Parallel Reachability Analysis for Hybrid Systems

Amit Gurung *, Arup Deka *, Ezio Bartocci †, Sergiy Bogomolov ‡,§, Radu Grosu † and Rajarshi Ray *
* National Institute of Technology Meghalaya, India
† Vienna University of Technology, Austria
‡ Institute of Science and Technology Austria, Austria
§ Australian National University, Australia

***Abstract*—We propose two parallel state-space-exploration algorithms for hybrid automaton (HA), with the goal of enhancing performance on multi-core shared-memory systems. The first uses the parallel, breadth-first-search algorithm (PBFS) of the SPIN model checker, when traversing the discrete modes of the HA, and enhances it with a parallel exploration of the continuous states within each mode. We show that this simple-minded extension of PBFS does not provide the desired load balancing in many HA benchmarks. The second algorithm is a task-parallel BFS algorithm (TP-BFS), which uses a cheap precomputation of the cost associated with the post operations (both continuous and discrete) in order to improve load balancing. We illustrate the TP-BFS and the cost precomputation of the post operators on a support-function-based algorithm for state-space exploration. The performance comparison of the two algorithms shows that, in general, TP-BFS provides a better utilization/load-balancing of the CPU. Both algorithms are implemented in the model checker XSpeed. Our experiments show a maximum speed-up of more than 2000× on a navigation benchmark, with respect to SpaceEx LGG scenario. In order to make the comparison fair, we employed an equal number of post operations in both tools. To the best of our knowledge, this paper represents the first attempt to provide parallel, reachability-analysis algorithms for HA.**

I. Introduction

Hybrid automaton (HA) is a popular formal framework for modeling and verifying safety properties in biological [1], [2] and cyber-physical systems [3]. HA combine the logical, discrete-mode-based representation of finite automata with a dynamical, continuous-state-based representation (for each mode) of differential equations. An HA is called safe, if a given set of bad states is not reachable from a set of initial states. Hence, safety can be proved in a HA through reachability analysis. Unfortunately, this is in most cases undecidable for HA [4]. SpaceEx [5], [6], [7], [8] is one of the most popular reachability-analysis tools for HA, using semi-decision procedures based on over-approximation techniques [9], [10]. The set of reachable states are kept as a collection of continuous sets, each of which being represented in a symbolic fashion. The main two challenges to be addressed within such set-based methods, are precision, and scalability. Recently, algorithms using convex sets, represented using support functions [9], [11] and zonotopes [10], have shown good scalability. However, all these techniques were originally conceived to run sequentially. Hence, they are currently not suitable to exploit the modern and powerful multi-core architectures that would enable them to improve further their scalability.

a) Our contribution: In this work, we provide two parallel algorithms, for the reachability analysis of a HA, that can leverage CPU multi-core architectures, to speed-up the performance. Both exploit Holzmann's lock-free, parallel, breadth-first-search algorithm (PBFS) [12], for traversing the discrete modes of an HA (note that PBFS was recently implemented in the SPIN model checker). The first algorithm adds to PBFS both symbolic reachable states, and flowpipe computations within these states. These are nontrivial extensions, enabling the reachability analysis in HA. Unfortunately, we notice that this algorithm is often not resulting in ideal load balancing, and many CPU's cores are underutilized. This happens when the number of symbolic states to be explored is less than the number of processor's cores or when the cost of flowpipe computation varies drastically for different symbolic states. For this reason, we provide a second algorithm, that improves considerably the load balancing, by estimating the cost of the post operations (both discrete and continuous), and using this estimate to balance the load. The two algorithms are implemented in the XSpeed [13] tool. XSpeed is now able to also handle SpaceEx models using the Hyst [14] model-transformation and translation tool for HA. Our experiments show a speed-up of up to 2000× on a navigation-benchmark instance with respect to SpaceEx LGG (Le Guernic-Girard) algorithm [5], [15]. To make the comparison fair, both tools run the same number of post operations. The tool and the benchmarks reported in the paper can be downloaded from http://nitmeghalaya.in/nitm_web/fp/cse_dept/XSpeed/index_new.html.

b) Related Work: In the last decade, there has been an increasing interest in developing efficient techniques for reachability analysis in HA, using different symbolical representations to compute and to approximate the reachable sets. PHAVer [16] and SpaceEx [5] use polytopes and support functions [9], [11] to compute the over-approximations with a certain level of precision of the reachable sets in linear HA. HModest [17] and Hydentify [6] use respectively PHAVer and SpaceEx in their toolchain as back-end applications for their analysis. Despite the limitation in scalability, intervals (boxes) [18] have been successfully applied for the analysis of non-linear hybrid systems in tools such as iSAT [19] and dReach [20]. Other tools such as Flow* [21] and nltoolbox [22] use polynomial approximations for the flowpipe computation such as Taylor models [21] and Bernstein polynomials [23].

All the aforementioned tools use only a single core of the CPU and they are currently not able to scale the reachability analysis by exploiting the powerful parallel multi-core architectures available for free in our modern CPUs.

In previous work [24], [13] we show how to speedup the reachability analysis of purely continuous systems (single mode HA) using parallel architectures. This paper extends this work to general HA. This extension is not trivial, as the interplay of discrete jumps and continuous flows can be often very intricate. Note that a discrete jump, when enabled, may switch the current continuous flow (dynamics) to a totally different flow. In this paper we address these challenges by showing how to properly combine parallel breadth first search (PBFS) (which was developed for discrete systems) with the parallel reachability analysis techniques for continuous systems. Hence, to the best of our knowledge, this paper represents the first attempt to provide parallel reachability-analysis algorithms for HA.

c) Paper organization: The rest of the paper is organized as follows. Section II provides the necessary background on hybrid automata and reachability analysis. In Section III, we show how to extend the Holzmann's lock-free parallel breadth first exploration algorithm [12] to handle the state space exploration in hybrid automata. Section IV addresses the load balancing problem introducing the notion of a task parallel algorithm. In Section V we provide a concrete example of a task parallel algorithm in the context of support-function-based reachability analysis. Section VI reports the experimental results to illustrate performance speed-up and CPU's core utilization. We conclude in Section VII.

II. Preliminaries

A hybrid automaton is a mathematical model of a system exhibiting both continuous and discrete dynamics. A *state* of a hybrid automaton is an n-tuple $(x_1, x_2, \ldots, x_n)$ representing the values of the n continuous variables in an n dimensional system at an instance of time.

Definition 1. *A hybrid automaton is a tuple* $(\mathcal{L}, \mathcal{X}, Inv, Flow, Init, \delta, \mathcal{G}, \mathcal{M})$ *where:*

- $\mathcal{L}$ *is the set of locations of the hybrid automaton.*
- $\mathcal{X}$ *is the set of continuous variables of the hybrid automaton.*
- $Inv : \mathcal{L} \rightarrow 2^{\mathbb{R}^n}$ *maps every location of the automaton to a subset of* $\mathbb{R}^n$*, called the invariant of the location. An invariant of a location defines the constraint on the states within the location of the automaton.*
- $Flow$ *is a mapping of the locations of the automaton to ODE equations of the form* $\dot{x} = f(x), x \in \mathcal{X}$*, called the flow equation of the location. A flow equation defines the evolution of the system variables within a location.*
- $Init$ *is a tuple* $(\ell_{init}, \mathcal{C}_{init})$ *such that* $\ell_{init} \in \mathcal{L}$ *and* $\mathcal{C}_{init} \subseteq Inv(\ell_{init})$*. It defines the set of initial states of the automaton.*
- $\mathcal{G} \subseteq 2^{\mathbb{R}^n}$ *is the set of guard sets of the automaton.*
- $\mathcal{M} \subseteq$ *set of functions* $\mathbb{R}^n \rightarrow \mathbb{R}^n$*, is the set of assignment maps of the automaton.*
- $\delta \subseteq \mathcal{L} \times \mathcal{G} \times \mathcal{M} \times \mathcal{L}$ *is the set of transitions of the automaton. A transition from a source location to a destination location in* $\mathcal{L}$ *may trigger when the* state s *of the hybrid automaton lies in the guard set from* $\mathcal{G}$*. The map* $\mathcal{M}$ *of the transition transforms the state* s *in the source location to a new state* s' *in the destination location.*

A reachable state is a state attainable at any time instant $0 \leq t \leq T$ under its flow and transition dynamics starting from an initial state in $Init$. The flow dynamics evolves a state (ℓ, x) to another state (ℓ, y) in a location ℓ after T time units such that $flow(x, T) = y$ and $flow(x, t) \in$ *Inv*(ℓ) for all $t \in [0, T]$, where $flow$ is the solution to the flow equation of $Flow(\ell)$. Reachability analysis tools produce a conservative approximation of the reachable states of the automaton over a time horizon. Reachable states can be expressed as a union of *symbolic states*. A symbolic state is a tuple (loc, $\mathcal{C}$) such that $loc \in \mathcal{L}$ and $\mathcal{C} \subseteq Inv(loc)$.

In Algorithm 1 [5], we show a generic reachability algorithm for hybrid automata. The algorithm maintains two data structures, $Wlist$ and R. $Wlist$ is the list of symbolic states waiting to be explored for reachable states and R is the list of already visited reachable symbolic states. $Init$ is a symbolic state depicting the initial states given as an input. $Wlist$ and R are initialized to $Init$ and $\emptyset$ respectively in line 2. A symbolic state S is removed from the $Wlist$ at each iteration and explored for reachable states in line 4. The operators $PostC$ and $PostD$ are applied consecutively. $PostC$ takes a symbolic state as argument and computes a conservative over-approximation of the reachable states under the continuous dynamics of the location, commonly referred as the *flowpipe*. The flowpipe is a symbolic state, say R' which is included in R in line 12. The operator $PostD$ is applied on R' which returns a set of symbolic states under the discrete dynamics, shown as R'' in line 6. If R'' is not contained in R, it is added to $Wlist$ for further exploration in line 8.

Algorithm 1 Reachability Algorithm for Hybrid Automata

```
1:  procedure REACH-HA(ha,Init)
2:      Wlist ← Init; R ← ∅;
3:      while Wlist ≠ ∅ do
4:          delete S from Wlist; R' ← PostC(S)
5:          R ← R ∪ R'
6:          R'' ← PostD(R');
7:          if R'' ⊆ R then go to step 3
8:          else  add R''/R to Wlist
9:          end if
10:     end while
11: end procedure
```

III. Parallel Breadth First Search

We observe that the $PostC$ computation for symbolic states in $Wlist$ is independent of each other and therefore, can be potentially parallelized. In this work, we exploit this inherent

parallelism and propose a parallel breadth first search (BFS) algorithm. In a multi-threaded implementation, threads can compute $PostC$ and $PostD$ operations in parallel, however the access to the shared data structure $Wlist$ and R has to be mutually exclusive to avoid race condition. Mutual exclusion can be accomplished by using locks or semaphores, however at the price of additional overhead. Moreover, such an implementation may not ensure an effective load balancing, which is illustrated in Section III-A.

In Algorithm 2 we show how to avoid the overhead of the mutual exclusion discipline by adapting the parallel lock-free breadth first search algorithm proposed in [12] to hybrid system state-space exploration. Our adapted algorithm is referred as A-GJH (Adapted Gerard J. Holzmann's) algorithm in the paper. The algorithm uses a three dimensional array $Wlist$ as the data structure for storing the symbolic states to be explored, shown in line 3. Essentially, it provides two copies of $Wlist$, each being a two-dimensional list of symbolic states. At each iteration, symbolic states are read from the $Wlist[t]$ copy and new symbolic states are written to the $Wlist[1-t]$ copy. At the first iteration the value of t is 0 and at the end of each iteration (see line 22) of the main while loop it is assigned to $1-t$. In this way, in the next iteration the write $Wlist$ copy becomes the read $Wlist$ copy and vice-versa. There are N threads, one thread per core, which computes the post operations in parallel. The reachable states explored by the thread indexed at w, are accumulated in the data structure $R[w]$, as shown in line 12. All the symbolic states present in the row $Wlist[t][w]$ are sequentially processed by the thread indexed by w, shown in line 8-15. When $PostD$ operator is applied on a symbolic state, a list of successor symbolic states results, to be processed in the next iteration. Each successor state is added to the list $Wlist[1-t][w'][w]$, where w' is randomly selected between 0 to $N-1$ (line 15). This random distribution of new states across rows of $Wlist[1-t]$ is for load balancing. Since each thread indexed at w writes to the list at $Wlist[1-t][w'][w]$, there is no write contention in $Wlist[1-t]$. Checking containment of symbolic states in the result data structure R is avoided in the algorithm since it is an expensive operation. The exploration is instead bounded by the number of levels of breadth first exploration of the automaton. When all the N threads terminate and synchronize at line 18, the exploration of symbolic states of $Wlist[t]$ is completed. The synchronization of threads ensure a breadth-first exploration. The algorithm terminates when there are no successor states in $Wlist[1-t]$ for further exploration or when the breadth exploration reaches a certain bound. A proof of correctness of the algorithm is shown in the appendix.

A. Load Balancing

The clever use of the data structures in A-GJH algorithm provides freedom from locks and reasonable load balancing when there are sufficiently large number of symbolic states in the waiting list. However, the load balancing in A-GJH does not perform well when the number of symbolic states in the waiting list is less than the number of cores of the processor.

Algorithm 2 Adapted G.J. Holzmann's Algorithm

```
1:  procedure REACH-PBFS(ha, Init, bound)
2:      t = 0, level = 0, N = Cores
3:      Wlist[2][N][N]                          ▷ 2 × N × N array
4:      Wlist[t][0][0] = Init
5:      R[0] ← Init
6:      repeat
7:          Fork threads with id w = 0 to N − 1 and execute
8:          for q = 0 to N − 1 do
9:              for each s in Wlist[t][w][q] do
10:                 delete s from Wlist[t][w][q]
11:                 R'[w] ← PostC(s)
12:                 R[w] ← R[w] ∪ R'[w]
13:                 R''[w] ← PostD(R'[w])
14:                 w' = choose random 0 ... N − 1
15:                 add s' ∈ R''[w] to Wlist[1 − t][w'][w]
16:             end for
17:         end for
18:         Barrier synchronization ▷ Thread synchronization
    to ensure BFS
19:         if Wlist[1 − t] is empty then
20:             done = true;
21:         else
22:             t = 1 − t                       ▷ Read/Write switching
23:             level ← level + 1
24:         end if
25:     until done OR level = bound
26: end procedure
```

Models	Breadths	CPU Utilization (%)			Speed-up	
		SpaceEx (LGG)	XSpeed (Seq-BFS)	XSpeed (A-GJH)	A-GJH vs. Seq-BFS	A-GJH vs. SpaceEx
Circle	4	12.50	12.50	12.53	0.9	1.1
	6	12.50	12.50	19.80	1.3	1.4
Nav 3x3 (inst# 1)	3	12.50	12.50	23.51	1.6	10.7
	5	12.50	12.50	39.28	1.8	8.2
Nav 5x5	5	12.50	12.50	33.87	1.6	6.9
	7	12.50	12.50	65.38	2.4	11.8

Fig. 1. Moderate CPU utilization and performance speed-up with A-GJH algorithm.

Since the symbolic states are distributed randomly to the N cores in line 15 for exploration, some of the cores remain idle when there are not enough states to be explored. For this reason there are benchmarks for hybrid systems reachability analysis where an incorrect load balancing results in a low utilization of the available cores. For example, Figure 1 shows that while the A-GJH running in a 4 core processor provides some improvements in performance compared to SpaceEX LGG (Le Guernic-Girard) and the sequential BFS, the CPU core utilization remains very modest. The best utilization is 65% in the NAV 5X5 benchmark for 7 levels of exploration and the worst is 12% in the Circle model. In the Circle model,

there are only 2 symbolic states for exploration at each BFS iteration, keeping most of the CPU cores idle.

Another principle problem in load balancing is that the cost of flowpipe computation may vary drastically for different symbolic states. This is illustrated on a 3×3 Navigation benchmark in Figure 2 [25]. The benchmark models the motion of an object in a 2D plane partitioned as a 3×3 grid. Each cell in the grid has a width and height of 1 unit and have a desired velocity v_d. In Figure 2 the cells are numbered from 1 to 9 and the respective desired velocities are shown with a directed vector. Note that there is no desired velocity shown in cell 3 and 7 to distinguish them from the others. Cell 3 is the target whereas cell 7 is the unsafe region. The actual velocity of the object in a cell is given by the differential equation $\dot{v} = A(v - v_d)$, where A is a 2×2 matrix. There is an instantaneous change of dynamics on crossing over to an adjacent cell. The green box is an initial set where the object can start with an initial velocity. The red region shows the reachable states under the hybrid dynamics after a finite number of cell transitions.

Figure 2 shows the reachable states after two and three levels of exploration in Algorithm 2. There are four symbolic states, S_1, S_2, S_3 and S_4, waiting to be explored after two levels of BFS, shown in blue. The symbolic states S_1, S_2 are $\{1,\ B_1\}$, $\{1,\ B_2\}$ and S_3, S_4 are $\{5,\ B_3\}$ and $\{5,\ B_4\}$ respectively, where 1 and 5 are the location ids and B_1, B_2, B_3 and B_4 are the blue regions in the boundary of location with ids 1 and 4, 1 and 2, 4 and 5, 2 and 5 respectively. Algorithm 2 spawns four threads, one each to compute the flowpipe from the symbolic states. In a four core processor, this seems an ideal load division. However, observe in Figure 2b that out of the four flowpipes, the two from S_1 and S_2 do not lead to new states since they start from the boundary of location id 1 and the dynamics pushes the reachable states outside the location invariant. This implies that the flowpipe computation cost for S_1 and S_2 are low and the two cores assigned to these flowpipe computation finish early and waits at the synchronization point until the remaining two busy cores complete. Such a situation keeps the available cores under-utilized due to the idle waiting of 50% of the cores. Similarly, the cost of $PostD$ operation also varies causing idle waiting of threads assigned to low cost computations.

IV. Task Parallel Algorithm

In the following, we propose an alternative approach to improve load balancing based on computational cost of the tasks encountered during the exploration. The idea is to identify the *atomic tasks* in a flowpipe computation ($PostC$). The atomic tasks across all flowpipe computations in a BFS iteration is a measure of the total work-load, at the present iteration of the BFS. For an effective load balancing, this work-load is distributed evenly between the cores of the processor. Similar tasks distribution can be applied also to the computation of discrete transitions ($PostD$). Algorithm 3 shows this approach.

In particular, the algorithm uses a two dimensional array $Wlist$ of list of symbolic states, shown in line 3. The first dimension of size two, provides two copies of $Wlist$, one for reading and one for writing, similar to Algorithm 2. The size of the second dimension is N and equals the total number of symbolic states in $Wlist$ waiting to be explored. Initially, N is set to one and it is modified after each iteration in line 29. The instruction in line 7 obtains an estimated cost of computing a flowpipe from a symbolic state using the function $flow\_cost$. After the flowpipe costs for all symbolic states in $Wlist$ are computed, line 8 breaks the flowpipe computations into atomic tasks and adds them into a tasks list. In line 12 the atomic tasks are evenly assigned to the processors cores. In line 17 the results of the atomic tasks computed in parallel are then joined together to obtain a flowpipe. Similar load division is carried out for $PostD$ operation. The successor states obtained from each flowpipe are added in the write list $Wlist[1-t][w]$, by a thread indexed at w in line 25. The exclusive access of the threads to the columns of $Wlist[1-t]$ eliminates write contention. A proof of correctness of the algorithm is shown in the appendix.

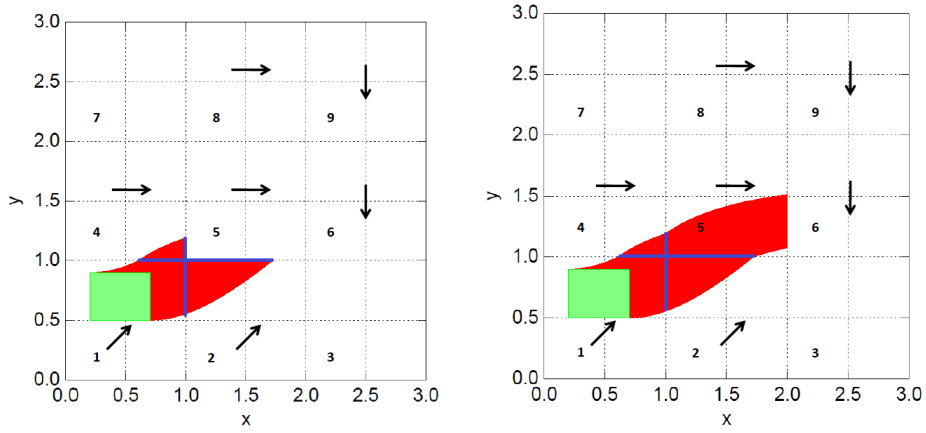
In this proposed algorithm, it is important to devise an efficient method to find the cost of $PostC$ and $PostD$ operations, for an effective load balancing. Efficient methods and data structures for splitting the $PostC$ and $PostD$ operations into atomic tasks and merging the task results, are important in order to ensure that the extra overhead incurred in the algorithm is not more than the gain due to parallelism. In the next section, we propose some procedures to efficiently compute the cost of post operations in a support-function algorithm.

V. Task Parallelism in Support Function Algorithm

A common technique in flowpipe computation consists in discretizing the time horizon T into N intervals with a chosen time-step $\tau = T/N$ and over-approximating the reachable states in each interval by a closed convex set, say Ω. A flowpipe can be represented as a union of convex sets as shown in Equation 1.

$$\begin{aligned} Reach_{[0,T]}(\mathcal{X}_0) &\subseteq \bigcup_{i=0}^{N-1} \Omega_i \\ Reach_{[i\tau,(i+1)\tau]}(\mathcal{X}_0) &\subseteq \Omega_i,\ \forall 0 \leq i < N \end{aligned} \tag{1}$$

The representation of the convex sets Ω is a key in the efficiency and scalability of reachability algorithms. Polytopes [26], [16] and Zonotopes[10] are popular geometric objects suitable to represent convex sets. More recently, support functions [9], [11] have been proposed as a functional representation of compact convex sets. SpaceEx [5] and XSpeed [13] tools implement support-function-based algorithms for flowpipe computation due to the promise it provides in scalability. We now present the preliminaries of support functions necessary to introduce the task parallelism in the algorithm.



(a) Reachable states after exploring 2 levels having 4 new symbolic states

(b) Reachable states after exploring 3 levels

Fig. 2. Illustrating Load Balancing Problem with Flowpipes of Varying Cost

A. Support functions

Definition 2. *[27] Given a nonempty compact convex set $\mathcal{X} \subset \mathbb{R}^n$ the* support function *of $\mathcal{X}$ is a function $sup_{\mathcal{X}} : \mathbb{R}^n \to \mathbb{R}$ defined as:*

$$sup_{\mathcal{X}}(\ell) = max\{\ell \cdot x \mid x \in \mathcal{X}\} \tag{2}$$

Definition 3. *[5] Given a support function $sup_{\mathcal{X}}$ of a nonempty compact convex set $\mathcal{X}$ and a finite set of directions $\mathcal{D}$, called* template directions*, a* template hull *of the convex set $\mathcal{X}$ is defined as:*

$$TH_{\mathcal{D}}(\mathcal{X}) = \{x \in \mathbb{R}^n \mid \bigwedge_{l_i \in \mathcal{D}} l_i \cdot x \leq sup_{\mathcal{X}}(\ell_i)\} \tag{3}$$

The support-function algorithm in [9] proposes a flowpipe computation by computing the template hull approximation of the convex sets Ω. Template hulls are computed by sampling the support functions of the convex sets in the template directions. The algorithm considers linear dynamics with non-deterministic inputs of the form:

$$\dot{x} = Ax(t) + u(t), \qquad u(t) \in \mathcal{U}, x(0) \in \mathcal{X}_0 \tag{4}$$

where $x(t), u(t) \in \mathbb{R}^n$, A is a real-valued $n \times n$ matrix and $\mathcal{X}_0$ and $\mathcal{U} \subseteq \mathbb{R}^n$ are the set of initial states and the set of inputs given as compact convex sets, respectively.

B. Flowpipe Cost Computation

We define the cost of computing a flowpipe by considering a support function sample as the logical atomic task.

Definition 4. *Given a time horizon T, a time discretization factor N, a set of template directions $\mathcal{D}$ and an initial symbolic state $S = (loc, \mathcal{C})$, the cost of computing the flowpipe with $PostC(S)$ is given by $flow\_cost(S) = j.|\mathcal{D}|$ where $j = max\{i \mid 0 \leq i \leq N$ and $\forall\ 0 \leq k \leq i, \Omega_k \vdash Inv(loc)\}$.*

We say that a convex set $\Omega \vdash Inv(loc)$ *if and only if* $\Omega \cap Inv(loc) \neq \emptyset$. Since computing polyhedral approximation of the convex sets Ω requires sampling support function in each direction of the set of template directions $\mathcal{D}$, the $flow\_cost$ essentially gives us the number of support function samplings, i.e. the atomic tasks, that is to be completed in order to compute the flowpipe. To compute $flow\_cost$, it is necessary to find the longest sequence Ω_0 to Ω_j satisfying the location invariant $Inv(loc)$. Assuming polyhedral invariants, checking the invariant satisfaction can be performed using the following proposition.

Proposition 1. *[28] Given a polyhedron $\mathcal{I} = \bigwedge_{i=1}^{m} \ell_i \cdot x \leq b_i$ and a convex set Ω represented by its support function sup_Ω, $\Omega \vdash \mathcal{I}$ if and only if $-sup_\Omega(-\ell_i) \leq b_i$, for all $1 \leq i \leq m$.*

A procedure to identify the largest sequence is to apply Proposition 1 to each convex set starting from Ω_0 iteratively until we find a Ω_j such that $\Omega_j \nvdash Inv(loc)$. The time complexity of the procedure is $O(m{\cdot}N{\cdot}f)$, where f is the time for sampling the support function, m is the number of invariant constraints and N is the time discretization factor. We propose a cheaper algorithm with fewer support functions samplings for a class of linear dynamics $\dot{x} = Ax(t) + u$, with u being a fixed input. Fixed input leads to deterministic dynamics allowing to compute the reachable states symbolically at any time point.

Proposition 2. *Given an initial set $\mathcal{X}_0$ and dynamics $\dot{x} = Ax(t) + u$ with A being invertible, the set of states reachable at time t is given by:*

$$X(t) = e^{At}x_0 \oplus A^{-1}(e^{At} - I)(v) \tag{5}$$

where $\oplus$ denotes minkowski sum operator. The idea of the procedure shown in Algorithm 4, is to use a coarse time-step to compute reachable states using Proposition 2 and detect an approximate time of crossing the invariant. Once the invariant crossing time is detected, similar search is followed by narrowing the time-step for a finer search near the boundary of the invariant for a desired precision. The procedure is

Algorithm 3 Task Parallel Breadth First Exploration

1: **procedure** REACH-TASK-PBFS(ha, $Init$, bound)
2: $t = 0$, $level = 0$, $N = 1$, $CostC = 0$
3: $Wlist[2][N]$ ▷ 2D list of symbolic states
4: $Wlist[t][0] = Init$
5: **repeat**
6: **for** each s in $Wlist[t]$ **do**
7: $CostC = CostC + flow\_cost(s)$
8: Break $PostC(s)$ into atomic tasks and add to Tasks list
9: **end for**
10: $Tasks\_Per\_Core$ = $\lceil CostC/\#Cores \rceil$ ▷ Even distribution of tasks to cores
11: **for** threads with id $w = 1$ to $\#Cores$ **do**
12: Execute $Tasks\_Per\_Core$ exclusive tasks from the Tasks list in parallel
13: Add results to Res
14: **end for**
15: Barrier Synchronization
16: **for** thread with id $w = 1$ to size of $Wlist[t]$ **do**
17: $R[w] = Res.merge()$ in parallel ▷ Merge task results to get flowpipe
18: **end for**
19: Barrier Synchronization
20: Similarly, repeat steps 6 to 15 with cost of $postD$ operations
21: **for** thread with id $w = 1$ to size of $Wlist[t]$ **do**
22: $R'[w] = Res.merge()$ in parallel ▷ Merge task results to get successor symb states
23: **end for**
24: Barrier Synchronization
25: Add each s in $R'[w]$ to $Wlist[1-t][w]$
26: **if** $Wlist[1-t]$ is empty **then** *done* = *true*
27: **else**
28: $t = 1 - t$ ▷ Read/Write switching
29: N = sum of size of all lists in $Wlist[t]$
30: Resize Wlist[1-t][N], $level = level + 1$, $CostC = 0$
31: **end if**
32: **until** $done$ *OR* $level = bound$
33: **end procedure**

illustrated on a toy model of a counter clockwise circular rotation dynamics as shown in Figure 3a. The model has two locations with the same dynamics but different invariants. The transition assignment maps does not modify the variables. Figure 3b illustrates the procedure. The initial set on the location is shown in blue. The red sets are the reachable images of the initial set computed at coarse time steps to detect invariant crossing, followed by computing the images at finer time-steps shown in green near the invariant boundary for detecting an upper bound on the time of crossing the invariant with a desired precision. After computing this time, say t', the $flow\_cost$ is obtained using Definition 4 with $j = t'/\tau$. However, the problem with the procedure is when it is possible for a reachable image to exit and re-enter the invariant within the chosen time-step. In such cases, the approximation error in the time returned by the procedure can be substantial. Constant dynamics and convex invariant $\mathcal{I}$ will not have such a scenario and the approximation error can be bounded.

Theorem 1. *For a class of dynamics $\dot{x} = k$, where k is a constant, let t be the exact time when reachable states from a given initial set $\mathcal{X}_0$ violate the convex location invariant $\mathcal{I}$. Let δ_C and δ_F be the coarse and fine time steps chosen to detect approximate time t' of invariant violation, then the approximation error $|t - t'| \leq \delta_F$.*

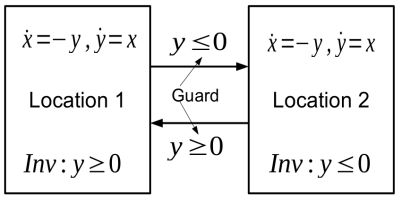
Proof. Constant dynamics have a fixed direction of evolution and therefore, convexity property of the invariant set $\mathcal{I}$ ensures that the reachable states cannot exit and re-enter the invariant set. The set of reachable states at time t, $X(t) = X_0 \oplus kt$ is a convex set when X_0 is a convex set, and can be exactly represented using its support function. Algorithm 4 samples the support function of $X(t)$ at δ_F time-steps to identify the time instant t' of crossing $\mathcal{I}$, which implies $|t - t'| \leq \delta_F$. □

Algorithm 4 Detecting time of invariant crossing with varying time-step

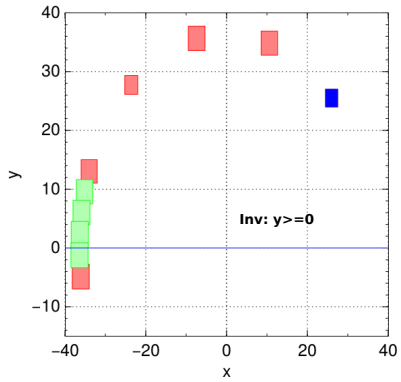
1: **procedure** INVARIANT-CROSSING TIME DETECTION($\mathcal{I}$, $\mathcal{X}_0$, T)
2: $discretization = 10$, $\tau = T/discretization$ ▷ Coarse Time-step
3: $i = 0$, $\mathcal{R}(0) = \mathcal{X}_0$
4: **while** $\mathcal{R}(\tau.i) \vdash \mathcal{I}$ **do** $i = i + 1$ ▷ Widened Search
5: **end while**
6: **if** $i > 1$ **then** $t1 = \tau * (i - 1)$
7: **else** *return* 0
8: **end if**
9: $\tau = \tau/discretization$, $i = 0$ ▷ Fine Time-step
10: **while** $\mathcal{R}(t1 + i * \tau) \vdash \mathcal{I}$ **do** $i = i + 1$ ▷ Narrowed Search
11: **end while**
12: return $t1 + i * \tau$ ▷ An upper bound on invariant crossing time
13: **end procedure**

C. Discrete-Jump Cost Computation

The $PostD$ computation performs the flowpipe intersection with the guard set followed by image computation. Considering a flowpipe having sets Ω_0 to Ω_j, each of these sets are applied with intersection operation and a map for non-empty intersection. Assuming intersection and image computation as the atomic task, the cost of $PostD$ on a flowpipe $\cup_{i=0}^{j}\Omega_i$ will be j, which can be obtained from the $flow\_cost$ computation in Definition 4. The addition of the cost of post operations for all symbolic states in the waiting list is used to uniformly distribute atomic tasks of post operations across the cores



(a) Hybrid Automaton



(b) Searching Time of Invariant Crossing with Widening-Narrowing Time-steps

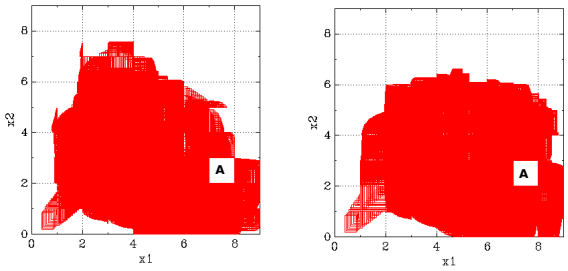
Fig. 3. (a) A hybrid automaton of a toy example with circular rotation dynamics. (b) Evaluation of the invariant crossing time detection algorithm on a circle model. The red sets are computed at coarse time-steps and the green sets computed at finer time-steps.

using multi-threading. Further details on the data-structures and task distribution is omitted for simplicity. The task parallel support-function-algorithm is referred as TP-BFS in the text that follows.

VI. Experiments

The parallel algorithms are implemented with multi-threading using OpenMP compiler directives. We conducted our experiments on a 12 core Intel Xeon(R) CPU E5-2670 v3, 2.30GHz and 62 GB RAM. We evaluate the performance of the proposed algorithms on varying the number of cores and with hyper-threading disabled. The benchmarks are described in detail in http://nitmeghalaya.in/nitm_web/fp/cse_dept/XSpeed/benchmarks.pdf. We report the performance speedup and CPU utilization of the proposed algorithms in comparison to SpaceEx (LGG) and sequential BFS in XSpeed on 4, 6, 8 and 12 cores in Figure 5 and Figure 6 respectively. The results are taken on the same 12 cores CPU by disabling some of the available cores. The results are for a time horizon of 10 units in each automaton location and box template direction as parameters. The sampling time in Switching Oscillator, Timed Bouncing Ball, Navigation 3×3 and 5×5 instances is $1e-4$, it is $1e-5$ in the Circle model and 0.1 units for Navigation 9×9 instance respectively. The number of BFS levels explored is 7 for the Nav (3x3) instances, 6 for the Switching Oscillator and Timed Bouncing Ball model, 8 for the Circle, 9 for the Nav (5x5) and 13 for the Nav(9x9) model respectively.

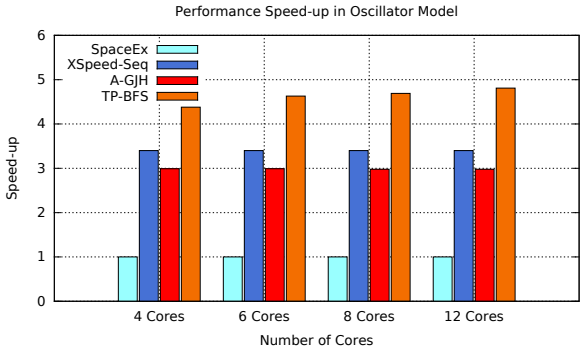
In order to relate our results with SpaceEx (LGG) scenario, we apply the same number of post operations in both XSpeed and SpaceEx, with similar parameters. We count the number of post operations for a given bound on the BFS level in XSpeed and the exploration in SpaceEx is bounded with the same count on post operations (by setting the argument iter-max). SpaceEx terminates on either finding a fixpoint or reaching the predetermined bound on the post operations, whichever is first. The number of post operations are chosen in the experiments such that SpaceEx does not find the fixpoint in those many post operations. Note that the cost of post operation could be different for the symbolic states in XSpeed and in SpaceEx. Therefore, comparison on the number of post operations is not perfectly fair but we could not find a better means of comparing. Figure 4 shows that the reachable region obtained from XSpeed on a Navigation benchmark after 13 BFS levels (105563 posts) is comparable to that obtained from SpaceEx (LGG) after 105563 posts (not finding fixpoint), and XSpeed computes the reachable region $2000\times$ faster. Note that the sequential BFS implementation in XSpeed is significantly better in performance than SpaceEx (LGG). We understand that this difference is due to the optimizations in XSpeed and also due to the fact that XSpeed does not perform fixpoint computation unlike SpaceEx and thus saves computation cost.



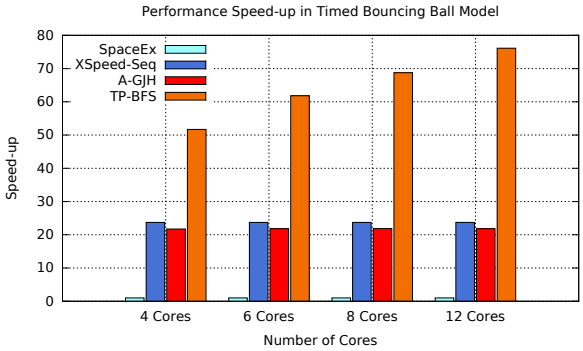
(a) XSpeed (b) SpaceEx(LGG)

Fig. 4. Reachable region of a 9×9 Navigation benchmark instance by (a) XSpeed after 13 BFS-levels (105563 post operations) and (b) SpaceEx (LGG) algorithm (105563 post iterations)

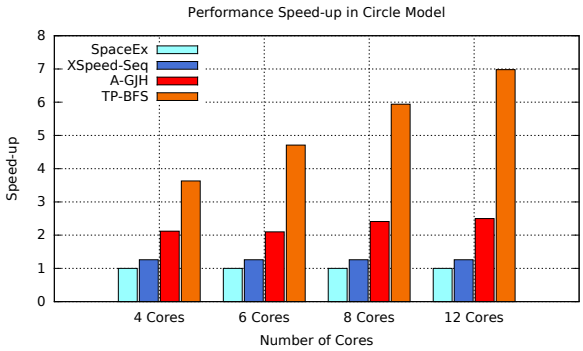
The performance and CPU utilization results for the Switching Oscillator and the Timed Bouncing Ball model is a good illustration of the effectiveness of the TP-BFS algorithm over A-GJH. In the Switching Oscillator model, BFS generates two new symbolic states at every level, one of which exits the location invariant early leaving only one expensive flowpipe to be computed at each level. We see that A-GJH is slower than Seq-BFS since it utilizes only one core for the flowpipe computation and incurs additional parallelization overhead. However, in case of TP-BFS the flowpipe task is distributed across all the available cores, making it faster than the Seq-BFS and A-GJH. For similar reason, A-GJH performs slower than sequential BFS in the Timed Bouncing Ball model. We observe a maximum speed-up of more than $100\times$ with TP-BFS in the Nav (3x3) (inst# 2) and $2000\times$ on the Nav (9x9) model with the A-GJH respectively in comparison to



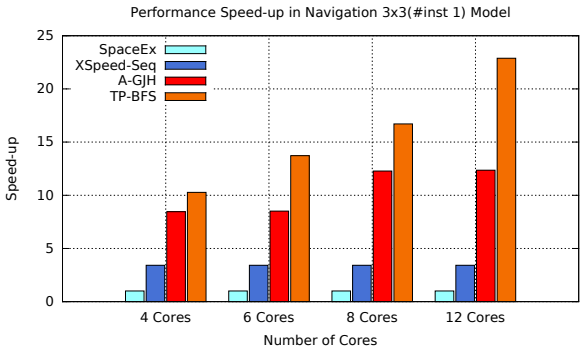
(a) Switching Oscillator Model



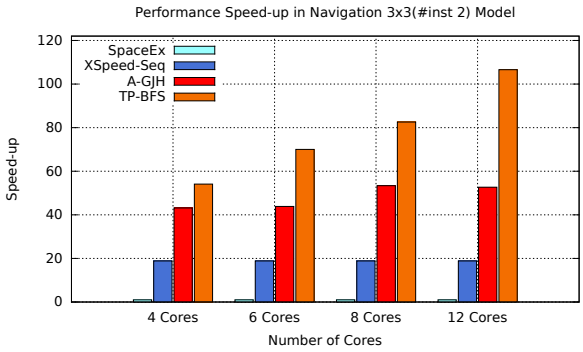
(b) Timed Bouncing Ball Model



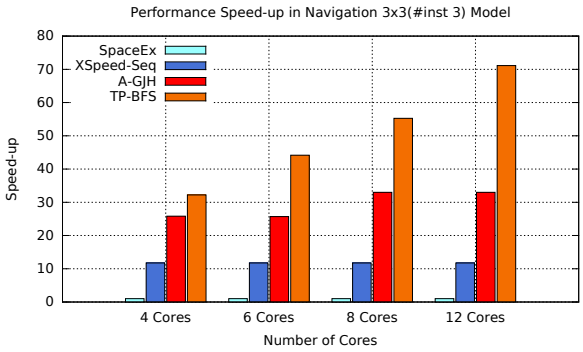
(c) Circle Model



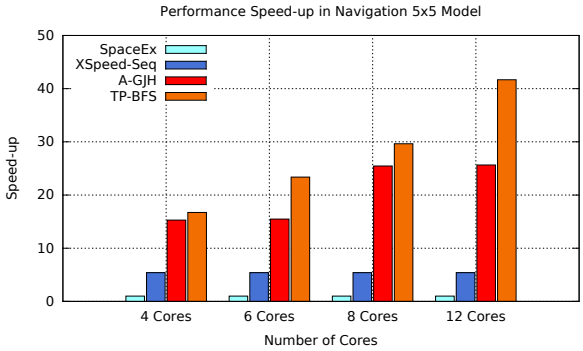
(d) Nav 3x3 (#inst 1) Model



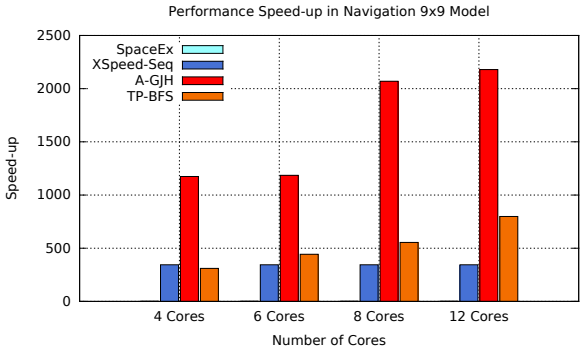
(e) Nav 3x3 (#inst 2) Model



(f) Nav 3x3 (#inst 3) Model



(g) Nav 5x5 Model

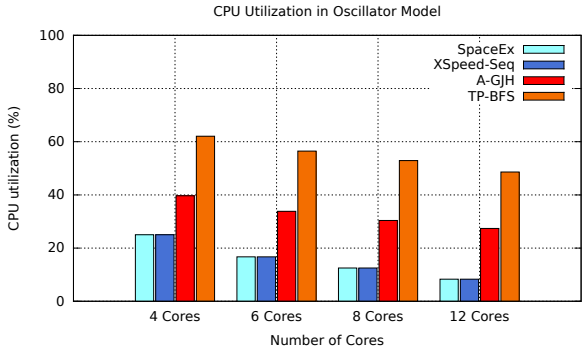


(h) Nav 9x9 Model

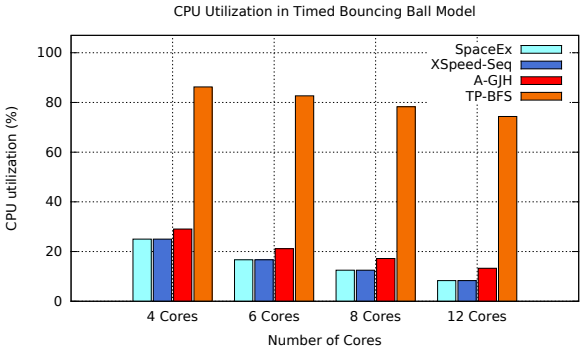
Fig. 5. Performance comparison of SpaceEx (LGG) , sequential BFS, A-GJH and TP-BFS on hybrid systems benchmarks.

SpaceEx (LGG). We also see an increase in the speed-up of the proposed parallel algorithms with the increase in the number of cores in the CPU, displaying scalability.

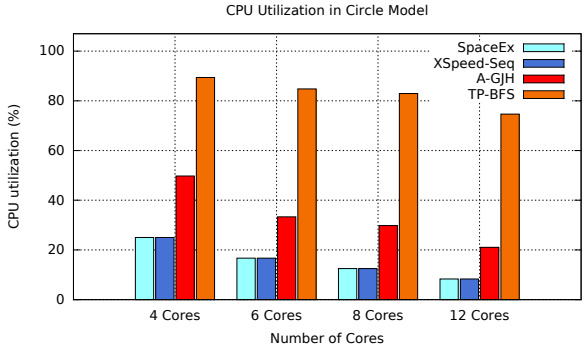
TP-BFS shows better performance and CPU utilization than A-GJH in all the models except Nav (9x9). We observe that for models where the number of symbolic states waiting to



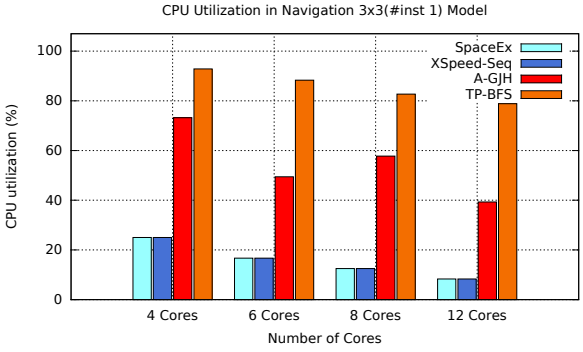
(a) Switching Oscillator Model



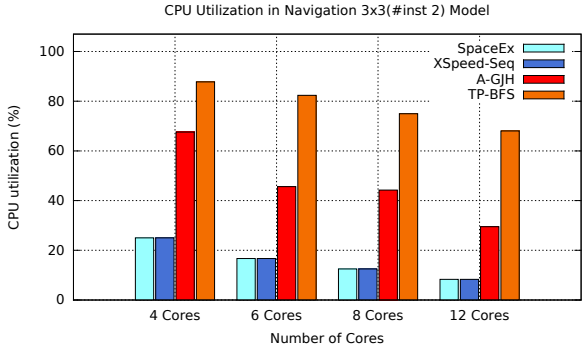
(b) Timed Bouncing Ball Model



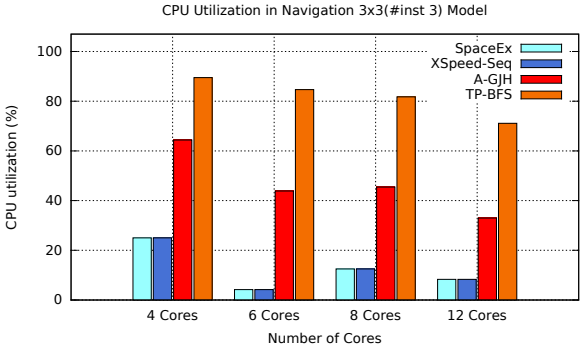
(c) Circle Model



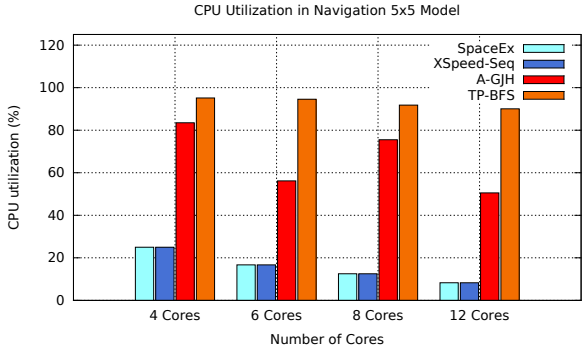
(d) Nav 3x3 (#inst 1) Model



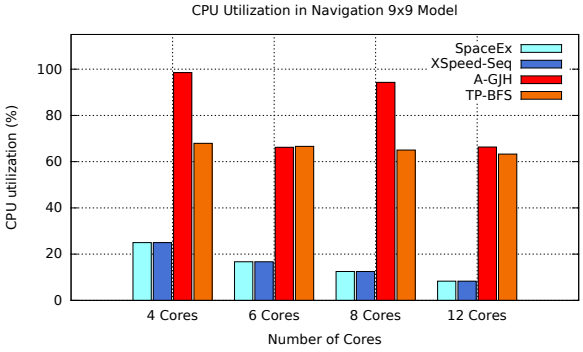
(e) Nav 3x3 (#inst 2) Model



(f) Nav 3x3 (#inst 3) Model



(g) Nav 5x5 Model



(h) Nav 9x9 Model

Fig. 6. CPU utilization of SpaceEx (LGG), sequential BFS, A-GJH and TP-BFS on hybrid systems benchmarks.

be explored in a BFS iteration is low to moderate, TP-BFS shows better performance and CPU utilization in comparison to A-GJH and SpaceEx (LGG). We observe that when there is a large number of symbolic states in the waiting list during BFS, as in the NAV 9×9 instance, the CPU core utilization and performance of A-GJH is better than TP-BFS. We believe

that this is because A-GJH keeps all available cores occupied, even if flowpipe computations are randomly assigned to cores, without taking their cost into consideration. In such a case, the extra overhead with task based load division becomes unnecessary as well as too expensive. This reduces CPU utilization (since flowpipe cost computation and load-division is a sequential routine) and performance. Figure 7 shows that the overhead of load balancing degrades the performance of TP-BFS with the increase in the number of symbolic states waiting to be explored during BFS, whereas the A-GJH algorithm consistently gains in performance and utilization. We verified that for the considered NAV 9×9 instance, the waiting list size in the BFS iterations are much larger than the available cores of the processor.

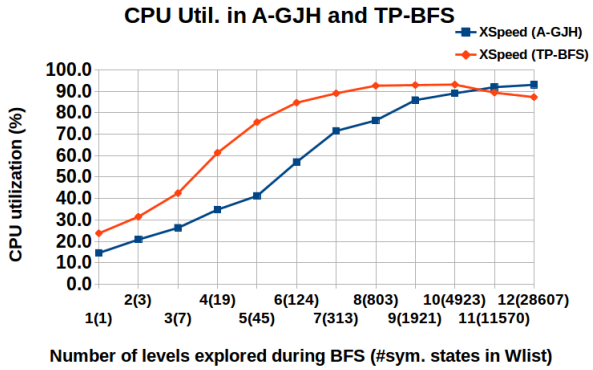


Fig. 7. Comparison of CPU Utilization between A-GJH and TP-BFS algorithm on a 9×9 Navigation model with sampling time as $1e-3$.

VII. Conclusion

We present an adaption of G.J. Holzmann's breadth first exploration algorithm of the SPIN model checker, for reachability analysis of hybrid systems. We show that due to the intricacies of post operators in hybrid systems, this first approach does not always produce an effective load balancing. We then propose an alternative approach for load balancing that splits reachability computation into atomic tasks and distributes them evenly according to an efficiently precomputed cost of the post operations. We provide an implementation of this approach using a support-function based algorithm. Our experiments show that this second approach shows in general a better load-balancing and performance with respect to the first one, with the exception when the number of symbolic states to be explored in the next step is considerably very large. Overall, the two proposed algorithms show a considerable improvement in performance with respect to the current state of the art.

VIII. Acknowledgement

This work was supported in part by DST-SERB, GoI under Project No. YSS/2014/000623 and by the Austrian Science Fund (FWF) under grants S11402-N23, S11405-N23 and S11412-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award) and by the ARC project DP140104219, "Robust AI Planning for Hybrid Systems".

References

[1] E. Bartocci, F. Corradini, M. R. D. Berardini, E. Entcheva, S. A. Smolka, and R. Grosu, "Modeling and simulation of cardiac tissue using hybrid I/O automata," *Theor. Comput. Sci.*, vol. 410, no. 33-34, pp. 3149–3165, 2009.

[2] E. Bartocci and P. Lió, "Computational modeling, formal analysis, and tools for systems biology," *PLoS Comput Biol*, vol. 12, no. 1, pp. 1–22, 2016.

[3] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*, second edition ed. E. A. Lee and S. A. Seshia, 2015.

[4] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *Journal of Computer and System Sciences*. ACM Press, 1995, pp. 373–382.

[5] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Proc. of CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 379–395.

[6] S. Bogomolov, C. Schilling, E. Bartocci, G. Batt, H. Kong, and R. Grosu, "Abstraction-based parameter synthesis for multiaffine systems," in *Proc. of HVC*, ser. LNCS, vol. 9434, 2015, pp. 19–35.

[7] S. Bogomolov, G. Frehse, M. Greitschus, R. Grosu, C. S. Pasareanu, A. Podelski, and T. Strump, "Assume-guarantee abstraction refinement meets hybrid systems," in *Proc. of HVC*, ser. LNCS. Springer, 2014, pp. 116–131.

[8] S. Bogomolov, A. Donzé, G. Frehse, R. Grosu, T. T. Johnson, H. Ladan, A. Podelski, and M. Wehrle, "Guided search for hybrid systems based on coarse-grained space abstractions," *STTT*, pp. 1–19, 2015.

[9] A. Girard and C. Le Guernic, "Efficient reachability analysis for linear systems using support functions," *Proc. of IFAC World Congress*, vol. 41, no. 2, pp. 8966–8971, 2008.

[10] A. Girard, "Reachability of uncertain linear systems using zonotopes," in *Proc. of HSCC 2015*, ser. LNCS, vol. 3414. Springer, 2005, pp. 291–305.

[11] C. L. Guernic and A. Girard, "Reachability analysis of hybrid systems using support functions," in *Proc. of CAV 2009*, ser. LNCS, vol. 5643. Springer, 2009, pp. 540–554.

[12] G. J. Holzmann, "Parallelizing the SPIN model checker," in *Proc. of SPIN 2012*, ser. LNCS, vol. 7385. Springer, 2012, pp. 155–171.

[13] R. Ray, A. Gurung, B. Das, E. Bartocci, S. Bogomolov, and R. Grosu, "XSpeed: Accelerating reachability analysis on multi-core processors," in *Proc. of HVC 2015*, ser. LNCS, vol. 9434, 2015, pp. 3–18.

[14] S. Bak, S. Bogomolov, and T. T. Johnson, "HYST: a source transformation and translation tool for hybrid automaton models," in *Proc. of HSCC'15*. ACM, 2015, pp. 128–133.

[15] C. Le Guernic and A. Girard, "Reachability analysis of linear systems using support functions," *Nonlinear Analysis: Hybrid Systems*, vol. 4, no. 2, pp. 250–262, 2010.

[16] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," *STTT*, vol. 10, no. 3, pp. 263–279, 2008.

[17] A. Hartmanns and H. Hermanns, "The modest toolset: An integrated environment for quantitative modelling and verification," in *Proc. of TACAS'14*, ser. LNCS, vol. 8413. Springer, 2014, pp. 593–598.

[18] N. Ramdani and N. S. Nedialkov, "Computing reachable sets for uncertain nonlinear hybrid systems using interval constraint-propagation techniques," *Nonlinear Analysis: Hybrid Systems*, vol. 5, no. 2, pp. 149–162, 2011.

[19] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, "Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 1, no. 3-4, pp. 209–236, 2007.

[20] S. Kong, S. Gao, W. Chen, and E. M. Clarke, "dReach: δ-reachability analysis for hybrid systems," in *Proc. of TACAS'15*, ser. Lecture Notes in Computer Science, vol. 9035. Springer, 2015, pp. 200–205.

[21] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Proc. of CAV'13*, ser. LNCS, vol. 8044. Springer, 2013, pp. 258–263.

[22] R. Testylier and T. Dang, "NLTOOLBOX: A library for reachability computation of nonlinear dynamical systems," in *Proc. of ATVA'13*, ser. LNCS, vol. 8172. Springer, 2013, pp. 469–473.

[23] T. Dang and D. Salinas, "Image computation for polynomial dynamical systems using the bernstein expansion," in *Proc. of CAV'09*, ser. LNCS, vol. 5643. Springer, 2009, pp. 219–232.

[24] R. Ray and A. Gurung, "Poster: Parallel state space exploration of linear systems with inputs using xspeed," in *Proc. of HSCC'15*. ACM, 2015, pp. 285–286.
[25] A. Fehnker and F. Ivancic, "Benchmarks for hybrid systems verification," in *Proc. of HSCC*, ser. LNCS, vol. 2993. Sprnger, 2004, pp. 326–341.
[26] A. Chutinan and B. H. Krogh, "Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations," in *Proc. of HSCC'99*, ser. LNCS, vol. 1569. Springer, 1999, pp. 76–90.
[27] R. T. Rockafellar and R. J.-B. Wets, *Variational Analysis*. Springer, 1998, vol. 317.
[28] C. Le Guernic, "Reachability analysis of hybrid systems with linear continuous dynamics," Ph.D. dissertation, Université Grenoble 1 - Joseph Fourier, 2009.

APPENDIX

Claim 1. *A-GJH algorithm performs a BFS of a HA with the number of BFS levels = bound.*

Proof. We show the correctness of the algorithm by the following loop invariant of the $repeat - until$ loop of the algorithm:

At the beginning of the iteration j of the *repeat-until* loop, the data structure R contains all the states of the HA reachable from $Init$ with $j - 1$ levels of BFS.

We use a *level* of a BFS to signify the frontiers of states reachable from $Init$. For example, $PostC(Init)$ denotes all states reachable upto a BFS level/frontier of 1 from $Init$, and $PostC(PostD(PostC(Init)))$ denotes all states reachable upto a BFS level/frontier of 2 from $Init$ and so on.

At *initialization*, $R[0]$ is assigned to $Init$. Therefore, the loop invariant is true at initialization, which says that at the beginning of the iteration 1 of the repeat-until loop, R contains all the states of the HA reachable from $Init$ with 0 levels of BFS.

We now show that the loop invariant is *maintained*. In lines 6 to 23, $PostC$ operator is applied to every symbolic state in $Wlist[t]$ and the result is included in R. The states reachable by $PostD$ transitions are added to $Wlist[1 - t]$ for exploration in the next iteration. Therefore, at each iteration, the BFS frontier is increased by 1, maintaining the loop invariant. Parallel exploration causes no race condition and write contention on the shared data-structure $Wlist$ and R. The justifications being the same as in the G.J. Holzmann's algorithm in the SPIN model checker [12].

The termination of the algorithm is evident from the termination condition of the $repeat-until$ loop. The loop terminates either when there are no new symbolic states for further exploration in $Wlist[t]$ or when the pre-determined $bound$ on the BFS levels is reached. It is clear that one of the condition must be eventually true and hence the algorithm must terminate.

At termination, the loop condition must be false, which means either $level = bound$ or $Wlist[t] = \emptyset$. In the former condition, the loop invariant at termination says that R contains all the states of the HA reachable from $Init$ with $bound$ levels of BFS, which is our claim. Termination due to the later condition implies that the fixed point has been found before BFS levels could reach the $bound$. In both cases, our claim holds.

□

Claim 2. *TP-BFS algorithm performs a BFS of a HA with the number of BFS levels = bound.*

Proof. The correctness of the algorithm can be proved using the same loop invariant used in the proof of claim 1. The arguments for the validity of the loop invariant are same except for the invariant *maintenance*. In lines 6 to 23, the $PostC$ and $PostD$ operations increases the BFS frontier/level by 1, maintaining the loop invariant but they are split into atomic tasks and inserted into a task list data structure. The algorithm provides a partitioned access of this task list to the cores, so that cores work on exclusive tasks with no race condition. The read/write switching of the data structure $Wlist$ is as in the A-GJH algorithm, which avoids locking during a BFS [12].

The termination proof of the algorithm is the same as the termination proof of claim 1.

□