A Cooperative Parallelization Approach for Property-Directed k-Induction

Martin Blicha[1,2], Antti E. J. Hyvärinen[1], Matteo Marescotti[1], and Natasha Sharygina[1]

[1] Università della Svizzera italiana (USI), Switzerland
{first.last}@usi.ch
[2] Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic

Abstract. Recently presented, IC3-inspired symbolic model checking algorithms strengthen the procedure for showing inductiveness of lemmas expressing reachability of states. These approaches show an impressive performance gain in comparison to previous state-of-the-art, but also present new challenges to portfolio-based, lemma sharing parallelization as the solvers now store lemmas that serve different purposes. In this work we formalize this recent algorithm class for lemma sharing parallel portfolios using two central engines, one for checking inductiveness and the other for checking bounded reachability, and show when the respective engines can share their information. In our implementation based on the PD-KIND algorithm, the approach provides a consistent speed-up already in a multi-core environment, and surpasses in performance the winners of a recent solver competition by a comfortable margin.

Keywords: Parallel model checking · Lemma sharing · Property-directed k-induction · IC3/PDR · Craig interpolation

1 Introduction

Safe inductive invariants of symbolically described, infinite-state transition systems are valuable artefacts when proving safety for example in software model checking. Algorithms suitable for obtaining such invariants include those based on k-induction [27,32] and IC3 [8]. These algorithms rely on descriptions in propositional or first-order logic that are solved with SAT and SMT solvers enhanced with over-approximation techniques based on Craig interpolation [12,7]. The elusive goal of such algorithms is to minimize the need for user intervention in model checking through well-defined tasks that can be turned into a symbolic traversal of a search space at the expense of increased computational cost.

Solvers for this problem have often a substantial heuristic component enabling different strategies in the algorithm execution. Recent results [10,30,24] show the use of varied strategies to be a powerful tool for parallelizing model-checking algorithms using algorithm portfolios. The abstract nature of the algorithmic components enables literally infinite possibilities for adjusting the model-checking algorithms, and the changes are known to affect dramatically not only

the algorithm run time but also its convergence. However, the key to truly scalable solving is the sharing of information among the solvers of the portfolio (see, e.g., [24]), a usually much more complicated task than constructing the portfolio.

This paper describes a parallelization approach for a recently introduced class (see [20,15]) of model-checking algorithms that combines the strength of k-induction with IC3-style search in finding safe inductive invariants. The algorithms consist of two components, the *induction-checking engine* and the *finite reachability engine*. We describe what information sharing means in a portfolio of instances of this class, and show with a robust experimental analysis on our implementation that the class can profit greatly from this type of parallelization already in a multi-core environment, surpassing in performance the state-of-the-art. While in the following we refer to the class with the acronym IcE/FiRE, we point out the two existing implementation that we are aware of, PD-KIND [20] and KIC3 [15].

An instance of determining the safety of a transition system $\mathcal{S}$ consists of a triple of predicates (I, T, P), where I describes the initial states of the system, T describes its transition relation, and P is a set of states to be tested to contain all reachable states of $\mathcal{S}$. The predicates are defined over a fixed set of state variables X, and, in the case of T, a copy X' of X. A solution to the instance, if one exists, is a predicate R containing I such that $R(X) \wedge T(X, X') \implies R(X')$ and R is contained in P.

In this paper we are studying a general class of algorithms that work on an over-approximation $\mathcal{F}$ of the states of $\mathcal{S}$ reachable in n steps or less for some $n \geq 1$. The idea is to maintain the invariant that predicate $\mathcal{F}$ does not intersect with $\neg P$, while trying to prove that $\mathcal{F}$ is (k-)inductive. When $\mathcal{F}$ is represented symbolically as a set of formulas, individual elements of $\mathcal{F}$ can be checked for inductiveness relative to $\mathcal{F}$ instead of checking $\mathcal{F}$ as a whole. Successfully checked elements are collected in a new set $\mathcal{G}$ which represents an over-approximation of the states of $\mathcal{S}$ reachable in m steps or less, for $m > n$. When $\mathcal{G} = \mathcal{F}$, such an $\mathcal{F}$ (or $\mathcal{G}$) has the properties of R and therefore is a solution for (I, T, P). This new class of algorithms, introduced in [20] and further refined in [15], is based on an observation that it is, from a pragmatic point of view, better to use an engine for k-induction instead of regular induction in showing $\mathcal{F}$ inductive. The class can be described as a combination of the algorithms based on k-induction and IC3. Intuitively it generalizes IC3, since k-induction is stronger than regular induction. In addition, instances from this class perform well in experimentations. For example, the model checker Sally [20], which implements PD-KIND, won the transition system division of the 2019 edition of the CHC competition.[3]

In this paper we describe a parallelization approach for the IcE/FiRE class of algorithms. We show that the algorithms allow sharing both the formulas constructed for $\mathcal{F}$ and the formulas inside the finite reachability engine. Our parallel algorithm, implemented for multi-core environments on top of PD-KIND [20], performs better than the state-of-the-art parallel and sequential solvers P3 [24], Z3/Spacer [22], and PD-KIND itself [20]. The implementation shows surpris-

[3] See `https://chc-comp.github.io`

ingly good, consistent, close to linear speed-up at least up to nine cores that is visible already for instances with run times as low as two seconds and tends to become more pronounced for higher run times. We show that both types of formula sharing are useful: the parallel solver solves more instances within our timeout and solves the easier instances faster. The implementation is particularly good at showing systems safe.

2 Related Work

Parallelization is a natural way of improving scalability of model-checking algorithms, for example when facing the complexity of real-world problems. We therefore review below only the work that we deem most relevant to our results.

In [24] we presented the P3 system for parallelizing the IC3-inspired algorithm IC3/PDR for computing clusters using portfolio of lemma-sharing solvers and search-space partitioning. The current work differs from that in several important aspects. First, we study a different class of algorithms, based on a combination of IC3 and k-induction. Second, in the implementation our emphasis in this work is on multicore environments instead of computing clusters. We also target a different application domain, studying transition systems instead of general constrained Horn clauses. Finally, in comparing the current system against P3 we measure a significant improvement on the set of instances that both tools can solve, providing practical evidence on the importance of the contribution.

Approaches for parallel IC3 were suggested, for example, in the original publication [8], and more recently in [10]. The current system differs from both, in addition to basing on k-induction, by allowing constraints expressible in first-order logic through an SMT encoding instead of purely propositional encoding, therefore being more readily applicable in software model checking.

The Tarmo system [34] allows SAT-based bounded model checkers to share learned clauses between queries of different execution bounds. The approach could be applied at least in the FiRE systems underlying our bounded reachability queries by allowing the SMT solvers to share clauses as in [25,18]. However, we leave the study of performance effects of such a technique for future work.

A system presented in [31] follows a different approach of determining the feasibility of symbolic execution paths in parallel. Our approach is more symbolic in the sense that it does not require the explicit enumeration of, in general, an exponential number of paths done in [31]. Algorithms for parallel LTL model checking are presented in [1]. The general approach relies on an automata-theoretic formulation of reducing model checking to determining the emptiness of Büchi automata. The parallelization idea focuses on using algorithms based on DFS and BFS for this purpose. We consider this approach orthogonal to ours, and leave it for future work to study the possible synergies. In [21] the authors use three processes to parallelize a standard k-induction algorithm enriched with invariants generated from predefined templates. This approach was generalized in [3] where program analysis with dynamic precision refinement generates continuously-refined invariants for the k-induction. Our approach is based

on the more general IcE/FiRE class, and allows scalability to arbitrary number of cores. In [30] the authors present a more general approach of parallelizing model checking by running several model checkers in parallel. However, the paper does not address the problem of sharing information between the solvers, a topic central to the current discussion.

Finally, our approach is greatly inspired by the sequential approaches combining k-induction with IC3, in particular the PD-KIND algorithm [20] but also the KIC3 framework [15]. In this work we aim at capturing the class of these algorithms from the point of view of information sharing between different solvers, and apply these results on parallelizing these algorithms.

A very recent, not yet published work [2] presents another approach of combining k-induction and IC3/PDR. It extends the framework of [3] and employs IC3/PDR (not only) for generation of auxiliary invariants for k-induction.

Combining and unifying different approaches to software verification, such as IC3/PDR [8,14], k-induction [32] and BMC [5], is becoming increasingly popular [3,4,9,15,20]. Both combination and parallelization techniques benefit from relentless continuous improvements [6,11,16,23,33] of the original algorithms.

3 Preliminaries

Let X denote a finite set of typed variables and let X' denote the set of primed versions of variables from X, i.e., the next-state variables. Then a *state formula* $F(X)$ is any quantifier-free formula over variables from X and a *transition formula* $T(X, X')$ is any quantifier-free formula over variables from both X and X'. A *transition system* $\mathcal{S}$ (over X) is a pair $\langle I, T\rangle$, where I is a state formula denoting the initial states of the system and T is a transition formula. A *state* s^X is a type-consistent assignment of variables from X, i.e., $s^X(x) \in Dom(x)$ for all $x \in X$. When clear from context, we omit X and write simply s. A state formula F holds in a state s if it evaluates to true under s, that is, $s \vDash F$. The states s such that $s \vDash F$ are called the *F-states*. A sequence of states $\langle s_0, s_1, \ldots, s_k\rangle$ is called a *trace* if $s_{i-1}^X, s_i^{X'} \vDash T(X, X')$ for all $1 \leq i \leq k$. A state s is *k-reachable* in $\mathcal{S}$ (reachable in k steps) if there exists a trace $\langle s_0, s_1, \ldots, s_k\rangle$ such that $s_0 \vDash I$ and $s_k = s$. A state is *reachable* if it is k-reachable for some finite k.

A state formula F is a *k-invariant* of the system if it holds in all states reachable in k or less steps. If F is a k-invariant then $\neg F$ is not reachable in k steps or less and we say that $\neg F$ is *k-inconsistent* with $\mathcal{S}$. When a concrete k is not important or not determined, or when we we refet to multiple k-invariants but with different values of k, we use a more general term *bounded invariants*. A bounded invariant F is thus a state formula for which there exists k such that F is a k-invariant. Similarly to IC3, we also use the term *lemma* to refer to a bounded invariant.

Given a transition system $\mathcal{S} = \langle I, T\rangle$, a state formula P and a set of state formulas $\mathcal{F}$, we say that P is *$\mathcal{F}^k$-inductive* if

$$\bigwedge_{i=0}^{k-1} ((\mathcal{F}(X_i) \wedge P(X_i)) \wedge T(X_i, X_{i+1})) \implies P(X_k) \tag{1}$$

If $\mathcal{F} = \{P\}$ and P is a $(k-1)$-invariant, then P is a *k-inductive invariant* of $\mathcal{S}$, meaning it is valid in all reachable states of $\mathcal{S}$. When P is not $\mathcal{F}^k$-inductive, the negation of (1) is satisfiable and each satisfying assignment defines a trace $\langle s_0, \ldots, s_k \rangle$ of $k+1$ states called a *counter-example to (k-)induction* (CTI). We say that a CTI is reachable in $\mathcal{S}$ when s_0 is reachable. A central task of the algorithm presented in this paper is to check if elements of $\mathcal{F}$ are $\mathcal{F}^k$-inductive. Checking this for an element P of $\mathcal{F}$ and placing P to another set $\mathcal{G}$ if P is $\mathcal{F}^k$-inductive is referred to as *pushing* P to $\mathcal{G}$.

Given a transition system $\mathcal{S}$ and a state formula P, the goal of verification is to prove that P is valid on all reachable states of $\mathcal{S}$, or equivalently that $\neg P$ is not reachable. We say that the system is *safe* with respect to P if P is indeed an invariant of the system, and we say that it is *unsafe* if there exists a finite trace starting from an initial state and ending in a $\neg P$-state. For the rest of the paper we make the assumption that the problem is non-trivial, meaning that the initial states satisfy the property P, or more formally, that $I \implies P$ is valid.

4 The IcE/FiRE Framework

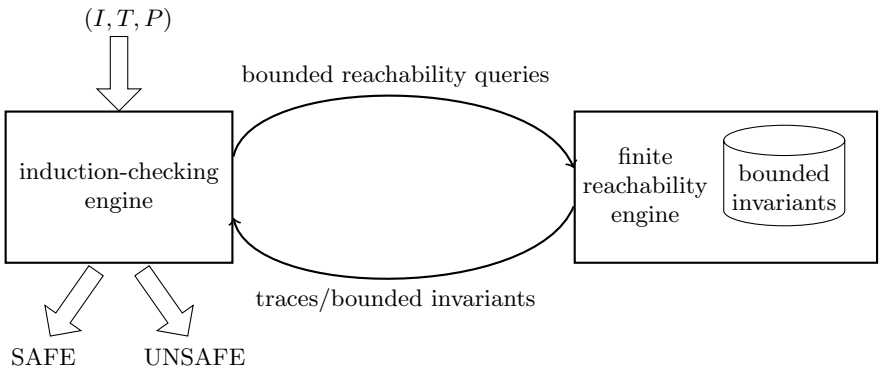

Fig. 1: The IcE/FiRE framework for solving safety of transition systems

This section formalizes a general approach for checking safety of symbolically represented transition systems in a way that allows us to present naturally our parallelization techniques. The approach splits the reasoning about the safety into two separate components (Fig. 1). The first, main, component is an *induction-checking engine* (IcE), also referred to shortly as induction engine. The goal of the induction engine is to decide the safety problem. It searches for a k-inductive strengthening of the property P being checked. If it finds such a strengthening it reports the system as safe. During the search it may discover that no such strengthening exists since the negation of the property is reachable from the initial states. In this case it reports the system as unsafe. To make progress in its search, to remove spurious counterexamples to induction, and to

confirm real ones, IcE relies on the services of the second component – *finite reachability engine* (FiRE). The role of FiRE is to answer *bounded reachability queries* issued by IcE. Given a state formula s and a number n, a bounded reachability query asks if any s-state is reachable from initial states in exactly n steps. The finite reachability engine answers these queries and provides a reason for the answer. In case of reachability, the reason is a trace of $n+1$ states leading from an initial state to an s-state. In case of unreachability, the reason is an n-invariant blocking s.

The cooperation of these two engines is depicted on Fig. 1. During the run, FiRE accumulates knowledge about the system in the form of bounded invariants. This knowledge helps it to answer the subsequent queries faster. The progress of IcE during its run is modelled using a set of rules that capture and evolve the state of IcE. We discuss the rules in the next section and discuss how IcE relies on FiRE when applying these rules.

The idea of separate components for inductive and bounded reachability reasoning is present already in [20]. However, our formalization enables us to easily extend the framework to parallel setting with information sharing and reason about its correctness. In addition, thanks to its abstract nature, it covers not only PD-KIND [20], but also other algorithms, such as KIC3 [15]. We show this for PD-KIND in Sec. 5, but omit the similar proof for KIC3 due to lack of space.

4.1 Induction-Checking Engine

Given a safety problem for a transition system (I, T, P) the induction-checking engine (IcE) searches for k-inductive strengthening of P. It maintains two distinct sets of state formulas: a *base* frame $\mathcal{F}$ and a *successor* frame $\mathcal{G}$. In addition, it maintains information about its current level n. Intuitively, if IcE is currently working on level n, it already knows that the system is safe up to level n, i.e., $\neg P$ is not reachable in n steps or less. The base frame $\mathcal{F}$ serves both as a witness that $\neg P$ is not reachable, as well as a candidate for the inductive strengthening of P. IcE maintains an invariant that on level n every element of $\mathcal{F}$ is an n-invariant. Moreover, P is always an element of $\mathcal{F}$. The successor frame $\mathcal{G}$ collects those elements of $\mathcal{F}$ that are $\mathcal{F}^k$-inductive for some fixed $k \leq n{+}1$. Since $\bigwedge \mathcal{F}$ is an n-invariant, this means that all elements of $\mathcal{G}$ are at least $(n{+}1)$-invariants. When all elements of the base frame are checked and either successfully pushed to $\mathcal{G}$ or dropped, and no termination condition has been hit, $\mathcal{G}$ becomes the new base frame and the successor frame is emptied. If at any point $\mathcal{F} = \mathcal{G}$ then $\mathcal{F}$ is a k-inductive strengthening of P, proving that P holds in the system (as shown later in Lemma 1). In addition to the two frames IcE maintains a queue Q. The queue contains the elements of $\mathcal{F}$ that still need to be processed at the current level. We also refer to the elements of Q as *obligations*.

We now formalize the workings of the induction engine as a set of rules that work on and modify the current state of IcE. The current state of IcE is a 5-tuple $\langle \mathcal{F}, \mathcal{G}, n, k, Q \rangle$ with $\mathcal{F}$ being the base frame, $\mathcal{G}$ the successor frame, n the current level, Q the current queue of obligations, and k defining the current depth of

induction. We refer to the state of IcE as *configuration*. For brevity we also sometimes refer to the elements of $\mathcal{F}$ as lemmas instead of bounded invariants. The initial configuration of IcE is $\langle \{P\}, \emptyset, 0, 1, \{P\} \rangle$ and IcE makes progress by applying the following rules. Note that the rules **Safe** and **Unsafe** are special, *terminating* rules.

Safe:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, \emptyset \rangle}{SAFE} \quad \textbf{if } \{ \mathcal{F} = \mathcal{G}$$

Unsafe:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \rangle}{UNSAFE} \quad \textbf{if } \{ \neg P \text{ is reachable in } [n+1, n+k] \text{ steps.}$$

Next-Level:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, \emptyset \rangle}{\langle \mathcal{G}, \emptyset, n', k', \mathcal{G} \rangle} \quad \textbf{if } \begin{cases} \mathcal{F} \neq \mathcal{G} \\ n' > n \\ \bigwedge \mathcal{G} \text{ is } n'\text{-invariant} \\ 1 \leq k' \leq n'+1 \end{cases}$$

Push-Lemma:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\} \rangle}{\langle \mathcal{F}, \mathcal{G} \cup \{l\}, n, k, Q \rangle} \quad \textbf{if } \{ l \text{ is } \mathcal{F}^k\text{-inductive}$$

Add-Lemma:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \rangle}{\langle \mathcal{F} \cup \{l\}, \mathcal{G}, n, k, Q \cup \{l\} \rangle} \quad \textbf{if } \{ l \text{ is an } n\text{-invariant}$$

Drop-Lemma:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\} \rangle}{\langle \mathcal{F}, \mathcal{G}, n, k, Q \rangle} \quad \textbf{if } \{ l \neq P$$

The rules of IcE, namely **Add-Lemma** and **Drop-Lemma**, are abstract in the sense that we do not prescribe when or how are the new lemmas learnt, nor when they should be dropped. In sequential setting, new lemmas are typically learnt from FiRE when a counter-example to induction of some obligation is showed to be unreachable by FiRE. We discuss this in detail in Sec. 5 when we instantiate the abstract IcE for a concrete algorithm.

One specific thing that we would like to point out is that **Add-Lemma** is general enough to cover not only the *internal* learning, but also *external* learning. By internal learning we mean the learning of lemmas from FiRE. The external learning means that the lemmas can come from any other source. This is important for parallelization as it enables incorporating bounded invariants discovered by other instances working on the same problem.

Correctness of the induction-checking engine. The abstract nature of the rules of IcE allows us to easily prove it correctness. That is, if the engine terminates by applying the rule **Safe** (**Unsafe**) then the system really is safe (unsafe).

Given our assumption that $I \implies P$, the following invariants are valid for the initial configuration and are maintained by every rule (excluding the terminating rules **Safe**, **Unsafe**):

1. $P \in \mathcal{F}$
2. For each $l \in \mathcal{F} \cup \mathcal{G} \cup Q$ at level n, l is an n-invariant of $\mathcal{S}$.
3. For each $l \in \mathcal{G}$, l is $\mathcal{F}^k$-inductive.

It is easy to verify that all invariants are valid for the initial configuration. The first invariant is trivially preserved by all rules except **Next-Level** as $\mathcal{F}$ either stays the same or grows. When **Next-Level** is applied that it must hold that $P \in \mathcal{G}$ since it is put in Q at the beginning of each level and can never be dropped. Since Q is empty when **Next-Level** is being applied, P must have been successfully pushed to $\mathcal{G}$ using **Push-Lemma**.

The second invariant is preserved by the rules **Next-Level**, **Push-Lemma** and **Drop-Lemma** since the set of formulas in consideration stays the same or becomes smaller. The invariant is also preserved by **Add-Lemma** because of the condition of the rule.

The third invariant trivially holds after applying **Next-Level** as the successor frame is empty at that moment. For the other rules, let us use $\mathcal{G}'$ to denote the successor frame after a rule has been applied. The invariant is also preserved by rules **Add-Lemma** and **Drop-Lemma** since $\mathcal{G}' = \mathcal{G}$. Finally, the invariant is preserved by **Push-Lemma** because of the condition of the rule.

Lemma 1. *When the algorithm terminates by applying **Safe**, the system satisfies the property P and $\bigwedge \mathcal{F}$ is a safe k-inductive invariant. When the algorithm terminates by applying **Unsafe**, the system can reach a state where P does not hold.*

Proof. The first part follows from the invariants. When **Safe** is applied, then it must be the case that $\mathcal{F} = \mathcal{G}$. This means that $\mathcal{F}$ is $\mathcal{F}^k$-inductive and consists of n-invariants of the system with $k \leq n+1$. It follows that $\bigwedge \mathcal{F}$ is a k-inductive invariant of the system. Moreover, $P \in \mathcal{F}$, so P is an invariant. The second part follows trivially from the condition of the rule **Unsafe**. □

4.2 Finite Reachability Engine

The finite reachability engine (FiRE) is responsible for answering *bounded reachability queries* issued by IcE. A bounded reachability query for a system $\mathcal{S}$ is simply a pair $\langle s, i \rangle$ where s is a state formula and i is a natural number. It represents a question if any s-state is reachable in $\mathcal{S}$ by exactly i steps. This is naturally generalized to queries of the form $\langle s, [i, j] \rangle$, meaning reachability in at least i and at most j steps. An answer to a bounded reachability query $\langle s, i \rangle$ is

either an i-invariant l such that $l \implies \neg s$ in case of unreachability, or a trace of $i+1$ states starting from an initial state and ending in an s-state in case of reachability.

We do not prescribe how FiRE should be implemented, but we note two known instances: bounded model checking [5] and IC3/PDR [8]. An interesting observation [20] is that when IC3/PDR only needs to answer bounded reachability queries then the requirements on the frames it maintains can be relaxed. The frames do not need to be inductive nor form a monotone sequence.

From the parallelization perspective the advantage of FiRE based on bounded invariants is two-fold. First, the correctness of FiRE is maintained when bounded invariants are exchanged between different instances. Second, there is freedom in generalizing the bounded invariants computed as certificates of unreachability and this freedom can be exploited for portfolio approach to discover a variety of interesting bounded invariants across multiple instances.

4.3 Cooperation of Multiple Instance

We base our parallelization on the portfolio approach running multiple instances of the same algorithm with different parameters on a single problem. However, we aim to go beyond that. We want the instances to *cooperate* and to *share* information they discover about the problem they are solving. Our approach to cooperation of multiple instances of IcE/FiRE framework is depicted in Fig. 2.

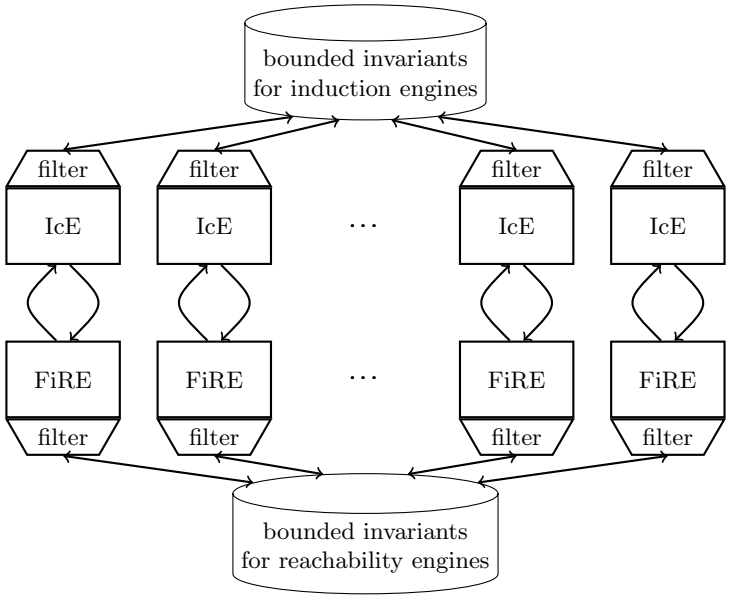

Fig. 2: Multiple instances of IcE/FiRE framework sharing information

In our approach, several instances of IcE/FiRE framework (see Fig. 1) work on the same problem and share information among themselves. However, the communication is split to that between the finite reachability engines and to that between induction-checking engines.

Cooperation of FiREs. Each reachability engine is gradually building and refining its representation of the state space by discovering and accumulating bounded invariants of the system. Since all instances work on the same transition system, a bounded invariant discovered by one instance is valid for other instances as well. Thus, multiple reachability engines can share their information through a global database of bounded invariants. Additionally, in this setting each FiRE has a *filter* which controls which invariants are sent and received. The filter can be set to send and receive all or none invariants, or it can implement a heuristic. For example, it might be beneficial to send out only sufficiently small invariants to avoid burdening the other instances too much.

Cooperation of IcEs. Unlike FiREs, it is not immediately obvious what information IcEs could share between themselves. Natural candidates are elements of the base frame or the successor frame. However, one needs to be careful since different IcEs could be working on different levels and thus directly including lemmas from other instance might violate the invariants of these frames. Our solution is to accept external information in a way that can be modelled using the rule **Add-Lemma** and thus guarantee to preserve the correctness of the engine. Each engine sends out elements of the successor frame $\mathcal{G}$. When an engine is working on a level n and a lemma is pushed to $\mathcal{G}$, it is guaranteed to be at least $(n{+}1)$-invariant. Moreover, it is an *interesting* bounded invariant in the sense that this engine so far believes it should be part of the inductive strengthening. The engine sends such lemma to the global pool for other instances to see. When another engine receives this $(n{+}1)$-invariant, it checks if it can apply **Add-Lemma** to add it to its base frame. If the engine's current working level is higher than $n{+}1$, such bounded invariant cannot be added. Moreover, our preliminary experiments showed that it is better to have additional checks in the filter for incoming lemmas in order not to spend too much time processing useless external lemmas. We discuss our implementation and the experimental results with different settings of sharing information in Sec. 6.

5 PD-KIND as an Instance of IcE/FiRE

In this section we reformulate the original description of PD-KIND [20] in terms of our IcE/FiRE framework. This reformulation enables us to identify the freedom in the algorithm that can be utilized for the portfolio approach to parallelization. Additionally, the techniques mentioned in Sec. 4 for sharing information between cooperating instances will become directly applicable for PD-KIND. On top of that, it allows us to prove the correctness of the parallel version of the algorithm.

5.1 Induction-Checking Engine of PD-KIND

The induction-checking engine of PD-KIND uses an extended configuration $\langle \mathcal{F}, \mathcal{G}, n, k, Q, n_{CTI} \rangle$, where n_{CTI} remembers the number of steps needed to reach a non-$\mathcal{F}$ state from an $\mathcal{F}$ state. This helps to determine $n' > n$ such that all elements of $\mathcal{G}$ are n'-invariants when applying **Next-Level**.

Additionally, IcE of PD-KIND maintains a mapping CEX of elements of $\mathcal{F}$ to potential counter-examples they block. Formally, CEX is a function from $\mathcal{F}$ to state formulas such that for each $l \in \mathcal{F}$, $l \implies \neg CEX(l)$ and every $CEX(l)$-state can reach a $\neg P$-state. Maintaining the potential counter-examples in addition to the bounded invariants allows for earlier discovery of real counter-examples. It also provides a possible fall-back in case the bounded invariant is too strong to be inductive.

The initial configuration of IcE is $\langle \{P\}, \emptyset, 0, 1, \{P\}, 1 \rangle$, with $CEX(P) = \neg P$, and the engine makes progress using the following set of rules.

Safe:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, \emptyset, n_{CTI} \rangle}{SAFE} \quad \textbf{if } \{ \mathcal{F} = \mathcal{G}$$

Next-Level:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, \emptyset, n_{CTI} \rangle}{\langle \mathcal{G}, \emptyset, n', k', \mathcal{G}, n' + k' \rangle} \quad \textbf{if } \begin{cases} \mathcal{F} \neq \mathcal{G} \\ n' = n + n_{CTI} \\ 1 \leq k' \leq n' + 1 \end{cases}$$

Push-Lemma:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\}, n_{CTI} \rangle}{\langle \mathcal{F}, \mathcal{G} \cup \{l\}, n, k, Q, n_{CTI} \rangle} \quad \textbf{if } \{ l \text{ is } \mathcal{F}^k\text{-inductive}$$

Unsafe:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\}, n_{CTI} \rangle}{UNSAFE} \quad \textbf{if } \{ CEX(l) \text{ is reachable in } [n{+}1, n{+}k] \text{ steps}$$

Add-Lemma:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q, n_{CTI} \rangle}{\langle \mathcal{F} \cup \{l'\}, \mathcal{G}, n, k, Q \cup \{l'\}, n_{CTI} \rangle} \quad \textbf{if } \begin{cases} \exists l \in Q \text{ s.t.} \\ \neg CEX(l) \text{ is not } \mathcal{F}^k\text{-inductive} \\ \text{with } c' \text{ being its } CTI \\ \textbf{Unsafe} \text{ is not applicable} \\ l' \text{ is } n\text{-invariant that blocks } c' \\ CEX(l') = c' \end{cases}$$

Bad-Lemma:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\}, n_{CTI} \rangle}{\langle \mathcal{F} \cup \{l'\}, \mathcal{G} \cup \{l'\}, n, k, Q, n'_{CTI}) \rangle} \quad \textbf{if } \begin{cases} N \in [n{+}1, n{+}k] \\ \neg l \text{ reachable in } N \text{ steps} \\ l' = \neg CEX(l) \\ \neg CEX(l) \text{ is } \mathcal{F}^k\text{-inductive} \\ n'_{CTI} = min(N, n_{CTI}) \end{cases}$$

Strengthen-Lemma:

$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\}, n_{CTI} \rangle}{\langle \mathcal{F} \cup \{l'\}, \mathcal{G}, n, k, Q \cup \{l'\}, n_{CTI} \rangle} \quad \textbf{if} \begin{cases} \neg CEX(l) \text{ is } \mathcal{F}^k\text{-inductive} \\ l \text{ is not } \mathcal{F}^k\text{-inductive} \\ \text{with } c' \text{ being } CTI \\ \textbf{Bad-Lemma} \text{ is not applicable} \\ l' \text{ is } n\text{-lemma s.t.} \\ l' \implies l \wedge \neg c' \\ CEX(l') = CEX(l) \end{cases}$$

A run of the engine starts from the initial configuration and applies the rules until **Safe** or **Unsafe** is applicable (which is generally not guaranteed to happen). The engine can be viewed as operating on a certain level, defined by the parameter n. At each level, the engine attempts to prove that the n-invariants from $\mathcal{F}$ are $\mathcal{F}^k$-inductive, strengthening the frame in the process if necessary or giving up on n-invariants that do not hold for higher levels. When all elements of the (refined) frame $\mathcal{F}$ have been processed two cases can happen. Either the whole frame $\mathcal{F}$ has been pushed, in which case the engine can terminate using **Safe**, or some element could not be pushed and thus **Next-Level** is applied.

If all elements have not been pushed yet, that is, Q is not empty, then an n-invariant l from Q is picked and processed in the following way: When l is $\mathcal{F}^k$-inductive then l, and consequently $\neg CEX(l)$, is in fact at least $(n{+}1)$-invariant. In this case **Push-Lemma** is applied and l is removed from Q.

If **Push-Lemma** is not applicable and $\neg CEX(l)$ is not $\mathcal{F}^k$-inductive then there exists a CTI witnessing this. This CTI can be either real (reachable in $\mathcal{S}$) or spurious (not reachable in $\mathcal{S}$). A bounded reachability query is issued to FiRE to determine the status. If it is real, the system $\mathcal{S}$ is unsafe because $CEX(l)$ is reachable and $\neg P$ is reachable from $CEX(l)$. In this case the algorithm terminates by applying **Unsafe**. If CTI is spurious then a new lemma blocking it is returned from FiRE and added to $\mathcal{F}$ by applying **Add-Lemma**.

The last possibility is that l is not $\mathcal{F}^k$-inductive but $\neg CEX(l)$ is $\mathcal{F}^k$-inductive. Now the reachability query regarding the CTI for l is issued to FiRE. If it is not reachable then l is strengthened using the reason of unreachability returned by FiRE – **Strengthen-Lemma** is applied. If it is reachable then l is not an invariant of the system and must be discarded. **Bad-Lemma** is applied and l is replaced by $\neg CEX(l)$. Since we already know that $\neg CEX(l)$ is $\mathcal{F}^k$-inductive, it can be immediately pushed to the next frame.

This formalization of PD-KIND allows us to prove its correctness, building on the correctness of the abstract induction-checking engine (see Lemma 1). We extend the proof for parallel version in Sec. 5.3.

Lemma 2. *If* PD-KIND *terminates using the rule* ***Safe*** *(****Unsafe****), the transition system is safe (unsafe).*

Proof. For **Safe**, notice that PD-KIND's run can be viewed as a run of the abstract engine (Sec. 4.1). To avoid name clashes we use a prime to denote the PD-KIND's rules in this proof. All four rules **Safe**', **Push-Lemma**', **Next-Level**'

and **Add-Lemma**' directly map to their abstract counterpart. **Bad-Lemma** is just **Drop-Lemma** applied on l followed by **Add-Lemma** and **Push-Lemma** on $\neg CEX(l)$. Finally, **Strengthen-Lemma** is **Drop-Lemma** applied on l, followed by **Add-Lemma** applied on l'. Consequently, each PD-KIND's run terminating with **Safe**' is mapped to an abstract engine's run terminating with **Safe**. By Lemma 1, the system is safe.

For **Unsafe**, we show that the following invariant is preserved throughout the run: For each l in $\mathcal{F} \cup \mathcal{G} \cup Q$, $CEX(l)$ can reach $\neg P$. The invariant holds for the initial configuration since $\mathcal{F} \cup \mathcal{G} \cup Q = \{P\}$ and $CEX(P) = \neg P$. **Add-Lemma** preserves the invariant since for the only new lemma l', $CEX(l')$ can reach $CEX(l)$, which can reach $\neg P$ by the induction hypothesis. The invariant is also preserved by **Bad-Lemma** and **Strengthen-Lemma** as $CEX(l') = CEX(l)$ for the only new lemma l' and an old lemma l. As the other rules do not change the set $\mathcal{F} \cup \mathcal{G} \cup Q$, we can conclude that the invariant is always preserved. Thus, when the algorithm terminates by rule **Unsafe**, $\neg P$ is reachable and the system is unsafe. □

5.2 Finite Reachability Engine of PD-KIND

The finite reachability engine used in PD-KIND [20] can be described as IC3-like algorithm. It answers the bounded reachability queries using a sequence of *reachability frames* and local reasoning only, i.e., it does not unroll the transition relation. A *reachability frame* at level n, $\mathcal{R}_n$, is a set of n-invariants. Consequently, the set of $\mathcal{R}_n$-states over-approximates the set of states reachable in n steps or less. Unlike IC3, there is no further condition on the reachability frames. They do not need to be monotone nor form an inductive sequence. Like IC3, when FiRE receives a query $\langle s, i \rangle$, it checks if it is reachable in one step from $\mathcal{R}_{i-1}$ using a simple satisfiability query $\mathcal{R}_{i-1} \wedge T \wedge s'$. If it is unreachable, then FiRE generalizes the reason for unreachability using *Craig interpolation* and returns the answer together with the reason. If it is reachable, then FiRE computes a predecessor t of s and recursively calls itself with query $\langle t, i-1 \rangle$. If this predecessor turns out to be unreachable, the $(i-1)$-invariant witnessing the unreachability is used to refine $\mathcal{R}_{i-1}$ and s is checked again. If the recursive sequences of calls ever reaches an initial state, then the information about reachability, together with the trace made of the predecessors is gradually returned.

Note that the only condition required for reachability frame $\mathcal{R}_n$ is that it consists of n-invariants. In sequential setting FiRE learns new bounded invariants on its own as it processes more and more reachability queries. However, in parallel setting it can also receive bounded invariants from external source. More specifically, it can receive bounded invariants discovered by other instances of the same engine working in parallel on the same problem. Additionally, different interpolation algorithms can be used in different instances, thus allowing the engines to spread the search for useful bounded invariants.

5.3 Parallel PD-KIND

Since PD-KIND is an instantiation of the IcE/FiRE framework, it can be readily plugged into the abstract parallel framework with information sharing described in Sec. 4.3.

The bounded reachability information is stored in form of reachability frames consisting of bounded invariants. Whenever FiRE learns new bounded invariant as a response to bounded reachability query made by IcE, it can send it to the other instances. It can also periodically query the common pool for new bounded invariants and when it receives an external i-invariant, it can directly add it to its reachability frame $\mathcal{R}_i$.

Similarly, IcE sends out bounded invariants when it manages to push them to the successor frame. When it receives an external bounded invariant, it must check the necessary condition for adding it to the base frame. If the condition is not met, it simply drops the lemma. Otherwise, it uses a heuristic to determine usefulness of the lemma. Since PD-KIND assumes that each element of the base frame is associated with a potential counter-example through the mapping CEX, each bounded invariant l that is sent out by IcE must also be accompanied by its companion $CEX(l)$.

It is important for the success of a parallel approach to *diversify* the search for the solution. It was not possible to discuss this for the abstract framework as it requires the concrete algorithm with its concrete settings that drive the behaviour of the algorithm. Here we identify the key points where the behaviour of PD-KIND can be adjusted and finally give an algorithm capturing PD-KIND as an instance of IcE/FiRE framework in the parallel setting.

Choosing the depth of induction. When the induction engine moves to the next level n by applying **Next-Level** there is freedom to choose a new value k of the induction depth from the interval $[1, n+1]$. The behaviour of the algorithm can be greatly influenced by the value of the induction depth it uses. For example, choosing large k requires large unwinding of the transition relation when SAT/SMT solver is used and the inductive checks become slower. On the other hand preferring larger k can lead to faster exploration of the search space. Moreover an obligation might be $\mathcal{F}^k$-inductive, and thus successfully pushed, but not $\mathcal{F}^{k'}$-inductive for $k' < k$. We denote the strategy to choose the new value of induction depth whenever **Next-Level** is applied as κ.

Obligation processing strategy. Several rules might be applicable given a configuration with nonempty queue of obligations Q. However, once the obligation to be processed is chosen, there is no more freedom. The conditions of the rules are mutually exclusive for a fixed obligation $l \in Q$. Which rule applies for a particular obligation l is determined by its properties and the properties of $CEX(l)$. Therefore, the behaviour of the algorithm can be controlled through the strategy determining the obligation to pick from the queue. We denote the strategy to pick the next obligation from Q by ω.

Learning strategy The finite reachability engine computes bounded invariants as certificates of unreachability. Theoretically, the certificate of unreachability for a query $\langle s, i \rangle$ could be $\neg s$. However, this leads to terrible performance in

practice as it excludes only s and nothing else. Therefore, FiRE uses more sophisticated techniques to compute bounded invariants that are stronger and exclude more unreachable states. FiRE of PD-KIND uses Craig interpolation for computation of bounded invariants. However, Craig interpolant for a given problem is in general not unique and there exist techniques for computing different interpolants in propositional logic and in theories of first-order logic. The use of different interpolation algorithms leads to different bounded invariants and this can have a huge influence on the performance of the whole algorithm (see Sec. 6). We denote the strategy for computing the bounded invariants as σ.

Algorithm 1 PD-KIND in the parallel setting of IcE/FiRE

```
1: procedure RUN(S, κ, ω, σ)
2:     C = ⟨F, G, n, k, Q, n_CTI⟩ ← ⟨{P}, ∅, 0, 1, {P}, 1⟩        ▷ Initial configuration
3:     while True do
4:         if Q = ∅ then
5:             if F= G then return SAFE                          ▷ Terminate using rule Safe
6:             else
7:                 Apply Next-Level on C with κ
8:                 continue
9:             end if
10:        end if
11:        FiRE.SendReceive()          ▷ FiRE sends and receives bounded invariants
12:        C ← IcE.Receive(C)                  ▷ IcE receives bounded invariants
13:        l ← ω(Q)                                          ▷ Pick obligation to process
14:        c ← CEX(l)
15:        switch ⟨l, c⟩                          ▷ Pick rule based on properties of l, c
16:            case l is F^k-inductive
17:                Apply Push-Lemma for l on C
18:                IcE.Send(⟨l, c, n+1⟩)          ▷ IcE sends pushed bounded invariant
19:            case c is reachable in [n+1, n+k] steps
20:                return UNSAFE                          ▷ Terminate using rule Unsafe
21:            case ¬c is not F^k-inductive
22:                Apply Add-Lemma with σ on C
23:            case ¬l is reachable in [n+1, n+k] steps
24:                Apply Bad-Lemma for l
25:            case None of the above condition is met
26:                Apply Strengthen-Lemma with σ on C for l
27:    end while
28: end procedure
```

The run of a single instantiation of IcE/FiRE as PD-KIND in a parallel setting with information sharing is presented in pseudocode as Algorithm 1. The input is a triple $\mathcal{S} = \langle I, T, P \rangle$ representing the transition system and the property together with the three strategies κ, ω, σ that determine the behaviour of the algorithm at the previously identified non-deterministic steps.

Lemma 3. *The parallel version of* PD-KIND *with information exchange is correct. If it reports SAFE (UNSAFE), the system is safe (unsafe).*

Proof. The correctness of exchanging the bounded invariants between reachability engines has been discussed already in Sec. 4.3. The only new step IcE does is incorporating an external lemma l from another PD-KIND instance, together with a potential counter-example that it blocks. This is done only if the condition of the abstract rule **Add-Lemma** is satisfied and thus the invariants ensuring the correctness of SAFE answer are preserved. Moreover, the invariant from the proof of Lemma 2 is preserved and thus also UNSAFE answer is correct. □

6 Implementation and Experiments

Our implementation of the parallel PD-KIND algorithm is based on the open-source model checker Sally [20] and uses the SMTS framework [26] for parallelization and information exchange. We have extended Sally with API for sending and receiving information. In our experiments Sally was using Yices [13] for checking satisfiability and OpenSMT [17] for the interpolation queries.[4]

The benchmarks were taken from the transition systems category of CHC COMP 2019[5], where the problem is encoded using the theory of linear real arithmetic. Out of 244 benchmarks, 7 problematic ones were excluded due to reasons such as the presence of a non-linear operation. All experiments were run on a single multi-core machine with 16 Intel® Xeon® X5687 @ 3.6 GHz CPUs and 180 GB of RAM. The resources were restricted to 1000 seconds of timeout and 6GB of memory *per one instance* of Sally. This means that configurations with more instances are effectively granted more memory and CPU time. This choice is in line with our goal of improving the solver's wall clock time.

All instances use the default strategy of Sally when they are choosing the depth of induction (κ from Algorithm 1). The obligation processing strategy ω is a priority queue based on a score assigned to obligations, randomized to diversify the behaviour of different instances. The learning strategy σ is diversified primarily by using different interpolation algorithms in OpenSMT and secondary by using different random seed for the SMT search. Three different LRA interpolation algorithms were used: Farkas interpolation algorithm [28], dual Farkas, and an interpolation algorithm based on decomposing Farkas interpolants [7]. We denote these as PF, DF and PD, respectively.

In the experiments we seek answers to the following questions:

1. How does the system compare to the state-of-the-art?
2. How important is the sharing of information between various instances?
3. How does the approach scale when the number of instances is increased?
4. How do different interpolation algorithms contribute to the overall performance?

[4] All benchmarks, tools and results are bundled together in an artifact available at `https://doi.org/10.5281/zenodo.3484097`

[5] https://github.com/chc-comp/chc-comp19-benchmarks/tree/master/lra-ts

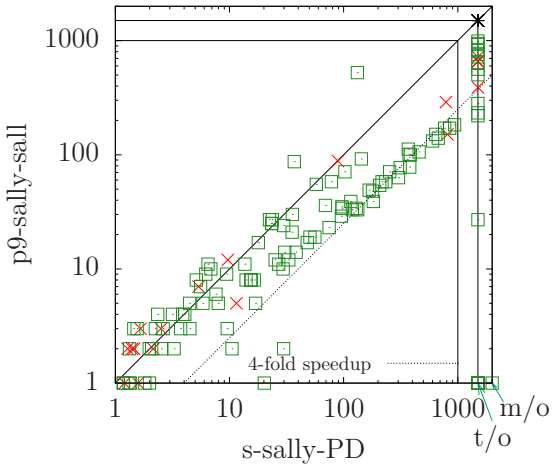

Fig. 3: Best parallel configuration against the winner of LRA-TS category of CHC COMP 2019

Comparison to the state-of-the-art. The main result of the experiments is summarized in Fig. 3 that compares the performance of the winner of the transition systems category of CHC COMP 2019 (sequential Sally using PD interpolation algorithm in OpenSMT) with our parallel implementation with nine instances sharing information between IcEs and between FiREs. The parallel implementation achieves 4-fold speedup on a significant number of instances and solves 224 instances compared to 197 instances solved by the sequential version.

We also compared our parallel implementation to P3 [24], the parallel implementation of Spacer [22] that also allows sharing information between solver instances. We also add the comparison with the sequential Spacer, the default Horn clause engine in Z3 [29].[6] The results are summarized in Fig. 4. Our framework significantly outperforms Spacer on safe instances. Interestingly, Spacer seems to fare better on unsafe instances.

Information sharing. Fig. 5 summarizes the performance of 4 configurations: no information sharing (sno), sharing between FiREs only (sreach), sharing between IcEs only (sind), and all sharing enabled (sall). In these configurations six instances were running in parallel (two instances for each interpolation algorithm PF, DF and PD). For comparison, the figure includes results of sequential versions with different interpolation algorithms. Note that the runtimes of the parallel implementation were rounded to the whole seconds and this creates an effect of "stairs" for the low runtimes in cactus plots with logarithmic scale. There is also a significant number of instances solved almost instantly and for this reason the axes start at 1 second runtime and 50 instances solved.

A clear gap is visible between the best sequential version and the parallel versions indicating that the parallel approach yields a significant improvement even without information sharing. Sharing information between FiREs is helpful, but

[6] Results for Z3-4.8.5 with default settings.

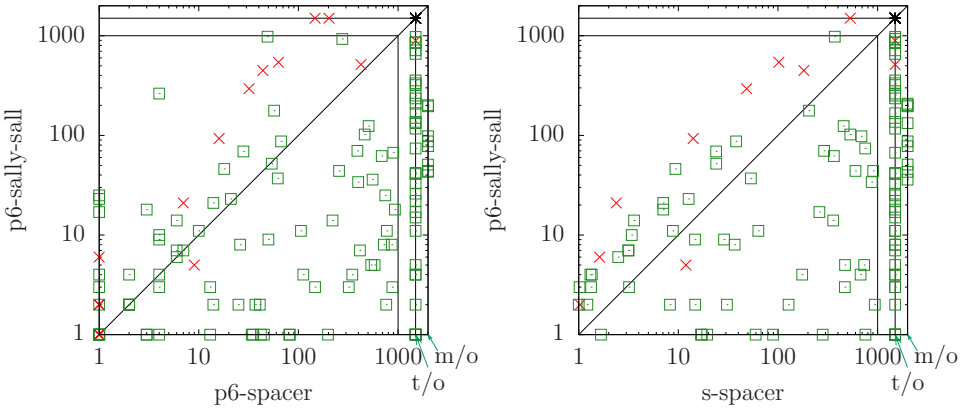

Fig. 4: Comparison of parallel SALLY and parallel SPACER using 6 communicating instances

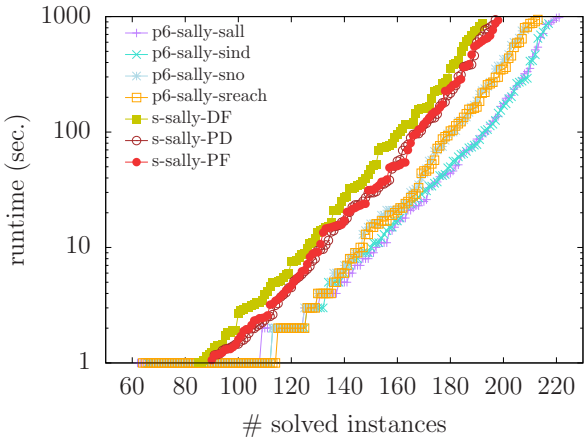

Fig. 5: The effect of sharing information

the effect is not that significant compared to sharing information between IcEs, which is crucial for improving performance on many benchmarks. Configurations with sharing reachability information disabled (**p6-sally-sno**, **p6-sally-sind**) do not profit much from enabling it (**p6-sally-sreach**, **p6-sally-sall**). However, some hard benchmarks could only be solved by allowing reachability information to be shared. On the other hand, enabling the sharing of induction information does boost the performance significantly. We conclude that the best performance was achieved by enabling sharing information between both IcEs and FiREs.

Scalability. We compared the performance of one, two, six and nine instances with all information sharing enabled. The results, summarized in Fig. 6, show that adding more instances improves the performance, both decreasing the run-

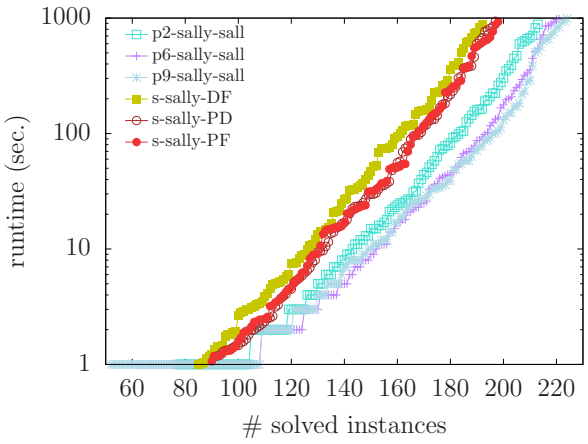

Fig. 6: Scalability experiments

time and solving more benchmarks with the configurations solving 197, 213, 221 and 224 instances, respectively.

The effect of interpolation. The large jump when moving from sequential solving to two instances running in parallel can be in part contributed to different interpolation algorithms. We investigate this further in Figure 7. We compared configurations using six instances when the interpolation algorithm varies (**p6-sally-sall**), when the interpolation algorithm is fixed to PF for all instance (**p6-sally-sall-PF**), and when it is fixed to PD (**p6-sally-sall-PD**). We also added a configuration of just two instances (one with PF, one with PD). The results show that varying the interpolation algorithm is very important as the performance of **p2-sally-sall** is comparable to that of **p6-sally-sall-PD** and **p6-sally-sall-PF** while **p6-sally-sall** performs significantly better.

The experiments show that our parallel algorithm performs substantially better than its sequential version. Its success can be contributed to more than one factor: The use of different interpolation algorithms helps to solve more benchmarks compared to a single interpolation algorithm used by all instances. Sharing information between solver instances can significantly reduce the runtime and thus solve more instances within the time limit. The major part of this can be contributed to the sharing of induction information, but sharing reachability information does help as well. The scalability experiments show continuing improvement up to nine instances. Additionally, our algorithm compares favorably with the state-of-the-art parallel implementation of Spacer, outperforming it significantly on the safe instances. Since Spacer is performing better on unsafe instances, the integration of the two algorithms within the SMTS framework to get the best of both tools is an interesting possibility for the future work.

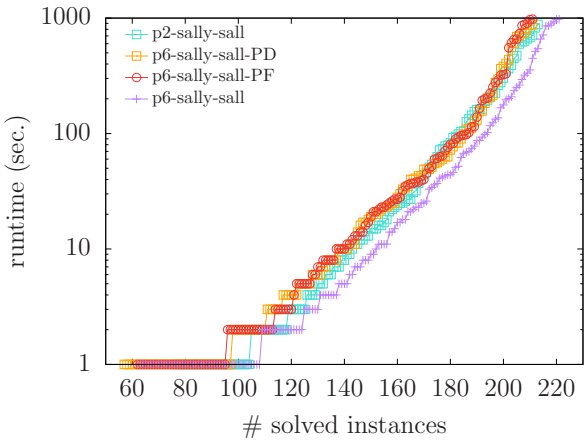

Fig. 7: The effect of using different interpolation algorithms

7 Conclusions

The IC3 algorithm [8] has arguably given a significant boost to symbolic model checking as witnessed by the number of new algorithms it has inspired. An early observation first made in [8] and later independently verified for example in [10,24] states that these algorithms are particularly amenable for parallelization. A recent pragmatic addition to the base algorithmic idea aims at obtaining higher quality reachability lemmas by k-induction and naturally splits the IC3 algorithm into two engines, one for induction and the other for computing bounded reachability.

This idea changes the way a lemma sharing parallel portfolio can be implemented for the class of algorithms, a question that was fundamental in IC3 from the beginning. In this work we provide the IcE/FiRE architecture that addresses this question by separating the two engines and their lemma storages and allowing parallel running solvers to share lemmas among their respective engines. We show experimentally that this approach provides a good speed-up in multi-core environments, and that the solver surpasses in speed and number of instances solved the current state-of-the-art on proving safety of transitions systems.

In future we plan to extend the presented idea in several ways. We will generalize the approach to solving constrained Horn clauses. We plan to study closer possible heuristics for sharing lemmas between the solvers, and to determine under what conditions the lemmas can be shared between an induction engine and a reachability engine. Aside from parallel portfolio, we would also like to study how search space partitioning and approaches such as the *parallelization tree* [19] could be applied in the context of the algorithm.

Acknowledgements. This work was partially supported by the Czech Science Foundation project nr. 18-17403S, by the Charles University institutional funding SVV 260451 and by the Swiss National Science Foundation (SNSF) grant 200020_166288.

References

1. Barnat, J., Bloemen, V., Duret-Lutz, A., Laarman, A., Petrucci, L., van de Pol, J., Renault, E.: Parallel model checking algorithms for linear-time temporal logic. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning., pp. 457–507. Springer (2018)
2. Beyer, D., Dangl, M.: Software verification with pdr: Implementation and empirical evaluation of the state of the art (2019)
3. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 622–640. Springer International Publishing, Cham (2015)
4. Beyer, D., Dangl, M., Wendler, P.: A unifying view on smt-based software verification. Journal of Automated Reasoning **60**(3), 299–335 (Mar 2018)
5. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Tools and Alg. for the Const. and Anal. of Systems (TACAS '99). LNCS, vol. 1579, pp. 193–207 (1999)
6. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (ctigar). In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 831–848. Springer International Publishing, Cham (2014)
7. Blicha, M., Hyvärinen, A.E.J., Kofroň, J., Sharygina, N.: Decomposing Farkas interpolants. In: Vojnar, T., Zhang, L. (eds.) Proc. TACAS 2019. LNCS, vol. 11427, pp. 3–20. Springer (2019)
8. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer (2011)
9. Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Blazy, S., Jensen, T. (eds.) Static Analysis. pp. 145–161. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
10. Chaki, S., Karimi, D.: Model checking with multi-threaded IC3 portfolios. In: Jobstmann, B., Leino, K.R.M. (eds.) Proc. VMCAI 2016. LNCS, vol. 9583, pp. 517–535. Springer (2016)
11. Cimatti, A., Griggio, A.: Software model checking via ic3. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 277–293. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
12. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. The Journal of Symbolic Logic **22**(3), 269–285 (1957)
13. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer (2014)
14. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. pp. 125–134. FMCAD '11, FMCAD Inc, Austin, TX (2011)
15. Gurfinkel, A., Ivrii, A.: K-induction without unrolling. In: Stewart, D., Weissenbacher, G. (eds.) Proc. FMCAD 2017. pp. 148–155. IEEE (2017)
16. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing – SAT 2012. pp. 157–171. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
17. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer, Cham (2016)

18. Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Heule, M., Weaver, S.A. (eds.) Proc. SAT 2015. LNCS, vol. 9340, pp. 369–386. Springer (2015)
19. Hyvärinen, A.E.J., Wintersteiger, C.M.: Parallel satisfiability modulo theories. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 141–178. Springer (2018)
20. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Piskac, R., Talupur, M. (eds.) Proc. FMCAD 2016. pp. 85–92. IEEE (2016)
21. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. Electronic Proceedings in Theoretical Computer Science **72**, 55–62 (Oct 2011)
22. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. pp. 17–34. Springer, Cham (2014)
23. Lange, T., Prinz, F., Neuhäußer, M.R., Noll, T., Katoen, J.P.: Improving generalization in software ic3. In: Gallardo, M.d.M., Merino, P. (eds.) Model Checking Software. pp. 85–102. Springer International Publishing, Cham (2018)
24. Marescotti, M., Gurfinkel, A., Hyvärinen, A.E.J., Sharygina, N.: Designing parallel PDR. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 156–163. IEEE Presss (2017)
25. Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: Artho, C., Legay, A., Peled, D. (eds.) Automated Technology for Verification and Analysis. pp. 428–443. Springer International Publishing, Cham (2016)
26. Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: SMTS: distributed, visualized constraint solving. In: LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018 (2018)
27. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2013. pp. 1–13. Springer, Heidelberg (2003)
28. McMillan, K.L.: An interpolating theorem prover. Theoretical Computer Science **345**(1), 101–121 (2005)
29. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. pp. 337–340. Springer, Heidelberg (2008)
30. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Sharygina, N., Veith, H. (eds.) Proc. CAV 2013. LNCS, vol. 8044, pp. 53–68. Springer (2013)
31. Rakadjiev, E., Shimosawa, T., Mine, H., Oshima, S.: Parallel SMT solving and concurrent symbolic execution. In: 2015 IEEE Trustcom/BigDataSE/ISPA. vol. 3, pp. 17–26 (Aug 2015)
32. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. pp. 127–144. Springer (2000)
33. Vediramana Krishnan, H.G., Vizel, Y., Ganesh, V., Gurfinkel, A.: Interpolating strong induction. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 367–385. Springer International Publishing, Cham (2019)
34. Wieringa, S., Niemenmaa, M., Heljanko, K.: Tarmo: A framework for parallelized bounded model checking. In: Brim, L., van de Pol, J. (eds.) Proc. PDMC 2009. EPTCS, vol. 14, pp. 62–76 (2009)