Parallel and Distributed Bounded Model Checking of Multi-threaded Programs

Omar Inverso
Gran Sasso Science Institute
L'Aquila, Italy
omar.inverso@gssi.it

Catia Trubiani
Gran Sasso Science Institute
L'Aquila, Italy
catia.trubiani@gssi.it

Abstract

We introduce a structure-aware parallel technique for context-bounded analysis of concurrent programs. The key intuition consists in decomposing the set of concurrent traces into symbolic subsets that are separately explored by multiple instances of the same decision procedure running in parallel. The decision procedures work on different partitions of the search space without cooperating, whence distribution follows effortlessly. Our experiments on a selection of complex multi-threaded programs show significant analysis speedups and scalability, and greater performance gains than with general-purpose parallel solvers.

CCS Concepts • **Software and its engineering → Software verification**; **Automated static analysis**.

Keywords Concurrency, Multithreading, Sequentialization, Software Verification, Parallel Analysis, Bounded Model Checking, SAT

1 Introduction

Concurrent programming is notoriously complex due to process interleaving leading to subtle undesirable situations that can be difficult to anticipate, even more so when sophisticated data-sharing mechanisms are adopted to increase the level of parallelism. For instance, techniques such as lock-free programming can effectively take advantage of modern computer architectures, but are particularly prone to errors. In fact, scalable techniques for automated software verification to assist programmers in such a cumbersome job are increasingly in demand.

Bounded model checking (BMC) reduces under-approximate reachability to propositional satisfiability (SAT) or generalisations thereof (SMT), successfully exploiting the outstanding performance gains of modern solvers [11]. BMC has been proved effective for the analysis of complex real-world sequential programs, as witnessed by the availability of several mature academic and commercial tools [16, 17, 22, 30] as well as recent competitions on software verification [6–8].

Sequentialization can seamlessly extend BMC to handle concurrent programs by transforming the initial program under analysis into a sequential one that preserves all the feasible execution traces up to some given bound. Most notably, it has been shown that combining tailored lazy sequentialization [28] with SAT-based BMC [16, 20] can be exceptionally effective for the analysis of complex multi-threaded programs that are out of reach for several other mature techniques [13, 28, 44, 45, 52].

However, BMC is limited by the inherent lack of parallelism of the underlying decision procedure [33], which prevents taking full advantage of modern multi-core hardware, or of the computational power of large clusters.

In this paper, we address this issue by introducing a parallelisation technique for bounded model checking of multi-threaded programs. The key intuition of our technique consists in decomposing the set of execution traces of the concurrent program under consideration into symbolic subsets that are separately explored by multiple instances of the very same decision procedure running in parallel.

Technically, we obtain different simple variations of the propositional formula initially generated by the bounded model checker for non-parallel analysis. By construction, satisfiability of any of the obtained formulae will indicate the presence of a bug in the program; unsatisfiability of all of them will prove the absence of bugs within the given bounds. We can thus spawn a separate solver for each formula, terminating the overall procedure as soon as one of the solvers finds a satisfiable assignment, or once all of them have run to completion. Since the solvers do not cooperate but only communicate upon termination, the technique naturally lends itself to distribution.

To evaluate the efficacy of our technique, we build a prototype tool and experiment with it on a selection of multi-threaded programs that are known in the literature for being particularly hard to analyse [13, 14, 25, 36, 38–40, 43, 51, 53]. The experimental results show that our technique can effectively distribute the overall computational load among

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374529

multiple units, achieving significant analysis speedups and scalability. The relatively simple structure-aware partitioning of the search space turns out to be quite effective, especially on the most complex programs for larger bounds, and for both reachable and unreachable bugs within the bounds. This positive trend seems to be further improved by distribution. Our technique outperforms general-purpose state-of-the-art parallel solvers.

The rest of the paper is organised as follows. Section 2 presents some preliminary concepts and definitions used in the rest of the paper. Section 3 describes our parallelisation schema. An experimental evaluation of our approach is presented in Section 4. Related work is discussed in Section 5. Final remarks and possible directions for future work are reported in Section 6.

2 Preliminaries

In this section, we review the basic concepts and the terminology used in the rest of the paper. For further details, we refer the reader to the references within the appropriate section.

2.1 Non-deterministic Multi-threaded Programs

A shared-memory multi-threaded program is composed of multiple threads of computation interacting via the shared memory. Besides local computations and operations on the shared memory, threads can perform concurrency-specific operations such as dynamic thread creation, join, synchronisation via locks or conditional variables, and so on.

For simplicity, in this paper we assume *sequential consistency* and *atomicity* of statements. In the presence of weaker consistency models, it is possible to capture the finer-grained thread interactions by means of appropriate program transformations, so that reasoning on the initial program under weaker consistency soundly reduces to reasoning on the transformed counterpart under sequential consistency [4]. As for atomicity of statements, it is always possible to rewrite a non-atomic statement as a sequence of atomic statements, each involving one shared variable at most [37].

The execution model of a multi-threaded program can intuitively be described as follows. At any given time during the execution of the program, only the *active* thread can perform computations. New threads can be spawned from any active thread, and added to the pool of *inactive* threads. At a *context switch* the currently active thread is suspended and becomes inactive, and one of the inactive threads is resumed and becomes the new active thread. When a thread becomes active for the first time its execution starts from the beginning, otherwise it continues from where the thread was last suspended. We refer to this mechanism as the *thread scheduling*.

In the following, we describe the syntax and semantics of C-like multi-threaded programs. For a comprehensive description of the execution model for shared-memory concurrent programs in the C language, along with the programming interface and ancillary data structures, we refer the reader to the POSIX-thread specifications [29].

Syntax. The syntax of multi-threaded programs is defined by the grammar shown in Figure 1. Terminal symbols are set in typewriter font. Notation $\langle n\ \mathtt{t}\rangle^*$ represents a possibly empty list of non-terminals n that are separated by terminals `t`; x denotes a local variable, y a shared variable, m a mutex, t a thread identifier and p a procedure name. All variables involved in a sequential statement are local. Expressions e can be local variables, integer constants, and can be combined using mathematical operators. Boolean expressions b can be `true` or `false`, or Boolean variables, and can be combined using the usual Boolean operations.

A *multi-threaded* program consists of a list of *global* variable declarations (i.e., *shared* variables), followed by a list of procedures. Each procedure has a list of zero or more typed parameters, and its body has a declaration of *local* variables followed by a statement. A statement is either a sequential or a concurrent statement, or a sequence of statements enclosed in braces.

A *sequential statement* can be an `assume` or `assert`, an assignment, a call to a procedure that takes multiple parameters (with an implicit call-by-reference parameter passing semantics), a `return`-statement, a conditional statement, a `while`-loop, a labelled sequential statement, or a jump to a label. Local variables are considered uninitialised right after their declaration, therefore they can non-deterministically assume any value allowed by their type. We also use the symbol `*` to explicitly denote the non-deterministic choice of any possible value of the corresponding type. We use the term *non-deterministic variable* to refer to such variables.

A *concurrent statement*, also known as *visible statement*, can be an assignment from a local to a shared variable (or the other way around), a call to a thread routine, or a labelled concurrent statement. Unlike local variables, global variables are always assumed to be initialised to a default value. For simplicity, we assume that the default value is always `0` regardless of the type of the variable. A thread creation statement $t = \mathtt{create}(p, e_1, \ldots, e_n)$ spawns a new thread from procedure p with expressions $e_1, \ldots, e_n$ as arguments. A thread join statement, $\mathtt{join}(t)$, pauses the current thread until the thread identified by t terminates its execution. Lock and unlock statements respectively acquire and release a mutex. The lock operation is blocking.

In the rest of the paper, we focus on *multi-threaded programs* in a C-like syntax under the POSIX-thread execution model. We assume the existence of a `main` procedure, which represents the only existing thread at the beginning of the execution, namely the *main thread* of the program.

Semantics. A *thread configuration* for a given thread of a multi-threaded program is a triple $\langle locals, pc, stack\rangle$, where

$prog ::= (dec;)^* \; (type\, p\, (\langle dec, \rangle^*) \; \{(dec;)^*\, stmt\})^*$
$dec ::= type\; z$
$type ::=$ `bool` | `int` | `void`
$stmt ::= seq;\, |\, conc;\, |\, \{stmt^+\}$
$seq ::=$ `assume`(b) | `assert`(b) | $x = e$ | $x = *$ | $p(\langle e, \rangle^*)$ | `return` e | `if`(b) $stmt$ `else` $stmt$ | `while`(b) $stmt$ | l: seq | `goto` l
$conc ::= x = y$ | $y = x$ | t = `create`$(p, \langle e, \rangle^*)$ | `join`(t) | `init`(m) | `lock`(m) | `unlock`(m) | `destroy`(m) | l: $conc$

Figure 1. Syntax of multi-threaded programs

locals is a valuation of the local variables of the thread, *pc* is the *program counter* that tracks the statement currently being executed, and *stack* models procedure calls by storing or retrieving the program counter of the caller and the current valuation of its local variables. Any other statement follows the usual C-like semantics.

A *multi-threaded program configuration* c, assuming n threads with identifiers $\{i_1, \ldots, i_n\}$, is a tuple of the form $\langle sh, en, th_{i_1}, \ldots, th_{i_n} \rangle$, in which sh is a valuation of the shared variables, $en \in \{i_1, \ldots, i_n\}$ is the identifier of the active thread, and th_{i_j} is the configuration of the thread i_j. In the *initial configuration* c, sh contains the default evaluation of the shared variables, $n = i_1 = 1$, and th_1 is the initial configuration of the `main` thread.

```
int i,j;

void t1() {
 for (int k=0; k<N; k++)
      i = i+j;
}

void t2() {
 for (int k=0; k<N; k++)
      j = j+i;
}

void main() {
 int tid1, tid2;
 int max;

 i = j = 1;

 tid1 = create(t1);
 tid2 = create(t2);

 join(tid1);
 join(tid2);

 max = fib(2*N+2);

 assert(j<max);
 assert(i<max);
}
```

Figure 2. `Fibonacci`

A *transition* of a multi-threaded program P corresponds to the execution of a statement by the active thread that moves the program from one configuration to another. If the statement being executed is sequential, only the configuration th_{en} of the active thread is updated. In particular, an `assume`-statement will not allow any further transitions from that thread if its condition does not hold. In contrast, the execution of an `assert`-statement on a condition that does not hold, will cause the whole program to terminate immediately. In this case the program is said to be *unsafe*, or to contain a an *reachable assertion violation.*

A thread creation operation adds to the configuration of the program a new thread configuration th_{n+1} with a fresh identifier. A thread join operation on a thread identifier t will not allow any further transition for the invoking thread until the thread identified by t terminates its execution. We omit the description of `lock` and `unlock` operations as they are not required in this paper. Upon termination, a thread is removed from the pool of threads for the program. The next active thread is then non-deterministically selected from the pool of available threads in c'.

Example. *The program of Figure 2 calculates Fibonacci numbers using concurrent computations over two shared variables* `i` *and* `j`*. The* `main` *thread creates threads* `t1` *and* `t2`*, waits until they both run to completion, and finally checks the property of interest expressed in the last two statements in the form of an assertion. The two threads* `t1` *and* `t2` *repeatedly increment the shared variables* `i` *and* `j` *by* `j` *and* `i`*, respectively. We assume a correct implementation for function* `fib`*.*

This program is unsafe as there exists an execution, starting with the `main` *thread, then continuing with alternating execution contexts of* `t1` *and* `t2`*, context-switching right after each assignment every time, leading to violating either of the two assertions at the end of the main thread. More concretely, let us suppose* `N=3`*. Assume that the* `main` *thread is preempted immediately after spawning* `t1` *and* `t2`*, then* `t1` *is scheduled, followed by* `t2`*, then again* `t1`*, and so on. During this alternation, variables* `i` *and* `j` *take on the consecutive values from the Fibonacci series. Eventually both* `t1` *and* `t2` *terminate and* `i=fib(7)=21`*. At that point the* `main` *thread is resumed, and the second assertion fails. Symmetrically, if* `t2` *is scheduled right after the* `main`*,* `j=fib(7)=21`*. Any other schedule will lead to smaller values for* `i` *and* `j`*, thus cannot violate either of the assertions.*

2.2 SAT-based BMC of Concurrent Programs via Sequentialization

Lazy sequentialization [28] takes as input a multi-threaded program P and two parameters u and r, corresponding respectively to the *loop unwinding bound* and the *round bound*, i.e., the number of round-robin schedules. It returns as output a sequential program $Q_{u,r}$ that preserves all the possible executions of P up to the given bounds. The output program $Q_{u,r}$ is obtained by applying a sequence of two program transformation passes, namely program unfolding and sequentialization.

Program unfolding consists in unwinding all the loops and inlining all the function calls in the program except those functions used for spawning threads. Recursive functions are inlined up to the unwinding bound. This process produces an intermediate *bounded program* P_u whose transition relation is the unfolded version of the initial program.

In practice, P_u preserves all the feasible behaviours of the initial program P up to u iterations of any cycle therein, and can spawn at most t threads. The number t of activations for each function is bounded because P_u is unfolded. P_u therefore

consists of a set of declaration statements for shared variables and t+1 function definitions: one for the main function plus t definitions for the functions used to spawn a thread, i.e., the *thread functions*.

The sequentialization pass consists in transforming P_u into a sequential program $Q_{u,r}$ that simulates all the executions of P_u within r round-robin schedules. Each thread function in P_u is transformed into a *thread simulation function* in $Q_{u,r}$. All calls to concurrency-specific routines (e.g., to create a thread or lock a mutex) are replaced by calls to functions that simulate them. The main function of P_u is treated as a thread function and thus translated into a thread simulation function as well. The new main function of $Q_{u,r}$ simulates the thread scheduler, invoking for each round the thread simulation functions in a fixed order.

The simulation functions can non-deterministically exit at any visible statement to simulate a context switch, but maintaining the program location of the simulated context-switch point, so that the computation can correctly be resumed at the next round.

On resuming a thread, the control jumps back to the location stored as above, then executes, again, a non-deterministically selected number of statements. The local variables of the thread simulation functions are persistent across function calls, so that there is no need to recompute them. This suspend-resume mechanism based on jumps is possible because P_u is bounded, and so it is possible to associate unique labels to the statements of every function from which a thread is spawned.

```
#include "lazycseq.h"
#define J(A,B)
        if(pc[t]>A | A>=cs')
        goto B

bool active[T] = {1,0,0};
int pc[T] = {0,0,0};
int size[T] = {5,3,3};
int r, t, cs;
-----------------------------
int i,j;

void f1() {
0: J(0,1); i = i+j;
1: J(1,2); i = i+j;
2: J(2,3); i = i+j;
}

void f2() {
   ...
}

void f0() {
    static int tid1, tid2;
    static int max;

    static i = j = 1;

0: J(0,1); tid1 = create(f1);
1: J(1,2); tid1 = create(f2);

2: J(2,3); join(tid1);
3: J(3,4); join(tid2);

           max = fib(2*N+2);

4: J(4,5); assert(j<max);
           assert(i<max);
}
-----------------------------
main() {
 int cs[R][T] = *;

 for (r=0; r<R; r++) {
  for (t=0; t<T; t++) {
   if (active[t]){
    assume(cs[r][t] ≥ pc[t]);
    assume(cs[r][t] ≤ size[t]);

    cs' = cs[r][t];
    ft();
    pc[t] = cs[r][t];
   }
  }
 }
}
```

Figure 3. Seq'd `Fibonacci`

The sequentialized program $Q_{u,r}$ consists of a static portion of code that simulates the concurrency-specific routines, a thread simulation function for each thread in P_u, and a new main function parameterised in t and r that simulates the thread scheduler.

Embedding the scheduler within the sequentialized program can be particularly convenient. In particular, the scheduling policy can be altered in such a way as to control the set of program behaviours of interest at a finer grain, for example by lifting the round-robin assumption on the schedules. Later in this paper, we indeed take advantage of this feature to implement our parallelisation approach.

Example. *The sequentialized version of program* `Fibonacci` *from Fig. 2 is shown in Fig. 3. The 3-unfolding is simplified for the sake of conciseness.* `R`, `T`, *and* `cs'` *correspond to the round bound, the number of threads, and to the next context-switch point, respectively.*

The differences from the original (unfolded) program are greyed out. The listing is divided into three sections. The transformed program is shown in the middle. The ancillary code and the scheduler to simulate context switching are respectively at the beginning and at the end of the listing.

2.3 SAT-based BMC of Sequential Programs

Bounded model checking (BMC) is a symbolic technique for under-approximate analysis of sequential programs [10, 11]. SAT-based BMC [16] is an efficient mechanised reduction from bounded reachability to propositional satisfiability.

Typically, the bound u is on the program unwinding. An initial procedure enforces the bound by transforming the input program P into a *bounded program* P_u by unwinding all loops and inlining all function calls. The bounded program is then converted into a simplified *intermediate representation* where there are no function calls, all jumps are forward, and each variable is assigned at most once. The intermediate representation of the bounded program is in turn compiled into a logical expression, known as *verification condition*, by encoding one instruction at a time as a separate conjunct.

A procedure known as *bit-blasting*, parameterised on the target system architecture, transforms the logical expression into a propositional formula by exploding the variables of the program into as many propositional variables as required to achieve bit-precise representation of the data (e.g., introducing 32 bits to represent an integer on a 32-bit architecture), and encoding bit-level arithmetics similarly to hardware circuits.

The propositional formula produced by the process described above is satisfiable if and only if there exists a feasible execution of the program where a property is violated within at most u iterations of any loop. The formula is then converted into an equisatisfiable CNF formula that can be analysed by a SAT solver. Eventually, any satisfying assignment of variables of the propositional formula returned by the

solver can be converted into an *error trace*, i.e., a sequence of transitions to follow in the input program leading to an error state. In the absence of detected errors, the bound represents a direct measure of the guarantee of correctness.

2.4 DPLL-style SAT

A *propositional decision procedure* takes as input a given formula ϕ and returns as output any satisfiable assignment of the variables of ϕ, otherwise $\bot$. We assume that ϕ is in CNF format:

$$\phi \triangleq \bigwedge_i \left(l_{i_1} \vee \cdots \vee l_{i_{|c_i|}}\right)$$

where $c_i = (l_{i_1} \vee \cdots \vee l_{i_{|c_i|}})$ is i-th *clause* of ϕ, and $|c_i|$ is the *size* of the clause, i.e., the number of *literals* in it.

A basic DPLL-style procedure consists in repeatedly applying two main steps, i.e., *decision* and *propagation*, *backtracking* when necessary [18, 19]. The decision step chooses an unassigned variable and assigns it a value. The propagation step propagates the implications of the decision on the chosen variable and its value according to the following two simple rules.

If a clause is a *unit clause*, i.e., it contains only a single unassigned literal, then only one possible value of that literal can make the clause true. This prunes an otherwise exhaustive search space by avoiding unnecessary decisions. At the same time, *pure* literals that occur only with one polarity can be set to the only value that satisfies every clause containing them. These clauses can then be removed from the formula, as they no longer constrain the search. Both the above rules may be applied repeatedly, excluding large parts of the search space.

When propagation generates conflicts, the decisions are backtracked. The idea is to flip one of the decisions variables that was previously assigned but not flipped, mark it as flipped, and then re-apply propagation. To increase efficiency modern solvers implement conflict-driven clause learning (CDCL), based on different refinements of the basic backtracking algorithm, such as *backjumping* or *clause learning*, so to invalidate multiple decisions at once rather than just one. The formula is *unsatisfiable* when backtracking is no longer possible. The formula is *unsatisfiable* when there are no unassigned variables. We refer the interested reader to [34] for further readings on propositional satisfiability.

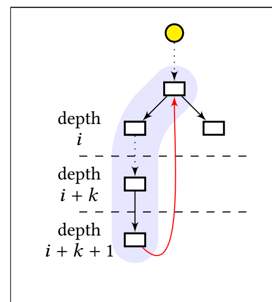

Figure 4. Decision graph

Visualising SAT. *A simple visualisation technique for the propositional satisfiability checking procedure described in this section is shown in Figure 4 [49]. The solver starts the search from the initial state (the round node) by iteratively choosing a variable and assigning it a value. Every choice generates a new node, increasing the depth by one. A conflict generated by the choice made at depth $i+k$ triggers backjumping at level $i+k+1$, invalidating all the chain of decisions starting from depth i. The search is then resumed from depth $i-1$, where a different variable is chosen, and so on. The temporal order of the choices corresponding to child nodes with a common root follows a left-to-right convention. Concrete examples of propositional decision graphs (simplified by removing backjump edges) are shown in Figure 6.*

3 Parallelisation Approach

The key intuition of our technique consists in decomposing the set of execution traces of a concurrent program into symbolic subsets. By exploring each such subset in isolation, the overall set of execution traces can be analysed in parallel. Before going further into the technical details, we first introduce the necessary definitions and notation.

3.1 Concurrent Execution Traces

Let P be a multi-threaded program (Sect. 2.1) and let us denote a transition for the active thread identified by j between a configuration c to a configuration c' by

$$c \xrightarrow[P]{j} c'. \qquad (transition)$$

In the rest of the section, we will sometimes use a placeholder ($\circ$) when a specific identifier for a configuration is not relevant. We can define the *transition relation* of P by combining the individual transitions of its threads (*transition*):

$$\xrightarrow[P]{} = \bigcup_j \circ \xrightarrow[P]{j} \circ\,. \qquad (tr\ relation)$$

A *trace* or *execution trace* of P is any sequence of transitions over (*tr relation*) connecting two configurations:

$$c \twoheadrightarrow_P c' = c \xrightarrow[P]{} \circ \cdots \circ \xrightarrow[P]{} c'. \qquad (trace)$$

If c is the initial configuration of the program, the trace is said *feasible* and c' is called a *reachable configuration* of P. If c is not an initial configuration of the program, the execution is an *execution fragment*.

An *execution context* or *context* for a thread t is an execution fragment where only transitions within the same thread take place:

$$c \overset{t}{\twoheadrightarrow}_P c' = c \xrightarrow[P]{t} \circ \cdots \circ \xrightarrow[P]{t} c'. \qquad (context)$$

A *k-context bounded execution* is an execution that can be obtained by concatenating at most k contexts of a multi-threaded program:

$$\circ \overset{t_1}{\twoheadrightarrow}_P \circ \cdots \circ \overset{t_k}{\twoheadrightarrow}_P \circ \qquad (k\text{-}cb\ exec)$$

where the sequence of thread identifiers $t_1, \ldots, t_k$ is the *schedule* of the execution. If $t_1, \ldots, t_k$ is a subsequence of a fixed schedule $\rho = s_1, \ldots, s_n$, the execution is called a *round-robin execution*, or *round* for short, with respect to ρ. A execution is *k-round-bounded* if it can be obtained by concatenating at most k rounds w.r.t. the same fixed schedule ρ. For instance, the execution

$$\circ \overset{s_1}{\underset{P}{\twoheadrightarrow}} \circ \overset{s_2}{\underset{P}{\twoheadrightarrow}} \circ \cdots \circ \overset{s_n}{\underset{P}{\twoheadrightarrow}} \circ \overset{s_{i \neq n}}{\underset{P}{\twoheadrightarrow}} \circ \qquad (k\text{-}rb\ exec)$$

has a round bound of 2, with the first n contexts spanning one round and the last context taking just one more round. The context bound for this execution is n+1.

3.2 Symbolic Partitioning of the Interleavings

Here we formally define the partitioning of the execution traces for a given concurrent program.

Let us consider a simple two-thread program, and suppose that one is interested in analysing all its possible executions up to one execution context. We can split the overall set of traces of the program in two subsets, collecting in the first subset all the execution traces whose schedule starts with the first thread, and the remaining traces in the other subset. By exploring each subset separately, the analysis can be performed in parallel using two computational units.

In the general case of a program P with n threads and m execution contexts, let us assume for the moment that all threads have been created beforehand and any of them can be scheduled from the initial configuration of P. Having denoted the set of threads of P with $T = \{t_1, \ldots, t_n\}$, the overall set of execution traces for n threads and within m contexts will then be captured by the following family of *symbolic* context-bounded executions:

$$\begin{array}{l} c_0 \overset{t_{1_1}}{\underset{P}{\twoheadrightarrow}} \circ \cdots \circ \overset{t_{1_m}}{\underset{P}{\twoheadrightarrow}} \circ \\ c_0 \overset{t_{2_1}}{\underset{P}{\twoheadrightarrow}} \circ \cdots \circ \overset{t_{2_m}}{\underset{P}{\twoheadrightarrow}} \circ \\ \qquad \cdots \\ c_0 \overset{t_{n_1}}{\underset{P}{\twoheadrightarrow}} \circ \cdots \circ \overset{t_{n_m}}{\underset{P}{\twoheadrightarrow}} \circ \end{array} \qquad (symbolic\ cb\ exec)$$

where c_0 represents the initial state of the program, $t_{i_1} = t_i$, and $t_{i_j} \in T$ (for $j \neq 1$) indicates the possible thread identifiers for the i-th expression above. Note that, since t_{i_j} is not fixed, each of the above expressions will combinatorially expand into different context-bounded executions as defined in Section 3.1 by (*k-cb exec*).

Now let T' and T'' be two partitions of T containing odd and even thread identifiers. Then, the overall set of m-context executions of P can be partitioned by separating the expressions defined by (*symbolic cb exec*) into two parts, based on whether the first scheduled thread (t_{i_1} for the symbolic schedule i) belongs to T' or T''. Splitting further ahead in the context can be done by restricting t_{i_j} to belong either to T' or T'' rather than to the whole T. In practice, the first scheduled thread is always the same in any possible execution (see Sect. 2.1), therefore it makes more sense to partition after the first execution context. Following this observation, up to 2^{m-1} partitions of the execution traces can be generated for a given program.

3.3 Parallel Bounded Model Checking Workflow

To illustrate how the parallelisation can be put in place, let us first shortly recap the existing workflow for non-parallel bounded model checking of concurrent programs.

Initially, the concurrent program of interest P is transformed into a sequential program $Q_{u,r}$ equivalent to P up to a given program flattening depth u, and a given number r of round-robin schedules (Sect. 2.2). Bounded model checking is then performed by encoding $Q_{u,r}$ into a propositional formula ϕ that is satisfiable if and only if there is a reachable error in $Q_{u,r}$, and thus in P, within the given bounds (Sect. 2.3). A propositional decision procedure checks whether ϕ is satisfiable, returning a satisfiable assignment for the variables of ϕ, in case ϕ is satisfiable (Sect. 2.4). If so, the satisfiable variable assignment is converted into an error trace for $Q_{u,k}$, and finally mapped back to the initial program P.

Our parallel analysis flow is similar. The main difference is that from the sequentialized program we generate multiple propositional sub-formulae rather than a single formula. Intuitively, each sub-formula will correspond to a different symbolic partition of the execution traces of the program (Sect. 3.2), and can be checked for satisfiability independently from the others. Therefore, satisfiability of any sub-formula will indicate a reachable bug in the program; unsatisfiability of all the sub-formulae will indicate the absence of reachable bugs within the given bounds.

Parallelisation also requires a few more slight alterations. In particular, we turn the initial round-bounded sequentialization schema into a context-bounded one, and alter the initialisation phase of the propositional solver. All the changes are discussed in detail in the rest of this section.

Changes to the Sequentialization Schema. We now show how to map the expressions (*symbolic cb exec*) into formulae obtained from ϕ that can be analysed in parallel. To that end, we need to introduce a few changes to the lazy sequentialization schema [28].

In the original lazy schema, the sequentialized program simulates an execution context for a given thread of the initial program by executing all the statements between two context-switch points in that thread (see Sect. 2.2). Specifically, we recall that $Q_{u,r}$ (see Fig. 3) simulates the execution context for thread `t` at round `r` by executing all the statements between `pc[t]` (i.e., the point where thread `t` was last pre-empted, initially set to the first statement of the thread) and the context-switch point `cs[r][t]`. Note

that the context-switch points are guessed up front via non-deterministic assignments, which allows to model any feasible interleaving up to the given number of round-robin schedules.

This schema cannot directly capture context-bounded executions as defined in Section 3.2 by (*symbolic cb exec*), because the scheduler assigns to each execution context a fixed thread identifier. To explicitly simulate single execution contexts rather than full rounds, we replace the scheduler with a context-bounded one (see Fig. 5). The new scheduler uses separate non-deterministic variables to symbolically represent not only the individual context-switch points (as in the old scheduler), but also the identifiers of the threads scheduled at each execution context (which in the old scheduler were fully deterministic instead), represented by the arrays `cs` and `tid`, respectively.

```
main() {
 // Context-switch point
 // and scheduled thread
 // at each execution context
 int cs[C] = *;
 int tid[C] = *;

 // Simulate context c
 // thread tid[c]
 for (c=0; c<C; c++) {
  t = tid[c];

  // Execute thread tid[c] from
  // the last pre-emption point
  // to the new one
  if (act[t]) {
   assume(cs[c] ≥ pc[tid[c]]);
   assume(cs[c] ≤ size[tid[c]]);

   cs' = cs[c];
   f_tid[c]();
   pc[tid[c]] = cs[c];
  }
 }
}
```

Figure 5. Context bounding

The revised sequentialization yields a program $Q_{u,k}$, where k is the context bound, and the identifiers of the scheduled threads at each execution context are modelled by a non-deterministic vector `tid`, whose elements correspond to the different t_{i_j} of expression (*symbolic cb exec*) defined in Section 3.2, except that in $Q_{u,k}$ the first scheduled thread is always the main thread t_0 (we recall that, as observed at the end of Sect. 3.2, the partitioning should take place from the second execution context on). It is worth to observe that the changes to the sequentialization schema only affect the scheduler; the rest of the program transformation is substantially unaffected.

Changes to the Bounded Model Checker. Due to bit-blasting (Sect. 2.3), each non-deterministic element of `tid` in $Q_{u,k}$ will correspond to a certain number of propositional variables of ϕ. For each execution context except the first and up to k, we can then append to ϕ an appropriate conjunct to restrict the propositional variable representing the least significant bit of the corresponding element of `tid`, achieving the partitioning of (*symbolic cb exec*) as wished for. In particular, restricting the least significant digit of `tid[1]` to be either 0 or 1 (observing that `tid[0]` is fixed) would split the search space in two.

In general, for m execution contexts, up to 2^{m-1} partitions can be induced by the following sub-formulae:

$$\begin{aligned} \phi_1 &\triangleq \phi \wedge a_1 \wedge \cdots \wedge a_{m-1} \\ \phi_2 &\triangleq \phi \wedge a_1 \wedge \cdots \wedge \overline{a_{m-1}} \\ &\cdots \\ \phi_{2^{m-1}} &\triangleq \phi \wedge \overline{a_1} \wedge \cdots \wedge \overline{a_{m-1}}. \end{aligned} \qquad (sf)$$

The actual number of partitions can be adjusted (in powers of two) by progressively removing assumptions. To clarify, the overall set of traces for 6 execution contexts can be partitioned into 32 sets by introducing 16 pairs of assumptions. Removing the last two pairs of assumptions $\overline{a_{m-1}}, a_{m-1}, \overline{a_{m-2}}$, and a_{m-2} would yield 8 sets instead. An alternative partitioning into four smaller sub-partitions of 8 sets each could be obtained by grouping the 32 partitions obtained in the beginning according to the four different combinations of values of a_1 and a_2. This is useful for distribution.

By construction, the set of satisfiable assignments of each ϕ_i is always a subset of those for ϕ, whence satisfiability of the former will imply satisfiability of the latter. Satisfiability of ϕ can be thus checked by considering each ϕ_i in isolation, terminating the overall procedure as soon as a satisfiable assignment is found, or once all the individual decision procedures have run to completion without satisfiable assignments.

We need to alter the bounded model checker and the built-in propositional solver to run under assumptions. The changes to the model checker itself are limited to keeping track of the propositional variables of interest, and handing over to the solver the assumptions about them. We recall that the propositional encoding produced by the model checker is completely independent from the assumptions.

Changes to the Propositional Solver. To fulfil the required assumptions, we override the variable selection heuristics of the solver at the beginning of its execution and force specific polarities for a set of given literals (see Sect. 2.4). More specifically, similarly to SAT solving under assumptions [21], within the solver we first convert all the assumptions into unit clauses, then force a propagation step (i.e., unit clause propagation and pure literal elimination, see Sect. 2.4). We also freeze all the assigned literals corresponding to the assumptions, to prevent the solver from backtracking to their decision level and flip their values.

It is worth to emphasise the effect of the assumptions both on the search space and on the propositional formula, due to the specific structure of the sequentialized program. In particular, since the propositional variables selected for the assumptions correspond to non-deterministic choices in the sequentialized program, the solver has to consider all their possible values in the worst case. At a propositional level, the search space reduces exponentially with the number of

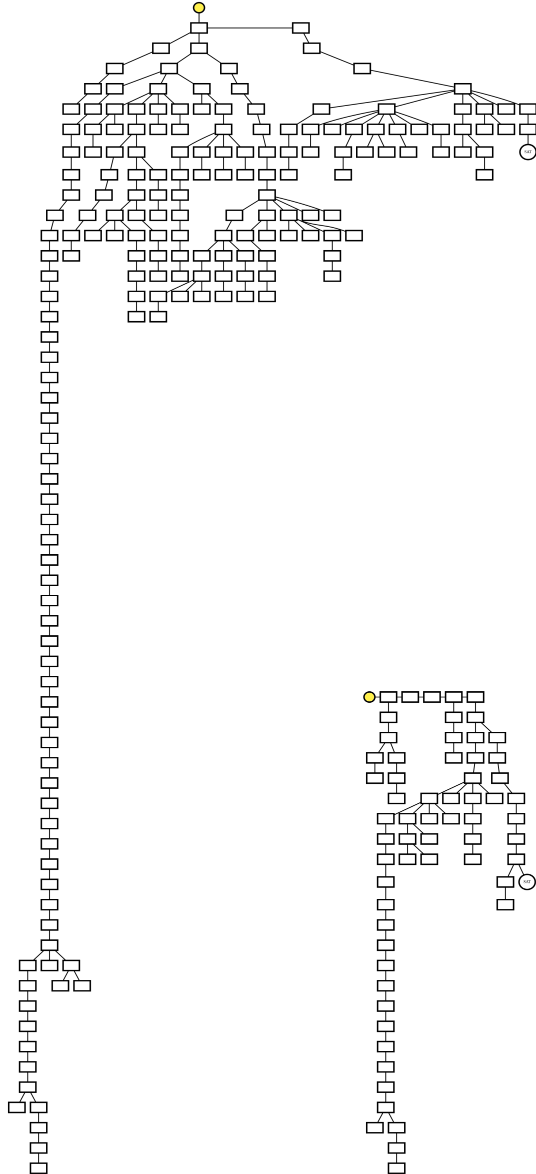

Figure 6. Propositional decision graphs with parallelisation (right) and without (left)

assumptions. Correspondingly, at the level of the program, every assumption will exclude half of the feasible execution traces.

In addition, the propagation step triggered right after appending the unit clauses to ϕ can have the effect of extensively pruning each ϕ_i. Intuitively, the elements of `tid` have a potentially great influence on the rest of the program (Fig. 3); similarly, the choice of specific polarities for the propositional variables that represent them will have a big impact on the whole ϕ_i. In practice, the solvers will start their search on more compact but equisatisfiable versions of the different ϕ_i. This can have a positive effect on the overall decision procedure runtimes, as any of the potentially very many subsequent steps of the solver is going to be performed on a smaller formula.

After the initialisation procedure, the solver carries on standardly. The assumptions are retained across possible restarts of the decision procedure.

Illustrative example. *Figure 6 reports a graphical comparison of the behaviour of the propositional solver with and without parallelisation. Following the procedures described in Section 2.2 (but replacing the scheduler with the context-bounded variant from Figure 5) and Section 2.3, a ϕ-formula of 3745 variables and 8874 clauses is generated from program* `Fibonacci` *(Fig. 3) using an unwinding and context bound of 2 and 6, respectively.*

The program contains a reachable assertion failure within these bounds, therefore ϕ is satisfiable. The two graphs follow the same structure as in Figure 4, described in Section 2.4 (for simplicity, backjumping edges are not drawn). The large graph on the left reproduces the execution of the solver on ϕ, without parallelisation. The small graph on the right represents instead the behaviour of the fastest solver that finds a satisfiable assignment for one of the sub-formulae (sf) of ϕ, say ϕ_i, when parallelising the analysis over 16 cores.

In the first case, the execution takes 268 decision steps with a maximal choice depth of 57 and 78 backjumps. In the second case, the decisions steps are 89 with a maximal depth of 28 and 26 backjumps. The performance difference is not appreciable in this simple example, as the solver can find a satisfiable assignment in the order of 0.01s for both ϕ and ϕ_i. Later in the paper, however, we shall see that on large formulae obtained from complex multi-threaded programs the performance gain can indeed be quite noticeable.

3.4 Prototype Implementation

We built a parallel prototype verifier upon the existing lazy sequentialization toolchain that is composed of CSeq [27], the C bounded model-checker CBMC [16] that CSeq relies upon for the analysis of the sequentialized program, and finally the MiniSat propositional solver [20] that CBMC uses as the decision procedure.

More concretely, in CSeq we replaced the scheduler of the default lazy sequentialization schema with the context-bounded version of Figure 5, and modified the feeder module to spawn different instances of CBMC with different assumptions on the selected set of propositional variables. To limit the overall engineering effort, CSeq invokes CBMC a first time only to generate the propositional formula along with a map from the variables of the program to variables in the propositional formula, so to create the required assumptions for partitioning. It then spawns different instances of CBMC, each performing the actual analysis under a different set of assumptions. Our implementation is based on CSeq 1.5, CBMC 5.4, and MiniSat 2.2.1 with simplifier.

Our prototype tool can, for example, spread the analysis over 8 computational cores by automatically picking three propositional variables and spawning a separate backend process for each of the 8 possible variable assignments. As soon as one of these verification sub-processes finds a feasible error trace, this is given as output and all other threads are terminated. In case a sub-process instead terminates without finding an error, the others keep going. When even the last sub-process terminates without finding errors, the program is reported to be verified safe within the given bounds.

User interface. For parallel analysis, the user provides the program filename, the unwind bound, the context bound (i.e., the number of context-switches minus one), and the number of cores or partitions (as a power of two not exceeding two to the number of execution contexts). For distributed analysis, the user additionally indicates the subset of partitions to analyse on each machine.

The following example command launches parallel analysis over 8 cores on a single machine:

```
./cseq.py -l lazy_cba_parallel
          -i bounded_buffer_unsafe.c
         --unwind 2 --contexts 5
         --cores 8
```

The following two commands can be run separately on different 4-cores machines to distribute the analysis over them:

```
./cseq.py -l lazy_cba_parallel
          -i bounded_buffer_unsafe.c
         --unwind 2 --contexts 5
         --cores 8
         --from 0 --to 3

./cseq.py -l lazy_cba_parallel
          -i bounded_buffer_unsafe.c
         --unwind 2 --contexts 5
         --cores 8
         --from 4 --to 7
```

Please note that our prototype does not currently implement the termination of processes across different machines.

A self-contained package for experimenting with our tool is available at https://doi.org/10.5281/zenodo.3594054.

4 Experimental Results

In this section, we present an experimental evaluation of our parallelisation technique on a selection of complex multi-threaded C programs. For all the experiments we used a dedicated machine equipped with 128GB of physical memory and a dual Xeon E5-2687W 8-core CPU clocked at 3.10GHz with hyper-threading disabled, running 64-bit GNU/Linux with kernel 4.9.95.

Benchmarks. Our selection of benchmarks consists of the four programs listed in Table 1.

Table 1. Main features of the considered programs and experimental results from SV-COMP 2019

Program	Lines	Threads	Error	Res. Limit	Correct
`Boundedbuffer`	368	5	13	1	**2**
`Eliminationstack`	331	8	9	6	**1**
`Safestack`	172	4	9	7	**0**
`Workstealingqueue`	262	4	8	2	**6**

`Boundedbuffer` [36] implements a shared buffer concurrently accessed by multiple processes using locks and conditional variables. It takes at least two unfoldings and five context switches for the error to show up, thus already exceeding existing estimates according to which the number of context switches typically needed for real-world bugs to manifest themselves is at most three [51].

`Eliminationstack` [13, 25] implements the elimination stack data structure proposed in [25] annotated with several assertions for verification purposes as described in [15]. This implementation is incorrect if memory is freed in pop operations. The bug requires three push operations to be concurrently executed with four pops.

`Safestack` [53] is a real-world benchmark implementing a lock-free stack for weak memory models that contains a subtle bug. This program was posted to the CHESS forum by its author as a challenge for existing testing tools. On our machine, the bug can be discovered in about 5 hours by CSeq within an unwinding and round bound of 3 and 4, respectively.

`Workstealingqueue` [14, 38–40, 43, 51] is a work-stealing queue with lock-free data structures. It takes 4 unwindings and 6 context switches to spot the error.

These programs are also part of the benchmarks for the `concurrency` category of SV-COMP 2019[1], in particular they represent the `concurrency/pthread-complex` subcategory. Table 1 summarises the experimental results of SV-COMP [8] on these files, obtained by the 16 tools participating in the `concurrency` category. Only two tools could produce a correct verification result for `Boundedbuffer`, one hit the resource limits and other 13 crashed or failed to produce any answer. For `Eliminationstack`, only one tool produced a correct result, six hit the resource limits, and nine did not produce any answer. None of the participants could successfully analyse `Safestack`, with seven of them hitting the resource limits and nine failing to produce any kind of answer. Successful analysis of `Workstealingqueue` was instead obtained by six of the tools, with two of them hitting the resource limits and eight crashing.

There are over one thousand programs in the `concurrency` category of SV-COMP 2019. Our experimental evaluation does not include further programs because many of the

[1] https://sv-comp.sosy-lab.org/2019/results/results-verified/META_ConcurrencySafety.table.html

participating tools can already verify them efficiently, with verification times in the order of a few seconds. It is worth to notice that SV-COMP assumes sequential consistency.

4.1 Scalability

We measured the performance of our prototype implementation on the selected programs over different bounds and while varying the numbers of available cores to partition the analysis. For all the experiments reported in this section we set a maximum runtime of 100k seconds and a memory usage limit of 64GB (which was never hit in any of our tests anyway).

The experimental results are reported in Table 2. The table contains four sections arranged vertically, a separate section for each program. The columns left to right represent the name of the program under analysis, the parameters for bounded exploration (unwind and context bounds), whether within the given bounds in the given program there is a reachable bug (indicated by a dot), the size of the generated propositional formulae (size of the DIMACS file, no. of variables and of clauses), the analysis times for the propositional formula (wall clock time, in seconds) over 1, 2, 4 and 8 cores, and finally the achieved speedups. Note the missing unwind bound for `Eliminationstack` due to the absence of loops in the program.

Our parallelisation technique turns out to be more effective on the two most complex programs, namely `Eliminationstack` and `Safestack`, and especially with larger bounds, i.e., a context bound of 5, and an unwind bound 3 of plus a context bound of 6 for the two programs, respectively (values boldfaced in the table).

The performance gains are quite large for some of the most complex configurations. For example, analysing 6 execution contexts of `Eliminationstack` requires about 95k seconds without parallelisation, but 25k seconds less with just one more core. This is rather interesting, considering that the partitioned sub-formulae only differ from the initial one in a single propositional variable out of almost 10 million. This nice scalability trend is consistently confirmed for the rest of the tests on up to 8 cores for `Eliminationstack` as well as on `Safestack` with 6 contexts, for instance.

We observe fluctuations in the speedup for some of the most simple configurations, e.g., `Boundedbuffer` with unwinding 3 and 5 contexts, and `Workstealingqueue` with unwinding 4 and 6 contexts. As suggested by the fact that different error traces are generated when changing the number of cores, this is due to the partitioning interfering with the heuristic decisions of the solvers. This is less likely to have an impact on performance when analysing safe instances, for which an exhaustive search is required.

Within the considered bounds and resource limits, our technique fails to spot the bugs in `Eliminationstack` and `Safestack`. Our tests time out on `Eliminationstack` for 7 contexts using up to 8 cores, and on `Safestack` with 8 contexts without parallelisation. From previous experiments, we know that the version of `Safestack` under consideration requires 4 round-robin executions, which correspond to 16 execution contexts. According to the error trace produced by CSeq, the execution contexts can be further trimmed down to 12. However, that is still out of reach for our current prototype implementation. Such inefficiency mostly stems from the inherent slowness of sequential analysis, and in particular to the structure-unawareness of decision procedure. In any case, our main goal is not parallel bug finding, but parallel analysis within the bounds. A more extensive analysis of `Safestack` where we show that by adding extra computational units it is indeed possible to terminate the analysis within the time limit is presented later in this section. Also note that the analysis of `Workstealingqueue` is faster with 6 contexts than with 5. This is not surprising, as a first bug is only exposed with a minimum of 6 contexts, while with 5 contexts the solver needs to perform an exhaustive search in order to conclude that the formula is unsatisfiable.

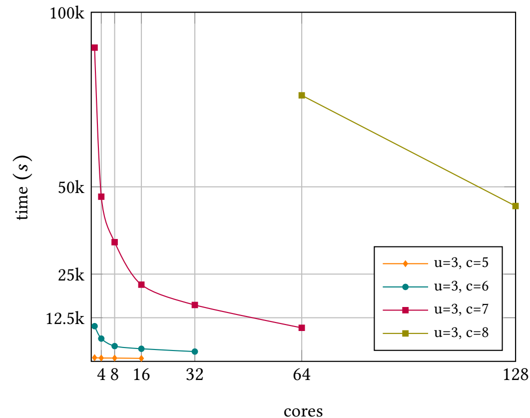

Figure 7. Distributed analysis of `Safestack`

To summarise, our prototype generally exhibits very good and consistent performance gains both in the case of reachable and unreachable bugs within the given bounds, respectively feasible and infeasible propositional expressions, especially for the most complex instances and with large bounds.

Distribution. We performed further experiments to assess the potential scalability of our approach when distributed over large computing clusters. Motivated by the limited scalability of the analysis of `Safestack` within 5 execution contexts, and in contrast to the very good speedups with one more context (see Table 2), we extended the analysis of the program to 8 context switches.

We split the partitions into chunks of 8 each, and analysed each chunk in separate runs by using 8 cores in parallel, so as to simulate a 128-core computing cluster of 16 machines with 8 cores each. We set a timeout of 100k seconds for each run. At the end of this procedure, having observed that still no

Table 2. Scalability of our prototype implementation of symbolic interleaving partitioning with MiniSat

Program	Params			Formula size			Time (s)				Speedup		
	u	c	reach	file (GB)	var (M)	cl (M)	1	2	4	8	2	4	8
Boundedbuffer	2	5	•	0.5	4.20	19.65	280	312	180	140	0.90	1.56	2.00
	2	6	•	0.6	5.27	24.64	842	955	584	352	0.88	1.44	2.39
	3	4		0.6	5.45	25.85	1063	703	472	397	1.51	2.25	2.68
	3	5	•	0.8	7.27	34.57	1052	570	1214	706	1.85	0.87	1.49
Eliminationstack	-	4		0.6	5.77	24.28	407	360	291	293	1.13	1.40	1.39
	-	5		0.8	7.70	32.53	7440	5041	2965	2019	1.48	2.51	**3.68**
	-	6		1.1	9.63	40.79	95518	69826	50601	34278	1.37	1.89	2.79
Safestack	3	4		0.4	3.97	16.80	685	547	482	366	1.25	1.42	1.87
	3	5		0.6	5.31	22.60	2607	1060	964	949	2.46	2.70	2.75
	3	6		0.7	6.65	28.41	14988	10088	6528	4374	1.49	2.30	**3.43**
Workstealingqueue	4	4		1.6	13.83	62.16	1239	930	830	929	1.33	1.49	1.33
	4	5		2.1	17.44	78.41	4545	4361	3391	2637	1.04	1.34	1.72
	4	6	•	2.5	21.06	94.69	1598	1104	1887	499	1.45	0.85	**3.20**

feasible error trace could be found within the given unwind and context bounds, we took the maximal amount of time required for a run to terminate.

The experimental results are shown in Fig. 7. The data points are missing for 7 contexts with one core and 8 contexts with less than 64 cores, due to the analysis of some of the chunks timing out. We observe the remarkable amount of time spent for the analysis of 7 execution contexts using only two cores (25h), which reduces to roughly half (13h) using four cores (linear speedup between 2 and 4 cores).

The main insight here is that scalability improves noticeably when increasing the context bound, confirming the performance trend (i.e., greater speedups for larger bounds) already observed for `Eliminationstack` and `Workstealingqueue` (see Table 2) with smaller parameters.

4.2 Comparison with Parallel SAT solvers

For comparison, we used parallel SAT solvers to analyse the propositional formulae produced by the bounded model checker from the sequentialized programs.

We considered the two top-ranked solvers in the parallel track of the two most recent editions of the SAT Competition (SC 2017[2] and SC 2018[3]) [1, 2].

Syrup [5] is a parallel version of Glucose with improved learnt clause minimisation. Glucose is built on top of MiniSat. Syrup ranked first in SC 2017. Plingeling [9] is a parallel version of Lingeling that uses satisfaction driven clause learning (SDCL), an extension of conflict-driven clause learning (CDCL). Plingeling ranked second in both SC 2017 and SC 2018. We exclude from the comparison painless-mcomsps [24] a portfolio-style solver instantiated within the PaInleSS framework that uses MapleCOMSPS as core sequential solver, in turn based on MiniSat. The tool ranked first in SC 2018, but we could not execute it because of an compatibility issues with our 64-bit system. We have been in contact with the authors and are still currently working to find a workaround.

For this part of the experiments we used Syrup 4.1 and Plingeling sc18, compiled from the source packages available at the SC 2018 online repository. We set a timeout of 20x the verification time achieved by our approach for the same formula with the same number of cores (Table 2).

The results are reported in Tables 3 and 4. Similarly to Table 2, columns Time and Speedup respectively report the required time for satisfiability check of the formula (wall clock time, in seconds), and the speedup over 1, 2, 4 and 8 cores. Column Performance Ratio measures the time taken by the parallel solver under consideration over the time taken by our approach (Table 2).

Despite some apparently large speedups in a few cases, we observe that in all cases our approach is faster than both Syrup and Plingeling, over 5x faster on about half of the tests, and up to 20x faster on the most complex cases.

5 Related Work

Effective parallelisation of program analysis is challenging and presents many interesting research directions at various levels. The following list is by no means exhaustive.

In general, parallelisation of propositional solving has been and still is the subject of extensive work, with several open challenges [33]. Attempts have been made to parallelise the analysis at the level of the decision procedure, where the major hindrance is in the loss of structure [46, 47]. In the attempt to shed some more light in that respect, we have included in our experimental evaluation a comparison between our relatively simple but structure-aware partitioning and more complex but general-purpose state-of-the-art parallel solvers [5, 9].

Propositional solving under assumptions by representing them as unit clauses to be propagated, rather than just

[2] https://baldur.iti.kit.edu/sat-competition-2017/
[3] http://sat2018.forsyte.tuwien.ac.at/

Table 3. Scalability and speedup of parallel solver Syrup vs. our approach

Program	Params			Time (s)				Speedup			Performance Ratio			
	u	c	reach	1	2	4	8	2	4	8	1	2	4	8
Boundedbuffer	2	5	•	1522	1899	529	461	0.80	2.88	3.30	5.44	6.09	2.94	3.29
	2	6	•	2004	1484	813	826	1.35	2.47	2.43	2.38	1.55	1.39	2.35
	3	4		3603	1748	1077	802	2.06	3.35	4.49	3.39	2.49	2.28	2.02
	3	5	•	2363	2716	2599	1086	0.87	0.91	2.18	2.25	4.76	2.14	1.54
Eliminationstack	-	4		1642	1170	953	647	1.40	1.72	2.54	4.03	3.25	3.27	2.21
	-	5		31593	19643	13539	7644	1.61	2.33	4.13	4.25	3.90	4.57	3.79
Safestack	3	4		1464	1222	847	696	1.20	1.73	2.10	2.14	2.23	1.76	1.90
	3	5		12563	8754	5677	5593	1.44	2.21	2.25	4.82	8.26	5.89	5.89
	3	6		157660	124578	86792	59245	1.27	1.82	2.66	10.52	12.35	13.30	13.54
Workstealingqueue	4	4		19636	14350	9991	6986	1.37	1.97	2.81	15.85	15.43	12.04	7.52
	4	5		>90900	75125	50358	30398	n/a	n/a	n/a	>20.00	17.23	14.85	11.53
	4	6	•	20731	20105	18012	3680	1.03	1.15	5.63	12.97	18.21	9.55	7.37

Table 4. Scalability and speedup of parallel solver Plingeling vs. our approach

Program	Params			Time (s)				Speedup			Performance Ratio			
	u	c	reach	1	2	4	8	2	4	8	1	2	4	8
Boundedbuffer	2	5	•	1884	2337	2145	561	0.81	0.88	3.36	6.73	7.49	11.92	4.01
	2	6	•	3989	2814	1399	1371	1.42	2.85	2.91	4.74	2.95	2.40	3.89
	3	4		3119	2492	2085	1385	1.25	1.50	2.25	2.93	3.54	4.42	3.49
	3	5	•	10001	7864	2647	1933	1.27	3.78	5.17	9.51	13.80	2.18	2.74
Eliminationstack	-	4		1699	1374	1292	1171	1.24	1.32	1.45	4.17	3.82	4.44	4.00
	-	5		40739	35187	29476	17711	1.16	1.38	2.30	5.48	6.98	9.94	8.77
Safestack	3	4		1858	998	819	537	1.86	2.27	3.46	2.71	1.82	1.70	1.47
	3	5		18008	17054	9182	5731	1.06	1.96	3.14	6.91	16.09	9.53	6.04
	3	6		>300000	>201760	>130560	>87480	n/a	n/a	n/a	>20.00	>20.00	>20.00	>20.00
Workstealingqueue	4	4		15717	10844	8865	7179	1.44	1.77	2.18	12.69	11.66	10.68	7.73
	4	5		>90900	>87220	>67820	49617	n/a	n/a	n/a	>20.00	>20.00	>20.00	18.82
	4	6	•	29165	>22080	13103	13387	n/a	2.22	2.17	18.25	>20.00	6.94	>20.00

first-chosen variables, was proposed in [41]. Integrating this approach with efficient preprocessing techniques has been shown to be effective for validating hardware designs [42]. Incremental SAT checking has been successfully integrated in many contexts, e.g., theorem proving [12].

Cooperation among multiple solvers has been shown to speed up the analysis [3, 54]. On the other hand, to avoid performance bottlenecks, modern parallel solvers (including those considered in our experimental evaluation) are particularly cautious about it [5]. Our approach is not exposed to this problem as the solvers do not cooperate.

Several tailored decision heuristics for SAT-based bounded model checking have been proposed, such as domain-specific variable ordering [50], or dynamic ordering to give priority to variables involved in recent conflicts [20]. Other approaches, specific to parallel SAT solving for BMC, have been considered in [32, 54]. An SMT-based parallel technique for bounded model checking sequential programs has been proposed in [48]. It is based on symbolic execution, and the partitioning is on the explicit execution paths of the program under analysis.

Similarly to our technique, structure-aware variable selection heuristics to guide the analysis of the search space have been adopted elsewhere. For instance, [31] combines random choices with heuristics that estimate the influence of a variable on the whole formula, resulting in a non-systematic search. We can rely on the specific structure of sequentialized programs, and thus our selection of variables can be straightforward and at the same time yield an effective partitioning of the search space. Adapting our own heuristics or others such as [31] to work in the general case would be very interesting, but is still an open question.

Like ours, other techniques use controlled schedules to steer the search too. Known approaches include stress-testing with probabilistic coverage guarantees via randomised priority-based scheduling [14, 43], and schedule sampling via lazy sequentialization [45]. Both [43] and [45] can perform parallel bug finding. In general, random scans of the search space can give some chance to spot bugs where more systematic analyses (including the one presented in this paper) may struggle, especially on programs where many different execution traces lead to a bug. In contrast, our technique

provides an effective partitioning of the search space either with or without reachable bugs, and full coverage up to the bounds. The two lines of research should therefore be considered orthogonal. Further empirical studies on controlled schedules have been considered in [51].

The lazy sequentialization schema [28] that we have amended in this paper can be seamlessly extended to weaker consistency models (such as those commonly adopted by modern commercial hardware) through an additional memory-management layer that in the sequentialized program simulates the possible reordering of memory operations [52]. This technique is modular with respect to our parallelisation approach, because it does not affect the overall structure of the sequentialized program, and in particular the scheduler. In general, support for weak memory models can be implemented at a preprocessing level by reducing program analysis under weak consistency to analysis under sequential consistency [4].

6 Conclusion and Future Work

We have experimented with partitioning, and then exploring in parallel, the set of possible traces of a concurrent program. To the best of our knowledge, ours is the first approach to concurrent (and possibly distributed) bounded model checking of multi-threaded programs. On complex programs, our prototype has shown very good speedups and scalability both in the case of reachable and unreachable error conditions. Our structure-aware technique is based on a relatively simple but effective partitioning of the search space and is more convenient than general-purpose state-of-the-art parallel solvers.

We plan to evaluate our technique on large computing clusters, and to experiment with more sophisticated program transformations and different assumption strategies for partitioning the analysis. An interesting variation to our approach would be dynamic assignment of sub-partitions of the execution traces to the cores as they become again available. An experimental comparison between our static partitioning approach and dynamic variants could yield some insights in the direction of cooperative parallelisation.

As for performance, it would be interesting to examine more in detail the runtime gaps of the solvers on the different sub-formulae and the speedup fluctuations while varying the number of cores, as well as the impact of different variable selection heuristics on load balancing. Performance models and probabilistic heuristics to relate the effect of splitting choices on the data or control flow of the program, and enhanced visualisation techniques for the behaviour of the solver could be of great help in that respect.

Symbolic partial order reduction has already been applied to round-bounded lazy sequentialization with positive results [26]. Our context-bounded analysis might benefit from partial order reduction even further, as in the round-bounded scheduler the threads are always executed in a fixed order, thus the chances of reducing the search space are considerably less. We also plan to experiment with static and dynamic partial order reduction [23, 35] directly at the level of the decision procedure.

It would be very interesting to generalise our methodology to other classes of complex programs, not necessarily concurrent ones, with adequate program transformations to lift the relevant structure and appropriate assumptions on it, or even general selection heuristics for the splitting variables. The latter would be quite valuable to improve the performance of sequential analysis too.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback that helped us to improve earlier versions of this paper. This work has been partially funded by MIUR projects PRIN 2017FTXR7S IT-MATTERS (Methods and Tools for Trustworthy Smart Systems) and PRIN 2017TWRCNB SEDUCE (Designing Spatially Distributed Cyber-Physical Systems under Uncertainty).

References

[1] 2017. *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions.* Department of Computer Science Series of Publications B, Vol. B-2017-1.

[2] 2018. *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions.* Department of Computer Science Series of Publications B, Vol. B-2018-1.

[3] Erika Ábrahám, Tobias Schubert, Bernd Becker, Martin Fränzle, and Christian Herde. 2011. Parallel SAT Solving in Bounded Model Checking. *J. Log. Comput.* 21, 1 (2011), 5–21.

[4] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013. Software Verification for Weak Memory via Program Transformation. In *Proceedings of the European Symposium on Programming (ESOP).* 512–532.

[5] Gilles Audemard and Laurent Simon. 2014. Lazy Clause Exchange Policy for Parallel SAT Solvers. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT).* 197–205.

[6] Dirk Beyer. 2016. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* 887–904.

[7] Dirk Beyer. 2017. Software Verification with Validation of Results - (Report on SV-COMP 2017). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* 331–349.

[8] Dirk Beyer. 2019. Automatic Verification of C and Java Programs: SV-COMP 2019. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* 133–155.

[9] Armin Biere. 2018. CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition. 13–14.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. *Advances in Computers* 58 (2003), 117–148.

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Proceedings of*

the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 193–207.

[12] Armin Biere, Ioan Dragan, Laura Kovács, and Andrei Voronkov. 2014. Experimenting with SAT Solvers in Vampire. In *Proceedings of Mexican International Conference on Artificial Intelligence (MICAI).* 431–442.

[13] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Tractable Refinement Checking for Concurrent Objects. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL).* 651–662.

[14] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 167–178.

[15] Omar Chebaro, Pascal Cuoq, Nikolai Kosmatov, Bruno Marre, Anne Pacalet, Nicky Williams, and Boris Yakobowski. 2014. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Automated Software Engineering* 21, 1 (2014), 107–143.

[16] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* 168–176.

[17] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. 2009. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* 137–148.

[18] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-proving. *Communications of ACM* 5, 7 (July 1962), 394–397.

[19] Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (July 1960), 201–215.

[20] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT).* 502–518.

[21] Niklas Eén and Niklas Sörensson. 2003. Temporal induction by incremental SAT solving. *Electronic Notes Theoretical Computer Science* 89, 4 (2003), 543–560.

[22] Stephan Falke, Florian Merz, and Carsten Sinz. 2013. The bounded model checker LLBMC. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* 706–709.

[23] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL).* 110–121.

[24] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. 2017. PaInleSS: A Framework for Parallel SAT Solving. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT).* 233–250.

[25] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *Proceedings of the annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 206–215.

[26] Vladimir Herdt, Hoang M. Le, Daniel Große, and Rolf Drechsler. 2019. Combining sequentialization-based verification of multi-threaded C programs with symbolic Partial Order Reduction. *International Journal on Software Tools for Technology Transfer (STTT)* 21, 5 (2019), 545–565.

[27] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* 807–812.

[28] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In *Proceedings of the International Conference on Computer-Aided Verification (CAV).* 585–602.

[29] ISO/IEC. 2009. *Information technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7, ISO/IEC/IEEE 9945:2009.*

[30] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. 2005. F-Soft: Software Verification Platform. In *Proceedings of the International Conference on Computer-Aided Verification (CAV).* 301–306.

[31] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015. Adaptive Concretization for Parallel Program Synthesis. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Daniel Kroening and Corina S. Pasareanu (Eds.). 377–394.

[32] Temesghen Kahsai and Cesare Tinelli. 2011. PKind: A parallel k-induction based model checker. In *Proceedings of the International Workshop on Parallel and Distributed Methods in verifiCation (PDMC).* 55–62.

[33] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. 2013. Resolution and Parallelizability: Barriers to the Efficient Parallelization of SAT Solvers. In *Proceedings of AAAI International Conference on Artificial Intelligence.* 481–488.

[34] Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures - An Algorithmic Point of View, Second Edition.* Springer.

[35] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron A. Peled, and Hüsnü Yenigün. 1998. Static Partial Order Reduction. In *Proceedings of International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS).* 345–357.

[36] Nuno Machado, Brandon Lucia, and Luís E. T. Rodrigues. 2015. Concurrency debugging with differential schedule projections. In *Proceedings of ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI).* 586–595.

[37] Markus Müller-Olm. 2006. *Variations on Constants - Flow Analysis of Sequential and Parallel Programs.* Vol. 3800. Springer.

[38] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI).* 446–455.

[39] Madanlal Musuvathi and Shaz Qadeer. 2008. Fair stateless model checking. In *Proceedings of ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI).* 362–371.

[40] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the International Symposium on Operating Systems Design and Implementation (OSDI).* 267–280.

[41] Alexander Nadel and Vadim Ryvchin. 2012. Efficient SAT Solving under Assumptions. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT).* 242–255.

[42] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. 2014. Ultimately Incremental SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT).* 206–218.

[43] Santosh Nagarakatte, Sebastian Burckhardt, Milo M. K. Martin, and Madanlal Musuvathi. 2012. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI).* 543–554.

[44] Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2017. Lazy-CSeq 2.0: Combining Lazy Sequentialization with Abstract Interpretation - (Competition Contribution). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* 375–379.

[45] Truc L. Nguyen, Peter Schrammel, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2017. Parallel bug-finding in concurrent programs via reduced interleaving instances. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* 753–764.

[46] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. 2002. Recovering and Exploiting Structural Knowledge from CNF Formulas. In *Proceedings of International Conference on Principles and Practice of Constraint Programming (CP)*. 185–199.

[47] Duc Nghia Pham, John Thornton, and Abdul Sattar. 2007. Building Structure into Local Search for SAT. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*. 2359–2364.

[48] Quoc-Sang Phan, Pasquale Malacaria, and Corina S. Pasareanu. 2015. Concurrent Bounded Model Checking. *ACM SIGSOFT Software Engineering Notes* 40, 1 (2015), 1–5.

[49] Mate Soos. 2019. SAT Solvers as Smart Search Engines. In *https://www.msoos.org/2019/02/sat-solvers-as-smart-search-engines/*.

[50] Ofer Strichman. 2000. Tuning SAT Checkers for Bounded Model Checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*. 480–494.

[51] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency testing using schedule bounding: an empirical study. In *Proceedings of the International Conference on Principles and Practice of Parallel Programming (PPoPP)*. 15–28.

[52] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2016. Lazy sequentialization for TSO and PSO via shared memory abstractions. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 193–200.

[53] Dmitry Vyukov. 2010. Bug with a context switch bound 5. In *Microsoft CHESS Forum*.

[54] Siert Wieringa, Matti Niemenmaa, and Keijo Heljanko. 2009. Tarmo: A Framework for Parallelized Bounded Model Checking. In *Proceedings of the International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*. 62–76.