Designing Parallel PDR

Matteo Marescotti*, Arie Gurfinkel†, Antti E. J. Hyvärinen*, Natasha Sharygina*
*Università della Svizzera italiana, Switzerland
†University of Waterloo, Canada

***Abstract*—Property Directed Reachability (PDR) is an efficient model checking technique. However, the intrinsic high computational complexity prevents PDR from meeting the challenges of real world verification. To address this problem, this paper introduces the parallel algorithm P3 based on: 1) partitioning of the input problem, 2) exchanging of learned reachability information, and 3) using algorithm portfolios. The generic nature of the proposed techniques makes them immediately suitable for software verification. This paper investigates the benefits of these techniques while taken individually and when combined together, implemented using distributed computing environment on top of the SMT-based software model checker SPACER. In our experiments over SV-COMP benchmarks we observe up to an order of magnitude speedup with respect to the sequential implementation with twice as many instances solved within a timeout.**

I. INTRODUCTION

Applying model checking to realistic, complex systems is highly non-trivial in part due to the computational complexity of the underlying decision problem. This paper studies how parallel computing can help in scaling model checking to such problems. We concentrate on parallelizing the execution of the Property directed reachability (PDR/IC3) algorithm [1], [5] using algorithm portfolios and approaches based on divide-and-conquer together with sharing of information learned in the model checking process. In particular, we study how a computational cluster can speed up model checking of software represented as sets of Constrained Horn Clauses (CHC) over Satisfiability modulo theories (SMT) constraints.

The PDR/IC3 algorithm (PDR in brief) is a relatively recent procedure that, given a transition system and a safety property, computes a safe inductive invariant or finds a counterexample for the safety of the system. During the computation, the algorithm maintains an increasingly long sequence of *frames* F_i representing symbolically safe over-approximations of states reachable from the initial state of the transition system in at most i steps. The frames are constructed as sets of PDR-*lemmas* that are computed to block spurious counterexamples. The algorithm terminates either by finding a concrete counterexample or by showing that the set of states described by F_{i+1} is a subset of the set of states described by its immediate predecessor frame F_i. In this case, the frame F_{i+1} is a safe inductive invariant for the program. Constructing the frame sequence is a heuristic process which can employ several different strategies within the PDR algorithm.

In this paper we introduce, to the best of our knowledge, the first divide-and-conquer technique for PDR (called partitioning throughout the paper), combine it with a portfolio of PDR solvers running different strategies, and allow the solvers to share PDR-lemmas. We combine these techniques in our new algorithm called P3 (Parallelly Performed PDR).

The P3 algorithm enables efficient parallel model-checking using PDR with the aim of improving the current state-of-the-art of parallel software verification. The approach is based on three concepts that expand the sequential PDR algorithm as follows: (i) P3 applies a portfolio of sequential PDR implementations parameterized to compute the frame sequence in different ways through different search strategies and by randomizing the search heuristic of the underlying SMT solver; (ii) P3 implements the novel partitioning approach introduced here by using the transition function of the program to compute, based on the negation of the safety property, its pre-images which are then distributed over the solvers in the portfolio as new safety properties; and (iii) the PDR implementations in the portfolio may share the PDR lemmas stored in the frames among each other. To the best of our knowledge, P3 enables parallel PDR for software verification for the first time.

We implemented the P3 algorithm using the sequential SPACER model checker [12] as a basis. We performed a thorough experimental analysis processing over 1000 instances from the Software Verification Competition 2016 (SV-COMP) with different configurations. The experiments were run in a computational cluster of 60 CPU cores. Our results show that a combination of divide-and-conquer, lemma sharing, and algorithm portfolio is capable of solving twice as many instances within our timeout of 1000 seconds and provides up to an order of magnitude speedup compared to the best sequential SPACER configuration. Our experimentations furthermore reveals that the choice of the heuristic for selecting which clauses to share is important, and that especially the partitioning technique benefits the most from it. The results are interesting also in the sense that they report the first experiments on running PDR on a computing cluster targeting in particular software verification with SMT-based Constrained Horn Clauses.

The paper is structured as follows. We compare with related work in Section II, and in Section III we review the basics of the PDR algorithm. In Section IV, we introduce our notion of distributed PDR and give details about our new parallelisation strategies. In Section V, we describe our implementation used for empirical evaluation presented in Section VI. Finally, we conclude the paper in Section VII.

II. RELATED WORK

The first attempt to parallelize PDR is mentioned in the original PDR paper [1], where the experimented parallel setting is based on sharing all the frames among different computing threads on the same machine. This work is further improved in [3] where they study different parallel approaches, all of them focused on multi-threaded portfolios and PDR-lemma sharing, addressing propositional PDR.

Both [1], [3] are limited to propositional PDR, making them suitable mainly for hardware verification. In contast to [1], [3], we propose and thoroughly evaluate different lemma sharing strategies by differentiating between k-invariants and ∞-invariants inside the frames. We introduce the novel partitioning technique for PDR, together with a concise algorithm capable of combining all these techniques in a sound manner. Moreover our implementation is based on the more scalable distributed computing, thus exploiting the much bigger computational power offered by cloud-computing environments compared to single-machine threads.

A first investigation of PDR in the setting of software verification is done in [4]. A different approach based on CHC for software verification is presented in [12]. We extend this work with the aim of improving the current state-of-the-art of parallel software verification.

There is a substantial amount of work on parallel SMT solving that can help software verification model checking techniques in general. The parallelization tree framework for combining divide-and-conquer and portfolio directly on SMT formulas is introduced in [10] and augmented with clause sharing in [14]. A parallel approach for model checking of concurrent programs is given in [9], while [15] presents a parallel symbolic execution involving several sequential SMT solvers.

III. Preliminaries

We assume that the reader is familiar with the basic notation and semantics of First Order Logic (FOL), and theories of Linear Integer Arithmetic (LIA) and Arrays. For a set of variables X, we write X' to denote the set of primed variables $X' = \{x' \mid x \in X\}$. We extend the notation to formulas, and write φ' for a formula obtained from φ by replacing all variables in φ with the corresponding primed variables. Furthermore we denote by $X^{[i]}$ the set of variables obtained by adding i primes to each $x \in X$.

A. Safety Properties

A program can be expressed as a transition system consisting of variables X, a formula $\mathit{Init}(X)$ describing the program's initial states, and a formula $\mathit{Tr}(X, X')$ describing the program's transition relation. Given a transition system, a *safety property* $\neg \mathit{Bad}(X)$ is a formula over the variables of the system. A set of states described by a formula F is *safe* if $F \wedge \mathit{Bad}$ is unsatisfiable. A transition system satisfies a safety property, i.e., is *safe* with respect to $\neg\mathit{Bad}$, if all its states reachable from the initial state with the transition relation are safe. The transition system is safe up to k steps if its states reachable by i applications of the transition relation, for all $0 \leq i \leq k$, are safe. In this paper, we are interested in determining whether a given program satisfies a given safety property. An instance of this problem is expressed as a triple $\mathcal{S} = \langle \mathit{Init}(X), \mathit{Tr}(X, X'), \mathit{Bad}(X)\rangle$. For simplicity, we assume that $\mathit{Init}(X) \implies \neg\mathit{Bad}(X)$, otherwise, $\mathcal{S}$ is unsafe and the counterexample is a trivial model over X that satisfies $\mathit{Init}(X) \wedge \mathit{Bad}(X)$.

Definition 1 (Post Image)**.** *Given a transition relation Tr and a set of states represented by F, the predicate $\mathit{post}^n_{\mathit{Tr}}(F)$ is the set of states reachable from any state in F after taking exactly n transitions of Tr. It is defined as follows:*

$$\mathit{post}^n_{\mathit{Tr}}(F) = \begin{cases} F & \text{if } n = 0, \\ \exists X' \cdot \mathit{post}^{n-1}_{\mathit{Tr}}(F)(X') \wedge \mathit{Tr}(X', X) & \text{if } n \geq 1. \end{cases}$$

$\mathit{post}^*_{\mathit{Tr}}(F)$ is the transitive closure of Tr:

$$\mathit{post}^*_{\mathit{Tr}}(F) = \bigvee_{n \geq 0} \mathit{post}^n_{\mathit{Tr}}(F)$$

The set of reachable states for a program $\mathcal{S}$ is $\mathit{post}^*_{\mathit{Tr}}(\mathit{Init})$. The PDR algorithm constructs an approximation of the reachable states by computing modularly overapproximations of states reachable by $\mathcal{S}$ in a certain number of steps. The algorithm presents these overapproximations as PDR-lemmas, formulas over variables X describing reachability information learned by PDR.

Definition 2 (Relatively Inductive and Invariant Lemmas)**.** *Given initial states represented by Init, transition relation Tr and a set of lemmas F, a PDR-lemma φ is* inductive relative to F *if and only if*

$$\mathit{Init} \implies \varphi \qquad\qquad \varphi \wedge F \wedge \mathit{Tr} \implies \varphi'$$

Whenever φ is inductive relative to true, we say that φ is an inductive lemma*. A PDR-lemma φ is an* invariant lemma *if it is true in all the reachable states, i.e., $\mathit{post}^*_{\mathit{Tr}}(F) \implies \varphi$. Every inductive PDR-lemma is invariant, but the converse is not true in general.*

An instance $\mathcal{S}$ is safe if there exists a safe inductive invariant $\mathit{Inv}(X)$ such that $\mathit{Inv}(X) \implies \neg\mathit{Bad}(X)$. $\mathcal{S}$ is unsafe if there exists an $n \in \mathbb{N}$ such that $\mathit{post}^n_{\mathit{Tr}}(\mathit{Init}) \wedge \mathit{Bad}$ is satisfiable. For an unsafe $\mathcal{S}$, a satisfying assignment for

$$\mathit{Init}(X^{[0]}) \wedge \mathit{Bad}(X^{[n]}) \wedge \bigwedge_{i=0}^{n-1} \mathit{Tr}(X^{[i]}, X^{[i+1]})$$

is called a *feasible counterexample*. The satisfying assignment corresponds to a sequence of states where the first state satisfies Init, each consecutive pair of states satisfies Tr, and the final state satisfies Bad, and can, therefore, be considered as an evidence for a programming error.

B. Property Directed Reachability (PDR)

In this section, we give a high-level overview of IC3/PDR algorithm. We refer the reader to [1], [5], [8], [6], [12] for the details of the original algorithm and its extensions to SMT.

Definition 3 (PDR Trace)**.** *Given an instance of the safety problem $\mathcal{S}$, a* PDR trace *for $\mathcal{S}$ is a sequence of frames $\mathcal{F} = \langle F_0, F_1, \ldots, F_N, \ldots\rangle$ such that each frame $F_i \in \mathcal{F}$ is a set* *of PDR-lemmas. Furthermore, the trace satisfies the following properties for $i \geq 0$:*

$$F_0 \equiv \mathit{Init} \tag{1}$$

$$F_i \wedge \mathit{Tr} \implies F'_{i+1} \tag{2}$$

$$F_i \implies F_{i+1} \tag{3}$$

$$i < N \implies (F_i \implies \neg \mathit{Bad}) \tag{4}$$

Intuitively, each frame $F_i \in \mathcal{F}$ over-approximates all the states reachable in at most i steps of the transition relation Tr from Init. Moreover, the trace proves that $\mathcal{S}$ is safe up to $N-1$ steps of Tr from Init.

PDR uses the trace to compute an increasing bound of steps from Init up to which $\mathcal{S}$ is safe. The algorithm works by iteratively adding an initially empty frame F_N at the end of the trace. PDR then tries to either prove safety of F_N by strengthening it, or to find a feasible counterexample based on it.

Definition 4 (Proof Obligation)**.** *Given an instance $\mathcal{S}$ of the safety problem and a* PDR *trace $\mathcal{F}$ for $\mathcal{S}$, a* proof obligation *is the pair $\langle \sigma, i \rangle$ where σ is a conjunction of predicates over state variables and $i \leq N$. In addition, the proof obligation satisfies the following:*

- *$\sigma \wedge F_i$ is satisfiable, and*
- *for all models m such that $m \models \sigma$, $\mathit{post}^*_{\mathit{Tr}}(m) \wedge \mathit{Bad}$ is satisfiable.*

The conjunction σ represents a set of the states consistent with a frame $F_i \in \mathcal{F}$, containing states that can reach Bad with a feasible path.

Given an instance $\mathcal{S}$, PDR computes a trace $\mathcal{F}$ of increasing length for $\mathcal{S}$ until either a fixed point is found for Tr or the algorithm determines a feasible counterexample. In the process, PDR constructs candidate counterexamples, proof obligations, that are stored in an *obligation queue* $\mathcal{Q}$. The proof obligations $\langle \sigma, i \rangle$ are propagated towards the initial state by computing their pre-image with respect to Tr, resulting in $\langle \sigma^-, i-1 \rangle$ which is then inserted to $\mathcal{Q}$. If a counterexample candidate is not feasible, a proof obligation will at some point be blocked by a frame. This happens if for a proof obligation $\langle \sigma, i \rangle$ it holds that $F_{i-1} \wedge \mathit{Tr} \wedge \sigma'$ is unsatisfiable. The clause $\neg\sigma$ is then simplified to a PDR-lemma and inserted to F_i.

PDR proves $\mathcal{S}$ safe if:

$$\exists i < N \cdot F_{i+1} \implies F_i$$

This simplifies Equation (2) to $F_i \wedge \mathit{Tr} \implies F'_i$. Thus together with Equations (1) and (4) F_i is proved to be both a fixed point for Tr and a safe inductive invariant for $\mathcal{S}$.

The way PDR computes the fixed point leaves room for some flexibility in how the lemmas are organized. In particular [6] suggests to separate the inductive lemmas to a distinct frame F_∞. Hence the frame F_∞ is initially empty and always consists of those lemmas $\varphi \in \bigcup F_i$ inductive relative to F_∞. Thus, $\mathcal{S}$ is safe when

$$F_\infty \implies \neg \mathit{Bad}.$$

PDR proves $\mathcal{S}$ unsafe whenever a proof obligation $\langle \sigma, 0 \rangle$ is added to the obligations queue. By Definition 4, σ represents a set of states in Init from which there is a feasible path leading to a state in Bad.

Definition 5 (PDR Configurations)**.** *Given an instance of the safety problem $\mathcal{S}$, a* PDR configuration *is the quadruple $\mathcal{C} = (N, \mathcal{F}, F_\infty, \mathcal{Q})$ where:*

- *$N \in \mathbb{N}$,*
- *$\mathcal{F} = \langle F_0, \ldots, F_N, \ldots \rangle$ is a trace of $\mathcal{S}$,*
- *F_∞ is the inductive frame of $\mathcal{F}$,*
- *$\mathcal{Q} = \{\langle \sigma, i \rangle, \ldots\}$ is the obligation queue, where $i \leq N$.*

The Initial gonfiguration *of* PDR *for $\mathcal{S}$ is $\mathcal{C}_0 = (1, \langle \mathit{Init}, \emptyset, \ldots \rangle, \emptyset, \emptyset)$*

Given a PDR configuration $\mathcal{C}$ of a safety problem $\mathcal{S}$, each of the following operation performed on $\mathcal{C}$ updates its components resulting in a new configuration $\mathcal{C}'$.

Candidate. If there exists σ such that $\sigma \implies F_N \wedge \mathit{Bad}$, then the proof obligation $\langle \sigma, N \rangle$ is added to $\mathcal{Q}$.

Predecessor. Given $\langle \sigma, i \rangle \in \mathcal{Q}$ with $i > 0$, if there exists a conjunction of predicates δ such that for all consistent m such that $m \models \delta$, $m \wedge \mathit{Tr} \models \sigma'$ holds, then $\langle \delta, i-1 \rangle$ is added to $\mathcal{Q}$.

Blocking. Given $\langle \sigma, i \rangle \in \mathcal{Q}$ with $i \geq 1$, if *Predecessor* is not applicable then remove $\langle \sigma, i \rangle$ from $\mathcal{Q}$ and add the PDR-lemma φ to all F_j with $1 \leq j \leq i$ such that

$$\mathit{Init} \implies \varphi \qquad \varphi \implies \neg\sigma \qquad F_{i-1} \wedge \mathit{Tr} \implies \varphi'$$

Unfold. If $\mathcal{Q} = \emptyset$ and *Candidate* is not applicable then $N := N + 1$.

Inductive. Given a subset of lemmas $\varphi \subseteq F_i$ with $0 \leq i < N$ s.t. $\varphi \wedge F_\infty \wedge \mathit{Tr} \implies \varphi'$, add φ to F_∞ (and to each F_i).

In addition PDR has the following two rules that guarantee the termination of the algorithm when always taken when they are applicable:

Safe. If $F_\infty \implies \neg \mathit{Bad}$ report *safe*.

Unsafe. If any $\langle \sigma, 0 \rangle \in \mathcal{Q}$ report *unsafe* and generate a counterexample.

In PDR with theories, and, therefore, in our implementation, the operation *Predecessor* employs Model-Based Projection [12] to ensure termination, while *Blocking* uses interpolation [8] to build the lemma.

Definition 6 (PDR Strategy)**.** *Given an intance $\mathcal{S}$ of the safety problem and a* PDR *configuration $\mathcal{C}$, a* PDR *strategy $\mathcal{T}_\mathcal{S}$ is a function that maps $\mathcal{C}$ to one of the possible* PDR *operations applicable for $\mathcal{C}$, based on $\mathcal{S}$.*

A PDR execution is a sequence of configurations $\langle \mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_T \rangle$ such that for every $i \in \{1, \ldots, T\}$, $\mathcal{C}_i$ is the result of the operation $\mathcal{T}_{\mathcal{S}}(\mathcal{C}_{i-1})$ on $\mathcal{C}_{i-1}$, $\mathcal{C}_0$ is the initial PDR configuration for $\mathcal{S}$, i.e. $(1, \langle Init, \emptyset, \ldots\rangle, \emptyset, \emptyset)$, and $\mathcal{T}_{\mathcal{S}}(\mathcal{C}_T) \in \{Safe, Unsafe\}$.

IV. The P3 Algorithm

In this section we introduce the P3 (Parallelly Performed PDR) algorithm for parallel model-checking with PDR.

P3 implements three parallelisation techniques for PDR: *portfolio*, *partitioning*, and *lemma sharing*. These techniques can be combined in order to exploit each one strengths.

Portfolio takes advantage by using different PDR strategies while partitioning focuses the search by constraining the problem. In general, partitioning means dividing a problem into several sub-problems. The PDR partitioning technique introduced in this paper partitions the problem by restricting the paths leading to the bad states.

Finally, lemma sharing provides each solver with useful information arising from search diversification, and possibly not derivable locally. The intuition is that PDR-lemmas express what is learned by each PDR execution. Since different executions employs different strategies, PDR-lemmas that are easily found by one strategy can be difficult to find by another.

In the rest of the section, we formalize portfolio, partitioning, and lemma sharing strategies and provide details of P3.

A. Portfolio

The most naïve parallel technique is a *portfolio* – concurrent and independent execution of multiple sequential PDR strategies on the same problem. A portfolio terminates as soon as one of its instances terminates successfully.

We define a notion of a distributed PDR configuration to model a PDR portfolio with any combination of lemma sharing and partitioning.

Definition 7 (Distributed PDR Configuration)**.** *Given an instance $\mathcal{S}$ of the safety problem, a distributed* PDR *configuration is a set of tuples:*

$$\mathcal{D}^n = \{(\mathcal{T}^i, \mathcal{C}^i)\}$$

where for each $i \in \{1, \ldots, n\}, n \in \mathbb{N}$:

- *$\mathcal{T}^i$ is a* PDR *strategy from Definition 6,*
- *$\mathcal{C}^i = \left(N^i, \mathcal{F}^i = \langle F_0^i, F_1^i, \ldots, F_{N^i}^i, \ldots, \rangle, F_\infty^i, \mathcal{Q}^i\right)$ is a* PDR *configuration from Definition 5.*

A distributed PDR configuration expresses a PDR portfolio when for every $i \in \{1, \ldots, |\mathcal{D}|\}$, $\mathcal{T}^i$ is a strategy for the input problem $\mathcal{S}$. That is, every strategy evolves the corresponding PDR-configuration independently, performing asynchronous and arbitrary choices based on $\mathcal{S}$.

A PDR portfolio $\mathcal{D}$ terminates when there exists $\mathcal{T}_{\mathcal{S}}^i(\mathcal{C}^i) \in \{Safe, Unsafe\}$ for some $i \in \{1, \ldots, |\mathcal{D}|\}$. Termination and soundness of this setting follows trivially from PDR because every execution is independent.

B. Partitioning

In this section, we define partitioning strategy and argue for its soundness.

Definition 8. *Given a safety problem $\mathcal{S}$, partition($\mathcal{S}$) is a set of problems $\{\mathcal{S}_{p_1}, \ldots, \mathcal{S}_{p_n}\}$, where each instance $\mathcal{S}_{p_i} = \langle Init(X), Tr(X, X'), p_i(X)\rangle$ is called a partition of $\mathcal{S}$, and such that*

$$\bigvee_i^n p_i \iff \exists X' \cdot Tr(X, X') \wedge Bad(X')$$

From Definition 8, it follows that $\mathcal{S}$ is safe if and only if all of its partitions are safe. A distributed PDR configuration $\mathcal{D}$ expresses partitioning if for each partition $\mathcal{S}_p \in partition(\mathcal{S})$ there is a pair $(\mathcal{T}_{\mathcal{S}_p}^i, \mathcal{C}^i) \in \mathcal{D}$. The result of a distributed configuration with partitioning is $Unsafe$ if there exists $\mathcal{T}_{\mathcal{S}_p}^i(\mathcal{C}^i) = Unsafe$, and the result is $Safe$ if for each $\mathcal{S}_p \in partition(\mathcal{S})$ there exist $\mathcal{T}_{\mathcal{S}_p}^i(\mathcal{C}^i) = Safe$.

The soundness is by construction: a counterexample for $\mathcal{S}_p$ is also valid for $\mathcal{S}$, while the safety of all $\mathcal{S}_p$ ensures the safety of $\mathcal{S}$ because every state leading to Bad in one step is expressed in a partition.

C. Lemma Sharing

In this section, we give the formal definition of lemma sharing and argue for its soundness in a distributed portfolio setting.

Definition 9 ((k-)invariant)**.** *A* PDR*-lemma ψ is k-*invariant *if it is true in all the states reachable in k steps or less, i.e., $post_{Tr}^k(Init) \implies \psi$. If φ is invariant, then it is k-invariant for any $k \in \mathbb{N}$.*

Following Definition 9, each frame F_k, $k \in \mathbb{N}$, is a set of k-invariants for $\mathcal{S}$, while F_∞ is a set of invariants for $\mathcal{S}$.

Theorem 1 (Lemma Sharing)**.** *Given a distributed* PDR *configuration $\mathcal{D}$ for an instance $\mathcal{S}$ of the safety problem, the* PDR*-lemma $\psi \in F_k^i, k \in \mathbb{N}$ is a k-invariant for $\mathcal{S}$ and the operation of adding ψ to any F_l^j with $i \neq j$ and $l \leq k$ keeps $\mathcal{C}^j$ a valid* PDR *configuration for $\mathcal{S}$.*
The same holds for $\varphi \in F_\infty^i$ when added to any F_∞^j, $i \neq j$.

Proof. The proof follows from Definition 2. Each $\varphi \in F_k^i$ is a k-invariant if $k \in \mathbb{N}$, or an invariant if $k = \infty$ and can be used to soundly refine a different abstraction of states reachable in up to k steps.

Similarly, sharing invariants is sound and makes every F_∞^i an invariant for $\mathcal{S}$. Thus, $\mathcal{S}$ is safe whenever any F_∞^i implies $\neg Bad$. □

D. Parallelly Performed PDR

The P3 algorithm is shown in Algorithm 1. It combines portfolio, lemma sharing, and partitioning. The algorithm works as follows. Until there are available computing resources, the procedure **Entrust** randomly selects a partition not yet solved, creates a new strategy and allocates the necessary resources in order to run PDR.

The procedure **Exclude** at line 12 is taken exactly once for each partition in $\mathcal{P}$ whenever a corresponding sequential PDR instance terminates. This happens finitely many times because there are finitely many partitions. Therefore, the algorithm

```
Input   : Safety problem
          $\mathcal{S} = \langle Init(X), Tr(X, X'), Bad(X) \rangle$.
Output  : $\{Reachable, Unreachable\}$
Data    : A distributed PDR configuration $\mathcal{D}$, a set of
          partitions $\mathcal{P}$.
Initially: $\mathcal{D} \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$.
Assume: $Init \wedge Bad$ is unsatisfiable.
1  $\mathcal{P} \leftarrow partition(\mathcal{S})$
2  while True do
3      Reachable: if $\langle \sigma, 0 \rangle \in \mathcal{Q}^i$ for some $i \in \{1, \ldots, |\mathcal{D}|\}$,
         return $Reachable$.
4      Unreachable: if $\mathcal{P} = \emptyset$, return $Unreachable$.
5      Lemma Sharing: copy a PDR-lemma $\varphi \in F^i_n$ to $F^j_n$
         with: $i, j \in \{0, \ldots, |\mathcal{D}|\}$, $i \neq j$ and $n \in \mathbb{N} \cup \{\infty\}$
6      Entrust: if computing resources are available, then:
7          select a partition $\mathcal{S}_p \in \mathcal{P}$
8          create a new PDR strategy $\mathcal{T}_{\mathcal{S}_p}$
9          create new $\mathcal{C} = (1, \langle Init, \emptyset, \ldots \rangle, \emptyset, \emptyset)$
10         set $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathcal{T}_{\mathcal{S}_p}, \mathcal{C})\}$
11         allocate computing resources for PDR($\mathcal{T}_{\mathcal{S}_p}, \mathcal{C}$)
12     Exclude: if there exists $\mathcal{S}_p \in \mathcal{P}$ such that
         $F^i_\infty \implies \neg p$ for some $i \in \{1, \ldots, |\mathcal{D}|\}$, then:
13         $\mathcal{P} \leftarrow \mathcal{P} \setminus \{p\}$
14         release computing resources used for each
             $(\mathcal{T}_{\mathcal{S}_p}, \mathcal{C}) \in \mathcal{D}$
15 end
```

Algorithm 1: The P3 algorithm.

terminates if all corresponding PDR instances terminate. Once exclude is taken, all the computing resources available might be reallocated on other partitions by several **Entrust** calls.

Lemma Sharing does not affect termination because it only refines the frames, leading the execution toward convergence. It is not possible for any frame to get weaker after lemma sharing is applied.

If $partition(\mathcal{S}) = \{\mathcal{S}\}$ at line 1, then partitioning is disabled, and if procedure **Lemma Sharing** at line 5 is never taken then lemma sharing is disabled. If both are disabled then the algorithm corresponds to a PDR portfolio. We assume that there is a global t to handle the desired setting.

V. Implementation

We implemented our parallel PDR algorithm using the tool SMTService [13], a framework for parallel and distributed solving already used in [14] for distributed SMT. A Graphical User Interface [2] is also available for analysing the parallel executions. We upgraded it to also support distributed PDR. SMTService is a client-server based framework. The *server* implements Algorithm 1. The *client* uses the SMT-based PDR model checker Spacer [12]. We modified Spacer in order to expose the API for lemma sharing. The overview of the architecture is shown in Figure 1.

The server is written in Python and its behaviour can be controlled through a *configuration* file. The server is in charge of pre-processing the input instances, managing clients connections and solving tasks and collecting statistics sent by the client solvers. The server stores its information in an SQLite3 database called *Logs*. The database is also used to analyse the steps of the parallel solving process. Instances to be solved are provided to the server at run time through the *control socket*. The control socket allows either the user or a tool (e.g., a model checker) to interact with the server.

The *partitioner* component creates the partitions. The partitioning process is guided by the CHC syntactic structure of the input instances. When partitioning is disabled, the partitioner creates a single partition corresponding to the bad states.

The *scheduler* keeps the list of all the instances and their partitions and manages connected clients. The scheduler solves one instance at a time. The partitions of the current instance are evenly distributed among the connected clients. Once a partition is proven unsatisfiable, the client working on it is provided with a remaining unsolved partition. Spacer implements three strategies, Spacer(DEF), Spacer(IC3), Spacer(GPDR), which differ in how they manage the queue of proof obligations. The scheduler proceeds in a best-effort way assigning each partition to 3 solvers, each configured with one strategy. If there are still solvers available then the same procedure is repeated using a different random seed in the underlying SMT solver of the client. Client failures and connection of new clients are handled in a sound way so that computational power may be added or removed on request.

The *Lemma Database* is implemented as a separate server. Each client periodically pushes learned lemmas to the Lemma Database accordingly to the *sharing strategy* provided by the server. The pull of the lemmas is also periodic and each client only pulls lemmas that are yet unknown for that client.

We implemented 4 lemma sharing strategies. The server reads from the configuration file which is the desired lemma sharing strategy and forwards it to the solver together with the instance. The procedure **Lemma Sharing** at line 5 in Algorithm 1 describes the strategy $*$-*invariants*, namely the exchanging of every learned PDR-lemma. Sharing only PDR-lemmas from F_∞ (∞-*invariants*) is done by fixing $n = \infty$, while sharing only k-*invariants* is achieved by constraining $n \in \mathbb{N}$. Finally, lemma sharing is disabled when the procedure **Lemma Sharing** is never taken. Once an instance is solved its lemmas are removed to reduce memory consumption. In the case of partitioning combined with lemma sharing, our implementation shares lemmas only among solvers working on the same partition.

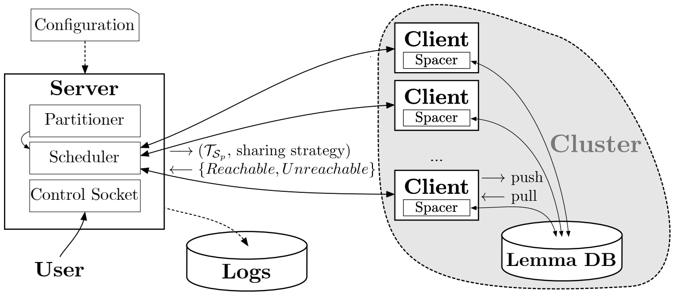

Figure 1. SMTService framework overview. The server implements Algorithm 1. Each client represents a different solver process in a computing node. Solid lines represent TCP/IP connections, while dashed lines represent disk I/O.

Table I
Summary of results showing the number of solved instances.

Technique	*less500*			*more500*		
	#reachable	#unreachable	#unknown	#reachable	#unreachable	#unknown
Spacer(GPDR)	63	175	13	0	8	317
Spacer(IC3)	64	155	32	2	9	314
Spacer(DEF)	64	155	32	2	13	310
portfolio	66	185	0	8	40	277
∞-invariants	66	185	0	7	49	269
k-invariants	66	182	3	7	90	228
$*$-invariants	66	185	0	7	90	228
partitioning	66	176	9	10	34	281
partitioning+∞-invariants	66	183	2	11	49	265
partitioning+k-invariants	66	182	3	11	115	199
partitioning+$*$-invariants	66	185	0	16	98	211
virtual best	66	185	0	18	132	175

VI. Experiments

This section presents an extensive experimental evaluation of the P3 algorithm on instances from the Software Verification Competition. We measure separately the performance of the algorithm on instances that are known to be easy and hard for the Spacer model checker, and study the performance of combinations of lemma sharing, portfolio, and partitioning. We also experiment on different lemma sharing heuristics on these settings.

All the reported experiments are executed on a cluster where each computing node is equipped with 2×Intel E5-2650 v3 CPU, 64 GB of RAM and Intel 40Gbps Infiniband network adapter. The parallel experiments are executed using 60 solvers on 6 computing nodes, with the server running on a separate node. The timeout is set to 1000 seconds (wall-clock time) on all the experiments.

The benchmark set used in our evaluations is constructed by SeaHorn [7], a fully automated analysis framework for LLVM-based languages. Given as input the source file, SeaHorn constructs the triplet $\langle Init(X), Tr(X, X'), Bad(X)\rangle$ expressed in SMT-LIB v2 like language and representing the input safety problem ready to be provided to Spacer. Our benchmark set is based on 1,802 C problems taken from the SV-COMP 2016 Device Drivers Linux 64-bit (LDV) categories available at https://github.com/sosy-lab/sv-benchmarks/tree/master/c and preprocessed by SeaHorn.

We first evaluate the benchmarks sequentially using the different strategies available in Spacer: IC3, GPDR and DEF. We call these settings *sequential*. Those benchmarks solved in less than one second are removed from the set and we experiment over the remaining 562 benchmarks.

Based on these evaluations we create two different benchmarks sets of easy and hard instances respectively:

- *less500*: benchmarks solved in less than 500 seconds by at least one strategy. It contains 251 benchmarks.
- *more500*: benchmarks solved in more than 500 or timed out by at least one strategy. It contains 325 benchmarks.

These benchmark sets partially overlap by having 14 benchmarks in common.

Table I shows an overall evaluation over all the experiments we carried out. For each technique, we report the number of instances proven reachable, unreachable, and those unsolved, for both the experimental sets. The table is partitioned into 4 parts. Going from top to bottom: part 1 contains the results from sequential executions; part 2 contains the result for lemma sharing strategies with pure portfolio; part 3 contains results with partitioning; and part 4 presents the results of the *virtual best* solver that uses the best configuration for each problem. This *virtual best* is achievable by running in isolation a portfolio of the 8 combinations, using 8×60 CPUs. Notably, every parallel technique outperforms sequential execution.

For the *more500* set the partitioning-based techniques perform the best. For the *less500* set, especially for reachable instances, portfolio-based technique is the best, matching the virtual best solver.

In Table II, we show average time speedups between sequential executions and the respective best parallel techniques for both sets, over benchmarks that did not time out. The columns *60 CPU* show the performance of the best technique: ∞-invariants for *less500* and partitioning+k-invariants for *more500*. An overview of performance over all the techniques is shown in Figure 2. We present the runtime performance for the three sequential strategies (IC3, GPDR and DEF) and all

Table II
Average speedup compared to sequential solving.

Sequential strategy	*less500*		*more500*	
	60 CPU	virtual best	60 CPU	virtual best
Spacer(GPDR)	8×	10×	59×	91×
Spacer(IC3)	26×	32×	56×	88×
Spacer(DEF)	27×	33×	53×	83×

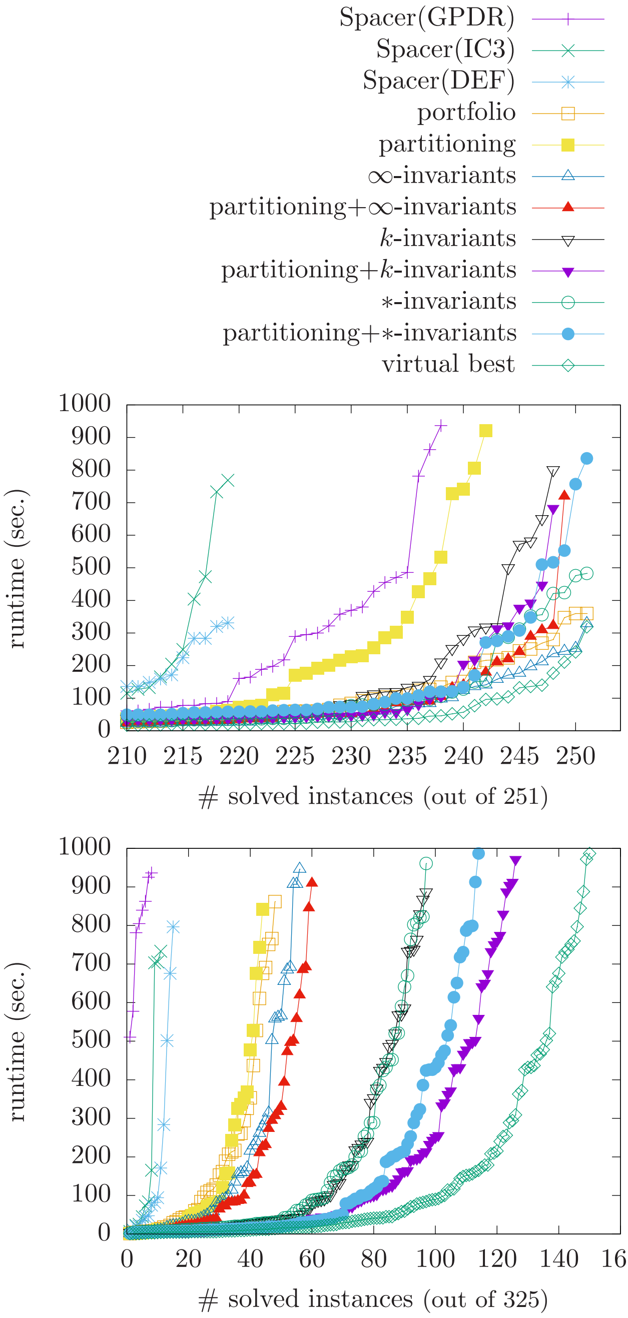

Figure 2. Comparison among all tested techniques on the sets *less500* (top) and *more500* (bottom). k-invariants refers to sharing only lemmas from the trace, ∞-invariants refers to sharing just F_∞, while $*$-invariants implements both. Finally, for each benchmark we report the virtual best runtime among all the tested techniques.

the parallel techniques tested in the cluster.

An overview for the set *less500* is given in Figure 2 (top). Sharing ∞-invariants over portfolio is the best technique for this set, already performing similarly to the virtual best. In fact, ∞-invariants solves all the benchmarks with an average 30% slowdown with respect to the virtual best, and up to 27× faster than sequential, as reported in Table II.

Figure 2 (bottom) shows a similar overview for *more500*. Regarding this set, partitioning techniques are the best choices. In fact, the best technique for this set is partitioning+k-invariants, followed by partitioning+$*$-invariants. However, 40 benchmarks are solved by only one of these two techniques, making them complementary rather than strictly better than the other.

Table III
LEMMA SHARING STATISTICS.

Parallel technique	*less500* time	*less500* #lemmas	*more500* time	*more500* #lemmas
portfolio +				
∞-invariants	0.35%	141	0.41%	670
k-invariants	1.24%	252	1.00%	347
$*$-invariants	1.55%	243	0.83%	348
partitioning +				
∞-invariants	1.46%	170	0.87%	403
k-invariants	3.51%	140	4.55%	238
$*$-invariants	3.27%	221	4.45%	320

A possible way to address the complementarity issue is by implementing a portfolio of multiple isolated parallel techniques, giving priority to the complementary ones. This approach is capable of gradually improving performance toward the virtual best results, where the best performance is reached with the highest CPU resource allocation.

Regarding *more500*, a portfolio of partitioning+k-invariants and partitioning+$*$-invariants is capable of solving 140 instances, 10 less than the virtual best and with an average 15% slowdown. Then, by adding $*$-invariants and k-invariants it is possible to solve 148 instances with 5% slowdown and half computing resources with respect to virtual best. The missing 2 instances are only solved by partitioning+∞-invariants and ∞-invariants.

Regarding *less500*, a similar setting can only decrease solving time. This is because ∞-invariants already solved all the benchmark. Figure 3 shows the comparison between partitioning and portfolio with $*$-invariant sharing being quite complementary. Thus, a portfolio of these two techniques is capable of solving the entire *less500* set using one fourth the CPU power requested by the virtual best, with a 14% slowdown.

Another important result shown in Figure 3 that partitioning often outperforms pure portfolio on reachable instances. This is because the first partition proven satisfiable also proves the entire problem satisfiability, and the focused search done in each partition helps the solvers to converge quickly.

Table III shows results about lemma sharing. The columns *time* show the average amount of time spent on lemma sharing push and pull, with respect to solving time. The columns *#lemmas* show the average number of PDR-lemmas exchanged. The amount of time spent on PDR-lemmas push and pull is about 1% and 3% of solving time respectively for portfolio based techniques, and partitioning based technique. The average amount of PDR-lemmas generated is significantly lower than in parallel SAT and SMT [10], [11]. While heuristics for clause sharing within parallel SAT and SMT are required due to the high throughput, in this settings lemma sharing heuristics are less crucial.

Overall, our experimental evaluations show that parallel techniques are highly beneficial. In particular, parallelization with portfolio combined with sharing PDR-lemmas from F_∞ is the best choice for easier instances, while partitioning with sharing PDR-lemmas from the trace is the best choice for harder instances. This demonstrate that the choice of the lemma sharing strategy is important.

More experimental results are available at http://verify.inf.usi.ch/content/p3-experimental-results-fmcad2017.

VII. Conclusions

This work introduces the first parallel approach of the IC3/PDR algorithm for software model checking based on constrained horn clauses and satisfiability modulo theories. The P3 algorithm is based on combining in a new and PDR-specific way algorithm portfolios, divide-and-conquer approaches, and sharing of information learned during the algorithm execution. We describe our algorithm and its parallel extensions in a unified framework that allows us to both reason about the correctness of the implementation, and study the effect of each component in relative isolation. In addition we identify two different types of PDR-lemmas, the k-invariants, and the ∞-invariants, and give the first results on constructing lemma sharing heuristics. To the best of our knowledge in particular the divide-and-conquer approach and the lemma sharing heuristics have not been previously used in the context of PDR. The techniques we propose improve the previously introduced powerful portfolio technique for PDR, and we believe that the contributions help in applying parallel and distributed computing both in hardware and software verification.

We implemented the P3 algorithm following the same principle of isolation between the different techniques. The implementation is based on the SPACER model checker and is adaptable to both distributed environments and multi-core computing throught our SMTSERVICE framework. Our experimental results obtained by running P3 on a representative set of software verification benchmarks from the SV-COMP 2016 competition show that the parallel approach is vastly superior to sequential SPACER configurations, solving over hundred more instances within our timeout and providing on the average super-linear speed-ups. We also show that each of the new techniques work in isolation and that they have interesting interaction with the different clause-sharing heuristics.

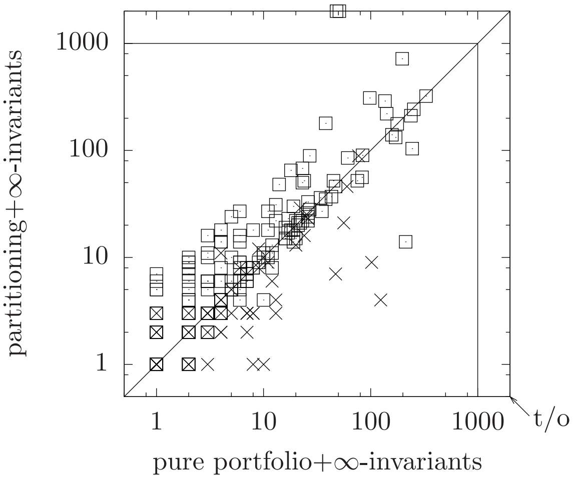

Figure 3. Comparison between sharing invariant with and without partitioning over *less500* sat (×) and unsat (⊡) instances. Compared to "best", this combination is overall 14% slower while using 75% less CPU. Moreover it is shown that partitioning outperforms on all sat instances.

Acknowledgements. This work is supported by the Swiss National Science Foundation (SNSF) grants 153402 and 166288.

References

[1] Bradley, A.R.: SAT-based model checking without unrolling. In: Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. pp. 70–87 (2011)

[2] Budakovic, J., Marescotti, M., Hyvrinen, A., Sharygina, N.: Visualising SMT-based parallel constraint solving. In: Proceedings of the 15th International Workshop on Satisfiability Modulo Theories. (2017)

[3] Chaki, S., Karimi, D.: Model checking with multi-threaded IC3 portfolios. In: Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. pp. 517–535 (2016)

[4] Cimatti, A., Griggio, A.: Software model checking via IC3. In: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. pp. 277–293 (2012)

[5] Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011. pp. 125–134 (2011)

[6] Gurfinkel, A., Ivrii, A.: Pushing to the top. In: Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015. pp. 65–72 (2015)

[7] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 343–361 (2015)

[8] Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings. pp. 157–171 (2012)

[9] Holzmann, G.J.: Cloud-based verification of concurrent software. In: Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. pp. 311–327 (2016)

[10] Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. LNCS, vol. 9340, pp. 369–386. Springer (2015)

[11] Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Incorporating clause learning in grid-based randomized SAT solving. Journal on Satisfiability Boolean Modeling and Computation 6(4), 223–244 (2009)

[12] Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design 48(3), 175–205 (2016)

[13] Marescotti, M.: SMTService Framework for distributed SMT and PDR. https://scm.ti-edu.ch/projects/smts (2017)

[14] Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. pp. 428–443 (2016)

[15] Rakadjiev, E., Shimosawa, T., Mine, H., Oshima, S.: Parallel SMT solving and concurrent symbolic execution. In: 2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 3. pp. 17–26 (2015)