Parallel Graph-Based Stateless Model Checking

Magnus Lång and Konstantinos Sagonas

Department of Information Technology, Uppsala University, Uppsala, Sweden
{Magnus.Lang,Konstantinos.Sagonas}@it.uu.se

Abstract. Stateless model checking (SMC) is an automatic technique with low memory requirements for finding errors in concurrent programs or for checking for their absence. To be effective, SMC tools require algorithms that combat the combinatorial explosion in the number of process/thread interactions that need to be explored. In recent years, a plethora of such algorithms have emerged, which can be classified broadly in those that explore *interleavings* (i.e., complete serializations of events) and those that explore *traces* (i.e., graphs of events). In either case, an SMC algorithm is *optimal* if it explores exactly one representative from each class of equivalent executions. In this paper, we examine the parallelization of a state-of-the-art graph-based algorithm for SMC under sequential consistency, based on the reads-from relation. The algorithm is provably optimal, and in practice spends only polynomial time per equivalence class. We present the modifications to the algorithm that its parallelization requires and implementation aspects that allow us to make it scalable. We report on the performance and scalability that we were able to achieve on C/pthread programs, and how this performance compares to that of other SMC tools. Finally, we argue for the inherent advantages that graph-based algorithms have over interleaving-based ones for achieving scalability when parallelism enters the picture.

1 Introduction

Stateless model checking (SMC) [12] is a fully automatic technique to systematically, and often *exhaustively*, test concurrent programs written in general-purpose programming languages for bugs and other concurrency issues. The programs, which must be data-deterministic and terminating, are executed many times under the control of a stateless model checker, each time controlled to exhibit a different interleaving. One approach to combating the combinatorial explosion in the number of executions that need to be explored in SMC is called *Dynamic Partial-Order Reduction* [11, 2], which drives the scheduling of the program, and systematically explores the different orderings of *dependent events*, for example conflicting accesses to the same shared variable. Recently, a new kind of SMC reduction algorithms [16, 17, 3] has emerged, which views program executions not as schedules but as *traces*, i.e., labelled *graphs* of program events. These new graph-based SMC algorithms are conceptually easier to understand and often simpler to implement in an efficient way. More importantly, as we will show in this paper, they are more amenable to effective parallelization.

In either type of SMC algorithms, the partial order relation and the traces, respectively, divide the interleavings of the program into equivalence classes. If an algorithm explores

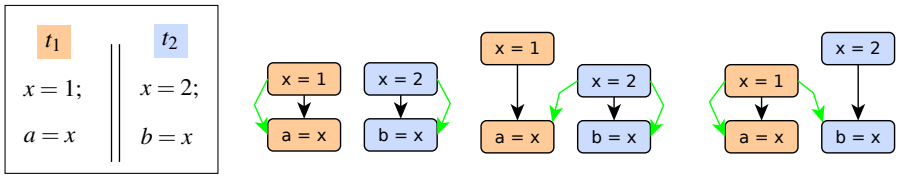

Fig. 1: Simple program (left) and its three READSFROM-SMC traces.

exactly one representative from each equivalence class, we say that it is *optimal*. Non-optimal algorithms may spend exponentially more time than optimal ones exploring multiple different elements of each equivalence class.

SMC algorithms have low memory requirements in practice, but run in time at least linear in the number of different concurrency behaviours of the program. In order to apply SMC to challenging programs, users are often required to carefully constrain the programs to minimize the number of concurrency behaviours that are independent of the part of the program they want to test. Having the ability to scale SMC algorithms using, for example, the nowadays abundant availability of multicore machines could allow users to apply stateless model checking more frequently, or to even more challenging programs. Last but not least, we hold that it is particularly desirable to parallelize optimal SMC algorithms, as the slowdown of non-optimality could easily trump the speedup gained by parallelization, no matter how powerful a platform one uses.

One example of an optimal graph-based SMC algorithm is the recently proposed READSFROM-SMC algorithm of Abdulla et al. [3], hereafter abbreviated RF-SMC, which runs programs under the *sequential consistency* (SC) memory model. In RF-SMC, traces consist of two types of edges: 1) "Program order", which contains the sequencing of operations from the program source code, but also orderings like a thread-spawn event before the thread-start, and 2) "Reads-from", which associates read events with the corresponding write event from which they got their value.

As an illustration, consider the program in Fig. 1. Two threads, t_1 and t_2, access a shared variable x. Each thread writes to the variable and reads from it into a local register, a resp. b. The three traces that RF-SMC will examine are also shown in the figure. Program order is drawn with black arrows, and Reads-from with green. We can see that either both threads read their own writes, or both threads read the same write (from t_1 resp. t_2). When RF-SMC explores the example program, it starts by running an arbitrary interleaving of the program events and constructs the corresponding trace. Let us assume that it is the first trace in Fig. 1. By analyzing the reads in this trace, the algorithm discovers that the source writes can be changed, leading to the two other traces, which it then proceeds to run, one at a time. They too will be analyzed, but will only lead back to the initial trace. Here, there is an opportunity to parallelize the algorithm by running and analyzing the second and third trace concurrently.

In this paper, we present PAR-RF-SMC (Sect. 4), a parallel version of the RF-SMC algorithm (Sect. 3), as well as an implementation on top of NIDHUGG and the changes that were needed to make the implementation parallel (Sect. 5). We experimentally evaluate its performance and scalability in Sect. 6. The paper ends by reviewing related work (Sect. 7) and some concluding remarks.

2 READSFROM-SMC by Example

Let us explore in depth how RF-SMC operates, using the program in Fig. 1 as an example. The algorithm represents an execution as a sequence of *events*. Each event records a side-effect of a program statement, such as a read or a write of a shared variable. There are two write events here $x=1$ and $x=2$, denoted e_1 and e_2. Read events record from which write event they read (the `rf` relation). The initialization of shared variables (x in the example) is also represented as write events, but we will omit them for brevity.

The goal of the algorithm is to explore all the execution graphs of the program, which we call *traces*. We represent a trace by a linearization of its events. For example, we may represent the first trace in Fig. 1 by $x=1\, a=x^{e_1}\, x=2\, b=x^{e_2}$, where $a=x^{e_1}$ denotes the event where $a=x$ reads from e_1. Note that these linearizations are not necessarily executions; i.e. read events do not necessarily read from the most recent write event.

To structure the exploration, RF-SMC maintains traces in an *exploration tree*. Branches of this tree are gradually pruned, achieving low memory consumption in practice, but in this section, for simplicity, we will show complete exploration trees.

The algorithm starts by running an arbitrary execution, and acquires its corresponding trace. Let's assume it is $x=1\, a=x^{e_1}\, x=2\, b=x^{e_2}$. The exploration tree is initialized with the first trace. In Fig. 2, it is shown as the leftmost trace, τ_1. The algorithm then explores if any if the reads in this trace can have their source writes changed. First, it considers if $b=x$ could have read from e_1 instead, and generates a trace prefix $x=1\, a=x^{e_1}\, x=2\, b=x^{e_1}$. In general, it may not always be possible to have read from this new source. In order to test it, RF-SMC employs a procedure called GETWITNESS (Sect. 3.1). This procedure will either report that the change would violate SC-consistency, or produces a *witness*, an execution in which that source is realized. In this case, the witness shown in the lower box in Fig. 2 is returned, and RF-SMC inserts the prefix in the exploration tree to create a new $b=x^{e_1}$ node, and associates it with the witness. Next, it considers if $a=x$ could read from e_2. For this, the prefix $x=1\, a=x^{e_2}$ would not be self-contained, since $e_2 : x=2$ is not included, so the missing event(s) are appended to create the prefix $x=1\, a=x^{e_2}\, x=2$. Again, note that the ordering of a linearization of a trace only needs to preserve program order. This prefix will also be found to be consistent, so this prefix, and its witness are inserted into the exploration tree, whose state at this point is as in the left of Fig. 2.

The sequential algorithm could in principle continue with either of these two witnesses, but will pick the leftmost one for space reduction reasons. Starting from the $b=x^{e_1}$ node, the algorithm takes the witness out of the tree, extends it to a complete execution, and adds any newly found nodes under $b=x^{e_1}$. In this case, the witness was already a complete execution. The algorithm then again looks for new sources for the read events in the trace. However, this time it finds that both the $b=x^{e_2}$ and $a=x^{e_2}$ nodes already exist in the exploration tree, and so does nothing. Last, it backtracks to the witness for $a=x^{e_2}$. This time, the execution is not complete and a final $b=x^{e_2}$ event is added to the end of the tree, to create the tree shown in the middle of Fig. 2. Once more, the algorithm will consider if any reads-from sources can be changed. For a, $a=x^{e_1}$ is still in the tree, but for b, the algorithm will generate the trace prefix $x=1\, a=x^{e_2}\, x=2\, b=x^{e_1}$. (Note that this does not correspond to the $b=x^{e_1}$ node already in the tree.) However, this trace is not consistent, as, under SC, there is no way to interleave the program statements

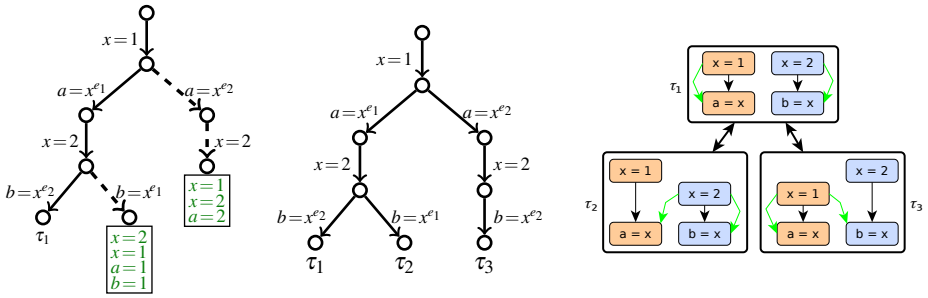

Fig. 2: Exploring the program of Fig. 1: (left) the first trace and two witnesses from it, shown in green. (middle) the tree of traces after the full exploration. (right) the "discoverability" relation of the three traces: there is an edge from trace τ to trace τ' if τ' is discoverable from τ. This affects available concurrency.

so that both a reads 2 and b reads 1. Thus, no more nodes are inserted into the exploration tree, and the algorithm terminates at this point.

As we can see, most of the algorithm is agnostic to the order that the newly found traces are explored. The only point where the logic differs is when the exploration tree is checked to see if a certain node already exists. We may further note that the only purpose of this check is to avoid redundant work, as the witness or lack thereof would be identical to that found the last time that trace prefix was checked. Simplifying slightly, the exploration of traces can be compared to exploration of a strongly connected digraph, where the set of all nodes is found by computing the neighbour sets of known nodes. Then, it should be clear why an algorithm on this form is parallelizable. Any time several new traces are discovered, these could in principle be explored in parallel.

The rightmost part of Fig. 2 shows this digraph of traces as induced by this example. If trace τ_1 is explored first, traces τ_2 and τ_3 may be explored in parallel. However, if exploration starts with trace τ_2 or τ_3, the remaining two traces can only be explored sequentially. Thus, the scheduling decisions made during arbitrary exploration can affect the available concurrency.

3 Sequential ReadsFrom-SMC

In this section, we review RF-SMC [3], a sequential SMC algorithm which optimally explores the complete consistent traces of a given concurrent program. The exploration algorithm is centered around a test GetWitness for checking whether a given trace is consistent under SC, described in Sect. 3.1.

For a read event e_R, let $e_R.src$ be the write event that it reads from, and for any read or write event e, let $e.var$ be the variable, or memory address, that is accessed. Recall that traces are labelled graphs represented by some linearization. We say that two traces τ and τ' are *equivalent*, denoted $\tau \equiv \tau'$, if they are linearizations of the same graph. For a trace τ, let $\leq_\tau$ be the relation containing all of the edges labelled "program order", and `rf` be the relation containing all edges labelled "reads-from". Define a *cut* of τ to be a subsequence τ' of τ such that whenever e and e' are events in τ, such that τ'

Algorithm 1: RF-SMC.

```
1  RF-SMC(τ, E)
2      extend E to a complete execution E · Ê where each event of Ê is unmarked
3      τ' := τ · Ê
4      for each read event e_R ∈ Ê do schedules(pre(τ', e_R)) := ∅
5      for each e_R, e_W ∈ τ' : e_W.var = e_R.var and e_W ≠ e_R.src and
6                               (e_R ∈ Ê or e_W ∈ Ê) and unmarked(e_R) and
7                               MayRead(τ', e_R, e_W) do
8          τ'' := pre(τ', e_R)
9          π := predecs(τ', e_W) ∩ post(τ', e_R)
10         σ := e_R[src := e_W] · mark(π)
11         E'' := GetWitness(τ'' · σ, E · Ê)
12         if E'' ≠ ⟨⟩ and ¬∃⟨σ', −⟩ ∈ schedules(τ'') : σ' ≡ σ then
13             add ⟨σ, E''⟩ to schedules(τ'')
14     for each read event e_R ∈ Ê starting from the end do
15         τ'' := pre(τ', e_R)
16         for each ⟨σ, E''⟩ ∈ schedules(τ'') do RF-SMC(τ'' · σ, E'')
17         erase schedules(τ'')
```

contains e and e' $[\leq_\tau \cup \texttt{rf}]^* e$, then τ' also contains e'. That is, a cut is closed under causal dependencies (in the sense of $\leq_\tau \cup \texttt{rf}$). Note that a cut is also a trace.

For a trace τ and an event $e \in \tau$ let

$pre(\tau, e)$ denote the prefix of τ up to, but not including, e;
$post(\tau, e)$ denote the suffix of τ after, but not including, e;
$predecs(\tau, e)$ denote the minimal cut of τ which contains e, i.e., the set of events (including e) on which e is causally dependent.

As an example, if τ is the trace τ_3 in Fig. 2, i.e., $\tau := x=1\, a=x^{e_2}\, x=1\, b=x^{e_2}$, then $predecs(\tau, b=x^{e_1})$ is $x=1\, x=2\, b=x^{e_2}$.

Algorithm 1 gives the pseudocode of RF-SMC(τ, E), where τ is an SC-consistent trace and E is an execution and a witness for τ. Exploration begins with a call RF-SMC$(\langle\rangle, \langle\rangle)$.

An important element in RF-SMC is to analyze an explored trace to see whether another trace can be formed by changing the source of one of its read events. In order to explore every consistent combination of sources for all the reads in the program exactly once, the reads are organized in the exploration tree, as shown in Sect. 2. For this to be sound, the order of read events in the traces must be preserved throughout the exploration. However, sometimes read events must be "lifted" because they are injected in the causal history of a prior read event when its source is changed. The algorithm must not explore alternative sources for lifted read events, or the algorithm might explore redundant traces, and even be unsound. To avoid this, we extend the representation of events with a field whose value is either $\top$ or $\bot$. When $\top$, we say that the event is *marked*; when $\bot$, that it is *unmarked*. For a sequence π of events, let $mark(\pi)$ be π but with each element marked. By marking any events that are "lifted" in the exploration tree, RF-SMC will not analyse any such event for alternate sources.

For some trace prefix τ in the exploration tree, RF-SMC represents the set of children of that node by a set $schedules(\tau)$. Each element is a tuple $\langle \tau', E' \rangle$ of the child

trace τ' and a witness E' of its consistency. As we saw in Sect. 2, the exploration tree serves two purposes. First, for any read event e_R in a trace τ, the set $\mathit{schedules}(\mathit{pre}(\tau, e_R))$ keeps track of all read sources for e_R that have been found so far. Secondly, *schedules* also keeps track of what trace prefixes, with associated executions, to explore in the future.

The algorithm is structured in three phases: i) exploration (lines 2–3); ii) new-source-detection (lines 4–13); and iii) recursive-exploration (lines 14–17).

In the exploration phase, RF-SMC(τ, E) extends E to an arbitrary complete execution $E \cdot \hat{E}$, and its complete trace τ' is computed. Correctness properties are checked on $E \cdot \hat{E}$.

In the new-source-detection phase, for every read event e_R, any possibly consistent source e_W from the same trace are considered. The MAYREAD(τ', e_R, e_W) predicate checks whether e_R reading from e_W would cause a causal loop, or whether there is another write event e'_W causally after e_W and before e_R. This is an efficient necessary but not sufficient check for whether e_R can read from e_W. If the check passes, the algorithm will construct the trace prefix $\tau'' \cdot \sigma$ containing e_R with the new source, as well as a sequence π of any new causal dependencies of e_R. On line 9, as an abuse of notation we take the intersection $\cup$ of two sequences, and mean the subsequence of both that contain the elements common to both sequences. Note that this is well-defined because both sequences agree on the ordering of common elements. The trace prefix $\tau'' \cdot \sigma$ is fed to the GETWITNESS decision procedure, along with the current execution $E \cdot \hat{E}$ as a hint. If it is found to be consistent, and there is no equivalent node already in the tree, it is inserted into the exploration tree along with the witness found by GETWITNESS.

In the recursive-exploration phase, the algorithm calls itself recursively on any consistent trace prefixes that were found by the new-source-detection phase, starting from the bottom of the tree. Note that the recursive calls to RF-SMC may add elements to $\mathit{schedules}(\tau'')$. When all sources of e_R have been recursively explored, the set $\mathit{schedules}(\tau'')$ may safely be erased to keep memory use low.

The algorithm satisfies the following three properties [3]: *Soundness*, *Completeness*, and *Optimality* (cf. Sect. 4).

3.1 Checking Consistency: the GETWITNESS Procedure

Let us briefly overview the GETWITNESS(τ, E) procedure. (For more details, refer to our previous paper [3].) The procedure checks the consistency of τ, returning either a witness or $\langle\rangle$. It takes an execution E as a hint for the ordering of write events to the same variable when it cannot infer the ordering, or when any ordering is valid.

The core of the procedure is a sound but incomplete heuristic which runs in polynomial time, but falls back on a sound and complete decision procedure which is polynomial time when the number of threads is fixed [3]. The heuristic is based on the concept of *saturation*. When a trace τ is saturated, the `rf` and $\leq_\tau$ relations are extended to a *saturated-happens-before* relation `shb`, which extends $\leq_\tau \cup$ `rf` by orderings that must be respected by any witness of τ. If `shb` is cyclic, then τ is inconsistent.

3.2 Implementation

An implementation of RF-SMC is available in the tool NIDHUGG [1]. In this section, we describe that implementation.

NIDHUGG takes C or C++ programs as input, but does its analysis on the level of LLVM IR, produced by the Clang compiler. Executions are checked for assertion violations and crashes, such as segmentation faults. For programs that do not terminate in bounded time, and hence have an infinite trace space, automatic loop bounding, sometimes called loop unrolling, can be requested by the user. As with any bounding technique, this makes the exploration exhaustive only up to the given bound, which means that bugs may be missed if they do not manifest in any trace within the bound.

In order to do efficient trace equivalence comparison, as needed on line 12 of Algorithm 1, NIDHUGG maintains a directed graph which is the union of all $(\leq_\tau \cup \texttt{rf})$ graphs for all the traces in *schedules*. Each node is duplicated for every possible reads-from assignment and program-order predecessor node. Insertions are *interned*, which means that if, when inserting a node for some event with some predecessor set, there is already a node for that program event with the same predecessor set, that node is reused. Thus, to compare two traces identified by the nodes of their last events, it suffices to compare the nodes for reference equality, which is an $O(1)$ operation. In the source code of NIDHUGG, this graph is called the *unfolding tree*.

As an optimization, in addition to the *schedules* sets, NIDHUGG maintains a cache of traces that GETWITNESS has found to be inconsistent. Before querying GETWITNESS, it checks both $\mathit{schedules}(\tau'')$ and the cache of inconsistent traces, so that consistency is never queried for the same trace twice.

As an additional optimization, for every read event e_R in τ tried in the new-source-detection phase, NIDHUGG caches the `shb` graph for $\mathit{pre}(\tau', e_R)$. Then, when `shb` is needed for a new trace τ'', the longest prefix of τ'' for which there is an `shb` cached is found and reused, adding only the missing suffix of events and re-saturating. In order to efficiently support this use, NIDHUGG represents `shb` graphs using persistent immutable data structures that provide $O(1)$ copying and $O(\log n)$ updates.

4 Parallelization of READSFROM-SMC

In this section, we present PAR-RF-SMC, a parallel version of RF-SMC. While the sequential version is expressed in a recursive form, PAR-RF-SMC is expressed in a task-based form, where each task explores one trace and spawns zero or more new tasks. Algorithm 2 shows its code. The algorithm consists of creating an initial task PAR-RF-SMC$(\langle\rangle, \langle\rangle)$, and terminates when all tasks have finished.

Recall from Sect. 3 that in RF-SMC, the global data structure *schedules* serves two purposes. It keeps track of both all read sources that have been found so far for any read event, as well as of what trace prefixes, with associated executions, to explore in the future. In PAR-RF-SMC, the trace prefixes to explore in the future are kept as tasks and do not need to be stored as global variables, but the set of all sources found for some read event e_R in some trace τ is still required to avoid redundant (duplicate) exploration. Therefore, PAR-RF-SMC uses a variable $\mathit{attempted}(\mathit{pre}(\tau, e_R))$, shared by all tasks, to keep track of this set. This is the only shared data structure.

Algorithm 2: PAR-RF-SMC.

```
1  PAR-RF-SMC(τ, E)
2      extend E to a complete execution E · Ê where each event of Ê is unmarked
3      τ' := τ · Ê
4      for each read event e_R ∈ Ê do
5          attempted(pre(τ', e_R)) := {pre(τ', e_R) · e_R}
6      for each e_R, e_W ∈ τ' : e_W.var = e_R.var and e_W ≠ e_R.src and
7                               (e_R ∈ Ê or e_W ∈ Ê) and unmarked(e_R) and
8                               MAYREAD(τ', e_R, e_W) do
9          τ'' := pre(τ', e_R)
10         π := predecs(τ', e_W) ∩ post(τ', e_R)
11         σ := e_R[src := e_W] · mark(π)
12         if ¬∃σ' ∈ attempted(τ'') : σ' ≡ σ then
13             add σ to attempted(τ'')  // atomically with the test above
14             E'' := GETWITNESS(τ'' · σ, E · Ê)
15             if E'' ≠ ⟨⟩ then spawn PAR-RF-SMC(σ, E'')
16     join all sub-tasks
17     for each read event e_R ∈ Ê do erase attempted(pre(τ', e_R))
```

The algorithm of PAR-RF-SMC is structured in three phases: i) exploration (lines 2–3); ii) new-source-detection (lines 4–15); and iii) cleanup (lines 16–17). As can be seen, there is no recursive-exploration phase. Instead, new tasks are spawned for all new sources found in the new-source-detection phase, and then the cleanup phase deletes all the *attempted* sets that are no longer needed after all sub-tasks have finished.

The exploration phase is identical to that of RF-SMC. A trace prefix τ and execution E is extended to an arbitrary complete execution $E \cdot \hat{E}$ and corresponding trace τ', and correctness properties are checked.

The new-source-detection phase is very similar to that of RF-SMC. The differences lie in the changes to the global data structure *attempted*. The set *attempted* of possible sources for a read event is initialized on line 5, just like *schedules* in RF-SMC. Possible alternative sources for reads are looped over on line 6, however on line 12 we see the first difference. Before we invoke the potentially expensive consistency check GETWITNESS, we first check that the source e_W for e_R has not been previously attempted, by this or any other thread. This ensures that the algorithm never queries consistency for the same trace prefix twice. Thus lines 12 and 13 need to be executed atomically, for example with a mutex guarding $attempted(\tau'')$. Finally, if we did not find this source in $attempted(\tau'')$ and GETWITNESS found it to be consistent, we add the new trace to the work-queue on line 15.

Just like RF-SMC, PAR-RF-SMC satisfies the following three properties:
(i) *Soundness*: each complete trace explored by the algorithm is a consistent trace of the program.
(ii) *Completeness*: the algorithm explores all consistent traces of the program.
(iii) *Optimality*: each trace is explored exactly once.

Proof. We can establish these properties by observing that any run of PAR-RF-SMC can be rearranged to produce an equivalent run of RF-SMC. Every time the sequential

algorithm adds an element to *schedules* for future exploration, the parallel algorithm spawns a task with the same parameters, and vice versa. Every time the parallel algorithm runs a new task, the sequential algorithm makes a recursive call to itself. In order to be allowed to rearrange the execution of the parallel algorithm like this, we must show that it does not affect the set of traces explored. To do that, it suffices to look at the only source of scheduling non-determinism in the algorithm; the access to the *attempted* set. When two tasks both try to insert equivalent σ's into *attempted*, whichever of them "wins" and gets to insert into *attempted* is exactly the one that will run GETWITNESS and, if σ is consistent, the one that will spawn a task to explore it. After that, the system is in the same state, no matter which task "won". □

The order in which tasks are scheduled is not specified. The algorithm keeps its correctness and optimality properties with any scheduling, but depth-first and left-to-right policies minimize memory use. Work-stealing scheduling policies [7], where each thread has its own depth-first queue but when empty "steals" a shallowest task from another thread's queue, may also be employed to maximize locality while bounding the increase in memory use.

As was described at the end of Sect. 2, the scheduling of the arbitrary execution during the exploration phase on line 2 affects the amount of concurrency exposed to this algorithm. It is possible to devise a program where under one scheduling, PAR-RF-SMC would explore it entirely sequentially, and under another scheduling, would find all other traces by examining the first one it explores. However, we have neither encountered nor we expect realistic programs with large numbers of traces to behave this way.

5 Implementation

PAR-RF-SMC has been implemented in NIDHUGG. The language NIDHUGG is written in, C++, does not have a task-based scheduler in its runtime system. There are libraries that provide such functionality, but we chose to write our own work-stealing task scheduler. The scheduler detects when there are no more tasks in the queue or running, and terminates at that point. Figure 3 shows a diagram of the components of the implementation.

In the sequential implementation, the *schedules* sets are stored as a stack, one entry per read event. However, in our parallel version, the *attempted* sets cannot be stored in a stack. Instead, they are organized in a tree. This tree is effectively the exploration tree, as described in Sect. 2. Unlike the pseudocode of Algorithm 2, our implementation does not do explicit deletions of *attempted* sets, as on lines 16–17. Rather, nodes in this *attempted* tree are reference counted, and each task holds references to the attempted sets of all read events in their input traces. Every node in the tree has an associated mutex which is held during the atomic check-and-insert operation on lines 12–13. The *attempted* tree is shown in the middle of Fig. 3.

We preserve the unfolding tree data structure from sequential RF-SMC, but we extend it with mutexes that guard the child lists used for interning. For the special list of root nodes for each thread, we employ a readers-writer mutex due to high contention and high hit-rate (i.e., most queries for a root node return an interned node, and thus need not modify the list). The unfolding tree is shown in the bottom of Fig. 3.

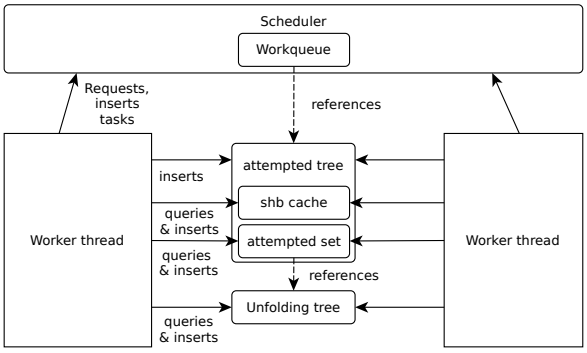

Fig. 3: Components of PAR-RF-SMC's implementation in NIDHUGG.

In sequential RF-SMC, the cache of `shb` graphs was represented using persistent immutable data structures [21], which had structural sharing that was memory managed through reference counting. These were stored along the sets on the *schedules* stack. For PAR-RF-SMC, we could simply move the cache into the nodes of the *attempted* tree, and ensure that write-accesses were properly synchronized. Luckily, the library that was used to provide these data structures was designed with threading in mind [21], and offered thread-safe atomic reference counting. However, as we were benchmarking our implementation, we found that the reference counting on these data structures became a scalability bottleneck. We were able to lift this bottleneck by redesigning these data structures to be more tailored to the needs of PAR-RF-SMC. In particular, because nodes in the *attempted* tree are always erased *after* all of their children, and because an `shb` graph is never updated after being installed into the cache, the memory management can be designed so that each piece of memory has exactly one owner, and thus does not require reference counting.

NIDHUGG supports early termination when it finds an error in the program, so naturally we wanted to support this in PAR-RF-SMC. We achieve this by telling the task scheduler to stop scheduling new tasks when an error is found. Thus, once all tasks that were running at that point terminate, so does the algorithm.

6 Performance and Scalability Evaluation

In this section, we report on the performance and scalability of NIDHUGG/rfsc. To put the numbers in perspective, we first compare its performance to three other SMC tools that implement state-of-the-art algorithms. Subsequently, we evaluate NIDHUGG/rfsc's scalability on a large multicore machine.

Tools Let us briefly present the SMC tools we compare against and the algorithms they employ. By VC-DPOR we refer to a prototype tool, based on NIDHUGG, that implements the recently proposed *Value-Centric Dynamic Partial Order Reduction* algorithm [9]. This algorithm, which is sensitive to the *values* used by the events during an execution rather than the read events themselves, in principle provides a coarser partitioning than

reads-from. However, neither the VC-DPOR algorithm nor its implementation provide any optimality guarantees and often explore —partially— considerably more executions than NIDHUGG/rfsc, as we will soon see. The second tool, CDSCHECKER [20], is a high-performance stateless model checker for C/C++11 programs. It employs a variant of the interleaving-based DPOR algorithm of Flanagan and Godefroid [11]. Although CDSCHECKER's implementation is well-engineered, the tool often explores a significant number of executions that are redundant, as this DPOR algorithm is not optimal. The last tool, GENMC [17], is a high-performance generic stateless model checker for concurrent C programs. As its algorithm is also graph-based, GENMC is the tool which is more similar to NIDHUGG/rfsc. However, rather than focusing on SC, GENMC provides a framework into which consistency checks for different (weak) memory models and program semantics can be plugged and even combined. GENMC offers a mode for SMC under `rf`-equivalence, which is the default, as well as a mode that tracks the modification order. We compare against the default mode of GENMC. In this mode, GENMC is optimal when consistency checks are not needed for SMC under SC. It is also faster than NIDHUGG/rfsc, both due to not checking consistency and due to being well-engineered.

Platform and Benchmarks Our benchmarking platform is a machine with two Intel(R) Xeon(R) Platinum 8168 CPUs (2.70GHz each with 24 cores and hyperthreading, giving a total of 48 physical/96 logical cores), has 192GB of RAM and ran Debian 10.3. All tools used Clang version 7.0.1 to translate the C source to LLVM IR. For benchmarks, we use the subset of programs from our previous paper [3] that can be handled by most tools and, more importantly for this paper, whose execution time is more than a few seconds, and hence their parallel execution makes sense. Refer to that paper for the programs' origin and characteristics, and to the artifact [4] of that paper for their sources.

Performance Table 1 shows the results: number of executions that the various tools explore and the time (in seconds) that this requires.[1] Since all these programs have a scaling parameter, often the number of threads involved, we show three rows for each. This allows to see the complexity of the different SMC algorithms and their scalability in terms of the number of executions explored as the state space increases. We notice the following:

– In terms of sequential performance, no tool is fastest overall. GENMC is fastest in the last six benchmarks where it is optimal and explores the same number of executions as NIDHUGG/rfsc. However, when it is not optimal and on circular-buffer, it is slower roughly by an order of magnitude compared to other tools (NIDHUGG/rfsc on fib-bench, VC-DPOR and NIDHUGG/rfsc on parker, and CDSCHECKER and NIDHUGG/rfsc on circular-buffer).

[1] In Table 1, entries n/a signify that the tool cannot handle that program; a ⏲ symbol that the benchmark does not complete after running for more than ten hours. The circular-buffer program contains a concurrency error which only manifests itself for parameter values ≥ 10. The CDSCHECKER tool finds this error immediately (within the first few executions), hence the † symbols for its circular-buffer(10) entries. The remaining three tools are not so lucky in their search, and catch the error after exploring many executions. The parallel version of NIDHUGG/rfsc detects this error at a point that is influenced by the distribution of tasks to threads, which also explains the slight variation in the curve of circular-buffer(10) in Fig. 4.

Table 1: Performance comparison of four SMC tools in terms of the number of executions that explore and the time (in secs) it takes to do so using one thread. The last column shows the time performance of parallel NIDHUGG/rfsc using 48 threads.

	CDSCHECKER		VC-DPOR		GENMC		NIDHUGG/rfsc		rfsc-48
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Time
fib-bench(4)	n/a	n/a	70937	6.93	34205	0.88	19605	1.66	0.15
fib-bench(5)	n/a	n/a	788940	87.69	525630	33.48	218243	21.28	0.76
fib-bench(6)	n/a	n/a	8543518	1182.25	8149694	3718.86	2364418	255.03	8.31
parker(12)	n/a	n/a	6601	0.92	69658	11.61	9407	2.24	0.13
parker(16)	n/a	n/a	11425	1.76	203754	43.60	21195	5.85	0.23
parker(20)	n/a	n/a	17561	3.09	475210	132.08	40087	12.73	0.41
circular-buffer(8)	12870	0.72	303149	50.44	12870	3.21	12870	2.51	0.15
circular-buffer(9)	48620	2.91	1147421	226.36	48620	13.58	48620	10.32	0.39
circular-buffer(10)	†	†	2964067	635.99	59279	19.13	59280	13.90	0.52
casrot(9)	372735	27.24	n/a	n/a	8597	0.08	8597	0.89	0.14
casrot(10)	3456845	284.27	n/a	n/a	38486	0.30	38486	4.28	0.23
casrot(11)	35407921	3230.99	n/a	n/a	182905	1.40	182905	22.39	0.89
lastzero(11)	184331	21.15	170515	33.09	7168	0.28	7168	1.13	0.13
lastzero(13)	1888624	255.84	1192108	317.12	32768	1.25	32768	5.89	0.26
lastzero(15)	19478080	3057.60	8264353	3061.91	147456	6.25	147456	30.76	1.00
readers(13)	13311	1.75	67108864	21224.93	8192	0.59	8192	1.25	0.14
readers(15)	53247	8.10	◷	◷	32768	2.46	32768	5.81	0.26
readers(17)	212991	37.24	◷	◷	131072	10.94	131072	25.23	0.89
sigma(7)	509861	48.08	46232	4.97	5040	0.23	5040	0.52	0.11
sigma(8)	9057756	977.89	409112	56.27	40320	1.67	40320	4.40	0.28
sigma(9)	180337837	22286.21	4037912	668.64	362880	15.94	362880	44.17	2.17
race-parametric(5)	34904	12.41	14967	3.92	8953	1.04	8953	3.60	0.21
race-parametric(6)	372436	134.75	88432	26.38	73789	8.24	73789	30.35	0.96
race-parametric(7)	4027216	1479.37	591352	209.40	616227	69.59	616227	255.43	7.69
approxds-append(5)	390728	25.69	121883	11.36	9945	0.60	9945	1.72	0.15
approxds-append(6)	30603290	2425.28	5353219	622.40	198936	12.83	198936	41.45	1.21
approxds-append(7)	◷	◷	◷	◷	4645207	342.52	4645207	1143.28	34.86

- VC-DPOR explores significantly less executions only on one program (parker) and only a few less on race-parametric(7). It is faster than the other tools only on parker. In the remaining seven programs, it examines a big number of partially explored executions —on readers even exponentially more!— and its numbers explode.
- The performance of the sequential NIDHUGG/rfsc is quite decent, but GENMC is 2.4 to five times faster than NIDHUGG/rfsc in the last four benchmarks where both tools explore the same number of executions. Also, in the casrot benchmark, GENMC is an order of magnitude faster. However, both tools scale similarly and better than the other two.
- When parallelism enters the picture, NIDHUGG/rfsc becomes the fastest tool across the board. (The last column of Table 1 shows times when executing with 48 threads, which is the number of physical cores in our machine.) Note that this would not have been possible if NIDHUGG/rfsc were examining a significant number of redundant executions (e.g., similar to those that CDSCHECKER or VC-DPOR often explore).

Scalability Let us now examine the scalability of PAR-RF-SMC compared to its sequential counterpart as implemented in NIDHUGG/rfsc. Figure 4 shows speedups obtained

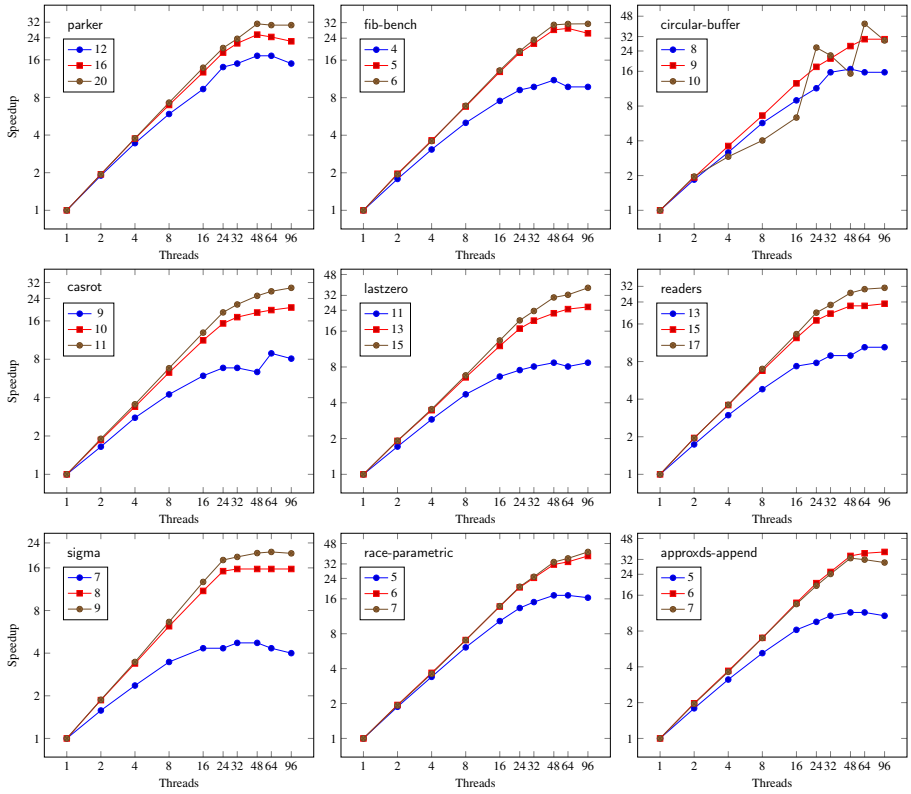

Fig. 4: Speedups (y-axes) obtained by running the benchmarks when varying the number of threads (x-axes) on a machine with 48 physical/96 logical cores.

for executions of all benchmark/parameter combinations. All graphs show a very similar picture. The speedup is almost linear up to 24 threads, which is the number of physical cores per chip on this machine, and becomes on average 32× when using all 48 physical cores (which are located on two different NUMA nodes). In most programs, there is a moderate speedup increase above 32× when hyperthreading is also used.

Two more points worth noting are that i) the speedups are in general highest for the benchmark configuration with the largest parameter value (brown lines in the plots), and ii) the speedup obtained for the configuration with the smallest parameter value (blue lines) often drops when using more threads than physical cores. This is due to threads not having much work to do after some point in time during the execution of the benchmark and/or trying to steal from other threads, causing memory traffic.

7 Related Work

To address the inherent complexity of testing concurrent software, researchers have developed a variety of methods for finding and reproducing concurrency errors. In the area of stateless model checking [12] numerous tools and research prototypes [19, 10,

1, 20, 16, 17] have been developed in the last decade, and SMC has been successfully applied to important concurrent programs (e.g., [13, 18]).

In recent years, a wide variety of SMC algorithms has been put forward (e.g., [11, 2, 26, 22, 5, 16, 8, 6, 9, 3]) with the aim to effectively combat the combinatorial explosion in the number of executions that must be explored. However, only a selected few of them [2, 6, 17, 3] come with optimality guarantees, and none of them has been parallelized. To the best of our knowledge READSFROM-SMC is the first *optimal* algorithm for SMC with a parallel implementation.

Still, non-optimal Dynamic Partial-Order Reduction (DPOR) algorithms have been parallelized in the past (e.g., by Yang et al. [25] and by Simsa et al. [23]), although the focus of those works has been on obtaining *distributed* versions of these algorithms rather than algorithms suitable for running on multicores. Also, their focus has been on techniques and heuristics on how to avoid situations where different workers end up exploring identical (N.B. not just from the same equivalence class!) parts of the search space, due to the non-local nature in which interleaving-based DPOR algorithms update their exploration frontier and the need, for scalability, to avoid a central coordinator.

Of course, distributed execution and parallelization of explicit state model checkers has also been investigated (e.g., [24, 14, 15]). Stateful exploration is less common for software model checking and often suffers from memory explosion.

8 Concluding Remarks

We have presented PAR-RF-SMC, the parallel version of a state-of-the-art graph-based SMC algorithm for SC. The algorithm retains its main properties (soundness, completeness and, most importantly, optimality), can be implemented with moderate additional effort on top of its sequential counterpart, and achieves very good scalability; on average 32 times speedup on a 48 core machine. Our performance evaluation shows that parallel NIDHUGG/rfsc currently outperforms all tools in its area, and offers the possibility for SMC to be applied to programs which are currently very challenging.

Acknowledgments. We would like to acknowledge the work of Nodari Kankava and Alexis Remmers for an initial prototype implementation of the algorithm which formed the basis for PAR-RF-SMC's implementation in NIDHUGG. This work has been partially supported by the Swedish Research Council through grant #621-2017-04812, and by the Swedish Foundation for Strategic Research through the aSSIsT project.

References

[1] Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 9035, pp. 353–367. Springer, Berlin, Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-46681-0_28

[2] Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: A foundation for optimal dynamic partial order reduction. Journal of the ACM 64(4), 25:1–25:49 (Sep 2017), http://doi.acm.org/10.1145/3073408

[3] Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. 3(OOPSLA), 150:1–150:29 (Oct 2019), https://doi.org/10.1145/3360576

[4] Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal Stateless Model Checking for Reads-From Equivalence under Sequential Consistency (Oct 2019), https://doi.org/10.5281/zenodo.3401442, artifact for the OOPSLA 2019 paper with the same title

[5] Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Computer Aided Verification. LNCS, vol. 10426, pp. 526–543. Springer, Berlin Heidelberg (Jul 2017), https://doi.org/10.1007/978-3-319-63387-9_26

[6] Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference. LNCS, vol. 10806, pp. 229–248. Springer, Cham (Apr 2018), https://doi.org/10.1007/978-3-319-89963-3_14

[7] Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM 46(5), 720–748 (Sep 1999), https://doi.org/10.1145/324133.324234

[8] Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. Proc. ACM on Program. Lang. 2(POPL), 31:1–31:30 (Jan 2018), http://doi.acm.org/10.1145/3158119

[9] Chatterjee, K., Pavlogiannis, A., Toman, V.: Value-centric dynamic partial order reduction. Proc. ACM Program. Lang. 3(OOPSLA), 124:1–124:29 (Oct 2019), https://doi.org/10.1145/3360550

[10] Christakis, M., Gotovos, A., Sagonas, K.: Systematic testing for detecting concurrency errors in Erlang programs. In: Sixth IEEE International Conference on Software Testing, Verification and Validation. pp. 154–163. ICST 2013, IEEE, Los Alamitos, CA, USA (Mar 2013), https://doi.org/10.1109/ICST.2013.50

[11] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of Programming Languages, (POPL). pp. 110–121. ACM, New York, NY, USA (Jan 2005), http://doi.acm.org/10.1145/1040305.1040315

[12] Godefroid, P.: Model checking for programming languages using VeriSoft. In: Principles of Programming Languages, (POPL). pp. 174–186. ACM Press, New York, NY, USA (Jan 1997), http://doi.acm.org/10.1145/263699.263717

[13] Godefroid, P., Hanmer, R.S., Jagadeesan, L.: Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 124–133. ISSTA, ACM, New York, NY, USA (Mar 1998), https://doi.org/10.1145/271771.271800

[14] Holzmann, G.J., Bosnacki, D.: The design of a multicore extension of the SPIN model checker. IEEE Trans. Softw. Eng. 33(10), 659–674 (Oct 2007), https://doi.org/10.1109/TSE.2007.70724

[15] Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification techniques. IEEE Trans. Softw. Eng. 37(6), 845–857 (Nov 2011), https://doi.org/10.1109/TSE.2010.110

[16] Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM on Program. Lang. 2(POPL), 17:1–17:32 (Jan 2018), https://doi.org/10.1145/3158105

[17] Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 96–110. PLDI 2019, ACM, New York, NY, USA (Jun 2019), https://doi.org/10.1145/3314221.3314609

[18] Kokologiannakis, M., Sagonas, K.: Stateless model checking of the Linux kernel's read–copy update (RCU). International Journal on Software Tools for Technology Transfer 21(3), 287–306 (Jun 2019), https://doi.org/10.1007/s10009-019-00514-6

[19] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. pp. 267–280. OSDI '08, USENIX Association, Berkeley, CA, USA (Dec 2008), http://dl.acm.org/citation.cfm?id=1855741.1855760

[20] Norris, B., Demsky, B.: A practical approach for model checking C/C++11 code. ACM Trans. Program. Lang. Syst. 38(3), 10:1–10:51 (May 2016), http://doi.acm.org/10.1145/2806886

[21] Puente, J.P.B.: Persistence for the masses: RRB-vectors in a systems language. Proc. ACM Program. Lang. 1(ICFP) (Aug 2017), https://doi.org/10.1145/3110260

[22] Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. In: 26th International Conference on Concurrency Theory (CONCUR 2015). LIPIcs, vol. 42, pp. 456–469. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (Aug 2015), http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.456

[23] Simsa, J., Bryant, R., Gibson, G.A., Hickey, J.: Scalable dynamic partial order reduction. In: Runtime Verification, Third International Conference, RV 2012. LNCS, vol. 7687, pp. 19–34. Springer (Sep 2012), https://doi.org/10.1007/978-3-642-35632-2_4

[24] Stern, U., Dill, D.L.: Parallelizing the murϕ verifier. Formal Methods in System Design 18, 117–129 (Mar 2001), https://doi.org/10.1023/A:1008771324652

[25] Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Distributed dynamic partial order reduction based verification of threaded software. In: Model Checking Software, 14th International SPIN Workshop. LNCS, vol. 4595, pp. 58–75. Springer (Jul 2007), https://doi.org/10.1007/978-3-540-73370-6_6

[26] Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: Programming Language Design and Implementation (PLDI). pp. 250–259. ACM, New York, NY, USA (Jun 2015), http://doi.acm.org/10.1145/2737924.2737956