Massively Parallel Bit-Precise Verification with Bitwuzla and Mallob*

Dominik Schreiber[1], Aina Niemetz[2], and Mathias Preiner[2]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
dominik.schreiber@kit.edu
[2] Stanford University, Stanford, CA, USA
{niemetz, preiner}@cs.stanford.edu

Abstract. We present a distributed platform for massively parallel SMT solving that supports various theories for bit-precise reasoning, with and without quantifiers and push-pop incrementality. Our system is based on an integration of the state-of-the-art SMT solver Bitwuzla into the distributed job scheduling and automated reasoning platform Mallob, which allows Bitwuzla to make heavy use of Mallob's distributed incremental SAT solving engine. Our experimental evaluation shows that this approach outperforms prior SMT parallelization approaches and achieves unprecedented speedups at up to 768 cores.

Keywords: SMT solving · Distributed computing · Verification · Propositional satisfiability

1 Introduction

Satisfiability Modulo Theories (SMT) solvers are integrated as the back-end reasoning engines in a wide range of applications of computer-aided verification, both in academia and industry. For many of these applications, bit-precise reasoning in the theory of fixed-size bit-vectors is a key requirement.

The dominant, state-of-the-art approach for solving bit-vector formulas in SMT is a technique called *bit-blasting* [23], an eager reduction of bit-vector constraints to propositional satisfiability (SAT). Bit-blasting is usually combined with aggressive simplifications of the input constraints prior to the actual reduction step. This technique is surprisingly efficient in practice, mainly due to the fact that state-of-the-art SAT solvers are able to efficiently deal with complex formulas over millions of variables. To render interrelated calls to the SAT solver more efficient, SMT solvers commonly exploit *incremental SAT solving*, where the SAT solver state is preserved across solving calls [3,13]. In the context of bit-blasting, such interrelated calls occur not only in incremental SMT solving, but also for abstraction-refinement-based techniques such as [32], and when combining the theory of fixed-size bit-vectors with other theories. Regardless, the back-end SAT solver of the procedure remains its main potential bottleneck.

* This work was supported in part by the Stanford Center for Automated Reasoning, the Stanford Center for Blockchain Research, and a gift from Amazon Web Services.

In recent years, parallel and distributed SAT solvers have increasingly gained traction [11, 20, 43]. Some of the best available systems use careful *clause sharing* across SAT solver threads to solve challenging problems hundreds of times faster than their state-of-the-art sequential counterparts [42, 44]. In particular, the state-of-the-art distributed SAT solving platform Mallob [44] allows to solve many SAT tasks in parallel and on demand, balances the available computational resources among them in a flexible fashion [37], and efficiently handles *incremental* SAT solving queries at a distributed scale for individual tasks [40,41]. Still, as of yet, we are unaware of any published works that exploit this power for SMT.

In this work, we specifically focus on accelerating the state-of-the-art SMT solver Bitwuzla [29], which supports reasoning over the (quantified and quantifier-free) theories of fixed-size bit-vectors and floating-point arithmetic, in combination with arrays and uninterpreted functions. The architecture of Bitwuzla is based on the counter-example guided abstraction refinement (CEGAR) paradigm [10]. Specifically, Bitwuzla implements the lazy SMT technique *lemmas on demand* [6, 28], but with a bit-blasting bit-vector solver (instead of a propositional abstraction) at its core. At the back end, the SAT solver is integrated in an offline fashion. This is in contrast to the classic CDCL($\mathcal{T}$) framework [15,33], which refines a propositional abstraction of the input via low-level, fine-grained interactions between the SAT solver and the theory solvers.

Contribution. We present an integration of the bit-blasting SMT solver Bitwuzla with distributed, on-demand incremental SAT solving in Mallob. Our work takes advantage of recent advances in parallel and distributed SAT solving in the context of SMT solving based on bit-blasting. This yields a scalable, parallel and distributed SMT solver for bit-precise reasoning.

Bitwuzla is integrated as a new application engine of the Mallob system and submits incremental SAT solving tasks to the distributed scheduler. This not only combines the advantages of bit-precise reasoning as implemented in Bitwuzla and parallel and distributed SAT solving as provided by Mallob, but offers remarkable synergies. First, we exploit the fact that almost all of Bitwuzla's non-trivial reasoning, and thus the vast majority of its solving time, is spent in plain incremental SAT solving, which is much easier to parallelize than the fine-grained interactions loop of CDCL($\mathcal{T}$). Second, using a flexible on-demand scheduler allows to maintain several distinct SAT solving tasks (a common pattern when spawning sub-solvers in quantified reasoning) and to shift the distributed resources to the task that has use for them. Third, the use of *incremental* SAT solving enables highly efficient and low-latency SAT queries while keeping distributed communication at a minimum. Finally, in contrast to more theory-specific parallelization efforts targeting the bit-vector theory [36, 49], replacing the bit-blasting solver's back-end SAT solver with a parallel and distributed SAT solver is theory agnostic. Parallelization on the SAT level benefits every supported theory (and combination) while retaining the full expressive power and versatility of Bitwuzla.

We evaluate our system, which we refer to as BITWUZLLOB, on up to 768 cores (16 compute nodes) and show that it outperforms and outscales prior related

parallel SMT approaches significantly. Our results suggest that BITWUZLLOB promises to serve as a solid base for future, theory-specific parallelization efforts in the context of bit-precise reasoning.

Related Work. State-of-the-art parallel and distributed SMT solving strategies can largely be divided into *portfolio* solving and *partitioning*. Portfolio solving configures multiple different solvers (or configurations of a solver) that attempt to solve the same SMT problem in parallel. Partitioning is a divide-and-conquer strategy where a problem is partitioned into independent subproblems that are solved in parallel. Some parallel SMT techniques are based on partitioning only [22,48], but the majority relies (at least partly) on a portfolio strategy, in combination with information sharing and partitioning [5,21,25,26,46]. Most of these approaches are in principle theory agnostic, but have not been evaluated in the context of incremental SMT solving and bit-precise reasoning. Furthermore, none of them exploit parallel and distributed SAT solving techniques.

Parallelization techniques for the theory of fixed-size bit-vectors have received little attention so far. Reisenberger [36] explored a cube-and-conquer-style partitioning technique, implemented as a prototype on top of an outdated version of Boolector [31], with limited success. STP-Parti-Bitwuzla [49], a recent participant in the parallel bit-vector track of SMT-COMP 2025 [1], implements a partitioning strategy similar to [48] on top of Bitwuzla. Recently, the shared-memory parallel SAT solver Gimsatul [16] was integrated as a SAT backend of Bitwuzla [30]. Gimsatul, however, does not support incremental solving, and is thus reinitialized for every SAT query.

2 Parallel and Distributed SAT Solving with Mallob

Consider a *propositional* formula $\phi = \bigwedge_{i=1}^{n} C_i$ in conjunctive normal form (CNF) where clauses $C_i = \bigvee_{j=1}^{k_i} l_{ij}$ are disjunctions of literals and each literal l_{ij} is a Boolean variable or its negation. The problem of *propositional satisfiability* (SAT) is to decide whether there exists a variable assignment that satisfies ϕ. Today's most efficient SAT solvers are based on Conflict-Driven Clause Learning (CDCL) [27], which involves a search over the space of partial variable assignments while bookkeeping *conflict clauses* derived from logical conflicts.

Many applications of SAT solvers exploit a mode of operation called *incremental SAT solving* [3, 13] where, once a solver returns a satisfiability result for some formula ϕ, the user can specify additional clauses C and query the same solver instance on the resulting formula $\phi' := \phi \cup C$. Furthermore, each satisfiability query can be supplied with a number of *assumption literals*, which are enforced to hold for this query *only*. Incremental SAT solving offers efficient interaction schemes on interrelated and evolving problems since it allows the solver to preserve its knowledge base across incremental solving calls.

In parallel and distributed SAT solving, a common approach is to run a sequential SAT solver thread on each available core and let the solver threads *cooperate* by means of exchanging useful *conflict clauses* they find during their

search [2,8,19,38,45,47]. Recent insights indicate that this *clause-sharing* strategy [44] can lead to good scalability up to hundreds, and even thousands of cores, even if all solver threads operate on the same, original formula and are configured and initialized in (almost) the exact same way [42].

Mallob is a distributed job scheduling and automated reasoning platform [44] that allows to schedule, balance, and solve many SAT (and other) tasks at once, in an on-demand fashion. The amount of computational resources allotted to a certain task can increase or decrease *during the task's execution* via so-called *malleable job scheduling*, subject to the overall system state (i.e., the number and demands of active jobs vs. the amount of available computational resources) [37]. The state-of-the-art parallel and distributed SAT solving engine MallobSat is tightly integrated in Mallob and offers full support for incremental SAT solving [40]. Recently, Mallob's combination of distributed incremental SAT solving with flexible multi-tasking was exploited for distributed MaxSAT solving [41].

3 Bit-Precise SMT Solving in Bitwuzla

Satisfiability Modulo Theories (SMT) is the problem of determining whether a *first-order* formula is satisfiable with respect to some background *theories* (for an introduction to SMT, see, e.g., [7]). Prominent examples of such background theories are the theories of fixed-size bit-vectors, arrays, integer arithmetic, real arithmetic and strings. In the following, we consider background theories as defined in SMT-LIB [4] and assume the usual notions and terminology of many-sorted first-order logic with equality (see, e.g., [14,24]).

Bitwuzla [29] is a state-of-the-art SMT solver specialized for bit-precise reasoning. It supports the theories of fixed-size bit-vectors and floating-point arithmetic, in combination with arrays and uninterpreted functions, with and without quantifiers. Bitwuzla implements an abstraction-refinement-based SMT paradigm called *lemmas on demand* [6,28], but with a bit-vector abstraction (and thus a bit-vector solver) instead of a propositional abstraction at its core. Atoms and terms that do not belong to the bit-vector theory are abstracted as uninterpreted Boolean and bit-vector constants, respectively. These abstracted terms are then handled by the corresponding theory solvers. The bit-vector abstraction is bit-blasted to propositional logic, and deciding its satisfiability is delegated to the back-end SAT solver. The general architecture of Bitwuzla is shown in Figure 1.

Given a formula ϕ, prior to solving, Bitwuzla aggressively applies simplification techniques as a preprocessing step. In the refinement loop of the lemmas on demand architecture, given the preprocessed formula ϕ' and the current set of refinement lemmas $\mathcal{L}$, the bit-vector solver is responsible for deciding the satisfiability of the bit-vector abstraction $\mathcal{A}(\phi' \cup \mathcal{L})$ (initially, $\mathcal{L} = \emptyset$). If the bit-vector solver determines that the abstraction is unsatisfiable, it concludes with *unsat* (since $\mathcal{A}$ is an over-approximation of the original problem ϕ). Otherwise, it produces a model for $\mathcal{A}$, which must be checked for consistency with respect to theory axioms by each theory solver. If a theory solvers determines that the model is inconsistent, it generates and adds a theory lemma to $\mathcal{L}$, thus refin-

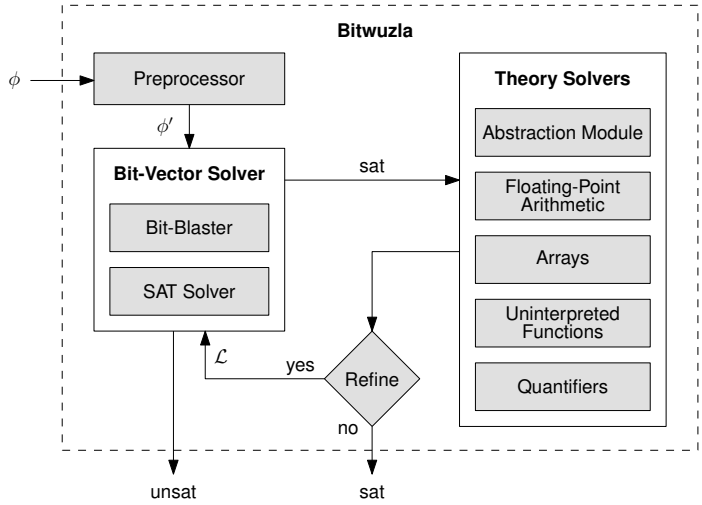

Fig. 1. Bitwuzla solver architecture.

ing $\mathcal{A}$ and ruling out the spurious model. This loop incrementally refines $\mathcal{A}$ until either $\mathcal{A}$ is determined to be unsatisfiable, or all theory solvers agree that the current model is consistent with their theory axioms, and thus, ϕ is *sat*.

Note that for floating-point arithmetic, Bitwuzla implements a technique called *word-blasting*, which translates floating-point terms to bit-vectors terms, thus heavily relying on the bit-vector solver. A more detailed introduction to Bitwuzla's architecture and theory solvers can be found in [29].

The main bit-vector solving procedure in Bitwuzla implements a CEGAR-based abstraction-refinement approach for bit-vector arithmetic based on bit-blasting [32]. This technique introduces and iteratively refines abstractions for arithmetic operations $\{\cdot, \div, \mathrm{mod}\}$, which are expensive for bit-blasting for large bit-widths due to the complexity of their bit-level circuit representation. It seamlessly integrates into Bitwuzla's lemmas on demand architecture.

Bitwuzla's bit-vector solver is used incrementally on different levels. On the user-facing side, Bitwuzla provides incremental solving capabilities via push-pop incrementality and solving under assumptions. But even for non-incremental SMT queries, Bitwuzla's bit-vector solver is used incrementally within the lemmas on demand loop. Consequently, Bitwuzla's bit-blasting solver heavily relies on the incremental solving capabilities of its default SAT backend CaDiCaL [9].

4 Integrating Bitwuzla with Mallob

We now describe our parallel and distributed SMT solver, which we refer to as Bitwuzllob, as an integration of Bitwuzla into the Mallob platform.

Figure 2 provides a general overview of our system. At the top level, the system can be used as a plain SMT solver similar to unmodified Bitwuzla: it

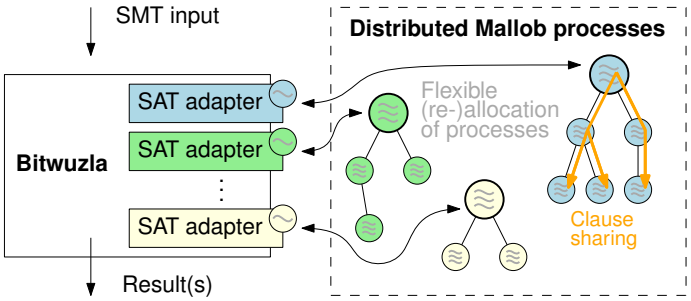

Fig. 2. Overview of our system's architecture (inspired by [41]). While only one SMT interaction is displayed, a single execution of the system can also handle several such interactions concurrently while balancing the available distributed resources among them. Each "∼" illustrates an incremental SAT solver thread.

processes user input in the SMT-LIB format and provides the expected output, e.g., the model of a satisfiable query after a `(get-model)` command. At the back end, however, we reworked how Bitwuzla performs SAT solving.

Depending on the input problem, Bitwuzla's execution involves either one single or several co-existing SAT solver instances. Since Bitwuzla is purely sequential, *at most one* of these instances is *active* at any point in time. In BitwuzLLOB, each of Bitwuzla's SAT solver instances now corresponds to a more complex *SAT adapter* object in the Mallob system instead of a plain CaDiCaL instance. We describe in Section 4.2 how we interfaced the two systems.

Each SAT adapter processes incoming SAT solving calls in two orthogonal ways in parallel. First, a *latency hiding head thread* runs a sequential CaDiCaL instance to intercept trivial SAT calls, as explained in more detail in Section 4.3. Second, a distributed incremental SAT solving task is deployed in Mallob to conquer non-trivial SAT calls. There is a 1:1 correspondence between SAT adapters and distributed incremental SAT solving tasks. However, a distributed task may be delayed in its deployment (if all SAT solving calls thus far are trivial, see Section 4.3) and a deployed, yet inactive distributed task may be terminated if too many inactive tasks are present in the system. Terminated tasks are potentially re-deployed at a later point (see Section 4.4 for further details).

4.1 User Interface

Launching BitwuzLLOB is done in the same way as launching Mallob. Since Mallob is based on the Message Passing Interface (MPI), an MPI wrapper such as `mpirun` or `mpiexec` is used to run $p \geq 1$ interconnected incarnations of the `mallob` executable, which can be distributed across $1 \leq m \leq p$ physical machines. The mapping of the p MPI processes to machines and cores can be configured via additional arguments to the MPI wrapper. For $m = p = 1$ (single-node, single-process), `mallob` can be executed directly without using an MPI wrapper.

```
$ mpirun -np 4 build/mallob -q -t=48 &  # Launch Mallob/Bitwuzllob
[1] 129857
$ mkfifo smt-output.pipe  # Create named pipe for output
$ cat job.json  # Show JSON submission file
{
    "user": "myname",
    "name": "job1",
    "application": "SMT",
    "files": ["smt-input.smt2"],
    "priority": 1.0,
    "configuration": {"smt-out-file": "smt-output.pipe"}
}
$ cp job.json .api/jobs.0/in/  # Submit SMT job
$ cat smt-output.pipe  # Fetch SMT solver output
sat
(
  (define-fun s () (_ BitVec 512) #b00000000...00000110)
  (define-fun t () (_ BitVec 512) #b00000000...00000000)
)
$ pkill -2 mpirun ; rm smt-output.pipe  # Clean up
```

Fig. 3. Example user interaction with BITWUZLLOB in its *on-demand* mode of operation. After launching BITWUZLLOB as a service (with four processes à 48 threads), an SMT solving task is submitted by copying a basic JSON file describing the task to a watched API directory. The output is then read from the specified output file (which, in this case, was set to a *named pipe* special file for direct inter-process communication). Note that the SMT input (specified as `smt-input.smt2` here) could also be set to a named pipe to feed SMT commands to BITWUZLLOB interactively.

Just like Mallob [39, 43], BITWUZLLOB supports two modes of operation: **mono** and **on-demand**. In the *mono* mode of operation, a path to a single SMT instance is given via the command-line (`-mono-app=SMT -mono=<file>`) when launching the system. BITWUZLLOB solves this instance and then terminates. In the *on-demand* mode, the system is launched without providing such a single instance to solve. Instead, a configurable subset of the system's processes, called **client processes**, *listen* for incoming SMT tasks, which users can submit to the system at any time over a filesystem-based JSON interface. Whenever a client receives an SMT task, it locally launches a procedure as shown in Figure 2 (left). Figure 3 illustrates an example for launching and using BITWUZLLOB in its *on-demand* mode to solve an SMT task. Internally, the *mono* mode is in fact only a special case of the *on-demand* mode, where exactly one client process is initialized and programmatically receives the singular input instance to process.

The distributed computational resources are balanced dynamically among all distributed SAT tasks spawned by the active SMT task(s) (cf. [37, 41]). If desired, the output of the internal Bitwuzla procedure of an SMT task can be directed to an output file by the client process that hosts the SMT task (specified via command-line option `-smt-out-file` in the *mono* setting, and JSON option

`"smt-out-file"` in the *on-demand* setting). For incremental and interactive SMT solving use cases, the SMT input file and/or output file are initialized as *named pipes*, which allows interacting with BITWUZLLOB by successively reading from and/or writing to these pipes. SMT solving can be configured globally via the command-line option `-bitwuzla-args`, or locally (i.e., specific to a certain SMT task) via statements in the SMT-LIB input.

4.2 Interfacing Bitwuzla and Mallob

In the following, we describe the interface we established between the Bitwuzla and Mallob systems, as well as the underlying rationale.

Exploiting Mallob for an application of incremental SAT solving is possible via (at least) two avenues. The first option is to *connect* the application to a running client process via Mallob's provided *bridge interface* [40] or similar interfacing code. Internally, the application uses a file system to submit tasks to a Mallob process and receive corresponding feedback. The application (Bitwuzla) and the distributed program (Mallob) thus remain as separate systems.

The second option is to *integrate* the application into Mallob itself (cf. [41]). This requires adding application code to the Mallob project, to be executed whenever a job for the target application arrives. This option allows the application to deploy sub-tasks (such as incremental SAT solving tasks) to Mallob *programmatically* and thus much more efficiently than with the first option.

For the sake of efficiency and ease of use, we decided on the second option and implemented BITWUZLLOB as a stand-alone distributed SMT solver by integrating Bitwuzla into Mallob as a new application engine. For this purpose, we extended the public API of Bitwuzla to allow the injection of an external SAT solver factory for constructing SAT solver instances. We further added an application handler for SMT to Mallob, which executes Bitwuzla on the input problem in a dedicated thread within the hosting client process. Instead of using Bitwuzla's internal SAT solver factory to create SAT solver instances, Mallob now acts as the SAT solver factory, constructing instances of our SAT adapter.

4.3 Latency Reduction

Bitwuzla commonly produces incremental SAT queries that feature a large number (potentially thousands) of solving calls, many of which are trivial. In practice, low latencies for individual, trivial SAT solving calls are as crucial for performance as high parallel speedups for challenging ones. Mallob has been tuned to reduce latencies for trivial SAT calls before [40, 41], but we observed that latencies of around 5-10 ms per call were still common.

In order to reduce latencies even further, we therefore introduce a so-called *latency-hiding head thread* within the host process. In addition to the distributed SAT job dispatch, we locally run a single sequential SAT solver thread that attempts to solve the same problems as the distributed solver in the background. As such, we have two independent SAT solver actors, a local one and a distributed one, for each incremental stream of SAT calls. If one of these actors

succeeds in solving incremental revisions 0 through k while the other actor is still occupied with revision $j < k$, the actor is interrupted. Subsequently, the interrupted actor receives revisions $j, j+1, \ldots, k$ as a single *contracted increment* that features the clauses of revision j to k and the assumption literals of k.

As a result, if an SMT input yields thousands of trivial SAT calls followed by a single challenging call, the distributed solver receives only one single, challenging increment. We even take this a step further by deferring the initialization of the distributed incremental SAT task alltogether, until an increment that cannot be solved by the latency-hiding head thread within a few milliseconds is reached.

In order to account for the high number of incremental revisions arising from some Bitwuzla runs, we also refactored Mallob's inter-process communication. In prior versions of Mallob, each MPI process transfers each formula increment to its dedicated SAT solving sub-process by means of *shared memory*: the MPI process allocates an appropriately sized shared memory segment, populates it with the clauses of the next increment, and then notifies the SAT sub-process that the segment can now be read. This eventually results in a large number of concurrently open shared-memory segments and corresponding (virtual) files in `/dev/shm/` that require significant amounts of RAM, which may become detrimental to performance. Instead, Mallob now transfers formula information in a buffered manner over a single, fixed-size shared memory segment. Double buffering and dedicated I/O threads are utilized to transfer data as fast as possible.

4.4 Limiting Concurrent Distributed Tasks

On SMT input problems with quantifiers, in each iteration of the lemmas on demand loop, Bitwuzla creates an internal sub-solver instance for performing checks on quantified formulas arising from model-based quantifier instantiation (MBQI) [17]. The number of incremental sub-solver checks depends on the number of active quantified formulas, and while performing an MBQI check for a quantified formula, a sub-solver may create nested sub-solver instances if the checked quantified formula contains nested quantified formulas. As a consequence, the quantifiers module may create a large hierarchy of sub-solvers. Each sub-solver is a fully functional (internal) Bitwuzla instance with a fresh SAT solver, potentially resulting in a large number of SAT solver instances.

While at most one SAT solver instance is active at any point in time, each suspended distributed solving task in Mallob *exclusively* blocks one MPI process, until the task is terminated. As such, in a naïve implementation, an execution with p Mallob processes will run into issues once Bitwuzla has created and used, but not destroyed, SAT adapters $T_1, \ldots, T_p$, and then creates and uses another SAT adapter T_{p+1}. While the latency-hiding solver thread of T_{p+1} still progresses, the corresponding distributed task's deployment is stalled indefinitely since no process is available to deploy T_{p+1} to.

We mitigate this issue by limiting the number of deployed distributed tasks per SMT solving task to a user-defined parameter $s \leq p$. If some SAT adapter is about to deploy a distributed task that would exceed this limit, another adapter's inactive distributed task is *evicted*, i.e., terminated and cleaned up. In principle,

any inactive task can be evicted since it can be re-deployed at a later point if and when the corresponding adapter receives another SAT call. In practice, choosing which task is best to evict can be non-trivial. Aspects to consider include the size of a task's formula (large problems take longer to re-deploy), the time of the task's creation (old solver instances might lose relevance), and the task's total solving time until now (significant knowledge may be lost when evicting a long running task). Considering Bitwuzla's approach to spawning sub-solvers, where the oldest SAT task may often correspond to Bitwuzla's main solver and is thus likely to be reused, we decided on an eviction strategy that picks the task with the *lowest total runtime* so far. We recommend to set $s \leq p/T$, where T is the maximum anticipated number of parallel SMT solving tasks, so that at least s processes per SMT solving task are in fact available. In our experiments, where only one SMT task is active at a time, we chose a conservative limit of $s = p/2$, thus ensuring that an active SAT task can exploit at least half of all processes.

5 Evaluation

We evaluate Bitwuzllob in terms of SMT solving performance and scaling behavior on non-incremental and incremental benchmarks of all supported logics of the SMT-LIB benchmark library [34,35]. Runs in distributed environments are costly and resource intensive, which requires a responsible use of computational resources in our experimental design. As in previous work on distributed SAT and MaxSAT solving [40, 41, 44], we therefore limit the wallclock runtime per instance to 300 s. This time limit is justified not only by saving resources but also by the observation that investing large amounts of resources should be traded off with much lower wallclock runtimes (when considering the same inputs).

Further, the set of benchmarks in supported logics contains 156,307 non-incremental and 25,464 incremental instances. Since considering the whole set is not feasible for our experimental evaluation, we selected a modestly sized but diverse and representative benchmark set as follows. We first ran sequential Bitwuzla on all non-incremental and incremental benchmarks with a time limit of 1200 seconds and a memory limit of 8GB. For each logic, we then grouped all benchmarks into six sets S_1–S_5 and S_u based on the observed runtime. Sets S_1–S_5 contain all *solved* benchmarks that were solved in more than n and at most m seconds, whereas S_u contains all *unsolved* benchmarks. We used the following runtime distribution for these sets: $(n, m) \in \{(5, 10), (10, 100), (100, 300), (300, 600), (600, 1200)\}$. As a final step, we randomly sampled benchmarks from each group with a distribution of 10/10/25/25/35/50 for $S_1/S_2/S_3/S_4/S_5/S_u$. In total, we selected 1098 (96/134/180/117/116/455) non-incremental and 288 (60/64/63/28/16/57) incremental benchmarks. Selecting benchmarks that were solved quickly by the sequential version allows us to measure the overhead of our distributed setup on easy instances, while more difficult benchmarks enable us to measure and analyze its scaling behavior. Overall, we selected a larger number of harder benchmarks, since our approach is more likely to be applied to hard instances.

We compare our system against several existing approaches. As a most natural comparison, we run plain, sequential Bitwuzla, i.e., with its default CaDiCaL backend. In addition, we test the three most relevant competitors in the parallel track of the 2025 SMT competition: (1) Bitwuzla+Gimsatul (*Gims+Bzla*), a version of Bitwuzla with the shared-memory parallel SAT solver Gimsatul [16] instead of CaDiCaL as its SAT backend [30]; (2) STP-Parti-Bitwuzla (*STP-P-Bzla*), a recent partitioning-based parallel SMT solver [48] that uses Bitwuzla (with CaDiCaL) as its sequential SMT backend and participated in the non-incremental `QF_BV` logic [49]; and (3) Yices2 (*Yices*), a parallel portfolio of Yices2 configurations that participated in the `QF_ABV`, `QF_AUFBV`, `QF_BV`, and `QF_UFBV` logics (among other logics not supported by Bitwuzla) [12].

We ran all experiments on the cluster SuperMUC-NG, where each node features a two-socket Intel Skylake Xeon Platinum 8174 processor clocked at 2.7 GHz with a total of 48 physical cores (96 hardware threads) and 96 GB of main memory. Nodes are interconnected via Intel OmniPath and run Linux SLES.[3]

5.1 Solving Performance

In our first experiment, we compare Bitwuzllob to existing approaches on commonly supported logics. Figure 4 (left) shows the results of all considered systems on the commonly supported non-incremental `QF_BV` logic on a single node with 48 cores. We ran this experiment on a single node since the competing approaches are not distributed. The explicit search space partitioning approach of STP-Parti-Bitwuzla results in a clear and consistent improvement over sequential Bitwuzla, but is outperformed by Bitwuzla with the parallel SAT solver Gimsatul as a backend. Bitwuzllob performs similar to Gims+Bzla at lower runtimes ($< 20\,\text{s}$) and significantly outperforms all other approaches for higher runtimes. The comparatively low number of instances solved by Yices can be attributed to the fact that its parallel portfolio consists of one default configuration for each logic, complemented with additional, orthogonal MCSAT-based [18] configurations. These MCSAT-based configurations are highly efficient on some fragments of `QF_BV` and less efficient on others, and our benchmark selection for this logic seems to be less favorable for MCSAT-based approaches.

Figure 4 (right) complements our comparison on `QF_BV` with a comparison of sequential Bitwuzla with the systems that support the entire set of selected benchmarks. Interestingly, Bitwuzla's parallel Gimsatul backend seems ineffective if evaluated over a combination of theories as it is outperformed by Bitwuzla with the sequential CaDiCaL backend by a significant margin. This is likely due to the fact that Gimsatul does not support incremental solving, and thus, Bitwuzla creates a fresh Gimsatul instance for each incremental SAT call. As a consequence, the performance of Gims+Bzla suffers on incremental (SAT) workloads, which are higher on instances that involve arrays, uninterpreted functions, and quantifiers, even on the non-incremental benchmarks. Bitwuzllob, in contrast, manages to drastically outperform sequential Bitwuzla.

[3] https://doku.lrz.de/supermuc-ng-10745965.html

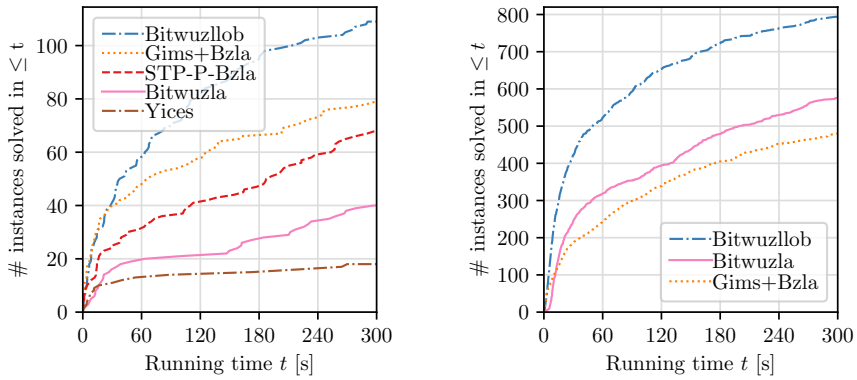

Fig. 4. Performance on non-incremental `QF_BV` instances (left) and the entire benchmark set (right) on a single node with 48 cores, with Bitwuzla as the sequential baseline.

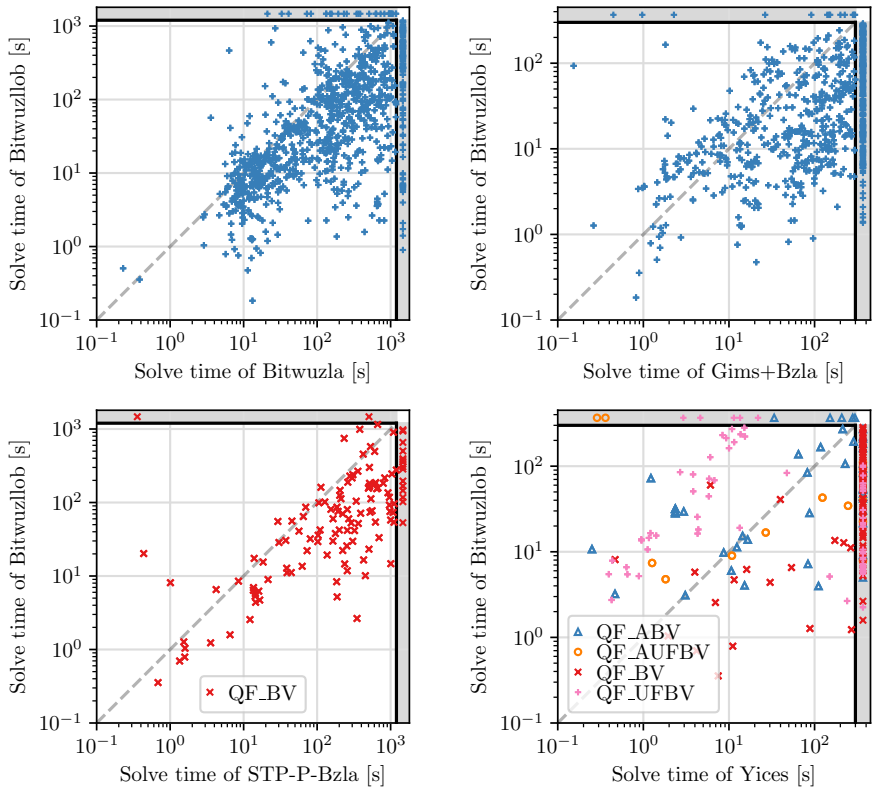

Fig. 5. Bitwuzllob versus Bitwuzla (top left), Gims+Bzla (top right), STP-Parti-Bzla (bottom left), and Yices (bottom right), on 48 cores, showing all instances from commonly supported logics. Left (right) plots use a time limit of 1200 s (300 s).

Figure 5 shows a per-instance comparison of our approach on a single node (48 cores) with the individual considered systems on the benchmarks of commonly supported logics. While we generally imposed a time limit of 300 s per instance, we also conducted additional runs with an increased time limit of 1200 s for a comparison of single-node BITWUZLLOB with sequential Bitwuzla and STP-Parti-Bitwuzla (the winner in logic `QF_BV` of the parallel track of SMT-COMP 2025) for a more complete picture, given in the left column of Figure 5. Overall, BITWUZLLOB clearly outperforms all other approaches. There are a few instances on which BITWUZLLOB seems to struggle that other parallel approaches solve quickly, indicated by points at the top-left corner of plots. On these instances, sequential Bitwuzla has a similarly bad performance behavior, which is also suggested by the comparison plot on the top left of Figure 5, as it does not show this behavior. We expect that improving the performance of the sequential solver on these instances will also improve the performance of BITWUZLLOB.

Tables 1 and 2 summarize the results of all considered systems with a time limit of 300 s on a single node with 48 cores in terms of solved instances, solved individual queries (for the case of incremental SMT instances), and Penalized Average Runtime (PAR-2), which penalizes each timeout with twice the time limit (i.e., 600 s). Logics not supported by a solver are indicated with a '–', and best results for each logic and metric are highlighted.

5.2 Scaling Performance

In our second experiment, we assess the scaling performance of our system. Figure 6 (left) provides a general overview for runs scaling from 4–768 cores on 1–16 nodes. Even at the lowest scale (1x4), our approach significantly outperforms sequential Bitwuzla.[4] We also observe consistent scaling-up to the highest scale (16x48), with clear diminishing returns beyond four nodes (192 cores).

Table 3 complements these findings with detailed *speedups* achieved over sequential Bitwuzla grouped by logic. Given a single SMT instance solved by the sequential baseline as well as the parallel run, the parallel speedup is the ratio between the sequential and the parallel (wallclock) runtime. We aggregate these speedup measures, using the geometric mean, while only considering instances of a certain minimum difficulty. For this, we first consider all *non-trivial* instances, i.e., instances that sequential Bitwuzla solved in ≥ 1 s, and then narrow this set down to the more *challenging* instances, i.e., instances solved by Bitwuzla in ≥ 60 s. Across the majority of logics, our system scales consistently when increasing the available computational resources, and also scales better for more challenging instances. The non-incremental logics that benefit the most from our parallelization are `QF_BV`, `FP`, and `QF_BVFPLRA`. Our system generally achieves lower speedups for incremental instances, albeit still very respectable in some cases (e.g., a speedup of 25.4 at 16 nodes for `QF_BVFP`).

[4] Note that comparing a run of our system with < 48 solver threads to a run on ≥ 1 entire nodes is not entirely fair because the run with fewer threads has access to relatively more resources (RAM, cache, hardware-threads for background tasks).

Table 1. Performance on *non-incremental* benchmarks in terms of number of solved instances (#) and Penalized Average Runtime (PAR-2) with a time limit of 300 s on a single node with 48 cores. The best result for each logic and metric is highlighted.

		Bitwuzla		Gim+B		B'b 1x48		B'b 4x48		B'b 16x48		Yices		STPPB	
#	Logic	#	PAR-2	#	PAR-2	#	PAR-2	#	PAR-2	#	PAR-2	#	PAR-2	#	PAR-2
80	QF_ABV	38	361.2	23	467.7	42	317.4	46	287.8	**47**	**280.2**	30	404.2	–	–
26	QF_ABVFP	21	141.9	21	147.5	23	84.1	**24**	66.8	**24**	**61.4**	–	–	–	–
1	QF_ABVFPLRA	**1**	85.9	**1**	49.4	**1**	17.5	**1**	**13.9**	**1**	**13.9**	–	–	–	–
23	QF_AUFBV	13	298.1	8	420.1	13	276.4	**14**	267.7	**14**	**259.7**	8	409.2	–	–
155	QF_BV	40	474.8	79	332.2	109	233.5	118	191.9	**122**	**169.3**	18	538.1	63	398.8
23	QF_BVFP	22	73.1	22	36.6	22	36.0	**23**	18.0	**23**	**14.1**	–	–	–	–
12	QF_BVFPLRA	8	248.8	**12**	75.8	**12**	11.4	**12**	9.0	**12**	**6.8**	–	–	–	–
73	QF_FP	38	348.0	26	409.9	51	217.5	**55**	189.9	**55**	**179.1**	–	–	–	–
10	QF_FPLRA	9	127.1	4	434.7	**10**	20.7	**10**	11.9	**10**	**11.3**	–	–	–	–
105	QF_UFBV	52	356.2	27	465.2	70	233.9	**76**	208.8	75	**201.6**	39	383.4	–	–
51	ABV	0	600.0	**6**	**529.4**	0	600.0	0	600.0	0	600.0	–	–	–	–
4	ABVFP	**1**	**455.0**	0	600.0	**1**	465.5	**1**	465.5	**1**	465.5	–	–	–	–
151	AUFBV	**51**	429.6	28	506.3	48	435.1	47	435.6	**51**	**424.6**	–	–	–	–
31	AUFBVFP	10	421.1	0	600.0	11	401.1	11	409.3	**14**	**368.1**	–	–	–	–
109	BV	43	398.6	36	427.9	50	359.8	**56**	**319.7**	54	324.2	–	–	–	–
20	BVFP	3	513.0	0	600.0	11	362.6	**13**	**323.3**	10	387.8	–	–	–	–
10	BVFPLRA	2	487.2	1	540.0	**3**	**465.1**	**3**	465.4	**3**	**465.1**	–	–	–	–
153	FP	42	465.4	74	352.4	**102**	**241.2**	91	268.7	94	254.9	–	–	–	–
4	FPLRA	**0**	**600.0**	**0**	**600.0**	**0**	**600.0**	**0**	**600.0**	**0**	**600.0**	–	–	–	–
56	UFBV	10	497.7	8	526.4	**11**	484.7	**11**	484.2	**11**	**483.3**	–	–	–	–
1	UFBVFP	0	600.0	0	600.0	**1**	277.9	**1**	**156.1**	0	600.0	–	–	–	–

Table 2. Performance on *incremental* benchmarks in terms of number of solved *check-sat* queries (q) and Penalized Average Runtime (PAR-2) with a time limit of 300 s on a single node with 48 cores. The best result for each logic and metric is highlighted.

		Bitwuzla		Gim+B		B'b 1x48		B'b 4x48		B'b 16x48	
q	# Logic	q	PAR-2	q	PAR-2	q	PAR-2	q	PAR-2	q	PAR-2
1017	79 QF_ABV	839	385.5	764	528.0	**944**	**260.4**	936	302.8	937	317.1
1040	45 QF_ABVFP	1014	321.1	856	493.1	**1016**	287.4	**1016**	**286.4**	925	297.5
1707	3 QF_ABVFPLRA	**1707**	29.6	583	600.0	**1707**	24.5	**1707**	25.1	**1707**	**24.4**
898	16 QF_AUFBV	887	248.9	834	481.3	889	234.5	893	182.0	**897**	**132.8**
14592	66 QF_BV	5819	291.4	2926	296.8	9700	226.2	9972	221.2	**9998**	**220.6**
53	31 QF_BVFP	36	224.2	39	210.5	**42**	97.3	**42**	89.8	**42**	**88.9**
26252	8 QF_BVFPLRA	**26252**	**17.9**	10487	487.2	**26252**	20.2	**26252**	20.2	**26252**	20.3
818	13 QF_UFBV	815	121.6	586	217.1	**818**	63.7	**818**	53.2	**818**	**46.3**
2108	2 ABVFPLRA	**2108**	**71.6**	450	600.0	**2108**	74.8	**2108**	73.3	**2108**	73.2
38856	18 BV	**35070**	**233.3**	23174	555.7	34931	273.4	34539	299.7	34678	275.2
458	1 BVFP	**458**	12.9	354	600.0	**458**	12.7	**458**	12.9	**458**	**12.5**
4855	6 BVFPLRA	3690	**213.9**	768	600.0	4047	216.1	3821	216.4	**4086**	216.5

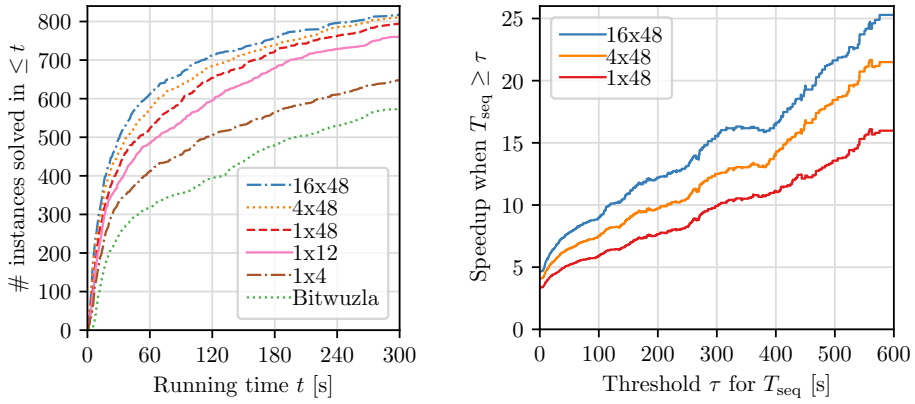

Fig. 6. Left: Scaling of our approach over all instances, with Bitwuzla as a sequential baseline. Right: *Weak scaling* of our approach. Each point (x, y) on a curve indicates that BITWUZLLOB achieved a mean speedup of y when only considering the (commonly solved) instances that took sequential Bitwuzla $\geq x$ seconds to solve.

Next, we analyze BITWUZLLOB's *Weak Scaling* behavior, i.e., how its speedups develop as the input difficulty is increased. While Table 3 shows speedups over instances with a runtime of sequential Bitwuzla of at least $\tau = 1\,\text{s}$ and $\tau = 60\,\text{s}$, Figure 6 (right) shows BITWUZLLOB's mean speedups for *all* runtimes $0 \leq \tau \leq 600\,s$. For instance, when considering the instances solved by Bitwuzla in $\geq 300\,\text{s}$, BITWUZLLOB at 1, 4, and 16 nodes achieves a geometric mean speedup over sequential Bitwuzla of 9.9, 12.5, and 15.6, respectively. Overall, the speedups achieved by BITWUZLLOB consistently increase with increasing input difficulty, while investing more resources also consistently results in higher speedups. At a threshold of 300 and 600 seconds, the single-node run achieves a mean resource-efficiency (i.e., ratio between speedup and number of cores) of 21% and 33%, respectively. While the runs at higher scales cannot yet reach such remarkably high (cf. [44]) efficiencies, we still consider the observed scaling highly encouraging for developing our approach further in the future.

5.3 Latency Hiding

In our last experiment, we analyze the impact of our techniques to reduce the minimum latencies of SAT calls (Section 4.3). Figure 7 shows the relative improvement in terms of runtime achieved by latency-hiding threads (LHT), based on how many SAT calls per instance were issued (geometric mean) for each incremental and non-incremental logic. Clearly, logics with a very high number of SAT calls per instance (hundreds to thousands) profit significantly from the LHT, with mean accelerations ranging from 60% to 590%. In particular, the four points in the upper right correspond to the *incremental* logics `ABVFPLRA`, `BV`, `BVFP` and `QF_BVFPLRA`, all with more than 1,000 incremental SAT calls per instance on average. For logics with less than a few hundred SAT calls per instance, LHT are

Table 3. Mean speedups of BITWUZLLOB on 1/4/16 nodes over sequential Bitwuzla. For each logic, we consider commonly solved instances with a minimum sequential runtime of $\{1, 60\}$ s and show the geometric mean speedup over the resulting instance set.

		Seq. time ≥ 1 s				Seq. time ≥ 60 s		
Logic	#	1x48	4x48	16x48	#	1x48	4x48	16x48
Non-incremental	QF_ABV	40	2.50	3.20	3.54	22	3.69	5.19	6.35
	QF_ABVFP	23	3.27	4.62	5.29	5	7.14	11.65	14.23
	QF_ABVFPLRA	1	4.89	6.17	6.20	1	4.89	6.17	6.20
	QF_AUFBV	13	2.07	1.76	2.16	4	3.18	1.90	3.67
	QF_BV	97	7.84	11.76	14.12	79	9.07	13.77	17.04
	QF_BVFP	22	4.14	6.61	7.75	3	6.47	12.48	20.21
	QF_BVFPLRA	11	9.05	10.86	14.06	5	34.28	42.60	65.58
	QF_FP	45	3.86	5.46	7.87	29	4.91	6.81	10.66
	QF_FPLRA	10	4.19	6.05	6.52	4	8.12	11.15	12.42
	QF_UFBV	53	2.26	2.41	3.17	32	3.70	4.24	5.77
	ABVFP	1	0.32	0.32	0.32	0	–	–	–
	AUFBV	35	2.03	2.11	2.15	19	3.30	3.08	3.49
	AUFBVFP	9	2.11	1.98	1.90	3	2.72	2.77	4.26
	BV	43	2.48	3.68	3.67	26	3.20	4.59	4.90
	BVFP	2	0.82	1.11	1.20	0	–	–	–
	BVFPLRA	2	0.38	0.38	0.38	0	–	–	–
	FP	91	8.94	11.95	15.12	75	10.62	14.75	19.21
	UFBV	10	4.07	5.24	7.10	1	27.02	8.88	76.09
Incremental	QF_ABV	37	1.61	1.79	1.67	22	1.86	2.13	2.02
	QF_ABVFP	24	1.87	1.91	2.01	7	3.11	3.50	4.13
	QF_ABVFPLRA	3	1.36	1.32	1.34	0	–	–	–
	QF_AUFBV	11	1.26	1.52	1.71	7	1.27	1.66	1.90
	QF_BV	44	2.60	2.87	2.93	21	4.47	5.26	5.37
	QF_BVFP	26	3.75	5.91	6.57	10	10.67	21.13	25.36
	QF_BVFPLRA	8	0.89	0.88	0.88	0	–	–	–
	QF_UFBV	13	1.50	1.79	1.80	6	1.72	2.23	2.56
	ABVFPLRA	2	0.99	0.99	1.00	1	0.95	0.98	0.98
	BV	10	0.65	0.66	0.65	2	0.78	0.78	0.78
	BVFP	1	1.01	1.00	1.03	0	–	–	–
	BVFPLRA	4	0.96	0.99	0.98	0	–	–	–

not significantly beneficial, and sometimes even detrimental, to performance. In particular, for non-incremental logics ABVFP, BVFP and BVFPLRA, the LHT incur extreme performance degradation (a mean slowdown of 50 to over 100).

This may be due to the fact that the behavior of the abstraction-refinement loop and the MBQI procedure in Bitwuzla are sensitive to the models of the intermediate SAT calls. Examining some of the concerned instances, we found that MallobSat's distributed SAT solver portfolio used by BITWUZLLOB reproducibly returns a different satisfying assignment to certain SAT queries than the LHT's default sequential CaDiCaL solver. Different satisfying assignments may guide Bitwuzla's search into different search spaces, leading to faster convergence of the search than others. Note that the affected logics for which the LHT models seem to have a higher-than-average negative impact contain only a small

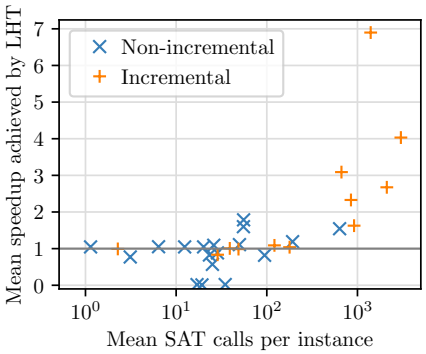

Fig. 7. Impact of latency-hiding threads (LHT), with one data point per logic. The x-coordinate denotes the geometric mean number of SAT calls across solved instances (without LHT), the y-coordinate denotes the geometric mean speedup achieved by enabling LHT over the commonly solved instances of the logic.

number of solved instances (15 in total). Overall, enabling the LHT significantly improves performance for instances with a high number of (fast) SAT calls. Exploiting the model sensitivity of our approach by retrieving *several* satisfying assignments from a single SAT query and then carefully branching the program flow over the resulting refinements in parallel may be an interesting direction for future work.

6 Conclusion

We have presented Bitwuzllob, a system for massively parallel SMT solving specialized for bit-precise reasoning. Our tool features the bit-blasting SMT solver Bitwuzla as an integrated module and deploys its (incremental) SAT queries to the distributed SAT solving system Mallob. Experiments with up to 768 cores of a distributed environment show that our system outperforms existing parallel SMT solvers for bit-precise reasoning, with unprecedented scaling on a wide range of incremental and non-incremental logics.

For future work, Bitwuzllob will serve as a solid foundation for exploring more advanced methods to further increase scalability. In particular, this may include modifying Bitwuzla's program flow to make *asynchronous* SAT calls (which would allow our system to actually solve several SAT tasks in parallel), as well as integrating theory-specific parallelization methods. Another promising direction could be to retrieve several different satisfying assignments from the distributed SAT solver and then branch over the respectively resulting refinements.

Acknowledgments. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding parts of this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de).

Data Availability Statement. The artifact accompanying this paper is archived and available in the Zenodo repository at https://zenodo.org/records/17478480.

References

1. SMT Competition 2025. https://smt-comp.github.io/2025 (2025)
2. Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.M., Piette, C.: Revisiting clause exchange in parallel SAT solving. In: Theory and Applications of Satisfiability Testing (SAT). pp. 200–213. Springer (2012). https://doi.org/10.1007/978-3-642-31612-8_16
3. Balyo, T., Biere, A., Iser, M., Sinz, C.: SAT race 2015. Artificial Intelligence **241**, 45–65 (2016). https://doi.org/10.1016/j.artint.2016.08.007
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.7. Tech. rep., Department of Computer Science, The University of Iowa (2025), available at http://smt-lib.org
5. Barrett, C.W., Chen, P., Cook, B., Dutertre, B., Jones, R.B., Le, N., Reynolds, A., Sheth, K., Stephens, C., Whalen, M.W.: SMT-D: new strategies for portfolio-based SMT solving. In: Narodytska, N., Rümmer, P. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024. pp. 1–10. IEEE (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_10
6. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2404, pp. 236–249. Springer (2002). https://doi.org/10.1007/3-540-45657-0_18
7. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_11
8. Biere, A.: Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In: SAT Competition 2013: Solver and Benchmark Descriptions. p. 1 (2013)
9. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: Cadical 2.0. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14681, pp. 133–152. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_7
10. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000). https://doi.org/10.1007/10722167_15
11. Cook, B.: Automated reasoning's scientific frontiers. https://www.amazon.science/blog/automated-reasonings-scientific-frontiers (2021), Amazon Science

12. Dutertre, B., Goel, A., Graham-Lengrand, S., Hader, T., Irfan, A., Jovanović, D., Lipparini, E., Mason, I.A., Nukala, K., Ruess, H.: Yices2 in SMT-COMP 2025. https://ahmed-irfan.github.io/smtcomp/yices2-smtcomp-2025.pdf (2025)
13. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4), 543–560 (2003). https://doi.org/10.1016/s1571-0661(05)82542-3
14. Enderton, H.B.: A mathematical introduction to logic. Academic Press (1972)
15. Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., Biere, A.: IPASIR-UP: user propagators for CDCL. In: Mahajan, M., Slivovsky, F. (eds.) 26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy. LIPIcs, vol. 271, pp. 8:1–8:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.SAT.2023.8
16. Fleury, M., Biere, A.: Scalable proof producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses. In: Pragmatics of SAT (2022)
17. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 306–320. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_25
18. Graham-Lengrand, S., Jovanovic, D., Dutertre, B.: Solving bitvectors with MCSAT: explanations from bits and pieces. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12166, pp. 103–121. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_7
19. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. JSAT **6**(4), 245–262 (2010). https://doi.org/10.3233/sat190070
20. Heisinger, M., Fleury, M., Biere, A.: Distributed Cube and Conquer with Paracooba. In: Pulina, L., Seidl, M. (eds.) Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12178, pp. 114–122. Springer (2020). https://doi.org/10.1007/978-3-030-51825-7_9
21. Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Heule, M., Weaver, S.A. (eds.) Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9340, pp. 369–386. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_27
22. Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Lookahead in partitioning SMT. In: Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021. pp. 271–279. IEEE (2021). https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_37
23. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View, Second Edition. Texts in Theoretical Computer Science. An EATCS Series, Springer (2016). https://doi.org/10.1007/978-3-662-50497-0
24. Manzano, M.: Introduction to many-sorted logic. In: Many-sorted Logic and its Applications, pp. 3–86. John Wiley & Sons, Inc., New York, NY, USA (1993)
25. Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: Artho, C., Legay, A., Peled, D. (eds.) Au-

tomated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9938, pp. 428–443 (2016). https://doi.org/10.1007/978-3-319-46520-3_27
26. Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: SMTS: distributed, visualized constraint solving. In: Barthe, G., Sutcliffe, G., Veanes, M. (eds.) LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018. EPiC Series in Computing, vol. 57, pp. 534–542. EasyChair (2018). https://doi.org/10.29007/FHGN
27. Marques-Silva, J., Lynce, I., Malik, S.: CDCL SAT solving. In: Handbook of Satisfiability. pp. 131–153. IOS Press (2021). https://doi.org/10.3233/faia200987
28. Moura, L.D., Rueß, H.: Lemmas on demand for satisfiability solvers. In: The 5th International Symposium on the Theory and Applications of Satisfiability Testing, SAT 2002, Cincinnati, USA, May 15, 2002 (2002)
29. Niemetz, A., Preiner, M.: Bitwuzla. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 3–17. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_1
30. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2025. https://bitwuzla.github.io/data/smtcomp2025/paper.pdf (2025)
31. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 587–595. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
32. Niemetz, A., Preiner, M., Zohar, Y.: Scalable bit-blasting with abstractions. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14681, pp. 178–200. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_9
33. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006). https://doi.org/10.1145/1217856.1217859
34. Preiner, M., Schurr, H., Barrett, C.W., Fontaine, P., Niemetz, A., Tinelli, C.: SMT-LIB release 2025 (incremental benchmarks) (May 2025). https://doi.org/10.5281/ZENODO.15493095
35. Preiner, M., Schurr, H., Barrett, C.W., Fontaine, P., Niemetz, A., Tinelli, C.: SMT-LIB release 2025 (non-incremental benchmarks) (Aug 2025). https://doi.org/10.5281/ZENODO.15493089
36. Reisenberger, C.: PBoolector: A Parallel SMT Solver for QF_BV by Combining Bit-Blasting with Look-Ahead. Master's thesis, Johannes Kepler University Linz (Nov 2014)
37. Sanders, P., Schreiber, D.: Decentralized online scheduling of malleable NP-hard jobs. In: Euro-Par 2022: Parallel Processing. pp. 119–135. Springer (2022). https://doi.org/10.1007/978-3-031-12597-3_8
38. Saoudi, M., Baarir, S., Sopena, J., Lejemble, T.: D-Painless: A framework for distributed portfolio SAT solving. In: Gurfinkel, A., Heule, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 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. Lecture Notes in Computer Science, vol. 15697, pp. 45–64. Springer (2025). https://doi.org/10.1007/978-3-031-90653-4_3
39. Schreiber, D.: Engineering HordeSat towards malleability: mallob-mono in the SAT 2020 cloud track. In: Proc. SAT Competition. pp. 45–46 (2020)
40. Schreiber, D.: Distributed incremental SAT solving with Mallob: Report and case study with hierarchical planning. CoRR **abs/2505.18836** (2025), https://doi.org/10.48550/arXiv.2505.18836
41. Schreiber, D., Jabs, C., Berg, J.: From scalable SAT to MaxSAT: Massively parallel solution improving search. In: Symposium on Combinatorial Search (SoCS) (2025). https://doi.org/10.1609/socs.v18i1.35984
42. Schreiber, D., Rigi-Luperti, N., Biere, A.: Streamlining Distributed SAT Solver Design. In: Berg, J., Nordström, J. (eds.) 28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 341, pp. 27:1–27:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025). https://doi.org/10.4230/LIPIcs.SAT.2025.27
43. Schreiber, D., Sanders, P.: Scalable SAT solving in the cloud. In: Li, C.M., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing – SAT 2021. pp. 518–534. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_35
44. Schreiber, D., Sanders, P.: MallobSat: Scalable SAT solving by clause sharing. Journal of Artificial Intelligence Research **80**, 1437–1495 (2024). https://doi.org/10.1613/jair.1.15827
45. Sinz, C., Blochinger, W., Küchlin, W.: PaSAT – parallel SAT-checking with lemma exchange: Implementation and applications. Electronic Notes in Discrete Mathematics **9**, 205–216 (2001). https://doi.org/10.1016/s1571-0653(04)00323-3
46. Wilson, A., Nötzli, A., Reynolds, A., Cook, B., Tinelli, C., Barrett, C.W.: Partitioning strategies for distributed SMT solving. In: Nadel, A., Rozier, K.Y. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023. pp. 199–208. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_28
47. Zhang, X., Chen, Z., Cai, S.: PRS: A new parallel/distributed framework for SAT. In: SAT Competition 2023: Benchmark, Solver and Proof Checker Descriptions. pp. 39–40 (2023)
48. Zhao, M., Cai, S., Qian, Y.: Distributed SMT solving based on dynamic variable-level partitioning. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14681, pp. 68–88. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_4
49. Zhao, M., Xu, Z., Lin, J., Cai, S.: STP-Parti-Bitwuzla at SMT-COMP 2025. https://github.com/shaowei-cai-group/STP-Parti-Bitwuzla-at-SMT-COMP-2025/blob/master/STP_Parti_Bitwuzla_at_SMT_COMP_2025.pdf (2025)