Parallel Refinement for Multi-Threaded Program Verification

***Abstract*—Program verification is one of the most important approaches to ensuring the correctness of concurrent programs. However, due to the path explosion problem, concurrent program verification is usually time consuming, which hinders the scalability to industrial programs. Parallel processing is the mainstream technique to deal with those problems which require mass computing. Hence, designing parallel algorithms to improve the performance of concurrent program verification is highly desired. This paper focuses on parallelization of the abstraction refinement technique, one of the most efficient techniques for concurrent program verification. We have proposed a parallel refinement framework which employs multiple engines to refine the abstraction in parallel. Different from existing work which usually parallelizes the search process, our method achieves the effect of parallelization by refinement constraint and learnt clause sharing, such that the number of required iterations can be significantly reduced and the performance of constraint solving can be improved. We have implemented this framework on the scheduling constraint based abstraction refinement method, one of the best methods for concurrent program verification. Experiments on SV-COMP 2018 show the encouraging results of our method. For those complex programs requiring a large number of iterations, our method can obtain a linear reduction of the iteration number and significantly improve the verification performance.**

***Index Terms*—concurrent program, abstraction refinement, scheduling constraint, parallel verification**

I. Introduction

Facilitated by the popularization of multi-core architectures, concurrent programs are becoming popular to take full advantage of the available computing resources. However, due to the nondeterministic thread interleavings, traditional approaches such as testing and simulation are hard to guarantee the correctness of such programs. Automatic program verification has become an important complementary to traditional approaches, as it automatically explores all the program behaviours to check the correctness.

However, due to the path explosion problem, concurrent program verification is usually time consuming, which significantly limits its scalability to industrial programs. Parallel processing is the mainstream technique to address those problems which require mass computing. Leveraging the availability of large number of processors may be a potential method to scale up the ability of verification. Hence, designing parallel verification algorithms to take full advantage of the hardware resources is highly desired.

Recently, researchers have tried to employ either multiprocessor architectures or distributed systems to do SAT solving, model checking, and program verification. Unfortunately, these are reasoning problems intrinsically, which are difficult to parallelize. For these, the subsequent analyses heavily depends on the previous ones, and it is difficult to partition the whole task into a set of independent sub-tasks which can be solved in parallel. In addition, optimizations tailored for sequential method is usually difficult to be ported into the parallel algorithms. As a result, the parallel method may even perform worse than the sequential method in some cases.

Scheduling constraint based abstraction refinement (SCAR) [1] is one of the most efficient methods for concurrent program verification. The tool implementing this method has won the gold medals in the ConcurrencySafety category of both SV-COMP 2017 and 2018 [2], [3]. It employs abstraction refinement to avoid the large formula in bounded model checking (BMC) of concurrent programs. Particulary, observed that the scheduling constraint usually dominates the monolithic encoding, it initially ignores the scheduling constraint and obtains an over-approximation abstraction of the original program, which is much smaller. If a counterexample is found, it is further validated and the abstraction is refined if the counterexample is infeasible. In this manner, a concurrent program is verified via dozens or hundreds of small constraint solving problems rather than a large monolithic problem. The authors have proved that this method is both sound and complete with respect to the given loop unwinding depth.

This paper aims to parallelize the SCAR method for multi-threaded program verification. Observing that the dependence among those refinements of different iterations is weak, we propose to perform those refinements concurrently. To realize this idea, we have devised a parallel refinement framework which employs multiple engines to refine the abstraction in parallel. Existing work of parallel verification usually partitions the search space into small ones, which are then searched concurrently by multiple engines. Different from these work, in our method, each engine performs abstraction refinement on the whole abstraction space. To make different engines perform diverse refinements, we employ a random searching strategy for each engine to make them refine different parts of the search space. Then we let all engines share their refinement constraints with each other. In this manner, each engine can obtain multiple refinement constraints in each iteration, such that the number of required iterations for each engine can be reduced. Ideally, this number may decrease linearly with the increase of engines, and the verification performance may be significantly improved compared with the sequential method. Moreover, observing that different engines solve the same verification problem, the learnt clauses for different engines can also be shared to improve the performance of constraint

solving, such that the verification performance can be further improved.

To the best of our knowledge, we are the first to perform a parallel abstraction refinement for concurrent program verification. Our method avoids the load balance problem of existing parallel verification techniques, and it can obtain a linear reduction of the required iterations with the increase of engines. Actually, our method is not limited to the SCAR method or multi-threaded program verification, it is a general method which can be easily extended to other abstraction refinement techniques. Moreover, given that we do not care the internal search process, all optimizations on the sequential method can be easily applied to our method. We have implemented our method on top of YOGAR-CBMC and evaluated it on the benchmarks in the ConcurrencySafety category of SV-COMP 2018. Experimental results show the encouraging results of our method. For those examples which require a large number of iterations, our method can obtain a linear reduction of the iterations and significantly improve the verification performance.

The contributions of this paper are listed as follows.

1) We have proposed a parallel refinement framework for program verification, which employs multiple engines to refine the abstraction diversely in parallel.
2) We have provided a set of techniques for diverse refinement and have implemented the parallel refinement framework on the SCAR method.
3) We have evaluated our method on the concurrency benchmarks of SV-COMP 2018. Experimental results show encouraging performance of our method.

The rest of this paper is organized as follows. Section II reviews the SCAR method for multi-threaded program verification. Section III introduces our parallel refinement method and discusses its efficiency and applicability. Section IV presents the algorithms and implementation details of parallelizing the SCAR method. Section V provides the experimental results. Section VI reviews the related work, and Section VII concludes the paper.

II. THE SCAR METHOD FOR MULTI-THREADED PROGRAM VERIFICATION

A. Multi-Threaded Program

A *multi-threaded* program consists of multiple concurrent threads. It contains a set of variables which are partitioned into *shared variables* and *local variables*. Each thread can read/write both the shared variables and its local variables. We focus on programs based on PThreads, one of the most popular libraries for multi-threaded programming. It uses `pthread_create(&t, &attrib, &f, &args)` to create a new thread `t`, and `pthread_join(t, &_return)` to suspend the current thread until thread `t` terminates[1]. In this paper, we assume each variable access is atomic. We also assume all functions are inlined and all loops are unwound by a bounded depth. Given a multi-threaded program, an *event* e is a read/write access to a shared variable. We use $\mathrm{var}(e)$ to denote the accessed variable of e. A read-write link (e_1, e_2) represents that "e_2 reads the value written by e_1". Therefore, e_1 is a write, e_2 is a read, $\mathrm{var}(e_1) = \mathrm{var}(e_2)$, and the value of e_2 is equal to that of e_1. In addition, there should be no other write of v happening between them.

Bounded model checking (BMC) is one of the most applicable techniques to alleviate the path explosion problem of concurrent programs. Instead of explicitly enumerating all thread interleavings, it employs a symbolic representation to encode the verification problem, which is then solved by a SAT/SMT solver. If a positive answer is given, then a satisfying assignment corresponding to a feasible counterexample is acquired. Otherwise, the program is proven safe w.r.t. the given loop unwinding depth.

In BMC, a multi-threaded program is usually encoded as a monolithic encoding $\alpha := \phi_{init} \wedge \rho \wedge \zeta \wedge \xi$, where ϕ_{init} is the initial states, ρ encodes each thread in isolation, ζ formulates that "each read of a variable v may read the result of any write of v", and ξ formulates the scheduling constraint, which defines the execution order requirements among all events of the program. The ξ is constituted of two parts: ξ^{po} and ξ^{rf}, where ξ^{po} requires that "the intra-thread execution order among those events should be consistent with the program order", and ξ^{rf} defines that "for any read-write link (e_1, e_2), there should be no other write of $\mathrm{var}(e_1)$ happening between them". A problem for BMC of multi-threaded programs is that the monolithic encoding usually generates a large formula which overwhelms modern constraint solvers.

B. The SCAR method

To avoid the large formula in BMC of concurrent programs, L. Yin et al. have proposed a scheduling constraint based abstraction refinement (SCAR) method for multi-threaded program verification, which can avoid the large formula [1]. Given a multi-threaded C program, observed that the encoding of the scheduling constraint ξ usually accounts for most of the monolithic encoding α, the SCAR method first ignores the scheduling constraint in the encoding. An over-approximated abstraction of the original program is then obtained as $\varphi_0 := \phi_{init} \wedge \rho \wedge \zeta$, which is much smaller than the monolithic encoding α. The initial abstraction φ_0 and the error states ϕ_{err} are then added to the abstraction model. If it is unsatisfiable, then the property is proven safe w.r.t. the given loop unwinding depth. Otherwise, a counterexample of the abstraction is provided. Given that the scheduling constraint ξ is ignored in the abstraction, this counterexample may be infeasible and further validation is required. If the counterexample is determined to be feasible, then a true counterexample is found and an execution trace to the error states can be returned. If the counterexample is determined to be infeasible, one should continue to search the rest space of the abstraction until either a feasible counterexample is found or no abstraction counterexample can be found any more. In each iteration, to prevent the infeasible counterexample from appearing again

[1]More information about PThreads can be found at https://computing.llnl.gov/tutorials/pthreads/

in future search, the counterexample should be negatived and formulated as a constraint to refine the abstraction. To prune more search space rather than just one counterexample, the kernel reasons of the counterexample are analyzed and formulated as a refinement constraint.

The efficiency of the SCAR method depends on the obtained refinement constraints. Ideally, they should have small sizes yet prune a large amount of search space. To obtain effective refinements, a notion of event order graph (EOG) is proposed and two graph-based algorithms over EOG have been devised in the SCAR method for counterexample validation and refinement generation, in which a small yet effective refinement can usually be obtained if the counterexample is infeasible.

Fig. 1 schematically depicts the general idea of the SCAR method. Given a multi-threaded program, we use A, E, and P to denote the abstraction space, the exact program space, and the property space where the property holds, respectively. Given that A is an over-approximation abstraction of E, we must have $\mathtt{E} \subseteq \mathtt{A}$. The property is said to be false on the program iff $\mathtt{E} \setminus \mathtt{P}$ is not empty, i.e., there exists some state of the program where the property does not hold. For the example shown in Fig. 1, it is obvious that the property is false in the program. However, given that an exact encoding of E requires a large formula, the problem of verifying whether the property is true on all states in E is usually difficult in practice.

Observed that the exact encoding of a multi-threaded program is dominated by the scheduling constraint, the SCAR method initially ignores the scheduling constraint and obtains an over-approximation abstraction A of the program, the encoding of which is much smaller. Then, the SCAR method verifies whether the property is true on all states in A. If it does, then the property must also be true on all states in E. Otherwise, if there exists some state $s \in \mathtt{A} \setminus \mathtt{P}$, further validation of whether $s \in \mathtt{E}$ is performed. If it is, then a true counterexample is found. Otherwise, s is a spurious counterexample introduced by the abstraction. The SCAR method analyzes kernel reasons making s infeasible, and negatives them as a refinement constraint to prune the search space of the abstraction A. Given that the kernel reasons must be implied by the program, the pruned space must contain no state of E. The rest of A must always be a superset of E.

The SCAR method repeats these steps of abstraction verification, counterexample validation, and refinement constraint generation, until either a true counterexample is found or the property holds on all rest states of A. As shown in Fig. 1, it may first find the counterexample ①. Given that ① is not in E, the space corresponding to the refinement constraint of ①, represented by the dotted ellipse containing ①, is pruned from A. Analogously, in the following iterations, the counterexamples ② to ⑦ are detected, and the spaces corresponding to their refinement constraints are pruned from A, until the true counterexample ⑧ is found. In this manner, the SCAR method converts the monolithic verification problem into a serial of small problems. Though the authors have tried their best to prune as much search space as possible in each iteration, for those complex programs, a large number of iterations may still be required to verify the program.

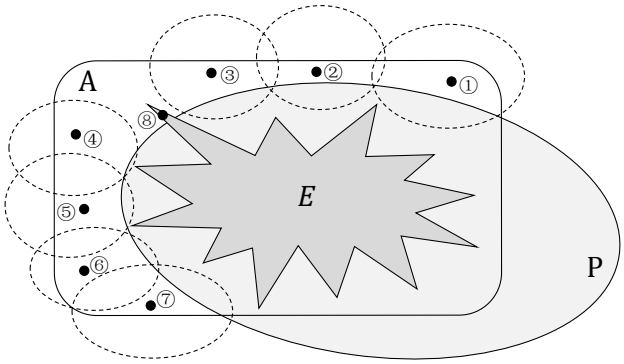

Fig. 1. General idea of the SCAR method.

III. Methodology of Parallel Refinement

From above introduction, in the SCAR method, each refinement prunes some part of the abstraction space. The dependence between subsequent refinements and previous ones is weak. It is just used to guarantee that different refinements prune different parts of the space. If this can be achieved by some other scheme, the refinements of different iterations can be performed concurrently, such that the number of required iterations for each engine can be reduced. Consider the example shown in Fig. 1. Suppose that four engines are used to refine the abstraction in parallel. In the first iteration, if the four engines obtain the counterexamples ① to ④ respectively, then we can prune those search spaces corresponding to the four counterexamples with just one iteration. In this manner, we may detect the feasible counterexample ⑧ with just two or three iterations.

A. The Parallel Refinement Framework

Based on this observation, we proposed the idea of parallel refinement for multi-threaded program verification. We have devised a parallel refinement framework which employs multiple engines to refine the abstraction in parallel. The framework is shown in Fig. 2. Given a multi-threaded program, same as the sequential method, it first employs a *Program Abstract* module to obtain the initial over-approximation abstraction φ_0. The property to be verified is encoded as ϕ_{err}. The sequential method iteratively verifies the abstraction, validates the counterexample (CE), and analyzes the refinement constraint (RC). In our parallel method, multiple engines are employed to perform the three steps individually on the whole abstraction space. Whenever an engine generates a refinement constraint, the engine shares the constraint with other engines and obtains all the refinement constraints generated by other engines. In this manner, at each iteration, each engine can obtain those refinement constraints generated by all engines, such that the number of required iterations for each engine can be significantly reduced.

The SCAR method uses constraint solving to verify whether the property is true on the abstraction. In this step, if any

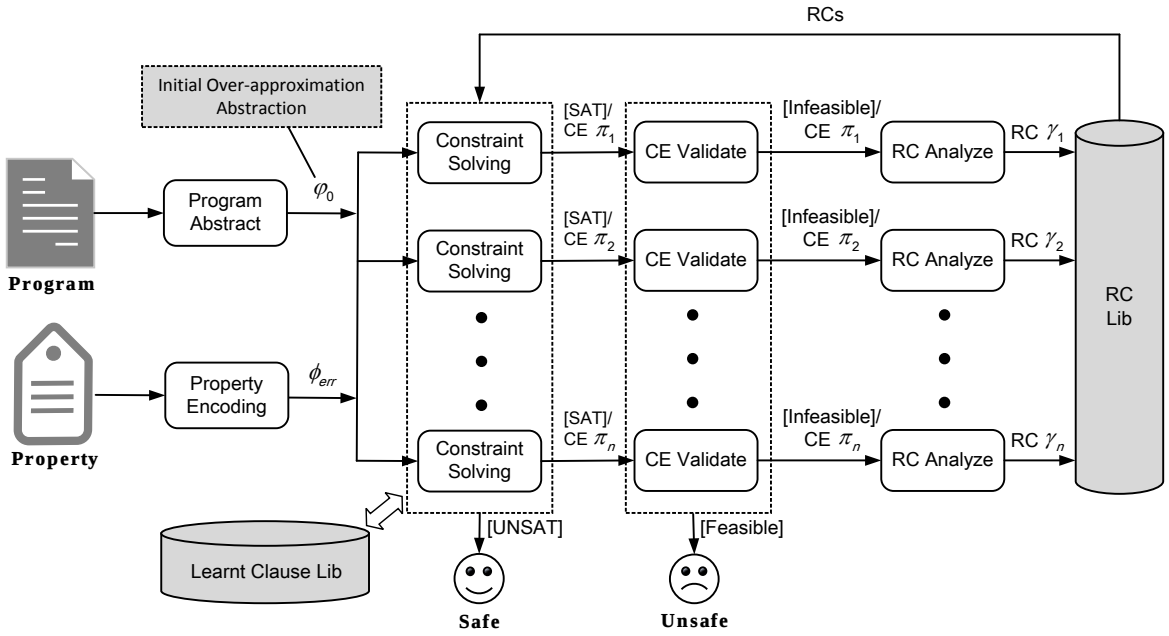

Fig. 2. Framework of parallel refinement

engine returns UNSAT, i.e., it claims that the property is true on its abstraction, then we can conclude that the property is safe on the original program. All engines will terminate immediately in this case. Here, we require that the initial abstraction φ_0 is an over-approximation abstraction of the original program. Furthermore, all refinement constraints should can be implied by the original program. As a result, all abstractions are over-approximations of the original program. Hence, we can conclude that the property is safe on the original program if some engine returns UNSAT.

If all engines return SAT, then each of them obtains a counterexample of its abstraction. Denote them by $\pi_1, \pi_2, \cdots, \pi_n$, respectively. To guarantee that the refinement constraints generated by different engines refine different parts of the search space, we require the n counterexamples vary significantly from each other, even though the abstraction problem may be the same for all engines. This is the critical step that guarantees the efficiency of our parallel refinement method. If the generated counterexamples are similar with each other, then the number of required iterations for each engine may be similar with the sequential method. However, if the generated counterexamples uniformly distribute on the search space, then the number of required iterations for each engine may decrease linearly with the increase of engines. In our method, we achieve this goal by a random search strategy.

In the counterexample validation step, each engine validates its counterexample individually. If some engine claims that its counterexample is feasible, then a true counterexample of the original program has been found. This engine will give an execution trace to the error states and notify all other engines that a true counterexample has been found. All engines will terminate immediately and our method will return `Unsafe`.

If all counterexamples are infeasible, each engine analyzes the refinement constraint corresponding to its counterexample individually. Denote them by $\gamma_1, \gamma_2, \cdots, \gamma_n$, respectively. All of them are then stored in a shared refinement constraint library (RC Lib), which can be accessed by all engines. Whenever a constraint is added, the library will be simplified to avoid redundant constraints. Then, all the new generated constraints are added to the abstractions of all engines. In this manner, in each iteration, each engine can obtain those refinement constraints generated by all engines, such that the number of required iterations for each engine can be reduced.

The SCAR method uses Minisat-2.2.0 as the constraint solver, which keeps a set of learnt clauses to prevent redundant search. In the SCAR method, those iterations of all engines solve the same problem, that is, whether the property is true on the original program. Hence, all learnt clauses generated by one engine can be used to help other engines avoid searching the same space again. To realize this idea, we let different engines share learnt clauses with each other. Particularly, we use a shared learnt clause library (Learnt Clause Lib) to maintain those shared clauses. Each engine periodically writes its learnt clauses to the library and read those clauses shared by other engines. In our method, the learnt clauses generated in an iteration of some engine can be used in the same iteration of the same engine, the same iteration of other engines, latter iterations of the same engine. These are the techniques of conflict clause learning, parallel SAT solving, and incremental SAT solving, respectively. The learnt clauses can also be used in latter iterations of other engines, and we call it the cross-engine sharing technique. Hence, with the learnt clause sharing strategy, our method integrates

four techniques together: conflict clause sharing, parallel SAT solving, incremental SAT solving, and cross-engine sharing.

B. Efficiency and Applicability

1) Efficiency: Our method can improve performance of the SCAR method in three aspects.

First, benefited from the refinement constraint sharing strategy, the number of required iterations for each engine can be significantly reduced. Ideally, given n engines, the number of required iterations may decrease to $1/n$, and our method may perform n times faster than the sequential method.

Second, benefited from the learnt clause sharing strategy, we have realized four techniques to improve the performance of constraint solving. Given that the time of each iteration stems mainly from constraint solving, our method can further improve the verification performance.

Third, as we all know, luck plays an important role in both refinement constraint generation and constraint solving. By sharing of refinement constraints and learnt clauses, our method puts all engines' good luck together. Therefore, remarkable improvement may be observed even without the first and second improvements.

The experimental results in Section V show that, for those complex programs which require a large number of iterations, the verification performance may be significant improved.

2) Applicability: Applicability is another important aspect to evaluate a method. From this point of view, our method enjoys the following advantages.

First, our parallel refinement framework is not confined to the SCAR method or multi-threaded program verification. It is a general framework which may be easily applied to other abstraction refinement methods which satisfy the following two requirements: a) All the obtained abstractions are over-approximations of the original program. This requires that the initial abstraction is an over-approximation, and all the generated refinement constraints are implied by the original program. b) Different engines refine different parts of the abstraction without search space partition. This requires that there exists some scheme to guarantee that different engines generate different counterexamples even for the same abstraction problem.

Second, all optimizations on the sequential method can be easily applied to our method. To improve the verification performance, many optimizations have been proposed on the sequential method, such as optimizations on abstraction generation, abstraction verification, counterexample validation, and refinement constraint generation, etc. In our method, each engine performs an individual abstraction refinement. All optimizations on the sequential method can be extended to our method by simply applying them to each engine. Similarly, our method can be applied to multi-threaded programs under both sequential consistency and weak memory models.

Third, our method is suitable for both true and false properties. From the framework, our method employs multiple engines to refine the abstraction in parallel. As long as the verification requires many iterations, our method may improve the performance by reducing the number of required iterations. Even for those programs require just several iterations, a better performance can also be obtained via improving the efficiency of constraint solving.

Last but not least, our method can be applied to both multi-processor architectures based on shared memory and distributed systems based on distributed memory. For shared memory systems, both the refinement constraint and learnt clause libraries can be implemented by the shared memory which can be accessed by all engines. For distributed systems, to reduce the communication overhead, one can disable the learnt clause sharing strategy and share only those refinement constraints, the sizes of which are usually very small.

IV. Algorithms and Implementation

A. Parallelization of the SCAR Method

We have implemented our method on multi-processor architectures based on shared memory. Each engine of the framework is implemented by a process. The communication among engines is realized by the shared memory and semaphore schemes of Linux processes. Shared memory is used to realize the sharing of refinement constraints and learnt clauses, while semaphore is utilized to synchronize different processes. Particularly, we use Linux APIs like `ftok()`, `shmget()`, `shmat()`, `shmdt()`, and `shmctl()` to implement the shared memory scheme, and we use Linux APIs like `semget()`, `semctl()`, `semop()` to implement the semaphore scheme.

Algorithm 1 demonstrates the master process of our parallel refinement method. The inputs include a multi-threaded program P which represents the program to be verified, and an integer n which is the number of parallel engines. In this paper, we focus on assertion properties which have been contained in the program already. The algorithm returns `Safe` if the property is not violated, `Unsafe` otherwise. If the property is violated, it can further provide an execution trace of P corresponding to the counterexample.

Algorithm 1 The Master Process

Require: The multi-threaded program P to be verified, and the number of parallel engines n;
Ensure: `Safe` if the property is true, `Unsafe` otherwise;
1: $CreateShareRes(sResult, sRefinements, sClauses)$;
2: $cmm = SingleProcessCmm(P)$;
3: **for** $i = 1$ to n **do**
4: $\quad SystemExec(cmm)$;
5: **end for**
6: $WaitForProcessFinish()$;
7: $result = sResult$;
8: $DeleteShareRes(sResult, sRefinements, sClauses)$;
9: **return** $result$;

At the beginning and end of our algorithm (Line 1 and 8), we use the $CreateShareRes$ and $DeleteShareRes$ functions to create and release the shared resources, respectively. Here, $sResult$, $sRefinements$ and $sClauses$ stand for the

verification result, the shared refinement constraints, and the shared learnt clauses, respectively. The shared resources also contain those semaphore resources which are used for process synchronization. Let cmm be the command that starts a single process for program verification (Line 2). We start n different processes in parallel which use the same command to verify the same program individually (Line 3-5). Different from the framework, in the implementation, to reduce the coupling degree of different processes, we let each engine perform a single program abstraction. The master process then waits until all the n processes have terminated (Line 6). To this end, a `Safe` or `Unsafe` result, which has been written in $sResult$, must have been obtained by some process. The algorithm returns this result and then terminates.

Algorithm 2 demonstrates the routine of a parallel process. At the beginning and end of this algorithm (Line 1 and 24), it uses the $BindShareRes$ and $UnbindShareRes$ functions to bind and unbind those shared resources, respectively, which are created by the master process. Then same as the sequential method, it obtains the initial over-approximation abstraction φ_0, and adds $\varphi_0 \wedge \phi_{err}$ to its solver, where ϕ_{err} represents the property (Line 2-4). A loop statement is then employed to perform the iterative abstraction refinements.

Before the start of each iteration, the algorithm first checks whether $sResult \neq unknow$. If it is true, then some process has obtained a `Safe` or `Unsafe` result and the algorithm terminates immediately (Line 6-8). If no conclusion has been made, the algorithm verifies the current abstraction (Line 9). If the returned value is UNSAT, both the abstraction and the original program is safe. We use the $WriteResult$ function to write this result to the shared memory $sResult$ (Line 10-13), which is monitored by other processes. If a SAT is returned, then a counterexample π of the abstraction can be obtained from the constraint solver (Line 14). The function $CEValidate$ is used to validate the feasibility of π. If it is feasible, then a true counterexample has been found and the original program is unsafe (Line 15). We again write this result to $sResult$ (Line 16). The $TracePrint()$ function is used to print the execution trace corresponding to π (Line 17). If π is infeasible, the function $RCGenerate$ is then used to generate the refinement constraint γ corresponding to π (Line 20). Then, we use the $RCShare$ function to write γ to the shared memory $sRefinements$, and read all those newly generated refinement constraints of other processes (Line 21). Let γ' be the set of obtained refinement constraints, which also contains γ. Then γ' is added to $solver$ to prune the abstraction (Line 23). From this algorithm, a parallel process terminates if a `Safe` or `Unsafe` result is obtained by either this process or other ones.

For learnt clause sharing, same as the SCAR method, we use Minisat-2.2.0 as our SAT solver. In most SAT solvers like Minisat, when the number of conflicts reaches some threshold, a restart strategy is employed to randomly try another variable decision order. Hence, the search process usually consists of multiple restarts. In our implementation, we use a function $LCShare()$ before each restart to share those newly generated learnt clauses of current process and read those of other processes. Given that each process may generate a large number of learnt clauses, sharing all of them may decrease the performance sometimes. As have done in [4], we employ an adaptive heuristic to share only clauses whose length is smaller than the continuously recalculated average length of all learnt clauses.

Algorithm 2 A Parallel Process

Require: The program P to be verified;
1: $BindShareRes(sResult, sRefinements, sClauses)$;
2: $solver = CreateSolver()$;
3: $\varphi_0 = ProgramAbstract(P)$;
4: $solver.add(\varphi_0 \wedge \phi_{err})$;
5: **loop**
6: **if** $sResult \neq unknow$ **then**
7: **break**;
8: **end if**
9: $result = solver.Solve()$;
10: **if** $result == unsatisfiable$ **then**
11: $WriteResult(Safe)$;
12: **break**;
13: **end if**
14: $\pi = CEGenerate(solver)$;
15: **if** $CEValidate(\pi) == feasible$ **then**
16: $WriteResult(Unsafe)$;
17: $TracePrint(\pi)$;
18: **break**;
19: **end if**
20: $\gamma = RCGenerate(\pi)$;
21: $\gamma' = RCShare(\gamma)$;
22: $solver.add(\gamma')$;
23: **end loop**
24: $UnbindShareRes(sResult, sRefinements, sClauses)$;

B. Randomization of Counterexample Generation

A critical issue of our method is to let different engines refine different parts of the abstraction without space partition. In our method, the refinement constraints are generated from counterexamples. Hence, different engines should generate different counterexamples, even if they verify the same abstraction. Ideally, the counterexamples generated by different engines should uniformly distribute on the search space to avoid redundant search.

In the SCAR method, a counterexample is generated according to the produced model of the verification problem if a SAT is returned. It employs Minisat-2.2.0 as the constraint solver which uses the DPLL framework. The DPLL framework is a DFS based search algorithm. It iteratively picks up a value of some Boolean variable to guide the search. Picking up either different value or different variable will lead the search to different search space. To make different engines search different parts of the abstraction space, one can let different engines employ different value and variable selection strategies. In our method, we employ a random search strategy

where a random value of some random variable is selected at each point. Given that an abstraction problem may contain millions of Boolean variables, it is scarcely possible for different engines to search the same space with this strategy. Compared with the sequential method, if their are a large number of counterexamples which uniformly distribute on the search space, the number of required iterations to detect all counterexamples may decrease linearly with the increase of engines.

A problem here is that selecting a random value and a random variable makes the search process to be non-deterministic for different runs, which interferes the result analysis. Minisat-2.2.0 uses a random strategy which generates a deterministic random value at each node. In this manner, the search process and generated model of a given SAT problem is deterministic. In our method, we employ a parameterized random strategy which uses the process ID ($1 \cdots n$) to guide the random search. Given an abstraction problem, the searches of each engine are deterministic for different runs, whereas the searches of different engines vary significantly from each other.

C. Synchronization verse Asynchronization

In our method, the parallel engines can execute in either synchronous or asynchronous mode. In the synchronous mode, when one engine generates a refinement constraint, it waits until all other engines have obtained their refinement constraints. Then all engines exchange their refinement constraints and each engine obtains those constraints generated by all engines. Hence, in this mode, different engines always solve the same abstraction problem at each iteration. However, each engine needs to wait for the termination of other engines' abstraction solving and refinement constraint generation at each iteration. Hence, a large amount of time may be wasted for those engines which run faster. In our experiments, we have observed that, even for the same abstraction problem, one engine may performs dozens of times faster than another one. Actually, the more engines we use, the more possible for a bad engine which runs much slower than other ones to appear. In this sense, the synchronous mode puts the bad luck of all engines together. As the number of engines increases, the time required at each iteration may also grow rapidly, which decreases the performance of our method.

In the asynchronous mode, whenever an engine generates its refinement constraint, it immediately writes the constraint into the shared library, and gets from the library those newly generated constraints of other engines. Then it starts its next iteration for abstraction verification without loss of time. In this mode, given that the performance of different engines may vary significantly from each other, it is possible that one engine is running the i-th iteration, while another engine is running the j-th iteration. Hence, no matter at the same iteration or at the same time, the abstraction problems for different engines are usually different from each other. However, given that all refinement constraints can be implied by the original program, they can all be safely added to other engines. Moreover, due to the non-deterministic interleaving problem of different engines, the abstraction problems of the same engine may vary significantly in different runs. Given that the generated counterexamples and refinement constraints depend significantly on the abstraction problems, in the asynchronous mode, even for the same program, the verification performance varies in different runs.

V. Experimental Evaluation

We have implemented our method on top of YOGAR-CBMC [5], which has won the gold medals in the ConcurrencySafety category of SV-COMP 2017 and 2018. We use the multi-threaded programs of SV-COMP 2018 [3] as our benchmarks. Our tool supports all features of C language in the experiments.

A. Benchmark

The open-source, representative, and reproducible benchmarks of Competition on Software Verification (SV-COMP) have been widely accepted for program verification. Given that these benchmarks are devised for comparison of those state-of-the-art techniques and tools, a significant number of studies on concurrent program verification have performed their experiments on them. The concurrency benchmarks of SV-COMP 2018 include 1047 examples and cover most of the publicly available concurrent C programs that are used for verification. Most of these examples are simple and can be verified quickly by YOGAR-CBMC. In our experiments, we choose those examples which require more than five seconds and a large number of iterations for YOGAR-CBMC as our benchmarks, which contain 16 examples. These programs contain hundreds of lines, complex structure variables with 2-dimensional pointers, and hundreds or even a thousand read/write accesses. With these complex features, these programs are challenging for state-of-the-art concurrency verification techniques and tools.

B. Experimental Setup

We conduct all of our experiments using a computer with 32 GB memory and two CPUs of Intel(R) Xeon(R) CPU E5645 2.40GHz. Each CPU has six cores and the computer contains 12 cores in total. A 900-second time limit is observed. Our experiments are presented in two parts. We first compare the performance of our method with 1, 2, 4, and 8 engines. Our method with one engine is the same with the sequential SCAR method. Then to further analyze the efficiency of different strategies, we compare the performance of our method with different strategies. These strategies include the learnt clause sharing and the execution mode of different engines (synchronous or asynchronous).

Similar as in [1], the loop unwinding depths in our experiments are dynamically determined through syntax analysis. In that work, the bound is set to n if some of the program's `for`-loops are upper bounded by a constant n, and 2 for other cases. Observed that most cases can be solved quickly for this bound, we enlarged the bound to be 4 if there is no `for`-loop.

TABLE I
EXPERIMENTAL RESULTS FOR DIFFERENT NUMBER OF ENGINES.

Example ID	Result	|Iterations|				Time (sec.)				Ratio	
		1 Engine	2 Engines	4 Engines	8 Engines	1 Engine	2 Engines	4 Engines	8 Engines	R-|Iters|	R-Time
1	Safe	33	19	10	6	6.2	5.7	4.6	3.8	5.5	1.6
2	Safe	37	18	11	6	14.4	10.4	9.2	8.7	6.2	1.7
3	Unsafe	54	13	13	7	6.9	2.9	3.7	3.5	7.7	2
4	Unsafe	137	111	65	38	42.8	44.5	32.4	28.7	3.6	1.5
5	Safe	263	136	70	36	551.4	283.6	213.8	121.4	7.3	4.5
6	Unsafe	268	129	68	35	166.1	70.6	57.7	38.6	7.7	4.3
7	Safe	317	151	76	41	12.2	7.8	5.1	3.7	7.7	3.3
8	Unsafe	647	389	201	102	50.7	30.6	17.7	11.5	6.3	4.4
9	Safe	658	333	199	91	9.7	5.4	3.8	2.5	7.2	3.9
10	Safe	862	398	201	102	44.6	27.8	17.3	11.1	8.5	4
11	Safe	1180	710	405	181	155.2	86	62.2	35.2	6.5	4.4
12	Safe	1610	785	389	203	27.9	19.7	14.7	11.4	7.9	2.4
13	Safe	1778	883	445	230	144.7	82.9	47	30.6	7.7	4.7
14	Safe	3206	1695	840	410	91.5	49.1	28.3	18.4	7.8	5
15	Safe	4060	2090	1009	504	304.9	173.1	102.2	75.5	8.1	4
16	Safe	4085	2055	1030	516	826.3	403	216.1	131.4	7.9	6.3
Total	-	19195	9915	5032	2508	2455.5	1303	836	536	7.6	4.6

C. Experiments for Different Number of Engines

TABLE I shows the results of our method with 1, 2, 4, and 8 parallel engines. Our method is in asynchronous mode, and it contains the sharing of both refinement constraints and learnt clauses. Note that in the asynchronous mode, due to the non-deterministic process interleavings, there exists a bit deviation between the verification performance of different runs. Hence, in our experiments, each example is performed three times and the average value is used in TABLE I. The `Example ID` column represents the ID of the example. Due to the space limit, the example names are ignored in this table. We sort all examples according to their required number of iterations under one engine (the third column). The `|Iterations|` columns represent the average iteration number of those parallel engines when the example is verified. Note that the iteration numbers of those engines may be different in the asynchronous mode. The `Time` columns represent the spent time of an engine to verify the program. We do not consider the abstraction generation time here, since we do not improve the performance of this step. In the `Ratio` columns, we compare the iteration number and spent time between one and eight engines. Particularly, the `R-|Iters|` column is the ratio between the third and the sixth column, and the `R-Time` column is the ratio between the seventh and the tenth column.

From the table, an encouraging result is that the required number of iterations decreases almost linearly with the increase of engines. With 1, 2, 4, and 8 engines, the total iteration numbers for one engine to verify all these examples are 19195, 9915, 5032, and 2508, respectively. With 8 engines, the required iteration number is reduced to 1/7.6 on average. This is consistent with our initial expectation. We employ multiple engines to refine the abstraction in parallel. Due to our random search strategy, different engines refine different parts of the abstraction in most cases. Hence, the required number of iterations decreases almost linearly with the increase of engines. From the table, we can observe that, the more iterations it requires, the more possible our method will be to obtain a linear reduction of the iteration number. We also observed that for Example 4, the iteration number does not decrease linearly as other examples. This is an unsafe program. In the sequential method, one may find the true counterexample of an unsafe program with fewer iterations if one has a good luck. However, under multiple engines, it rarely happens that all engines have the same good luck with some particular engine. Hence, a large amount effort may be spent on those counterexamples which are irrelative with the true counterexample.

The required time to verify all these examples under 1, 2, 4, and 8 parallel engines is 2455.5, 1303, 836, and 536 seconds, respectively. Benefited from the reduction of iterations, the verification performance can be significantly improved. However, the verification time does not decrease linearly with the increase of engines. For Example 1 to 4, compared with the sequential method, only less than two times of speed-up can be obtained with 8 engines. We have observed that, for these programs, the verification time does not distribute uniformly on those iterations. Most of their verification time is spent at the last iteration at which the program is verified. Though the iteration number is reduced, the last iteration, the time of which dominates the verification time, can not be avoided. Actually, the verification performance depends on many factors. Nevertheless, benefited from the reduction of iterations, our method can still obtain a 4 to 5 times of speed-up under 8 parallel engines.

We also observed from the table that our method is suitable for both safe and unsafe programs. For both the two kinds of programs, our method can obtain a linear reduction on the required number of iterations, and it can perform 4 to 5 times faster than the sequential method with 8 engines.

D. Experiments for Different Strategies

To analyze the efficiency of different strategies, further experiments are performed to compare the performance of our method under different strategies. Here, we are concerned about two issues: 1) the effectiveness of learnt clause

TABLE II
EXPERIMENTAL RESULTS FOR DIFFERENT STRATEGIES.

Example ID	Result	Iterations			Time (sec.)			Time Ratio	
		Asyn+RCS	Asyn+RS	Syn+RCS	Asyn+RCS	Asyn+RS	Syn+RCS	RS/RCS	Syn/Asyn
1	Safe	6	7	6	3.8	5.1	7.4	1.34	1.94
2	Safe	6	7	5	8.7	14.3	10.2	1.64	1.17
3	Unsafe	7	6	8	3.5	3.4	6.9	0.97	1.97
4	Unsafe	38	50	50	28.7	34.5	34.2	1.2	1.19
5	Safe	36	37	37	121.4	284.3	438.4	2.34	3.61
6	Unsafe	35	37	37	38.6	39.7	85.1	1.03	2.2
7	Safe	41	42	41	3.7	4.7	6.3	1.27	1.7
8	Unsafe	102	87	113	11.5	10.8	19.5	0.94	1.69
9	Safe	91	89	80	2.5	2.6	4.3	1.04	1.72
10	Safe	102	112	121	11.1	13.5	35.1	1.22	3.16
11	Safe	181	177	188	35.2	35.6	72.3	1.01	2.05
12	Safe	203	205	197	11.4	14.2	36.2	1.24	3.17
13	Safe	230	227	220	30.6	32.2	51.6	1.05	1.69
14	Safe	410	422	408	18.4	27.2	103.4	1.48	5.62
15	Safe	504	510	529	75.5	107.4	243.8	1.42	3.23
16	Safe	516	509	512	131.4	139.1	496.1	1.06	3.78
Total	-	2513	2529	2554	536	768.8	1650.8	1.43	3.08

sharing strategy, and 2) the efficiency of our method under the synchronous mode. For the first issue, we performed a further experiment on a version of our method which shares only the refinement constraints. To justify the second one, we performed a more experiment on another version of our method which shares both refinement constraints and learnt clauses yet uses the synchronous mode.

Experimental results are given in TABLE II. Here, we use `RS` to represent that only refinement constraints are shared, and use `RCS` to represent that we share also learnt clauses. We use `Asyn` and `Syn` to represent that we use the asynchronous and synchronous mode, respectively. All experiments were performed under 8 engines. Again, all the experimental data is the average value of three runs. In the `Time Ratio` columns, `RS/RCS` represents the ratio between the time of `Asyn+RS` and `Asyn+RCS`, and `Syn/Asyn` represents the ratio between the time of `Syn+RCS` and `Asyn+RCS`.

From this table, the number of required iterations for `Asyn+RS` is similar with that of `Asyn+RCS`, which indicates that the learnt clause sharing strategy does not affect the required number of iterations. For the verification time, there are 7 examples where the `Asyn+RCS` method performs a bit better than or similar with the `Asyn+RS` method. There are 9 examples upon those `Asyn+RCS` performs 1.2 to 2.3 times faster than `Asyn+RS`. This indicates that the learnt clause sharing strategy improves the performance of constraint solving for some of the examples. This is consistent with our knowledge of parallel SAT solving and incremental SAT solving. Given that adding too many learnt clauses may also lead to some overhead on the constraint solving, the improvement benefited from these strategies depends on examples. However, with this strategy, the total verification time is reduced from 768.8 to 536 seconds.

In the synchronous mode, the number of required iterations is similar with that of the asynchronous mode. Note that in the synchronous mode, all engines have the same number of iterations, while in the asynchronous mode, we use the average iteration number of all parallel engines. From the table, the asynchronous mode performs much better than the synchronous mode. The spent time in the synchronous mode is 3.08 times of that in the asynchronous mode. The reason is that in the synchronous mode, it waits until all engines have finished their constraint solving and refinement constraint generation at each iteration. As a result, the spent time at each iteration equals to that of the engine who runs slowest. As we have pointed out before, the more engines we use, the more possible for a bad engine to appear, which runs much slower than other engines. Hence, in the synchronous mode, though the required iteration number is similar with that in asynchronous mode, the required time to verify the program may be much more than that in asynchronous mode.

E. Threats to Validity

One threat to the validity is the limited benchmarks we have used. In our experiments, we select those programs in the ConcurrencySafety track of SV-COMP 2018 which require a large number of iterations as our benchmark. For those examples where counterexamples of the abstraction are not uniformly distributed on the search space, or the verification time is not uniformly distributed on those iterations, the efficiency of our method may be limited.

Another threat is the number of engines we have used. Given that the computer we used contains only 12 cores, we used 8 parallel engines at most in our experiments. Hence, it is difficult to evaluate the efficiency of our method if dozens of or even hundreds of parallel engines are used.

The third is our implementation of the learnt clause sharing strategy. Various clause exchange schemes have been studied by other researchers. However, this paper focuses mainly on the sharing of refinement constraints, and our implementation for learnt clause sharing is simple.

VI. RELATED WORK

Verification of concurrent programs have been extensively studied in recent years. The most successful techniques include

stateless model checking [6]–[9], bounded model checking [10]–[13], and abstraction refinement [14]–[17], etc. Parallel verification has been considered to be an efficient technique to deal with large problems. Various parallel verification algorithms have been designed on either shared memory architectures or distributed systems [18]. Existing work for parallel verification has considered both the stateless model checking and bounded model checking techniques.

For stateless model checking, the general idea is to explicitly enumerate all possible reachable states. To parallelize this technique, existing work usually partitions the state graph into disjoint subsets, which are explored by different engines. The seminal work for this idea is presented by Stern and Dill [19]. In this work, the authors partition the state table over the nodes of the parallel machine, and each new state is assigned randomly to one of the active CPUs. In [20], Holzmann and Bosnacki extended the SPIN verifier, one of the most widely used verification tools for concurrent systems, to multi-core shared-memory systems. The main problems for parallelizing the stateless model checking technique include the load balance problem, the communication balance problem, and the parallelization of those optimization strategies, etc. To deal with the communication overhead between parallel workers, in [21], Melatti et al. extended this technique to computer clusters where each node contains two CPUs. For parallelization of optimization strategies, as pointed out in [20], the benefit of parallelization is reduced with the partial order reduction (POR) optimization. In some cases, the performance does not improve or even degrades. To address this problem, various algorithms have been devised to integrate the advantages of both POR and parallelization [22]–[24]. Moreover, in recent years, to explore the aggressive power of GPU, several researches have been performed to employ the massive number of threads of GPU for the computationally intensive task of verification [25]–[29].

For bounded model checking, the general idea is to convert the verification problem into one or a serial of constraint problems, which are then solved by constraint solvers. Existing work for parallelizing this technique usually contains two directions. The first one is to employ parallel SAT solvers [30]–[32] for constraint solving. The general idea for parallel SAT solving is to exchange learnt clauses of different engines, just as we have done in our method. Another direction is for those verification techniques which convert the verification problem into a serial of constraint problems. For these techniques, one can employ different engines for different constraint problems. In [4], [33], the authors focus on tradition BMC which converts the verification problem into a serial of constraint problems of different lengths. Multiple engines are then employed to solve those problems of different lengths concurrently. In [34], Kahsai and Tinelli focus on k-induction and multiple engines are employed to solve the base and induction serial independently. In [35], Malacaria and Pasareanu focus on symbolic execution and a set of worker threads are utilized to run the decision procedures for different paths. In [36], [37], the authors focus on IC3, which converts the BMC problem into many small problems. To parallelize IC3, they employ multiple threads to solve those small problems in parallel.

In this paper, we focus on abstraction refinement of multi-threaded programs. To the best of our knowledge, we are the first to parallelize the abstraction refinement technique. We have proposed a parallel refinement framework which employs multiple engines to refine the abstraction in parallel. Similar as the parallelization of BMC, we also employ multiple engines to solve those small constraint problems. However, in the parallelization of BMC, the small problems are usually independent from each other, while in the abstraction refinement technique, all latter problems are obtained from their previous problems. We have analyzed the mechanism of the parallelization and present the details of the implementation. Then we parallelized the SCAR method, one of the most efficient methods for concurrent program verification.

Most of existing work for parallel verification focus on concurrent models, while we focus on multi-threaded programs. In [38], Nguyen et al. also focus on parallel verification of multi-threaded programs. To parallelize the verification, they employ a code-to-code translation to convert the program into a set of simpler program instances, each capturing a subset of the original program's interleavings. These instances are then checked independently in parallel. Their method for parallelization is totally different from ours. Their method is appropriate mainly for unsafe programs, while our method is suitable for both safe and unsafe programs.

VII. Conclusion

In this paper, we have parallelized the SCAR method, one of the most efficient methods for concurrent program verification. The SCAR method employs the abstraction refinement technique to convert the large monolithic verification problem into a serial of small problems. Observed that the dependence of those refinements of different iterations is weak, we have proposed a parallel refinement framework which employs multiple engines to refine the abstraction in parallel. We have implemented our method on top of YOGAR-CBMC, and evaluated it on the concurrent benchmark of SV-COMP 2018. Experiments show encouraging results of the presented method. For those complex programs which require a large number of iterations, a linear reduction of the required iterations can be obtained with the increase of engines, and the verification performance can be significantly improved. In the future, we will apply our method to other abstraction refinement methods and distributed systems.

References

[1] L. Yin, W. Dong, W. Liu, and J. Wang, "Scheduling constraint based abstraction refinement for multi-threaded program verification," *CoRR*, vol. abs/1708.08323, 2017. [Online]. Available: http://arxiv.org/abs/1708.08323

[2] SV-COMP, "2017 software verification competition." http://sv-comp.sosy-lab.org/2017/, 2017.

[3] ——, "2018 software verification competition." http://sv-comp.sosy-lab.org/2018/, 2018.

[4] S. Wieringa, M. Niemenmaa, and K. Heljanko, "Tarmo: A framework for parallelized bounded model checking," in *International Workshop on Parallel and Distributed Methods in verifiCation, PDMC*, 2009, pp. 62–76.

[5] L. Yin, W. Dong, W. Liu, Y. Li, and J. Wang, "YOGAR-CBMC: CBMC with scheduling constraint based abstraction refinement - (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, Thessaloniki, Greece, 2018, pp. 422–426. [Online]. Available: https://doi.org/10.1007/978-3-319-89963-3_25

[6] P. A. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas, "Optimal dynamic partial order reduction," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2014, pp. 373–384.

[7] J. Barnat, L. Brim, V. Havel, J. Havlícek, J. Kriho, M. Lenco, P. Rockai, V. Still, and J. Weiser, "Divine 3.0 - an explicit-state model checker for multithreaded C & C++ programs," in *International Conference on Computer Aided Verification, CAV*, 2013, pp. 863–868.

[8] K. E. Coons, M. Musuvathi, and K. S. McKinley, "Bounded partial-order reduction," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2013, pp. 833–848.

[9] J. Huang, "Stateless model checking concurrent programs with maximal causality reduction," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2015, pp. 165–174.

[10] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bounded model checking of concurrent software," in *International Conference on Computer Aided Verification, CAV*, 2013, pp. 141–157.

[11] H. Günther, A. Laarman, and G. Weissenbacher, "Vienna verification tool: IC3 for parallel software - (competition contribution)," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*, 2016, pp. 954–957.

[12] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Bounded model checking of multi-threaded C programs via lazy sequentialization," in *International Conference on Computer Aided Verification, CAV*, 2014, pp. 585–602.

[13] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato, "Verifying concurrent programs by memory unwinding," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*, London, UK, 2015, pp. 551–565. [Online]. Available: https://doi.org/10.1007/978-3-662-46681-0_52

[14] A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach, "Succinct representation of concurrent trace sets," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2015, pp. 433–444.

[15] N. Sinha and C. Wang, "On interference abstractions," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2011, pp. 423–434.

[16] A. Malkis, A. Podelski, and A. Rybalchenko, "Thread-modular counterexample-guided abstraction refinement," in *International Static Analysis Symposium, SAS*, 2010, pp. 356–372.

[17] A. Gupta, C. Popeea, and A. Rybalchenko, "Threader: A constraint-based verifier for multi-threaded programs," in *International Conference on Computer Aided Verification, CAV*, 2011, pp. 412–417.

[18] I. Cerná and B. R. Haverkort, "Parallel and distributed methods in verification," *J. Log. Comput.*, vol. 21, no. 1, pp. 1–3, 2011.

[19] U. Stern and D. L. Dill, "Parallelizing the murj verifier," *Formal Methods in System Design*, vol. 18, no. 2, pp. 117–129, 2001.

[20] G. J. Holzmann and D. Bosnacki, "The design of a multicore extension of the SPIN model checker," *IEEE Transactions on Software Engineering, TSE*, vol. 33, no. 10, pp. 659–674, 2007.

[21] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan, "Parallel and distributed model checking in eddy," *STTT*, vol. 11, no. 1, pp. 13–25, 2009.

[22] J. Barnat, L. Brim, and P. Rockai, "Parallel partial order reduction with topological sort proviso," in *IEEE International Conference on Software Engineering and Formal Methods, SEFM*, 2010, pp. 222–231.

[23] J. Barnat, L. Brim, M. Ceska, and P. Rockai, "Divine: Parallel distributed model checker," in *HiBi/PDMC*, 2010, pp. 4–7.

[24] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Distributed dynamic partial order reduction," *International Journal on Software Tools for Technology Transfer, STTT*, vol. 12, no. 2, pp. 113–122, 2010.

[25] T. Neele, A. Wijs, D. Bosnacki, and J. van de Pol, "Partial-order reduction for GPU model checking," in *International Symposium on Automated Technology for Verification and Analysis, ATVA*, 2016, pp. 357–374.

[26] J. Barnat, P. Bauch, L. Brim, and M. C. Jr., "Designing fast LTL model checking algorithms for many-core gpus," *J. Parallel Distrib. Comput.*, vol. 72, no. 9, pp. 1083–1097, 2012.

[27] S. Edelkamp and D. Sulewski, "Efficient explicit-state model checking on general purpose graphics processors," in *Model Checking Software*, 2010, pp. 106–123.

[28] A. Wijs and D. Bosnacki, "Gpuexplore: Many-core on-the-fly state space exploration using gpus," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, 2014, pp. 233–247.

[29] Z. Wu, Y. Liu, J. Sun, J. Shi, and S. Qin, "GPU accelerated on-the-fly reachability checking," in *International Conference on Engineering of Complex Computer Systems, ICECCS*, 2015, pp. 100–109.

[30] Y. Hamadi, S. Jabbour, and L. Sais, "Control-based clause sharing in parallel SAT solving," in *International Joint Conference on Artificial Intelligence, IJCAI*, 2009, pp. 499–504.

[31] R. Martins, V. M. Manquinho, and I. Lynce, "An overview of parallel SAT solving," *Constraints*, vol. 17, no. 3, pp. 304–347, 2012.

[32] G. Audemard and L. Simon, "Lazy clause exchange policy for parallel SAT solvers," in *Theory and Applications of Satisfiability Testing - SAT*, 2014, pp. 197–205.

[33] E. Ábrahám, T. Schubert, B. Becker, M. Fränzle, and C. Herde, "Parallel SAT solving in bounded model checking," *J. Log. Comput.*, vol. 21, no. 1, pp. 5–21, 2011.

[34] T. Kahsai and C. Tinelli, "Pkind: A parallel k-induction based model checker," in *International Workshop on Parallel and Distributed Methods in verification, PDMC*, 2011, pp. 55–62.

[35] Q. Phan, P. Malacaria, and C. S. Pasareanu, "Concurrent bounded model checking," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–5, 2015.

[36] A. R. Bradley, "Sat-based model checking without unrolling," in *International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI*, 2011, pp. 70–87.

[37] S. Chaki and D. Karimi, "Model checking with multi-threaded IC3 portfolios," in *International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI*, 2016, pp. 517–535.

[38] T. L. Nguyen, P. Schrammel, B. Fischer, S. La Torre, and G. Parlato, "Parallel bug-finding in concurrent programs via reduced interleaving instances," in *IEEE/ACM International Conference on Automated Software Engineering, ASE*, 2017, pp. 753–764.