond the scope of standalone solvers, and indeed, we see that this is the case. In many applications, the number of problems solved in a fixed time matters. We can also see (Figure 5) that SMTS and SMT-D solve a different subset of the benchmarks, so we know further improvement is still possible.

b) Comparison to partitioning cvc5: CVC5-P, the state-of-the-art parallel/distributed implementation of cvc5, uses a combination of portfolios and partitioning strategies. We implemented and ran the hybrid multijob approach of [22] and compared it with SMT-D. Fig. 6 and Table I show that SMT-D is significantly more effective at utilizing 64 copies of cvc5, resulting in a 52% improvement in PAR-2 score (vs 12% improvement by CVC5-P), and in 50 more problems being solved (159 vs 109).

D. Comparison to a Legacy Version of Z3

Z3 was the first SMT solver to implement a portfolio approach with clause sharing. However, this functionality is no longer supported in modern versions of Z3, and the latest release that we could find with this functionality is version 2.15 (Windows-only, from 2009). We attempted a comparison, for completeness, but are only able to draw limited conclusions, for various reasons, including: (*i*) Z3 2.15 runs on a different operating system than our other solvers; (*ii*) it crashes on any configuration with more than eight solvers; and (*iii*) it fails (parsing or execution) on 85 problems in our modern set of 214 benchmarks. When run on the remaining 129 SMT benchmarks, we obtain the results shown in Table III. However, even these results must be taken with a grain of salt, as they show that Z3 performs *worse* when enabling clause sharing, perhaps because of instability of the 2.15 implementation on modern benchmarks. Thus, while this early work in Z3 was important pioneering work, we believe that a fair comparison can only be achieved if the sharing functionality is restored in a modern version of Z3.

VII. Conclusion

SMT-D is a promising advancement in the realm of parallel, portfolio-based SMT solving. Leveraging a hub-and-spoke architecture with a tight CDCL(T) integration, lemma sharing, and guided randomization, SMT-D demonstrates significant improvements in scalability, outperforming not just sequential cvc5, but also pure portfolio (with sharing), and CVC5-P

(portfolio with partitioning). In addition, SMT-D demonstrates more improvement from clause sharing than SMTS and an early version of Z3 and has performance that is overall comparable with and complementary to the state of the art.

While SMT-D demonstrates solid progress in distributed SMT solving, many opportunities for future work remain. These include further parameter tuning, deeper integration with the underlying SAT solver, handling internally-introduced symbols, exploring additional sources of diversity (including using different solvers in the portfolio, such as OPENSMT2 or Z3), exploring additional filtering and redundancy-detection heuristics, and combining our approach with partitioning-based parallelism. In addition, we plan to extend the implementation and evaluation of SMT-D to the full set of logics and benchmarks in SMT-LIB.

REFERENCES

[1] Tomás Balyo, Peter Sanders, and Carsten Sinz. HordeSat: A massively parallel portfolio SAT solver. *CoRR*, abs/1505.03340, 2015.

[2] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *TACAS '22*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[5] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability, Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 33, pages 825–885. IOS Press, February 2021.

[6] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999. Springer Berlin Heidelberg.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. Springer Berlin Heidelberg.

[8] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[9] Google. grpc.io. https://grpc.io/. [Accessed 15-Mar-2023].

[10] A. Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*, 2015.

[11] Youssef Hamadi and Lakhdar Sais, editors. *Handbook of Parallel Constraint Reasoning*. Springer, 2018.

[12] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing*, 2016.

[13] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'11, page 50–65, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving SAT in grids. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 430–435, 2006. Springer Berlin Heidelberg.

[15] Matteo Marescotti, Antti E. J. Hyvärinen, and Natasha Sharygina. Clause sharing and partitioning for cloud-based SMT solving. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis*, pages 428–443, Cham, 2016. Springer International Publishing.

[16] Matteo Marescotti, Antti E. J. Hyvärinen, and Natasha Sharygina. SMTS: distributed, visualized constraint solving. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 534–542. EasyChair, 2018.

[17] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, nov 2006.

[18] SMT-COMP Organizers. SMT-COMP. https://smt-comp.github.io/, 2023.

[19] Dominik Schreiber and Peter Sanders. Scalable SAT solving in the cloud. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 518–534, Cham, 2021. Springer International Publishing.

[20] Roberto Sebastiani and Silvia Tomasi. Optimization modulo theories with linear rational costs. *ACM Transactions on Computational Logic*, 16, 10 2014.

[21] Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is at least 14. In *International Conference on Theory and Applications of Satisfiability Testing*, 2022.

[22] Amalee Wilson, Andres Nötzli, Andrew Reynolds, Byron Cook, Cesare Tinelli, and Clark W. Barrett. Partitioning strategies for distributed SMT solving. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, pages 199–208. IEEE, 2023.

[23] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. A concurrent portfolio approach to SMT solving. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 715–720, 2009. Springer Berlin Heidelberg.

[24] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008.