Parallel and Incremental Verification of Hybrid Automata with Ray and Verse

Haoqing Zhu(✉), Yangge Li, Keyi Shen, and Sayan Mitra

Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Champaign, USA
{haoqing3,li213,keyis2,mitras}@illinois.edu

Abstract. Parallel and distributed computing holds a promise of scaling verification to hard multi-agent scenarios such as the ones involving autonomous interacting vehicles. Exploiting parallelism, however, typically requires handcrafting solutions using knowledge of verification algorithms, the available hardware, and the specific models. The Ray framework made parallel programming hardware agnostic for large-scale Python workloads in machine learning. Extending the recently developed Verse Python library for multi-agent hybrid systems, in this paper we show how Ray's fork-join parallelization can help gain up to 6× speedup in multi-agent hybrid model verification. We propose a parallel algorithm that addresses the key bottleneck of computing the discrete transitions and exploits concurrent construction of reachability trees, without locks, using dynamic Ray processes. We find that the performance gains of our new reachset and simulation algorithms increase with the availability of larger number of cores and the nondeterminism in the model. In one experiment with 20 agents and 399 transitions, reachability analysis using the parallel algorithm takes 35 min on a 8 core CPU, which is a 6.28× speedup over the sequential algorithm. We also present an incremental verification algorithm that reuses previously cached computations and compare its performance.

1 Introduction

The hybrid automaton framework is useful for precisely describing and simulating scenarios involving interacting autonomous vehicles and other types of intelligent agents [1,4,8,9,11,15,20,23]. Parallel computing holds promise in scaling the verification of such hybrid models to scenarios with many agents, which in turn, multiply the number of expensive mode transitions that have to be computed. Despite several recent efforts [3,7,12] that we discuss in Sect. 2, building effective parallel verification algorithms remains a difficult art. It requires detailed knowledge of the computing hardware, the parallelism in the target models and how parallelism could be exploited in the verification algorithm.

This research was funded in part by NASA University Leadership Initiative grant (80NSSC22M0070) and a research grant from the NSF (SHF 2008883).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
É. André and J. Sun (Eds.): ATVA 2023, LNCS 14215, pp. 95–114, 2023.
https://doi.org/10.1007/978-3-031-45329-8_5

Parallelization is particularly important for multi-agent hybrid scenarios. As the agents interact, they make decisions that are modeled as discrete transitions and the number of transitions to be computed usually blows up over the analysis time horizon. For *reachability analysis* algorithms—a mainstay for verification—the set of reachable states is computed using two functions: (a) a *postCont* function computes the set of states that can be reached over a fixed time, while the agents follow a given dynamics (or mode). And, (b) a *postDisc* function computes the change in state when an agent makes a decision or a transition. For nondeterministic decisions (e.g., brake *or* steer to avoid collision), multiple postDisc computations have to be performed. A reachability algorithm thus builds a tree, the *reachtree*, where each branch corresponds to the unique choices made by all the relevant agents. Even with deterministic models, as a set of states is propagated forward in reachability analysis, this set can trigger multiple transitions. All of this contributes to an explosion of the reachtree, which often grows exponentially with longer analysis horizons and larger numbers of agents.

The Ray framework from RISELab has made it easier for application developers to scale-up their data science and machine learning algorithms through parallel computing, without having them worry about compute infrastructure details [19,21]. Ray supports the fork-join style of parallelization with remote function calls that run on other cores on the same machine or other machines. This style of parallelization is hardware agnostic and is easier to use as it does not involve the explicit use of synchronization primitives like locks.

In this paper, we propose and implement a Ray-based parallel reachability algorithm that utilizes the fact that different branches in the reachtree can be computed independently. This approach allows us to compute multiple branches of the tree simultaneously, which improves analysis performance. We use the recently developed Python Verse library [18] for multi-agent hybrid systems, as the underlying framework within which we develop our parallelized algorithm. In our experiments, we see that the parallel reachability algorithm can be effective in improving performance, especially in more complex hybrid automata models where more nondeterministic branching can occur. In an experiment involving several vehicles in a 4-way intersection (called `isect(4,20)` in Sect. 5), we get a 6.28× performance boost over the original sequential version of the reachability algorithm in Verse.

In industrial applications, verification procedures can be integrated in the CI/CD pipelines [6,22]. For engineers to use the verification results, it has been observed that the algorithms should run in 15–20 min and not over hours [5,22,24]. One of the ways in which this level of performance is achieved in the above studies is by performing *incremental verification.* That is, the verification algorithm only runs on the relevant part of the codebase that changes in each developer commit, and the algorithm *reuses* proofs from in previous verification runs. Inspired by these observations, our second contribution in this paper is the development of an incremental verification algorithm for hybrid scenarios.

This algorithm maintains a cache which contains information on previously computed trajectories and discrete transitions. These results are indexed with the state of the system. If the system reaches a state similar to one in the cache,

then its result can be reused. Instead of caching the result of every timestep, the incremental verification algorithm uses the same task unit as the parallel algorithm, making it possible to use both algorithms at the same time and have the best of both worlds. In the experiments we see that the incremental verification algorithm is able to reuse some computations and significantly speed up analysis in some situations, while in others it provides no benefits.

The rest of the paper is organized as follows: In Sect. 2 we discuss related work on parallel verification of hybrid systems. Section 3 introduces hybrid multi-agent scenarios and how they translate into hybrid automata. We will present the design and the correctness of our parallel and incremental verification algorithms in Sect. 4. Finally, in Sect. 5 we will present the experimental evaluation of these algorithms.

2 Related Work

A number of software tools have been developed for creating, simulating, and analyzing hybrid system models. Table 1 summarizes the actively maintained tools and those that have incorporated some form of parallelization.

Table 1. Parallelization methods used in hybrid verification tools.

Tool name	Parallelization target	Supported dynamics	Language/library
C2E2 [8]	None	nonlinear	C++
SpaceEx [11]	None	linear	Java
Flow* [4]	None	nonlinear	C++
DryVR [10]	None	nonlinear	Python
XSpeed [12]	CPU & GPU	linear	C++
JuliaReach [3]	CPU	nonlinear	Julia
CORA [1]	None	nonlinear	Matlab
dreach [17]	None	nonlinear	C++
HyLAA [2]	None	linear	Python
PIRK [7]	CPU & GPU	nonlinear	C++, pFaces
this paper	CPU	nonlinear	Python, Ray

Both CPUs and GPUs can be used to parallelize computation, but they differ in the type of tasks suitable for parallelization on each. The complex cores of CPUs today make them good at computing complex algorithms serially, but common desktop-grade CPUs only have 8 to 32 cores. On the other hand, GPUs have much simpler cores but many of them, from hundreds to thousands. This makes them efficient at performing many numerical calculations at the same time, but unsuitable for any algorithm involving complex logic.

JuliaReach [3] utilizes CPU-based parallelization for boosting the performance of reachability computation. PIRK [7] uses the pFaces [16] runtime to parallelize reachability of continuous states using both CPUs and GPUs. XSpeed [12] besides CPU also support using GPUs for accelerating the computation of continuous and discrete state evolutions. The Ray framework [21] was developed at RISELab at UC Berkeley with the aim to make parallel and distributed computing easier for researchers by helping them focus on application development work, regardless of the specifics of their compute cluster. The Ray framework has been used in several successful projects, including Uber using it for performing large-scale deep learning training for autonomous vehicles. To the best of our knowledge, our work is the first to parallelize hybrid system reachability analysis with Ray.

One approach to parallelize reachability analysis algorithms is to spawn a collection of threads, each of which explores a part of the reachable state space. The downside of this approach is that resources, such as memory, are shared between different threads, which in turn, implies the need for locking. We have chosen to use Ray [21], a popular parallelization framework for Python, for implementing parallel verification algorithms. The use of Ray allows us to implement algorithms without using locks.

XSpeed is a hybrid automaton verification tool that incorporated several parallelization algorithms [12]. We have adapted the *AsyncBFS* algorithm implemented in XSpeed to Verse. However, there are several differences in the designs. First, XSpeed is only able to handle linear systems, more specifically, invertible linear systems, while our algorithm is able to handle non-linear systems. Second, explicit locking is no longer necessary due to Ray's design. Last but not least, we have chosen a different granularity of parallelization in Verse that is coarser than the algorithm presented in XSpeed. XSpeed's algorithm assigns one pair of postCont and postDisc as the task for a single thread. However, we have chosen to use a batch of several post operations as a task. This is mainly due to one of Python's limitations. The Python interpreter uses a Global Interpreter Lock (GIL) which prevents Python code from utilizing multi-threading capabilities, and multiple processes have to be used instead to achieve parallelism. In this case, all the resources, including the input hybrid automaton and computed reachable sets, have to be copied back and forth between different processes and the overhead for this can be high. Therefore, we choose to use the batch operation to reduce the number of copying needed.

3 Preliminaries: Hybrid Multi-agent Scenarios

In this paper, we are basing our algorithms on hybrid multi-agent scenarios. Each of these scenarios contains a collection of agents interacting in an environment. We will describe the agents and scenario in this section and in Sect. 3.2 we will discuss how a scenario formally defines a hybrid system.

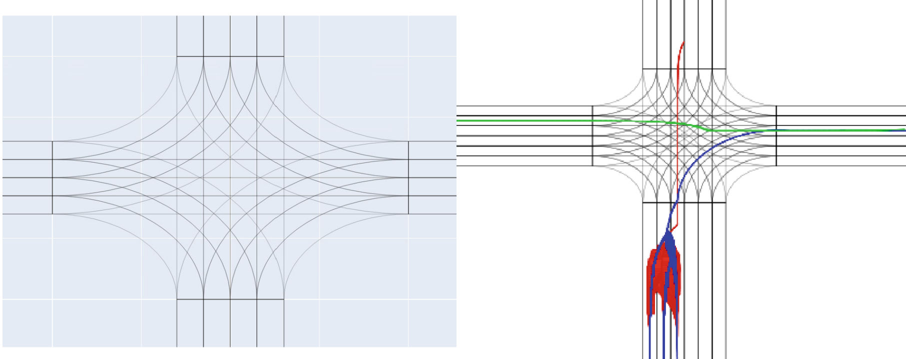

Fig. 1. *Left:* A 4-way intersection scenario with 2 lanes in each direction, showing the lane boundaries. Each lane extends very far outside of the picture. *Right:* Computed reachsets with 3 agents (represented with 3 different colors) on the intersection scenario with 3 lanes in each direction. The red car goes from north to south, the green car goes from east to west, and the blue car goes from east to south. (Color figure online)

3.1 Agents in Hybrid Scenarios

An *agent* is a hybrid automaton that reacts based on the states of all agents in the scenario. An $\mathcal{A}$ in a scenario with $k-1$ other agents is defined by a tuple $\mathcal{A} = \langle X, D, X^0, d^0, G, R, F\rangle$, where

1. X and D are the *continuous state space* and *discrete mode space* respectively, X^0 is the set of initial continuous states and d^0 is the initial mode;
2. G and R are the guards and resets functions for the agent, which jointly define the discrete transitions for the agent;
3. F is the *flow* function, which defines the evolution of the continuous states.

We will describe each of these components briefly.

X is the agent's continuous state space. In Fig. 1 we show the environment of an intersection example. The continuous state variables can be x, y, θ, v for the position, heading, and speed for the vehicle agents in this example. D is the agent's discrete mode space. In the intersection example, some of the modes can correspond to the internal state of the agent, while others can correspond to its location in the environment. One possible discrete mode could be $\langle$`SW-0`, `Normal`$\rangle$, where `SW-0` means the agent is tracking the leftmost lane going from south to west, and `Normal` means the agent is cruising along the road.

The guard G and reset R functions jointly define the discrete transitions. For a pair of modes $d, d' \in D$, $G(d, d') \subseteq X^k$ defines the condition under which a transition from d to d' is enabled, and $R(d, d') : X^k \to X$ defines how the continuous states of the agent are updated when the mode switch happens. Both of these functions take as input the full continuous states of all the other

$k-1$ agents in the scenario. This means that the transitions of every agent can depend on its own state that also on the observable[1] state of the other agents. For a single agent, we refer to the combination of guards and resets for that agent as a *decision logic*.

The *flow* function $F : X \times D \times \mathbb{R}^{\geq 0} \rightarrow X$ defines the continuous time evolution of the continuous state. For any initial condition $\langle x^0, d^0 \rangle \in Y$, $F(x^0, d^0)(\cdot)$ gives the continuous state of the agent as a function of time. In this paper, we use F as a black-box function (see footnote[2]).

3.2 Scenario to Hybrid Verification

A scenario SC is defined by a collection of k agent instances $\{\mathcal{A}_1 ... \mathcal{A}_k\}$. We assume agents have identical state spaces, i.e., $\forall i, j \in \{0, ..., k-1\}, X_i = X_j, D_i = D_j$, but they can have different decision logics and different continuous dynamics. We make this assumption to simplify the implementation of the handling of the decision logic. This does not affect the expressive power of Verse as different state variable and mode types could be unioned into a single type.

Next, we define how a scenario SC specifies a hybrid automaton $H(SC)$. We will use a hybrid automaton close to that in Definition 5 of [10]. As usual, the automaton has discrete and continuous states and discrete transitions defined by guards and resets.

Given a scenario with k agents, $SC = \{\mathcal{A}_1, ... \mathcal{A}_k\}$, the corresponding hybrid automaton $H(SC) = \langle \mathbf{X}, \mathbf{X}^0, \mathbf{D}, \mathbf{D}^0, \mathbf{G}, \mathbf{R}, \mathbf{TL} \rangle$, where

1. $\mathbf{X} := \prod_i X_i$ is the *continuous state space*. An element $\mathbf{x} \in \mathbf{X}$ is called a *state*. $\mathbf{X}^0 := \prod_i X_i^0 \subseteq \mathbf{X}$ is the set of *initial continuous states*.
2. $\mathbf{D} := \prod_i D_i$ is the *mode space*. An element $\mathbf{d} \in \mathbf{D}$ is called a *mode*. $\mathbf{d}^0 := \prod_i d_i^0 \subseteq \mathbf{D}$ is the *initial mode*.
3. For a pair of modes $\mathbf{d}, \mathbf{d}' \in \mathbf{D}$, $\mathbf{G}(\mathbf{d}, \mathbf{d}') \subseteq \mathbf{X}$ defines the continuous states from which a transition from $\mathbf{d}$ to $\mathbf{d}'$ is enabled. A state $\mathbf{x} \in \mathbf{G}(\mathbf{d}, \mathbf{d}')$ iff there exists an agent $i \in \{1, ..., k\}$, such that $\mathbf{x}_i \in G_i(\mathbf{d}_i, \mathbf{d}_i')$ and $\mathbf{d}_j = \mathbf{d}_j'$ for $j \neq i$.
4. For a pair of modes $\mathbf{d}, \mathbf{d}' \in \mathbf{D}$, $\mathbf{R}(\mathbf{d}, \mathbf{d}') : \mathbf{X} \rightarrow \mathbf{X}$ defines the change of continuous states after a transition from $\mathbf{d}$ to $\mathbf{d}'$. For a continuous state $\mathbf{x} \in \mathbf{X}$, $\mathbf{R}(\mathbf{d}, \mathbf{d}')(\mathbf{x}) = R_i(\mathbf{d}_i, \mathbf{d}_i')(\mathbf{x})$ if $\mathbf{x} \in G_i(\mathbf{d}_i, \mathbf{d}_i')$, otherwise $= \mathbf{x}_i$.
5. $\mathbf{TL}$ is a set of pairs $\langle \xi, \mathbf{d} \rangle$, where the *trajectory* $\xi : [0, T] \rightarrow \mathbf{X}$ describes the evolution of continuous states in mode $\mathbf{d} \in \mathbf{D}$. Given $\mathbf{d} \in \mathbf{D}, \mathbf{x}^0 \in \mathbf{X}$, ξ should satisfy $\forall t \in \mathbb{R}^{\geq 0}, \xi_i(t) = F_i(\mathbf{x}_i^0, \mathbf{d}_i)(t)$.

[1] The observable state is defined by a *sensor* function; here we assume that the full state is observable.

[2] This design decision is relatively independent. For reachability analysis, we currently uses black-box statistical approaches implemented in DryVR [10] and NeuReach [25]. If the simulator is available as a white-box model, such as differential equations, then the algorithm could use model-based reachability analysis.

In [18] it is shown that $H(SC)$ is indeed a valid hybrid automaton for a scenario with k agents $SC = \{\mathcal{A}_1, ..., \mathcal{A}_k\}$ provided that all agents have identical sets of states and modes, $Y_i = Y_j, \forall i, j \in \{0, ..., k-1\}$.

For some trajectory ξ we denote by $\xi.\mathit{fstate}$, $\xi.\mathit{lstate}$, and $\xi.\mathit{ltime}$ the initial state $\xi(0)$, the last state $\xi(T)$, and $\xi.\mathit{ltime} = T$. For a sampling parameter $\delta > 0$ and a length m, a δ-*execution* of a hybrid automaton $H = H(SC)$ is a sequence of m labeled trajectories $\alpha(\mathbf{x}^0, \mathbf{d}^0; m) := \langle \xi^0, \mathbf{d}^0 \rangle, ..., \langle \xi^{m-1}, \mathbf{d}^{m-1} \rangle$, such that

(1) $\xi^0.\mathit{fstate} = \mathbf{x}^0 \in \mathbf{X}^0, \mathbf{d}^0 \in \mathbf{D}^0$;
(2) $\forall i \in \{1, ..., m-1\}, \xi^i.\mathit{lstate} \in \mathbf{G}(\mathbf{d}^i, \mathbf{d}^{i+1})$ and $\xi^{i+1}.\mathit{fstate} = \mathbf{R}(\mathbf{d}^i, \mathbf{d}^{i+1})(\xi^i.\mathit{lstate})$;
(3) $\forall i \in \{1, ..., m-1\}, \xi^i.\mathit{ltime} = \delta$ for $i \neq m-1$ and $\xi^i.\mathit{ltime} \leq \delta$ for $i = m-1$.

We define the first and last state of an execution $\beta = \alpha(\mathbf{x}^0, \mathbf{d}^0; m) = \langle \xi^0, \mathbf{d}^0 \rangle, ..., \langle \xi^{m-1}, \mathbf{d}^{m-1} \rangle$ as $\beta.\mathit{fstate} = \xi^0.\mathit{fstate} = \mathbf{x}^0$, $\beta.\mathit{lstate} = \xi^{m-1}.\mathit{lstate}$ and the first and last mode as $\beta.\mathit{fmode} = \mathbf{d}^0$ and $\beta.\mathit{lmode} = \mathbf{d}^{m-1}$.

3.3 Bounded Reach Sets

We will define a pair of *post* operators, that will be useful in the computation of executions. Consider a scenario SC with k agents and the corresponding hybrid automaton $H(SC)$. For any $\delta > 0$, continuous state $\mathbf{x} \in \mathbf{X}$ and a pair of modes $\mathbf{d}, \mathbf{d}'$, the discrete $\mathit{post}_{\mathbf{d},\mathbf{d}'} : \mathbf{X} \to \mathbf{X}$ and continuous $\mathit{post}_{\mathbf{d},\delta} : \mathbf{X} \to \mathbf{X}$ operators are defined as follows:

$$\mathit{post}_{\mathbf{d},\mathbf{d}'}(\mathbf{x}) = \mathbf{x}' \iff \mathbf{x} \in \mathbf{G}(\mathbf{d}, \mathbf{d}') \text{ and } \mathbf{x}' = \mathbf{R}(\mathbf{d}, \mathbf{d}')(\mathbf{x})$$
$$\mathit{post}_{\mathbf{d},\delta}(\mathbf{x}) = \bigcup_{t \in [0,\delta)} \prod_{i \in \{1,\ldots,k\}} F_i(\mathbf{x}_i, \mathbf{d}_i, t)$$

These operators are also lifted to sets of states in the usual way. If part of the input states are not contained within the guard conditions, they will be ignored in the returned result by $\mathit{post}_{\mathbf{d},\mathbf{d}'}$.

In addition, we define $\mathit{post}_{\mathbf{d},\delta}(\mathbf{x}).\mathit{lstate} = \prod_{i \in \{1,\ldots,k\}} F_i(\mathbf{x}_i, \mathbf{d}_i, \delta)$, in other words $\mathit{post}_{\mathbf{d},\delta}(\mathbf{x}).\mathit{lstate}$ represents the frontier of the continuous states after δ-time. We conclude this section with the definition of the bounded reachable states of $H(SC)$.

Definition 1. *The bounded reachable states of* $H(SC)$ *is*

$$\mathit{Reach}(\boldsymbol{X}^0, \boldsymbol{d}^0, \delta, T_{\max}) = \bigcup_{\boldsymbol{x}^0 \in \boldsymbol{X}^0} \alpha(\boldsymbol{x}^0, \boldsymbol{d}^0; m)$$

where: (1) $\boldsymbol{X}^0 \subseteq \boldsymbol{X}$ *and* $\boldsymbol{d}^0 \in \boldsymbol{D}$ *are the initial states of the hybrid automaton; (2)* $T_{\max}$ *is the time horizon; (3)* $\alpha(\boldsymbol{x}^0, \boldsymbol{d}^0; m)$ *is a valid execution; (4)* $m = \lceil \frac{T_{\max}}{\delta} \rceil$ *is the length of execution.*

4 Parallel and Incremental Verification Algorithms

In this section, we will describe the parallel and incremental algorithms for computing reachable states that we have implemented in Verse. We will also discuss their correctness. For the sake of a self-contained presentation, we will first introduce several important notations and subroutines in the context of the sequential reachability algorithm. Then in Sect. 4.2 and 4.3, we will add in optimizations before getting to the final version of the algorithm.

4.1 Reachability Analysis

Recall that for a scenario SC and its hybrid system model $H(SC)$, $\mathbf{X}$ and $\mathbf{D}$ are respectively the continuous state space and discrete mode space. The building blocks for all reachability algorithms are two functions $\texttt{postCont}(\mathbf{d}^0, \delta, \mathbf{X}^0)$ and $\texttt{postDisc}(\mathbf{d}^0, \mathbf{d}^1, \mathbf{X}^0)$ that compute (or over-approximate) $post_{\mathbf{d}^0,\delta}(\mathbf{X}^0)$ and $post_{\mathbf{d}^0,\mathbf{d}^1}(\mathbf{X}^0)$, respectively. Similar to Sect. 3.3, we will use $\texttt{postCont}(\mathbf{d}^0, \delta, \mathbf{X}^0)$. *lstate* to denote the frontier of $\texttt{postCont}(\mathbf{d}^0, \delta, \mathbf{X}^0)$. In Verse, `postCont` is implemented using algorithms in [10,25].

The sequential `verify` function implements a reachability analysis algorithm using these post operators (Algorithm 1). This algorithm constructs an execution tree $Tree = V$ up to depth $T_{\max}$ in breadth-first order. Each node $N = \langle \mathbf{X}, \mathbf{d}, t, stride, children \rangle \in V$ is a tuple of a set of states, a mode, the start time, the stride, which can be computed by `postCont`, and children of the current node. The root is $\langle \mathbf{X}^0, \mathbf{d}^0, 0, stride, children \rangle$ given a initial set of states $\mathbf{X}^0$ and mode $\mathbf{d}^0$. The *children* field of each node provides the edge relations for the tree. There is an edge from $\langle \mathbf{X}, \mathbf{d}, t, stride, children \rangle$ to $\langle \mathbf{X}', \mathbf{d}', t', stride', children' \rangle$ if and only if $\mathbf{X}' = post_{\mathbf{d},\delta}(post_{\mathbf{d},\mathbf{d}'}(\mathbf{X})).lstate$ and $t' = t + \delta$. We will use the dot field access notation to refer to fields of a node. For example for a node N, $N.stride$ and $N.\mathbf{X}$ refers to the stride and the set of initial states in N.

Note that one of the arguments to the `verify_step` function is a node with only the $\mathbf{X}$, $\mathbf{d}$ and t fields populated. After this function executes, it populates the *stride* and *children* fields of the node N and returns a completed node.

To show the correctness of the `verify` algorithm, we will first show some key properties of the `verify_step` function in Proposition 1. Throughout this section, we fix a scenario SC and the corresponding hybrid automaton $H(SC)$. Let $\mathbf{X}$ and $\mathbf{D}$ be the continuous and discrete state spaces of $H(SC)$.

Algorithm 1

1: **function** `verify_step`(N, δ) **where** $N.stride = \varnothing$, $N.children = \varnothing$
2: $N.stride \leftarrow$ `postCont`$(N.\mathbf{X}, N.\mathbf{d}, \delta)$
3: **for** $\mathbf{d}' \in \mathbf{D}$ s.t. $\mathbf{G}(N.\mathbf{d}, \mathbf{d}') \cap N.stride.lstate \neq \varnothing$ **do**
4: $\mathbf{X}' \leftarrow$ `postDisc` (N.*stride.lstate*, N.**d**, **d**')
5: $N.children \leftarrow N.children \cup \langle \mathbf{X}', \mathbf{d}', N.t + \delta, \varnothing, \varnothing \rangle$
6: **return** N

7: **function** `verify`$(H, \mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max})$
8: $queue \leftarrow [\langle \mathbf{X}^0, \mathbf{d}^0, 0, \varnothing, \varnothing \rangle]$
9: $reachset \leftarrow \varnothing$
10: **while** $queue \neq \varnothing$ **do**
11: $N \leftarrow queue$`.dequeue()`
12: $N \leftarrow$ `verify_step`(N, δ)
13: $reachset \leftarrow reachset \cup N.stride$
14: **for** $N' \in children$ s.t. $N'.t < T_{\max}$ **do**
15: $queue$`.add`(N')
16: **return** *reachset*

Proposition 1. *For any set of states* $\boldsymbol{X}^0 \subseteq \boldsymbol{X}$*, mode* $\boldsymbol{d}^0 \in \boldsymbol{D}$ *and time* t*, the node* $N =$ `verify_step`$(\langle \mathbf{X}^0, \mathbf{d}^0, t, \varnothing, \varnothing \rangle, \delta)$ *satisfies the following:*

$$post_{\boldsymbol{d},\delta}(\boldsymbol{X}) \subseteq N.stride \tag{1}$$

$$\forall N' \in N.children, \boldsymbol{G}(N.\boldsymbol{d}, N'.\boldsymbol{d}) \cap N.stride.lstate \neq \varnothing \tag{2}$$

$$\forall N' \in N.children, post_{N.\boldsymbol{d},N'.\boldsymbol{d}}(post_{N.\boldsymbol{d},\delta}(N.\boldsymbol{X}).lstate) \subseteq N'.\boldsymbol{X}. \tag{3}$$

Proof. For (1), from line 2 in Algorithm 1, $N.stride =$ `postCont`$(N.\mathbf{X}, N.\mathbf{d}, \delta)$. As we assumed about `postCont` in the start of Sect. 4.1, $post_{N.\mathbf{d},\delta}(N.\mathbf{X}) \subseteq N.stride$ and $post_{N.\mathbf{d},\delta}(N.\mathbf{X}).lstate \subseteq N.stride.lstate$.

For (2), from the loop condition at line 3, for every children $N' \in N.children$: $\mathbf{G}(\mathbf{d}^0, N'.\mathbf{d}) \cap N.stride.lstate \neq \varnothing$

For (3), for every children $N' \in N.children$:

$$\begin{aligned} post_{N.\mathbf{d},N'.\mathbf{d}}(post_{N.\mathbf{d},\delta}(N.\mathbf{X}).lstate) &\subseteq post_{N.\mathbf{d},N'.\mathbf{d}}(N.stride.lstate) \\ &\subseteq \texttt{postDisc}(N.stride.lstate, N.\mathbf{d}, N'.\mathbf{d}) \\ &= N'.\mathbf{X} \end{aligned}$$

□

Proposition 2. *Given initial states* $\boldsymbol{X}^0 \subseteq \boldsymbol{X}$ *and* $\boldsymbol{d}^0 \in \boldsymbol{D}$*, time step* δ *and time horizon* $T_{\max}$*,*

$$Reach(\boldsymbol{X}^0, \boldsymbol{d}^0, \delta, T_{\max}) \subseteq \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}).$$

Proof. Let $m = \lceil \frac{T_{\max}}{\delta} \rceil$ be the height of the reachset tree, and $\epsilon = T_{\max} - (m-1) \times \delta$. According to the Definition 1 of bounded reachable states, we have $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, (m-1) \times \delta + \epsilon) = \bigcup_{\mathbf{x}^0 \in \mathbf{X}^0} \alpha(\mathbf{x}^0, \mathbf{d}^0; m)$.

We will prove by induction on the height of the reachset tree $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, (m-1) \times \delta + \epsilon)$. For the base case, $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, \epsilon) \subseteq \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, \delta)$. Let $N = \langle \mathbf{X}^0, \mathbf{d}^0, 0, \varnothing, \varnothing \rangle$, then it follows immediately that $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, \epsilon) = \mathit{post}_{\mathbf{d}^0, \epsilon}(\mathbf{X}^0) \subseteq \texttt{postCont}(\mathbf{X}^0, \mathbf{d}^0, \delta) = \texttt{verify_step}(N, \delta).\mathit{stride} = \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, \delta)$.

Induction hypothesis: Given $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, k \times \delta + \epsilon) \subseteq \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (1 + k) \times \delta)$ where $k \in [1, m-1)$, show $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta + \epsilon) \subseteq \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+2) \times \delta)$.

Induction step:

$$\begin{aligned}
\texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+2) \times \delta) &= \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta) \\
&\quad \cup \bigcup_{\mathbf{X}^k, \mathbf{d}^k} \texttt{verify_step}(\mathbf{X}^k, \mathbf{d}^k, \delta).\mathit{stride} \\
\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta + \epsilon) &= \bigcup_{\mathbf{x}^0 \in \mathbf{X}^0} \alpha(\mathbf{x}^0, \mathbf{d}^0; k+1) \\
&= \bigcup_{\mathbf{x}^0 \in \mathbf{X}^0} \alpha(\mathbf{x}^0, \mathbf{d}^0; k) \cup \bigcup_{\mathbf{x}', \mathbf{d}'} \alpha(\mathbf{x}', \mathbf{d}'; 1)
\end{aligned}$$

where $\mathbf{d}' \in \mathbf{D}$ s.t. $\alpha(\mathbf{x}^0, \mathbf{d}^0; k).\mathit{lstate} \in \mathbf{G}(\alpha(\mathbf{x}^0, \mathbf{d}^0; k).\mathit{lmode}, \mathbf{d}')$ and $\mathbf{x}' = \mathbf{R}(\alpha(\mathbf{x}^0, \mathbf{d}^0; k).\mathit{lmode}, \mathbf{d}')(\alpha(\mathbf{x}^0, \mathbf{d}^0; k).\mathit{lstate})$.

$\mathbf{X}^k$ and $\mathbf{d}^k$ come from the nodes in the queue. Since the `verify` algorithm uses BFS, these nodes will be all the *children* from the last layer in $\texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta)$, thus $\bigcup_{\mathbf{X}^k, \mathbf{d}^k} \texttt{verify_step}(\mathbf{X}^k, \mathbf{d}^k, \delta) \supseteq \bigcup_{\mathbf{x}', \mathbf{d}'} \alpha(\mathbf{x}', \mathbf{d}', \epsilon)$.

Combining this with the induction hypothesis:

$$\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, k \times \delta + \epsilon) = \bigcup_{\mathbf{x}^0 \in \mathbf{X}^0} \alpha(\mathbf{x}^0, \mathbf{d}^0, k+1) \subseteq \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (1+k) \times \delta)$$

we get $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta + \epsilon) \subseteq \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+2) \times \delta)$, which completes the proof. □

4.2 Parallel Reachability with Ray

In this section, we show how we parallelize the verification algorithm shown above using Ray [21]. Ray uses remote functions as an abstraction for performing parallelization. Remote functions in Ray can be called on one process but will

be executed in another process. These processes can be on different cores of the same machine, or cores on other network-connected machines. Throughout this paper, we will assume that the remote functions execute on other cores within the same machine. However, we note that thanks to Ray's abstraction, our implementation can as easily take advantage of networked clusters.

For a Python function with arguments `f(args)`, the function `f` can be turned into a remote function by decorating the definition of the function with the `ray.remote` decorator. Such remote functions can be called via `f.remote(args)`. In order to simplify the pseudocode, we will simply use a **remote** keyword instead.

In Ray, two processes communicate through a distributed database. For a remote function, both the arguments and the return value will be stored in the database. From the caller's side, when a remote function is called, the arguments to the function will be sent automatically to the database, and a reference to the return value of the function is returned immediately. For the remote process, the arguments are first fetched from the database, then the function will run, and lastly, the return value is sent back to the database. The caller can poll and fetch the value back by using the `ray.wait()` function. For an array of value references `refs`, `ray.wait(refs)` blocks until one of the references in `refs` is available, fetches and returns that value along with the rest of the references.

The basic parallel algorithm is shown in Algorithm 2. The `verify_parallel` algorithm uses a queue to explore the tree just like `verify`, however there are two branches in the loop. One of them pops nodes from the queue and calls `verify_step` on the node as a remote function, while the other waits for the results to come back, processes the result, and adds new nodes to the queue. The algorithm prioritizes sending out computations, which means there can be multiple remote computations inflight at the same time, increasing parallelism and thus speedup. As more branching in the scenario benefits the algorithm more, we can use the number of leaves in the reachtree as a simple metric to measure this potential benefit. In other words, the more leaves there are in a scenario's reachtree, the more speedup we expect to see. Note that as several nodes can happen in parallel, they may be computed in a different order compared to `verify`.

Proposition 3. *For any set of states $\boldsymbol{X}^0 \subseteq \boldsymbol{X}$ and mode $\boldsymbol{d}^0 \in \boldsymbol{D}$,*

$$\texttt{verify_parallel}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}) = \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max})$$

Proof. To prove the equality, we can show that the set of calls to `verify_step` in `verify` and `verify_parallel` are the same. In `verify`, `verify_step` is called at line 12; in `verify_parallel`, `verify_step` is called at line 8 as a remote function call. We assume that remote calls in Ray will always return and that given the same arguments, remote function calls to `verify_step` will return the same values as non-remote calls. We can then compare the tree generated

Algorithm 2

```
1:  function verify_parallel(H, X^0, d^0, δ, T_max)
2:      queue ← [⟨X^0, d^0, 0, ∅, ∅⟩]
3:      refs ← ∅
4:      reachset ← ∅
5:      while queue ≠ ∅ ∨ refs ≠ ∅ do
6:          if queue ≠ ∅ then
7:              N ← queue.dequeue()
8:              refs.add(remote verify_step (N, δ))
9:          else                                    ▷ wait only when queue is empty
10:             ⟨N, refs⟩ ← ray.wait(refs)
11:             reachset ← reachset ∪ N.stride
12:             for N' ∈ N.children do
13:                 if N'.t < T_max then
14:                     queue.add(N')
15:     return reachset
```

by both `verify` and `verify_parallel` and prove by induction on the height of the tree currently computed. Note that due to the nondeterministic ordering of node traversal, the `verify_parallel` can begin computing nodes that have $k+1$ depth before finishing nodes at depth k.

Base Case: Given a set of initial states $\mathbf{X}^0 \subseteq \mathbf{X}$ and initial mode $\mathbf{d}^0 \subseteq \mathbf{D}$, we want to show that

$$\texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, 1 \times \delta) = \texttt{verify_parallel}(\mathbf{X}^0, \mathbf{d}^0, \delta, 1 \times \delta)$$

and the children of both trees are the same.

Let $N = \texttt{verify_step}(\langle \mathbf{X}^0, \mathbf{d}^0, 0, \varnothing, \varnothing \rangle, \delta)$, then:

$$\begin{aligned}\texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, 1 \times \delta) &= N.stride \\ &= \texttt{verify_parallel}(\mathbf{X}^0, \mathbf{d}^0, \delta, 1 \times \delta)\end{aligned}$$

Thus, the two trees are equal. The children for both are $N.children$, and they are equal.

Induction Step: Given $\texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, k \times \delta) = \texttt{verify_parallel}(\mathbf{X}^0, \mathbf{d}^0, \delta, k \times \delta)$ where $k \in [1, m)$, and their children are equal, show

$$\texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta) = \texttt{verify_parallel}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta)$$

Since the children of all nodes at depth k for `verify` and `verify_parallel` are the same, they must generate the same set of nodes at depth $k+1$, which gives the same set of reachable states. □

Note that in practice, computing `verify_step` is cheap. Calling small remote functions like this will incur a lot of overhead due to the cost of communication and serialization/deserialization of data. When implementing the algorithm `verify_parallel`, we have chosen to batch together these computations, so that each remote function call computes as many timesteps as possible until a discrete mode transition is hit.

4.3 Incremental Verification

In this section, we show how we implement an incremental verification algorithm on top of `verify_parallel`.

Consider two hybrid automata $H_i = H(SC_i)$, $i \in \{1, 2\}$ that only differ in the discrete transitions. That is, (1) $\mathbf{X}_2 = \mathbf{X}_1$, (2) $\mathbf{D}_2 = \mathbf{D}_1$, and (3) $\mathbf{TL}_2 = \mathbf{TL}_1$, while the initial conditions, the guards, and the resets are slightly different[3]. SC_1 and SC_2 have the same sensors, maps, and agent flow functions. Let $Tree_1 = V_1$ and $Tree_2 = V_2$ be the execution trees for H_1 and H_2. Our idea of incremental verification is to reuse some of the computations in constructing the tree for H_1 in computing the same for H_2.

Recall that in `verify`, expanding each vertex N_1 of $Tree_1$ with a possible mode involves a guard check, a computation of $post_{\mathbf{d},\delta}$ and $post_{\mathbf{d},\mathbf{d}'}$. The algorithm `verify_incremental` avoids performing these computations while constructing $Tree_2$ by reusing those computations from $Tree_1$, if possible. To this end, `verify_incremental` uses a *cache (C)* that stores the result of a batch of `verify_step`. This is the same as that in Sect. 4.2, which simply batches together all the adjacent `verify_step` that have the same discrete modes. We'll call this batch operation `verify_batch`. The pseudocode for `verify_batch` is described in Algorithm 3. Formally, here are the properties of `verify_batch`:

Proposition 4. *For any set of states $\boldsymbol{X}^0 \subseteq \boldsymbol{X}$, mode $\boldsymbol{d}^0 \in \boldsymbol{D}$, time step δ and time horizon $T_{\max}$, let $\langle reachset, branches, N^0\rangle = \texttt{verify_batch}(\langle \mathbf{X}^0, \mathbf{d}^0, 0, \varnothing, \varnothing\rangle, \delta, T_{\max})$. Then for the ith node explored in* `verify_batch`*, we have:*

$$N^i \in \texttt{verify_step}(N^{i-1}, \delta).children$$
$$N^i.\boldsymbol{d} = N^{i-1}.\boldsymbol{d}$$

we further have:

$$branches = \bigcup_{i\in[0,k)} \{N' \mid N' \in N^i.children \text{ s.t. } N'.\boldsymbol{d} \neq N^i.\boldsymbol{d}\}$$
$$reachset = \bigcup_{i\in[0,k)} N^i.stride$$
$$N^0 = \langle \boldsymbol{X}^0, \boldsymbol{d}^0, 0, \varnothing, \varnothing\rangle; \forall N \in branches, N.\boldsymbol{d} \neq \boldsymbol{d}^0; k \leq \lceil \frac{T_{\max}}{\delta} \rceil$$

where k is the number of nodes in the batch.

[3] Note that in this section subscripts index different hybrid automata, instead of agents within the same automaton (as we did in Sects. 3 and 3.2).

For verify_batch($\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}$), the cache C will be indexed by $\langle \mathbf{X}^0, \mathbf{d}^0 \rangle$, and the value will be the same as that of verify_batch. Unlike normal caches, a cache hit can happen for C when the incoming key $\langle \mathbf{X}', \mathbf{d}' \rangle$ satisfies $\mathbf{X}' \subseteq \mathbf{X}^0 \wedge \mathbf{d}' = \mathbf{d}^0$.

The incremental verification algorithm is presented in Algorithm 3. The function verify_incremental checks C before every *post* computation to retrieve and reuse computations when possible. The cache can save information from any number of previous executions, so verify_incremental can be even more efficient than verify_parallel when running many consecutive verification runs.

The correctness property of verify_incremental is the same as that of algorithm verify_parallel in Sect. 4.2, i.e. the reachset computed by algorithm verify_incremental for SC_2, when given a cache populated with data from SC_1, is an overapproximation of the reachset computed by verify. More formally:

Proposition 5. *Given scenarios SC_1 and SC_2 with the same sensors, map, and agent flow functions, for cache $C = \varnothing$, any initial conditions $\boldsymbol{X}_1^0, \boldsymbol{X}_2^0 \subseteq \boldsymbol{X}$, $\boldsymbol{d}_1^0, \boldsymbol{d}_2^0 \in \boldsymbol{D}$, timestep δ and time horizon $T_{\max}$, after* verify_incremental($H_1, \mathbf{X}_1^0, \mathbf{d}_1^0, \delta, T_{\max}, C$) *is executed,*

$$\texttt{verify}(H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}) \subseteq \texttt{verify_incremental}(H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}, C)$$

Proof. For any initial conditions $\mathbf{X}_1^0, \mathbf{X}_2^0 \subseteq \mathbf{X}$, $\mathbf{d}^0 \in \mathbf{D}$, timestep δ, time horizon $T_{\max}$ and time t, let:

$$\langle \mathit{reachset}_1, \mathit{branches}_1 \rangle = \texttt{verify_batch}(\mathbf{X}_1^0, \mathbf{d}^0, \delta, T_{\max}, t)$$
$$\langle \mathit{reachset}_2, \mathit{branches}_2 \rangle = \texttt{verify_batch}(\mathbf{X}_2^0, \mathbf{d}^0, \delta, T_{\max}, t)$$

Given that verify_batch simply batches together postCont and postDisc operations, $\mathbf{X}_1^0 \subseteq \mathbf{X}_2^0 \implies \mathit{reachset}_1 \subseteq \mathit{reachset}_2$. As the cache C just stores the result of verify_batch, $\mathbf{X}_1^0 \subseteq \mathbf{X}_2^0 \implies \mathit{reachset}_1 \subseteq C(\mathbf{X}_2^0, \mathbf{d}^0)$. That is, verify_incremental($H_2, \mathbf{X}_2^0, \mathbf{d}^0, \delta, T_{\max}, \varnothing$) $\subseteq$ verify_incremental($H_2, \mathbf{X}_2^0, \mathbf{d}^0, \delta, T_{\max}, C$). That is, the reachset returned from a verify_incremental with caches would be an overapproximation of a version that doesn't have caches.

From Proposition 4, a verify_batch call can simply be decomposed into multiple verify_step calls. With the two conditions stated above, the algorithm for verify_incremental can be simplified to be the same as the algorithm of verify_parallel, which we have proven to be equivalent to verify. Thus, it follows that verify($H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}$) = verify_parallel($H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}$) $\subseteq$ verify_incremental($H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}$). $\square$

Algorithm 3

```
1: function VERIFY_BATCH(N^0, δ, T_max) where N^0.stride = ∅, N^0.children = ∅
2:   branches ← ∅
3:   reachset ← ∅
4:   N ← N^0
5:   while t < T_max do
6:     N ← verify_step(N, δ)
7:     reachset ← reachset ∪ N.stride
8:     if ∃N' ∈ N.children s.t. N'.d = N.d then
9:       branches ← branches ∪ (N.children \ N')
10:      N ← N'
11:      t ← t + δ
12:    else
13:      return reachset, branches ∪ N.children, N^0
14: function verify_incremental(H, X^0, d^0, δ, T_max, C)
15:   queue ← [⟨X^0, d^0, 0, ∅, ∅⟩]
16:   refs ← []
17:   reachset ← ∅
18:   while queue ≠ ∅ ∨ refs ≠ ∅ do
19:     if queue ≠ ∅ then
20:       N ← queue.dequeue()
21:       if C(N.X, N.d) ≠ ∅ then                          ▷ queries the cache
22:         ⟨subreachset, branches⟩ ← C(N.X, N.d)
23:         reachset ← reachset ∪ subreachset
24:         for N' ∈ branches s.t. N'.t < T_max do
25:           queue.add(N')
26:       else
27:         refs.add(remote verify_batch (N, δ, T_max))
28:     else                                    ▷ wait only when queue is empty
29:       ⟨⟨subreachset, branches, N⟩, refs⟩ ← ray.wait(refs)
30:       C(N.X, N.d) ← ⟨subreachset, branches⟩      ▷ update the cache with results
31:       reachset ← reachset ∪ subreachset
32:       for N' ∈ branches s.t. N'.t < T_max do
33:         queue.add(N')
34:   return reachset
```

5 Experimental Evaluation

We have implemented parallel and incremental verification algorithms in the Verse library [18], and in this section we will evaluate their performance against Verse's original sequential algorithm. Our goal is to glean qualitative lessons. We are not comparing against parallel tools mentioned in Sect. 2 because (a) it is not straightforward to implement multi-agent models in these other tools and (b) it is hard to draw fair conclusions comparing running time and memory usage across C++ and Python tools.

We perform these experiments on scenarios described in Table 2. Besides the intersection scenario, we also adopt some examples from [18]. Types of agents

include: (1) 4-d vehicle with bicycle dynamics and Stanley controller [13], and (2) 6-d drone with a NN-controller [14]. Some of the agents have collision avoidance logics (CA) for switching tracks. All experiments were performed on a desktop PC with 8 core (16 thread) Intel Xeon E5-2630.

Table 2. Name and description of scenarios.

Scenario	Description
`isect(`l`,`n`)`	4-way intersection of Fig. 1 with l lanes and n vehicles, all having CA
`drone`	3 straight parallel tracks in 3D space with 3 drones, one having CA
`drone8`	3 Figure-8 tracks in 3D space with 2 drones, both having CA
`curve`	3-lane curved road with 3 vehicles, one having CA
`wide(`n`,`a`)`	5-lane straight road with n vehicles, a of which having CA
`race`	3-lane circular race track with 3 vehicles, one having CA

5.1 Parallel Reachability Speeds up with Cores and Branching

Table 3 shows the experimental results on running `verify_parallel` on these scenarios on using 2, 4, and 8 CPU cores. The data is sorted according to the number of leaves of the reachtree, which we use as a metric to measure the potential parallelism in a scenario, as we discussed in Sect. 4.2.

Table 3. Runtime for verifying the examples in Table 2. Columns are: name of the scenario (name), number of timesteps simulated (length), number of mode transitions (#Tr), the width of the execution tree (#leaves), the run time of the `verify` (serial), the run time of `verify_parallel` using 2, 4, or 8 cores and the corresponding speedup in parentheses. All running times are in seconds.

name	length	#Tr	#leaves	serial	2 cores	4 cores	8 cores
`curve`	400	4	2	50	56 (0.89)	55 (0.91)	56 (0.89)
`drone`	450	7	2	29	37 (0.78)	30 (0.97)	29 (1)
`race`	600	7	2	220	162 (1.36)	156 (1.41)	157 (1.4)
`drone8`	400	8	4	31	36 (0.86)	36 (0.86)	37 (0.84)
`wide(7,2)`	1600	37	7	188	135 (1.39)	136 (1.38)	137 (1.37)
`isect(4,9)`	1000	37	11	342	349 (0.98)	190 (1.8)	130 (2.63)
`isect(4,10)`	800	59	15	587	606 (0.97)	318 (1.85)	197 (2.98)
`wide(8,3)`	600	105	20	311	313 (0.99)	172 (1.81)	104 (2.99)
`isect(4,15)`	1000	102	37	2115	2085 (1.01)	1081 (1.96)	653 (3.24)
`isect(4,20)`	1060	399	140	13100	8416 (1.56)	4477 (2.93)	2085 (6.28)
`isect(4,12)`	800	589	225	7136	4445 (1.65)	2214 (3.3)	1302 (5.62)

First, we observe that for scenarios with more than 37 transitions, parallelization speeds up reachability analysis with at least 4 cores. Secondly, the number of leaves in a scenario roughly correlates to the speedup gained. This is illustrated in the experiments as we move down the table, we can see that the speedup ramp-up. Lastly, for each scenario the speedup generally increases with the number of cores. Maximum gains are made with the `isect(4,20)` scenario, which has a wide execution tree with many transitions. The `verify_parallel` algorithm reduces the running time from over 3 and a half hours to a little over 30 min. This performance gain can make a tool usable, where previously it was not [22,24].

Because of the overhead of parallelism, some scenarios can be slower while using `verify_parallel` than `verify`. The overhead mainly comes from 2 areas: process creation and communication costs. As the number of parallel tasks increase, Ray creates more processes dynamically to handle those work, and the time caused by creating and initializing those processes can be larger than the benefit provided by parallelism. In addition, it takes time to send the inputs to and receive the results from the remote processes. This overhead gets larger with more agents in the scenario, larger state and/or mode spaces, longer time horizon, and more complex decision logics. However, from our results, this overhead does not overshadow the savings the `verify_parallel` algorithm introduce.

5.2 Incremental Verification Can Speed up Reachability Across Model Updates

To test our incremental verification algorithm, we apply it to several scenarios undergoing changes and edits. In this section we will report on the `drone8`, `wide(8,3)`, and `isect(4,15)` scenarios. We will modify the initial condition or behavior of agents in the scenario. We measure the similarity across the models using the earliest time (T_{change}) when the automaton's behavior changes: for two identical scenarios T_{change} would be ∞, when the initial conditions of one of the agents is changed then $T_{\text{change}} = 0\%$, and when the decision logic code of one of the agents is changed T_{change} indicates the time where the change affects runtime behavior. We run each of the experiments with the `verify`, serialized `verify_incremental`, `verify_parallel`, and parallelized `verify_incremental` algorithms. The results are shown in Table 4.

From the table we can observe that when compared to the non-incremental versions of the algorithms, the incremental versions provide more speedup when the behavior of the automata are closer. We can see from the table that as T_{change} goes higher, the speedups provided by incremental verification trend upwards. In the ideal case, where the same scenario is verified again, the verification time is reduced from 575 s to 20 s for `isect(4,15)`. For a case when the scenario is changed, the maximum gain we observe is from 404 s to 82 s for `isect(4,15)`.

Secondly, we observe that combination of incremental verification and parallelization can sometimes give us more savings. For example, in row 5 of Table 4, we can observe that the parallelized `verify_incremental` algorithm, which

Table 4. Runtime for verifying the examples in Table 2. Columns are: name of the scenario (name), number of timesteps simulated (T), single or repeated runs (single), number of mode transitions (#Tr), the width of the execution tree (#leaves), the time for the first change in automaton behavior (T_{change}), the run time of `verify` (ser), the run time and hit rate of `verify_incremental` without parallelization (inc ser, hit rate), the run time of `verify_parallel` (par), the run time and hit rate of `verify_incremental` with parallelization (inc par, hit rate). The unit of run time is in seconds.

name	T	#Tr	#leaves	T_{change}	ser	inc ser	hit rate	par	inc par	hit rate
`drone`	450	9	3	75%	53	29	90%	51	30	90%
`wide(8,3)`	600	51	10	15.67%	173	20	100%	80	51	96.74%
`wide(8,3)`	600	105	20	∞	335	17	100%	104	16	100%
`wide(8,3)`	600	49	8	0%	182	129	95.31%	72	53	87.95%
`isect(4,15)`	400	5	2	0%	116	98	59.57%	100	83	29.79%
`isect(4,15)`	400	37	12	∞	575	36	100%	189	20	100%
`isect(4,15)`	400	24	9	82.25%	404	35	100%	159	82	39.7%

takes 83 s to finish, out performs both the serialized `verify_incremental` algorithm and the `verify_parallel` algorithm alone, which takes 98 s and 100 s respectively. However, in some situations the serial `verify_incremental` algorithm can be much faster than the parallel `verify_incremental` algorithm. This typically happened when the number of leaves is small as shown in row 7 of the table. The situation is caused by the overhead introduced from parallelization. Even though the cache provides run time savings, the trajectories will need to be copied to the remote processes when they are in the cache, which causes more overhead.

6 Conclusions and Future Directions

In this paper, we presented parallel and incremental verification algorithms for hybrid multi-agent scenarios using Ray and Verse. For large scenarios with more than 10 agents, and large number of branches, the speedup can be 3 or 6 times. The incremental verification algorithm `verify_incremental` makes it faster to iterate on models. When the states and decision logics do not change much, the algorithm can give significant speedups.

This work suggests several directions for future research. Currently the algorithms parallelize the computation on a per-branch basis. Agent-level parallelization could be useful for large scenarios with clusters of non-interacting agents. Being able to divide up tasks at a finer scale would mean more opportunities for parallelization, but we would also need to be careful of too small task sizes and develop batching algorithms that both take advantage of the parallelization and minimizing the overhead induced. In incremental verification, the algorithm currently redoes the computation as soon as any of the agents reaches

states or exhibits behaviors never seen before. This is fairly evident from the experiments, where the caching is not able to provide runtime improvements when the initial conditions change. Finer-grained analysis of agent interactions can be done so that changes in agents' states or behaviors will only trigger recomputation of agents that will be affected. Moreover, in some situations the `verify_incremental` algorithm can become slower due to enabling parallelization. We can improve this by avoiding using remote functions when the result is already in the cache.

References

1. Althoff, M.: An introduction to CORA 2015. In: Proceedings of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
2. Bak, S., Duggirala, P.S.: HyLAA: a tool for computing simulation-equivalent reachability for linear systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, pp. 173–178. ACM (2017)
3. Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: JuliaReach: a toolbox for set-based reachability. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 39–44 (2019)
4. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
5. Chong, N., et al.: Code-level model checking in the software development workflow. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2020, pp. 11–20. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3377813.3381347
6. Chudnov, A., et al.: Continuous formal verification of Amazon s2n. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 430–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_26
7. Devonport, A., Khaled, M., Arcak, M., Zamani, M.: PIRK: scalable interval reachability analysis for high-dimensional nonlinear systems. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 556–568. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_27
8. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_5
9. Fan, C., Qi, B., Mitra, S., Viswanathan, M.: DryVR: data-driven verification and compositional reasoning for automotive systems. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, Part I. LNCS, vol. 10426, pp. 441–461. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_22
10. Fan, C., Qi, B., Mitra, S., Viswanathan, M.: DryVR: data-driven verification and compositional reasoning for automotive systems. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 441–461. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_22

11. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
12. Gurung, A., Ray, R., Bartocci, E., Bogomolov, S., Grosu, R.: Parallel reachability analysis of hybrid systems in XSpeed. Int. J. Softw. Tools Technol. Transf. **21**(4), 401–423 (2018). https://doi.org/10.1007/s10009-018-0485-6
13. Hoffmann, G.M., Tomlin, C.J., Montemerlo, M., Thrun, S.: Autonomous automobile trajectory tracking for off-road driving: controller design, experimental validation and racing. In: 2007 American Control Conference, pp. 2296–2301 (2007)
14. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 169–178 (2019)
15. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science. Morgan Claypool (2005). Also available as Technical Report MIT-LCS-TR-917
16. Khaled, M., Zamani, M.: PFaces: an acceleration ecosystem for symbolic control. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, pp. 252–257. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3302504.3311798
17. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: δ-reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_15
18. Li, Y., Zhu, H., Braught, K., Shen, K., Mitra, S.: Verse: a python library for reasoning about multi-agent hybrid system scenarios. In: Enea, C., Lal, A. (eds.) CAV 2023. LNCS, vol. 13964, pp. 351–364. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37706-8_18
19. Liang, E., et al.: RLlib: abstractions for distributed reinforcement learning. In: International Conference on Machine Learning, pp. 3053–3062. PMLR (2018)
20. Mitra, S.: Verifying Cyber-Physical Systems: A Path to Safe Autonomy. MIT Press, Cambridge (2021)
21. Moritz, P., et al.: Ray: a distributed framework for emerging {AI} applications. In: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 2018), pp. 561–577 (2018)
22. O'Hearn, P.W.: Continuous reasoning: scaling the impact of formal methods. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, pp. 13–25. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3209108.3209109
23. Platzer, A.: Differential logic for reasoning about hybrid systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 746–749. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71493-4_75
24. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at google. Commun. ACM **61**(4), 58–66 (2018)
25. Sun, D., Mitra, S.: NeuReach: learning reachability functions from simulations. In: Fisman, D., Rosu, G. (eds.) TACAS 2022, Part I. LNCS, vol. 13243, pp. 322–337. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_17