Enhanced Distributed Behavioral Cartography of Parametric Timed Automata*

Étienne André, Camille Coti, Hoang Gia Nguyen

Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, Villetaneuse, France

Abstract. Parametric timed automata (PTA) allow the specification and verification of timed systems incompletely specified, or subject to future changes. The behavioral cartography splits the parameter space of PTA in tiles in which the discrete behavior is uniform. Applications include the optimization of timing constants, and the measure of the system robustness w.r.t. the untimed language. Here, we present enhanced distributed algorithms to compute the cartography efficiently. Experimental results show that our new algorithms significantly outperform previous distribution techniques.

Keywords: parametric verification, distributed algorithms, real-time systems

1 Introduction

Systems combining concurrent aspects with real-time constraints are notoriously difficult to exhaustively test, and their failure due to unsuspected bugs may lead to dramatic consequences. Model checking concurrent real-time systems aims at formally verifying the correctness of the system model w.r.t. a property.

The notion of timed automata (TA) is a well-known formalism for specifying and verifying concurrent real-time systems. TA extend finite-state automata with a set of clocks (real-time variables growing linearly) that can be compared with integer constants. TA are used in several powerful tools such as Uppaal [LPY97] or PAT [SLDP09]. However, the binary answer ("yes" or "no") output by model checking is not always satisfactory: indeed, it does not allow to change or optimize some values of the system constants, nor (in general) to evaluate the system robustness, *i.e.* the infinitesimal variation of timing constants while preserving the reachability or language. Parametric timed automata (PTA) [AHV93] extend TA with rational-valued parameters allowed in place of constants.

In [AF10], the behavioral cartography (BC) of PTA was proposed: given a bounded parameter domain D, BC partitions D in *tiles*, *i.e.* in parameter

* This is the author version of the paper of the same name accepted for publication at ICFEM 2015. The final publication is available at http://www.springer.com. This work was partially supported by a BQR grant "SynPaTiC", by the ANR national research program "PACS" (ANR-2014), and the INS2I PEPS JCJC 2015 "PSyCoS" project.

subspaces where the discrete (untimed) behavior is uniform. That is, the set of satisfied linear time properties is the same for any rational-valued parameter valuation ("point") in a tile. This helps to identify robust subspaces, in which the timing constants can vary with no harm w.r.t. the system correctness expressed in terms of the untimed language. In [ACE14], we sketched two master-worker point distribution algorithms to compute BC in a distributed fashion.

Contribution The goal of this paper is to propose efficient distributed algorithms to compute BC efficiently using parallel, distributed computing resources. We formalize the existing point-by-point distribution algorithms (Seq and Random), that were only informally sketched in [ACE14].[1] Then, our main contribution is to propose three new distributed algorithms to speed up the cartography: the first one (Static) is a static domain decomposition scheme, where each node works independently on its own parameter subdomain; the second one (Shuffle) addresses the drawbacks of Seq and Random; finally, the third one (Subdomain) is a new master-worker, dynamic, distributed domain decomposition process. We then evaluate our algorithms on real-time case studies. In all cases, our new algorithms Shuffle and (a variant of) Subdomain outperform the algorithms of [ACE14]. We also discuss how to choose the appropriate algorithm depending on the case study.

Related works The design of efficient parameter synthesis techniques has been tackled in various works, *e.g.* using SMT-based model checking techniques [CGMT13], or using symbolic techniques for integer synthesis [JLR15]. BC helps to quantify the system robustness; this has also been tackled using the "ASAP" semantics [DWDR05] (see, *e.g.* [Mar11] for a survey), but usually in only one dimension (a single variation δ of the timing delays is considered, whereas BC allows as many dimensions as parameters). To the best of our knowledge, with the exception of [ACE14], distributed computing techniques were not applied yet to parameter synthesis for PTA.

Formal verification can be made in parallel in two ways: modeling languages can be designed to be easy to use in a distributed fashion, or the verification algorithms themselves can be parallelized. Our approach fits in the second category. In recent years, some model checkers were extended to parallel computing, *i.e.* running on multicore computers. This is the case of PKind [KT11], APMC (a probabilistic model checker) [HBE+10], and FDR3 (for CSP refinement checking). More recently, two algorithms were proposed to address multi-core LTL verification [ELPP12] and emptiness checking of timed Büchi automata [LOD+13]. However, with the exception of FDR3 (that can run either on multicore or on clusters), these works run verification on multicore computers (with a shared memory) whereas our primary goal is to run verification on a cluster (where each node has its own memory). Furthermore, none of these works considered parameter synthesis.

[1] [ACE14] was published in a distributed computing community and focused on the parallelization technique used for this particular application, and the paper did not go into formal details. This is not an actual contribution of the current paper, but makes it standalone.

Outline We introduce the necessary notations in Section 2. We briefly define in Section 3 the static domain decomposition algorithm (Static). Then, we formalize in Section 4 the master-worker scheme and the two point distribution algorithms of [ACE14]; we also introduce a third point distribution algorithm (Shuffle). We introduce in Section 5 our new dynamic domain decomposition algorithm (Subdomain). We conduct experiments in Section 6 and conclude in Section 7.

2 Preliminaries

Parameter Constraints We assume here a set $X = \{x_1, \ldots, x_H\}$ of *clocks*, *i.e.* real-valued variables that evolve at the same rate. A clock valuation w is a function $w : X \to \mathbb{R}_+$. We denote by $X = 0$ the conjunction of equalities that assigns 0 to all clocks in X.

We assume a set $P = \{p_1, \ldots, p_M\}$ of *parameters*, *i.e.* unknown constants. A *parameter valuation* v is a function $v : P \to \mathbb{Q}_+$. We will often identify a valuation v with the *point* $(v(p_1), \ldots, v(p_M))$. An *integer* point is a valuation $v : P \to \mathbb{N}$. We denote by $\mathbf{0}$ the valuation assigning 0 to all parameters.

An *inequality* over X and P is $e \prec 0$, where $\prec \in \{<, \leq, \geq, >\}$, and e is a linear term $\sum_{1 \leq i \leq N} \alpha_i z_i + d$ for some $N \in \mathbb{N}$, where $z_i \in X \cup P$, $\alpha_i \in \mathbb{Q}$, for $1 \leq i \leq N$, and $d \in \mathbb{Q}$. A (linear) constraint over X and P is a set of inequalities over X and P. We define in a similar manner inequalities and constraints over P. A *guard* is a set of inequalities each of them referring to at most one clock.

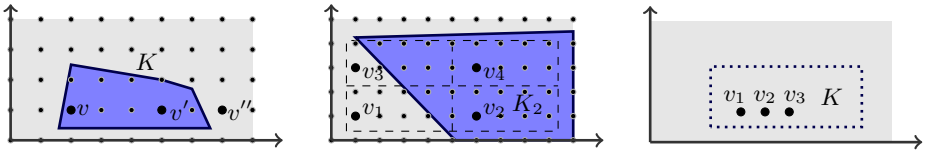
Given a parameter valuation v, $C[v]$ denotes the constraint over X obtained by replacing each parameter p in C with $v(p)$. We say that v *satisfies* C, denoted by $v \models C$, if the set of clock valuations satisfying $C[v]$ is nonempty.

We denote by $C{\downarrow_P}$ the projection of C onto P, *i.e.* obtained by eliminating the clock variables (using existential quantification). We define the *time elapsing* of C, denoted by $C^{\nearrow}$, as the constraint over X and P obtained from C by delaying an arbitrary amount of time. Given $R \subseteq X$, we define the *reset* of C, denoted by $[C]_R$, as the constraint obtained from C by resetting the clocks in R, and keeping the other clocks unchanged.

Definition 1. *A PTA $\mathcal{A}$ is a tuple $\mathcal{A} = (\Sigma, L, l_0, X, P, I, E)$, where: 1) Σ is a finite set of actions, 2) L is a finite set of locations, 3) $l_0 \in L$ is the initial location, 4) X is a set of clocks, 5) P is a set of parameters, 6) I is the invariant, assigning to every $l \in L$ a guard $I(l)$, and 7) E is a set of edges (l, g, a, R, l') where $l, l' \in L$ are the source and destination locations, g is the transition guard, $a \in \Sigma$, and $R \subseteq X$ is a set of clocks to be reset.*

Given a PTA $\mathcal{A} = (\Sigma, L, l_0, X, P, I, E)$, and a parameter valuation v, $\mathcal{A}[v]$ denotes the TA obtained from $\mathcal{A}$ by substituting every occurrence of a parameter p_i by the constant $v(p_i)$ in the guards and invariants.

Symbolic semantics. A symbolic state is a pair (l, C) with l a location, and C a constraint over $X \cup P$. The initial state of $\mathcal{A}$ is $s_0 = (l_0, (X = 0)^{\nearrow} \wedge I(l_0))$, *i.e.* clocks are initially set to 0, and can evolve as long as $I(l_0)$ is satisfied. The



(a) Graphical example (b) Redundancy (c) Choosing points

Fig. 1: Graphical representations and challenges

computation of the state space is as follows: Given a symbolic state $s = (l, C)$, $\mathsf{Succ}(s) = \{(l', C') \mid \exists (l, g, a, R, l') \in E \text{ s.t. } C' = \left([(C \wedge g)]_R\right)^{\nearrow} \cap I(l')\}$.

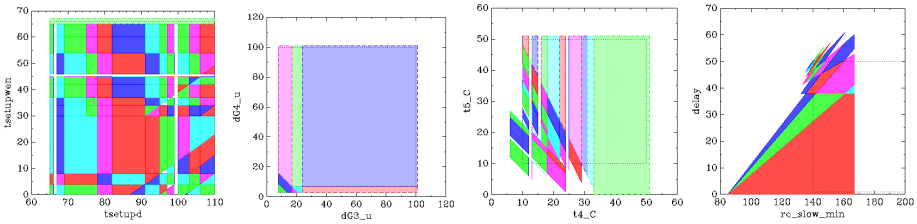
A symbolic run of a PTA is an alternating sequence of symbolic states and actions of the form $s_0 \overset{a_0}{\Rightarrow} s_1 \overset{a_1}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} s_m$, such that for all $i = 0, \ldots, m-1$, $a_i \in \Sigma$, and $s_i \overset{a_i}{\Rightarrow} s_{i+1}$ is such that s_{i+1} belongs to $\mathsf{Succ}(s_i)$ and is obtained via action a_i. In the following, we simply refer to the symbolic states belonging to a run of $\mathcal{A}$ starting from s_0 as states of $\mathcal{A}$. Given a run $(l_0, C_0) \overset{a_0}{\Rightarrow} (l_1, C_1) \overset{a_1}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} (l_m, C_m)$, its corresponding *trace* is $l_0 \overset{a_0}{\Rightarrow} l_1 \overset{a_1}{\Rightarrow} \cdots \overset{a_{m-1}}{\Rightarrow} l_m$. The set of all traces of a TA is called its *trace set*.

The Inverse Method The *inverse method* (IM) [AS13] generalizes the behavior of $\mathcal{A}[v]$ in the form of a *tile*, *i.e.* a parameter constraint K where the discrete behavior is uniform (see Fig. 1a, where $K = \mathsf{IM}(\mathcal{A}, v)$). That is, for any point v' satisfying K, the trace sets of $\mathcal{A}[v']$ and $\mathcal{A}[v]$ are equal. Hence any linear-time property (expressed in, *e.g.* LTL) valid in $\mathcal{A}[v]$ is also valid in $\mathcal{A}[v']$. Note that, in general, tiles have no predefined "shape": they are general polyhedra in $|P|$ dimensions that can have arbitrary size, number of vertices, and edge slope. The computation time of IM also greatly varies, from milliseconds to several hours, depending on the complexity of the model, and the size of the trace set.

The Behavioral Cartography Given a PTA $\mathcal{A}$ and a bounded parameter domain D (usually a hyperrectangle in $|P|$ dimensions), the *behavioral cartography* (BC) [AF10] repeatedly calls IM on (some of the) integer points of D (of which there is a finite number), so as to cover D with tiles. The result gives a tiling of D such that the discrete behavior (trace set) is uniform in each tile.

In Fig. 1a, BC first considers point v, and computes $K = \mathsf{IM}(\mathcal{A}, v)$. Then, BC iterates on the subsequent points, all already covered by K, until it meets v'', that is not yet covered. Hence, BC will then compute $\mathsf{IM}(\mathcal{A}, v'')$, and so on, until all integer points in D are covered.

BC can be used for several applications: first, it identifies the system robustness in the sense that, in each tile, parameters can vary as long as they remain in the tile, without impacting the system's discrete behavior. Second, BC can be used to perform parameter optimization; the weakest conditions of the input signal of an industrial asynchronous memory circuit (SPSMALL) were derived using BC [AS13]. Third, given a set of linear time properties (*i.e.* that can be verified on the trace set), it suffices to compute only once BC, and then to check



(a) SPSMALL (b) Flip-flop circuit (c) Schedulability (d) RCP

Fig. 2: Examples of graphical behavioral cartographies in 2 dimensions

each property on the trace set generated for each tile in order to know a complete (or nearly complete) set of parameter valuations satisfying each property.

Remark 1. BC does not guarantee the full, dense coverage of D for two reasons. *1)* IM may not terminate, as the corresponding problem is undecidable [AM15]. In our implementation of BC, this is addressed using a timeout: if $\mathsf{IM}(\mathcal{A}, v)$ does not terminate within some time bound, BC switches to the next integer point, and v will (most probably) never be covered. However, although it was shown possible in theory, this never happened in any of our experiments. *2)* IM generalizes integer points in the form of dense, rational-valued constraints, but it could happen in rare cases that some tiles do not contain any integer points. This sometimes happened in our experiments (*e.g.* in Fig. 2a around $x = 100$ and $y = 55$); usually, calling BC on multiples of $\frac{1}{3}$ instead of integers was empirically shown to be sufficient in most cases (although in theory there might be an infinite number of tiles in a bounded domain). Conversely, note that BC frequently covers (parts of) the parametric space beyond D; this is the case in Figs. 2b to 2d (in Fig. 2b, the entire parametric space is even covered).

Also note that the motivation for considering integer points is that, in most cases, considering integers is sufficient to cover entirely (or almost entirely) the domain D. However, as said above, our implementation allows any "step" instead of integers (*e.g.* multiples of $\frac{1}{3}$).

3 Static Domain Decomposition

In order to tackle larger case studies, our objective is to take advantage of the iterative nature of the cartography (in contrast to most, if not all, other known parameter synthesis algorithms), and to distribute it on N processes. There is no theoretical obstacle in doing so, since all calls to IM are independent from each other. The challenge is rather to select efficiently the points on which IM is called, so that as few redundant constraints as possible are computed.

In this section, we briefly describe a static domain decomposition ("Static"). That is, the rectangle D is split into N subdomains, and then each process is responsible for handling its own subdomain in an independent manner (with no

communication). This domain decomposition method is often used for regular data distributions, where all subdomains require the same processing time, and preferably on domain shapes such as rectangles or hypercubes, that can easily be mapped on a grid of processes.

Each node i performs the following procedure:

1. split D into N subdomains;[2]
2. execute BC on the ith subdomain, *i.e.* iteratively select integer points and call IM until all integer points in the ith subdomain are covered by tiles.

For example, in Fig. 1b, the domain D (the external dashed rectangle) is split into four equal subdomains (the four internal dashed rectangles); v_i, $1 \leq i \leq 4$ represents a possible first point on which to call IM in each subdomain. (K_2 in Fig. 1b will be used later on.)

This static decomposition is straightforward but is not satisfactory for BC for three main reasons.

First, the general "shape" of the cartography is entirely arbitrary and unknown beforehand, since tiles can themselves have any shape. Fig. 2 gives examples of cartographies in 2 parameter dimensions: although the geometrical distribution of the tiles of Fig. 2a within D is rather homogeneous, this is not true at all for the others. For example, splitting the domain of Fig. 2b (resp. Fig. 2d) into four equal parts would be very unfair for the node responsible of the lower-left (resp. upper-right) subdomain, since most tiles are concentrated there; this would also be inefficient, since the other nodes will rapidly become idle.

Second, the geometrical distribution of the tiles says nothing on the *time* necessary to compute each tile. Recall that the computation of IM can be very long (up to several hours). Even when the tiles are homogeneously located within D, some tiles may require much more time than others. For example, in Fig. 2a (where the geometrical distribution of the tiles is rather homogeneous), it could happen that the bottom-left tiles require much more time than others, resulting in this node to work much longer, while the other nodes would rapidly finish their duty. Again, this would result in a loss of efficiency due to load unbalance since not all of the nodes are working actively.

Third, the absence of communication between nodes may result in redundant computations. Let us go back to the example of cartography in Fig. 1b. Assume that node 2 finished first to compute a tile, say K_2. This tile not only covers the entire subdomain of node 2, leading to the termination of process 2, but it also covers node 4's subdomain entirely and a large part of node 2's subdomain. Without communication, these nodes will keep working without knowing that their subdomain has already been covered. In contrast, a smarter distribution scheme should be such that, in this situation, nodes 2, 3 and 4 would go to help node 1 finish its (not much covered yet) subdomain. We will address this efficiency issue in the remainder of this paper.

[2] Alternatively, a single node could perform the split and then send to each other node its own subdomain (at the cost of additional communications).

Master tag	Argument
POINT(v)	parameter valuation
STOP	-
SUBDOMAIN(sd)	new subdomain
TILES(T)	latest tiles

Worker tag	Argument
COMPLETED	-
NOTIFYPOINT(v)	parameter valuation
REQTILES	-
RESULT(K)	constraint computed

Table 1: Tags for master-worker communications

4 Master-Worker Point Distribution Algorithms

We first recall our master-worker scheme (Section 4.1); then, we formalize the abstract algorithm for the master (Section 4.2), the Seq (Section 4.3) and the Random point distribution (Section 4.4) – only informally described in [ACE14]. Additionally, we introduce a new point distribution Shuffle (Section 4.5).

4.1 Principle: Master-Worker

Workers ask the master for a point v, then execute $\mathsf{IM}(\mathcal{A}, v)$, and finally send the corresponding result K to the master. The master does not call IM itself, but instead distributes points to the workers. Whereas this may be a loss of efficiency for few processes, this shall be compensated for a large number of processes. Moreover, this parallel computation scheme balances the load between workers automatically.

The master and workers communicate with each other by sending messages that are labeled using *tags*, using two asynchronous functions $send(n, msg)$ and $receive()$. Function $send(n, msg)$ sends a tagged message msg to node n. Function $receive()$ is a blocking function that waits until a message is received, and returns a pair (n, msg), where msg is the tagged message that has been received from node n. Based on the tag of the message, receiving processes can decide what to do with the message itself. Note that workers never communicate with each other. We assume that messages are made of a tag and zero or one argument: for example, POINT(v) sends a POINT tag together with the parameter valuation v. We give the list of tags used throughout this paper in Table 1.

4.2 An Abstract Algorithm for the Master

We first formalize in Algorithm 1 the "abstract" master algorithm sketched in [ACE14]; this algorithm contains variation points that can be instantiated to give birth to concrete master algorithms. In this section, we only use the worker tag RESULT and the master tags POINT and STOP. The workers only call the inverse method on the point they receive from the master, and send the result back, until a STOP tag is received (formalized in Algorithm 8 in Appendix B.1).

Algorithm 1 takes as input a PTA $\mathcal{A}$ and a parameter domain D; it is also parameterized by a *point distribution mode* M. Each mode is responsible for instantiating the variation points to give birth to a concrete algorithm. The master

Algorithm 1: Abstract algorithm for the master

```
   input  : PTA 𝒜, domain D, number of processes N, mode M
   output : Set of tiles T
   // Initialization phase
1  T ← ∅  ;  M.initialize()
2  foreach process n ∈ {1,...,N} do send(n, POINT(M.choosePoint())) ;
   // Main phase
3  while there are uncovered integer points in D do
4  |  n, RESULT(K) ← receive()  ;  T ← T ∪ {K}
5  |  send(n, POINT(M.choosePoint()))
   // Finalization phase
6  foreach process n ∈ {1,...,N} do
7  |  n, RESULT(K) ← receive()  ;  T ← T ∪ {K}  ;  send(n, STOP)
8  return T
```

starts by creating an empty set of tiles and then calls the mode initialization function M.*initialize*(), that initializes the various variables needed by the concrete algorithms (line 1). Then, the master sends a point to each node n; the way these points are chosen among D (M.*choosePoint*()) is decided by the mode (line 2). Then the master enters the main loop (line 3 to line 5): while there are uncovered points, every time a node n sends a constraint K and asks for work, the master stores the result in its list of tiles; then, it selects a point according to M and sends it to n. Finally, once all integer points are covered, the master receives results from the remaining nodes and sends STOP tags (line 6–line 7).

The way points are picked by the master to be distributed to the workers is a highly critical question. Choosing points in a wrong manner can lead to a dramatic loss of efficiency. For example, choosing points very close to each other would most probably lead to the (redundant) computation of the same tile. This situation is depicted graphically in Fig. 1c, where points v_1, v_2, v_3 may yield the same tile K. In the next three subsections, we formalize three master modes; these modes will define additional global variables and must instantiate *initialize*() and *choosePoint*().

4.3 Sequential Point Distribution

The first point distribution algorithm (Seq) is a direct extension of the monolithic (*i.e.*, non-distributed) algorithm: as in the non-distributed BC, it enumerates all the points of D in a sequential manner starting from $\mathbf{0}$. Seq assumes a function *nextPoint* that, given a parameter valuation v and a parameter domain D, returns the next point in D for some lexicographic order on the points of D. Seq maintains a single global variable v_{prev}, storing the latest point sent to a worker. The initialization function Seq.*initialize*() sets v_{prev} to a special value $\perp$ such that *nextPoint*($\perp$) returns the smallest point in D (*e.g.* $\mathbf{0}$ if $\mathbf{0} \in D$).

Algorithm 2: Seq.*choosePoint*()

variables : Point v_{prev}
output : Point v

1 $v \leftarrow v_{prev}$
2 **repeat** $v \leftarrow nextPoint(v, D)$ **until** v *is not covered by any tile in* T;
3 $v_{prev} \leftarrow v$; **return** v

Seq.*choosePoint*() (given in Algorithm 2) returns the next point of D not covered yet by any tile.

The main advantage of Seq is that it is inexpensive on the master's side. Its main drawback is the risk of redundant computations by the workers, due to the situation depicted graphically in Fig. 1c: for instance, at the beginning, the N processes will ask for work, and the master will give them the first sequential N points, all very close to each other, with a high risk of redundant computation.

4.4 Random + Sequential Point Distribution

The second point distribution algorithm (Random) selects points randomly, and then in a second phase performs a sequential enumeration to check the full coverage of integers in D. This second phase is necessary to guarantee that all the integer points have been covered. The second phase starts after a given number MAX of consecutive failed attempts to find an uncovered point randomly. Indeed, simply stopping BC after MAX tries could give a probabilistic coverage (*e.g.* 99 %) of integer points, but cannot guarantee the full coverage. Since finding the points not covered by a list of tiles has no efficient practical solution, this sequential check is the only concrete option we have.

Random maintains two global variables. First, *seqPhase* acts as a flag to remember whether the algorithm is in the first or second phase. Second, v_{prev} stores the latest point sent to a worker (just as in Seq). Random.*initialize*() initially sets *seqPhase* to `false` and v_{prev} to $\perp$.

We give Random.*choosePoint*() in Algorithm 3. In the first phase (line 1 to line 7), Random.*choosePoint*() randomly computes a point, and then checks whether it is covered by any tile; if not, it is returned. Otherwise, a second try is made, and so on, until the maximum number MAX of attempts is reached. In that latter case, it switches to the second phase (line 8 to line 11), consisting in a sequential enumeration of all the points just as in Seq.*choosePoint*().

4.5 Shuffle Point Distribution

The main problem of Random is the fact that the second phase, necessary to check the full coverage of integers, may be costly and even useless if almost all the points have already been covered. To alleviate this problem, we propose a new algorithm Shuffle that first computes statically a list of all integer points in D, then shuffles this list, and then selects the points of the shuffled list in a

Algorithm 3: Random.*choosePoint*()

```
   variables : Point v_prev, flag seqPhase
   output    : Point v
   // First phase
 1 if ¬seqPhase then
 2 |   nbTries ← 0
 3 |   while nbTries < MAX do
 4 |   |   v ← randomPoint(D)
 5 |   |   if v is not covered by any tile in T then return v;
 6 |   |   nbTries ← nbTries + 1
 7 |   seqPhase ← true
   // Second phase
 8 if seqPhase then
 9 |   v ← v_prev
10 |   repeat v ← nextPoint(v) until v is not covered by any tile in T;
11 |   v_prev ← v ; return v
```

sequential manner. The sequential phase of Random is then dropped, at the cost of being able to compute, store statically and shuffle a large quantity of points.

Shuffle maintains a single global variable, *i.e.* the list *allPoints* of all the points in D that has been shuffled. The Shuffle.*initialize*() function assigns *shuffle*(*allIntegers*(D)) to *allPoints*. (We assume here that function *allIntegers*(D) returns the list of all the integer points of D, and function *shuffle*(L) shuffles the elements of a list L.)

Then, the Shuffle.*choosePoint*() function simply consists in selecting the next uncovered point in *allPoints*. That is, it performs *pop*(*allPoints*), until the point output is not covered by any tile, in which case it returns it (we assume here that function *pop*(L) pops the first element of the list L and returns it).

5 Dynamic Domain Decomposition

The most intuitive solution for distributing BC is the Static distribution scheme of Section 3, *i.e.* to split D into N subdomains, and then ask each process to handle its own subdomain in an independent manner. As said in Section 3, this may lead to inefficient computations (which will be confirmed by our experiments in Section 6). Still, we use this idea to set up a *dynamic* domain decomposition algorithm. This algorithm is different from the previous ones, in the sense that it does not fit in the abstract master algorithm formalized in Section 4.2.

Initially, the master splits in D into N subdomains, and distributes the subdomains to the workers. In contrast to the algorithms of Section 4, the workers are now responsible for checking whether all the points in their subdomain have been covered yet or not. This mechanism reduces the load on the master without leading to redundant point coverage checks. Then, when a worker has covered

Algorithm 4: Subdomain: Master

```
   input  : PTA 𝒜, domain D, number of processes N
   output : Set of tiles T
   // Initialization phase
1  T ← ∅  ;  SD, currentPoints ← initialSplit(D, N)
2  foreach process n ∈ {1, ..., N} do send(n, SUBDOMAIN(SD[n])) ;
   // Main phase
3  while a subdomain in SD can be split do
4      switch receive() do
5          case n, NOTIFYPOINT(v): currentPoints[n] ← v ;
6          case n, RESULT(K): T ← T ∪ {K} ;
7          case n, REQTILES: send(n, TILES(T)) ;
8          case n, COMPLETED:
9              n', sd1, sd2 ← split(SD, currentPoints, n)
10             send(n, SUBDOMAIN(sd1))  ;  send(n', SUBDOMAIN(sd2))
   // Finalization phase
11 switch receive() do
12     case n, RESULT(K): T ← T ∪ {K} ;
13     case n, COMPLETED: send(n, STOP) ;
14 return T
```

all the integer points in its subdomain (because the points are covered by tiles computed either by this worker, or by other workers), it informs the master; the master dynamically splits a subdomain (typically, one that has only been covered a little) and sends it back to the idle worker.

The main idea is that the master is responsible for handling the dynamic distribution of the subdomains (including detecting the slowest workers to split their subdomain), whereas the workers are responsible for covering all the points in their subdomain in a sequential manner. There is no need for more complex algorithms, since each worker is working on its own in its own subdomain.

5.1 Master Algorithm

In the following, we assume several functions. We believe that understanding the role of these functions is straightforward; in practice, they lead to very tricky implementation issues (especially for the *split* function with arbitrary numbers of processes and parameter dimensions).

We give the master algorithm in Algorithm 4. Besides the list of tiles T, the master maintains two arrays of size N: the array SD associating with each node its current subdomain, and the array *currentPoints* associating with each node its latest known point (used to understand how advanced a worker is in its subdomain). These two arrays are initialized using the function *initialSplit* that splits D into N subdomains (line 1). Then the master sends its subdomain (line 2) to each node.

The algorithm then enters its main phase (line 3 to line 10). The master waits for incoming messages received via the asynchronous, blocking function *receive*(). If a new point is received (line 5), the master updates the *currentPoints* array (this is needed to perform splits using the most up-to-date data). If a result is received (line 6), the master stores it. If a request for tiles is received (line 7), the master sends all the tiles back so that n can update its local list.[3][4] If the master is notified that a worker n has completed its subdomain, *i.e.* all of its points have been covered (line 8), the master finds out which subdomain is the least covered, *i.e.* which workers are the most in need for assistance; this is performed by $split(SD, currentPoints, n)$, that returns the node n' needing help, and two new subdomains sd_1 and sd_2 split from n''s former subdomain, while updating SD (line 9). The master then informs both nodes of the split (line 10).

Finally, when no subdomain can be split (*i.e.* all non-completed subdomains contain only one point), the master stores the last tiles it receives (line 12) and sends a STOP signal to the workers (line 13).

5.2 Worker Algorithm

We give the Subdomain worker algorithm in Algorithm 5. Each worker waits for messages from the master: whenever a STOP signal is received from m (m stands for the master node id), the worker terminates (line 3). Otherwise, a subdomain sd is received (line 4): the worker then covers sd with tiles (line 5 to line 11) by calling IM sequentially on consecutive integers as in the Seq (master) algorithm. The worker selects a point, sends it to the master for update purpose, calls IM on that point, sends the result to the master, asks for an update of the list of tiles, and so on. When sd is covered, the worker notifies the master (line 12), and then waits again for a new message from the master until termination.[5]

5.3 An Additional Heuristic

It may happen that, while a node is calling IM on a point v, another node has covered v with its own tile. For example, in Fig. 1b, node 2 calls IM on point v_2, while node 4 calls IM on point v_4. Assume calling IM on point v_2 yields K_2, that incidentally covers v_4. It is more efficient to stop the computation of IM on v_4, so that node 4 moves to another point instead of computing a redundant tile. We hence improve Subdomain by adding a heuristic that prevents this situation as

[3] For efficiency purpose, in our implementation, the master only sends the new tiles since n's latest request (which is ensured using additional queue data structures).

[4] The local list is necessary to detect whether a point in the worker's subdomain is covered by a tile computed by another worker.

[5] Additionally, the worker checks whether the master has split its subdomain, because some other worker completed its own subdomain. In our implementation, this requires on the worker's side frequent (but inexpensive) checks whether the master has split the worker's current subdomain and, if so, a simple update of the subdomain.

Algorithm 5: Subdomain: Worker n

```
   input     : PTA 𝒜
   variables : Set of tiles T, point v_prev
1  while true do
2      switch receive() do
3          case m, STOP:  return ;
4          case m, SUBDOMAIN(sd):
5              while there are uncovered points in sd do
6                  v ← Seq.choosePoint()
7                  send(m, NOTIFYPOINT(v))
8                  K ← IM(𝒜, v)
9                  send(m, RESULT(K))
10                 m, TILES(receivedTiles) ← receive()
11                 T ← T ∪ receivedTiles
12             send(m, COMPLETED)
```

follows: the master keeps track of all the points currently processed by each node; whenever a constraint computed by a node i covers the current node processed by another node j, the master informs immediately node j, and this node stops its computation to move to the next point. We refer to Subdomain augmented with this heuristics as Subdomain + H. This heuristic might be expensive, both on the master side and on the worker side (frequent checks to perform, and more communication), hence we will study both Subdomain and Subdomain + H.

6 Experiments

We implemented our algorithms in the working version (2.7) of IMITATOR [AFKS12].[6]

We are presenting here results using seven case studies: Flip-flop4 is a 4-parameter dimension asynchronous flip-flop circuit. RCP is a parametric model of the root contention protocol (inspired by the TREX [ABS01] model). Sched3-2, Sched3B-2, Sched3B-3 and Sched5 are parametric schedulability problems, where the goal is to find tiles where the system is robustly schedulable. SiMoP is a parametric networked automation system [AS13]. More details on the configuration of the case studies are given in Appendix C.2. We give in the "model" part of Table 2 the number of clocks, of parameters, and of integer points in D for each case study. In the "cartography" part, we give the number of tiles and the time (in seconds) to compute the non-distributed cartography ("monolithic"). Note that the number of tiles gives an upper bound on the number of nodes above which a perfect distribution algorithm cannot become more efficient: if each node computes a different tile, then using more than n nodes cannot be faster than n nodes. Hence, we bound the analysis to the smallest power of 2 greater or equal to # Tiles ("N_{max}").

[6] Sources, models and results are available at www.imitator.fr/static/ICFEM15/.

Methodology We compute BC for each algorithm, for a number of nodes from 4 to 128 (see Appendix C.3 for all details and plots). For sake of brevity, we study here the performances at $n = N_{max}$. The execution time (in seconds) is given in the third part ("Execution time") of Table 2. (The algorithm Hybrid will be explained later on.)

We use two metrics to evaluate our algorithms. The first metric is the following ratio, that compares algorithms with each other, independently of their absolute performances: for each algorithm and each case study, we compute the time for this case study and this algorithm for N_{max} nodes divided by the maximum over all algorithms for this case study for N_{max} nodes, and multiplied by 100. A ratio equal to 100 means that this algorithm is the slowest for this case study, and a small ratio indicates a more efficient algorithm.

The second metrics is the speedup, that evaluates the scalability of each algorithm: for each algorithm and each case study, we compute the time for this case study and this algorithm for N_{max} nodes divided by the time needed for a perfect algorithm (*i.e.* the monolithic time divided by N_{max}), and multiplied by 100. Here, a number close to 100 means a very scalable algorithm, whereas a number close to 0 indicates an algorithm that does not scale well.

In the following, we describe the performance of each algorithm according to Table 2, before concluding which is the most efficient strategy.

Static This static domain decomposition algorithm is clearly not efficient, which shows that BC cannot be efficiently distributed using classical techniques for regular data distribution. Static is the worst algorithm twice (for RCP and Sched3B-2), and never the most efficient; a surprise is the very good performance for Flip-flop4, which probably comes from the fact that the tiles are very homogeneous geometrically for this case study, making a static distribution efficient.

Seq Although it is easy to implement, this algorithm is terribly inefficient: with 3 case studies for which it is the worst algorithm, it is also the worst in average. This comes from the fact that Seq is very likely to distribute to different nodes points that are close to each other, leading to redundant computations.

Random This algorithm behaves well for case studies with relatively few points in D, but it is always behind Shuffle in that case. It does not perform as well on case studies with large D, most likely because of the sequential enumeration of all points in the second phase of Random.

Shuffle With four case studies for which it is the best one, Shuffle is very efficient when D does not contain too many points; shuffling the points guarantees a good random repartition of the points, without entailing complex operations at the master side... at the cost of being able to shuffle a large quantities of points. This latter aspect certainly explains the low performances for Flip-flop4 and Sched3B-3.

Subdomain This algorithm is always outperformed by its variant Subdomain + H; it seems that the cost of checking which node is computing which point and the additional necessary communications are largely compensated by the benefit of preventing redundant computations brought by stopping ongoing executions.

Subdomain + H This algorithm has the best average speedup (17 %). Although it clearly outperforms Shuffle for only two experiments (Flip-flop4 and Sched3B-3), Subdomain + H is for no case study very far from the best algorithm. This could make a good candidate for the best distribution algorithm – but we advocate in the following for a better proposition.

Conclusion: Hybrid From the experiments, we notice that Subdomain + H is always among the most efficient, but is outperformed by Shuffle for case studies with relatively few points in D. Hence, we propose the following "algorithm": if D contains relatively few points (say, less than 100,000), use Shuffle, otherwise use Subdomain + H. Note that the condition (number of points in D) only depends on the input of the analysis, and can be checked very easily. This new algorithm "Hybrid" is always the best one – except for RCP, for which it is very slightly slower than Subdomain + H despite a small number of points (3,050). In addition, Hybrid gets the smallest average ratio (31 %) and the highest speedup (20 %).

Discussion An average speedup of 20 % at N_{max} for Hybrid can seem relatively low; this means that a perfect distribution algorithm (that would always divide the monolithic computation time by N) would be 5 times faster. Still, we find it promising. First, all distributed algorithms suffer from the time spent in communication, which always lowers the speedup. Second, this confirms that distributing BC is far from trivial, due to the unknown shape of the cartography, the unknown computation time for each tile, and the risk for redundant computations. Third, and most importantly, a speedup of 20 % means that, when using 128 nodes, the computation time is still divided by more than 25 – which leads to an impressive decrease of the verification time.

7 Final Remarks

We proposed here distribution algorithms to compute the cartography relying on the inverse method. In fact, one can use other algorithms than IM to obtain different "cartographies"; this is the case of [ALNS15] where we use a reachability preservation algorithm ("PRP") instead of IM so as to obtain, not a behavioral cartography, but a simple "good/bad" partition with respect to a reachability property. Distributing PRP using Subdomain often outperforms the monolithic bad-state reachability synthesis (*e.g.* [AHV93,JLR15]). Hence, we believe that our point distribution algorithms can be reused for different purposes than just BC.

In addition to using distributed computing resources, our aim is to design multicore algorithms for parameter synthesis, in the line of [ELPP12,LOD+13] – and then combine both approaches.

Finally, we would like to formally verify the master-worker communication scheme of Sections 4 and 5, so as to avoid potential deadlocks caused by a node waiting for a message that cannot arrive at that point.

Case study	Flip-flop4	RCP	Sched3-2	Sched3B-2	Sched3B-3	Sched5	SiMoP	Average
Model								
Clocks	5	6	13	13	13	21	8	
Parameters	4	2	2	2	3	2	2	
$\|D\|$	386400	3050	286	14746	530856	1681	10201	
Cartography								
# Tiles	190	19	59	71	378	177	48	
N_{max}	128	32	64	128	128	128	64	
Monolithic	1341.0	1992.0	46.0	61.2	865.0	3593.0	111.6	
Execution time at N_{max} (s)								
Static	33.0	2108.0	4.0	26.6	181.0	213.0	21.4	
Seq	2059.0	653.0	4.6	11.0	810.0	219.0	36.1	
Random	652.0	635.0	3.6	8.4	524.0	148.0	23.6	
Shuffle	670.0	624.0	3.1	7.6	243.0	140.0	18.7	
Subdomain	48.0	1286.0	7.2	15.8	217.0	273.0	32.4	
Subdomain + H	24.0	622.0	4.0	11.0	81.0	199.0	23.2	
Hybrid	24.0	624.0	3.1	7.6	81.0	140.0	18.7	
Ratio at N_{max} w.r.t. slowest at N_{max} (%)								
Static	2	100	56	100	22	78	59	60
Seq	100	31	64	41	100	80	100	74
Random	32	30	50	32	65	54	65	47
Shuffle	33	30	43	29	30	51	52	38
Subdomain	2	61	100	59	27	100	90	63
Subdomain + H	1	30	56	41	10	73	64	39
Hybrid	1	30	43	29	10	51	52	31
Speedup at N_{max} (%)								
Static	32	5	19	3	4	13	11	12
Seq	1	16	17	8	1	13	6	9
Random	2	17	22	10	1	19	10	11
Shuffle	2	17	25	11	3	20	12	13
Subdomain	22	8	11	5	3	10	7	10
Subdomain + H	44	17	19	8	8	14	10	17
Hybrid	44	17	25	11	8	20	12	20

Table 2: Summary of experiments

References

ABS01. Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX: A tool for reachability analysis of complex systems. In *CAV*, Lecture Notes in Computer Science, pages 368–372. Springer, 2001.

ACE14. Étienne André, Camille Coti, and Sami Evangelista. Distributed behavioral cartography of timed automata. In *EuroMPI/ASIA*, pages 109–114. ACM, 2014.

AF10. Étienne André and Laurent Fribourg. Behavioral cartography of timed automata. In *RP*, volume 6227 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2010.

AFKS12. Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *FM*, volume 7436 of *Lecture Notes in Computer Science*, 2012.

AHV93. Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *STOC*, pages 592–601. ACM, 1993.

ALNS15. Étienne André, Giuseppe Lipari, Hoang Gia Nguyen, and Youcheng Sun. Reachability preservation based parameter synthesis for timed automata. In *NFM*, volume 9058 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2015.

AM15. Étienne André and Nicolas Markey. Language preservation problems in parametric timed automata. In *FORMATS*, Lecture Notes in Computer Science. Springer, 2015. To appear.

AS13. Étienne André and Romain Soulat. *The Inverse Method*. ISTE Ltd and Wiley & Sons, 2013.

CGMT13. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Parameter synthesis with IC3. In *FMCAD*, pages 165–168. IEEE, 2013.

DWDR05. Martin De Wulf, Laurent Doyen, and Jean-François Raskin. Almost ASAP semantics: From timed models to timed implementations. *Formal Aspects of Computing*, 17(3):319–341, 2005.

ELPP12. Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco Van De Pol. Improved multi-core nested depth-first search. In *ATVA*, volume 7561 of *LNCS*, pages 269–283, 2012.

HBE+10. Khaled Hamidouche, Alexandre Borghi, Pierre Esterie, Joel Falcou, and Sylvain Peyronnet. Three high performance architectures in the parallel APMC boat. In *PMDC*. IEEE, 2010.

JLR15. Aleksandra Jovanović, Didier Lime, and Olivier H. Roux. Integer parameter synthesis for timed automata. *IEEE Transactions on Software Engineering*, 41(5):445–461, 2015.

KT11. Temesghen Kahsai and Cesare Tinelli. PKind: A parallel k-induction based model checker. In *PDMC*, volume 72 of *EPTCS*, pages 55–62, 2011.

LOD+13. Alfons Laarman, Mads Chr. Olesen, Andreas Engelbredt Dalsgaard, Kim Guldstrand Larsen, and Jaco Van De Pol. Multi-core emptiness checking of timed Büchi automata using inclusion abstraction. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*. Springer, 2013.

LPY97. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

Mar11. Nicolas Markey. Robustness in real-time systems. In *SIES*, pages 28–34. IEEE Computer Society Press, 2011.

SLDP09. Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

Appendix

A Existing Algorithms

A.1 The Inverse Method Algorithm

We recall here IM from [AS13].

Given a parameter valuation v, a state (l, C) is said to be v-compatible if $v \models C$. We extend Succ to sets of states as follows: given a set S of states, $\mathsf{Succ}(S) = \{s' \mid \exists s \in S \text{ s.t. } s' \in \mathsf{Succ}(s)\}$. Given a set S of symbolic states, we denote by $\mathsf{Succ}^j(S)$ the set of states reachable from S in exactly j steps, *i.e.* the composition of j times Succ.

Here, we consider PTA extended with an initial parameter domain (as considered in, *e.g.* [AS13]). That is, these PTA have their possible parameter valuations restricted to belong to the set defined by a parameter constraint K. When clear from the context, given a PTA $\mathcal{A}$ and a constraint K, we denote by $\mathcal{A}(K)$ the PTA initially constrained by $\mathcal{A}$. (This can be simulated using an initial gadget that will ensure this constraint over the parameters before the actual initial location of the PTA.)

We use notation $\mathsf{Succ}_{\mathcal{A}(K)}$ to denote that the Succ operation is applied to PTA $\mathcal{A}(K)$.

Algorithm 6: Inverse method $\mathsf{IM}(\mathcal{A}, v)$

input : PTA $\mathcal{A}$, parameter valuation v
output : Constraint K over the parameters

1 $i \leftarrow 0$; $K_c \leftarrow \texttt{true}$; $S_{new} \leftarrow \{s_0\}$; $S \leftarrow \{\}$
2 **while** `true` **do**
3 **while** *there are v-incompatible states in S_{new}* **do**
4 Select a v-incompatible state (l, C) of S_{new} ;
5 Select a v-incompatible J in $C{\downarrow_P}$;
6 $K_c \leftarrow K_c \wedge \neg J$; $S \leftarrow \bigcup_{j=0}^{i-1} \mathsf{Succ}^j_{\mathcal{A}(K_c)}(\{s_0\})$; $S_{new} \leftarrow \mathsf{Succ}_{\mathcal{A}(K_c)}(S)$
7 **if** $S_{new} \subseteq S$ **then**
8 **return** $K \leftarrow \bigcap_{(l,C)\in S} C{\downarrow_P}$
9 $i \leftarrow i + 1$; $S \leftarrow S \cup S_{new}$; $S_{new} \leftarrow \mathsf{Succ}_{\mathcal{A}(K_c)}(S)$

IM [AS13] is a breadth-first algorithm (given in Algorithm 6), that maintains an integer i (which corresponds to the exploration depth), the current constraint K_c (initially set to `true`, *i.e.* the parameter constraint corresponding to all parameter valuations), the set S_{new} of states computed at the latest iteration, and the set S of states computed at all previous iterations. IM iteratively explore states and refines the constraint K_c: when a v-incompatible state (l, C) is met (line 4), then a v-incompatible inequality is selected within $C{\downarrow_P}$ (line 5),

and added to K_c (line 6). When a fixpoint is reached, *i.e.* when no more new states are generated (line 7), then the intersection of the projection onto P of all reachable states is returned (line 8).

The inverse method can be characterized as follows.

Theorem 1 ([AS13]). *Let $\mathcal{A}$ be a PTA and v be a parameter valuation. Assume $\mathsf{IM}(\mathcal{A}, v)$ terminates with result K. Then*

1. *$v \models K$, and*
2. *for all $v' \models K$, the trace sets of $\mathcal{A}[v]$ and $\mathcal{A}[v']$ are the same.*

A.2 The Behavioral Cartography Algorithm

We recall the original non-distributed behavioral cartography algorithm from [AF10] in Algorithm 7. We extend the $\models$ notation as follows: given a set T of tiles, we write $v \models T$ if there exists some K in T such that $v \models K$.

Algorithm 7: Behavioral Cartography $\mathsf{BC}(\mathcal{A}, D)$

```
   input  : PTA 𝒜, point v
   output : Set of tiles T
1  T ← ∅
2  foreach integer point v ∈ D do
3      if v ⊭ T then T ← T ∪ {IM(𝒜, v)};
4  return T
```

B Master-worker Point Distribution Algorithms

B.1 Worker Algorithm

Algorithm 8: Worker algorithm

```
   input  : PTA 𝒜
1  while true do
2      switch receive() do
3          case m, STOP: Terminate ;
4          case m, POINT(v):
5              K ← IM(𝒜, v)
6              send(m, RESULT(K))
```

B.2 Sequential Point Distribution: Initialization Algorithm

We give the Seq.*initialize*() function in Algorithm 9.

Algorithm 9: Seq.*initialize*()

variables : Point v_{prev}
1 $v_{prev} \leftarrow \perp$

B.3 Random Point Distribution: Initialization Algorithm

We give the Random.*initialize*() function in Algorithm 10.

Algorithm 10: Random.*initialize*()

variables : Point v_{prev}, flag *seqPhase*
1 *seqPhase* ← `false`
2 $v_{prev} \leftarrow \perp$

B.4 Shuffle Point Distribution: Algorithms

Algorithm 11: Shuffle.*initialize*()

variables : List of points *allPoints*
1 *allPoints* ← *shuffle*(*allIntegers*(D))

Algorithm 12: Shuffle.*choosePoint*()

variables : List of points *allPoints*
output : Point v
1 $v \leftarrow pop(allPoints)$
2 **while** v *is covered by a tile* **do**
3 $\quad v \leftarrow pop(allPoints)$
4 **return** v

C Experiments

C.1 Description of the Experimental Testbed

We ran our experiments on two clusters of Grid'5000: Pastel (located in Toulouse, France), and Griffon (located in Nancy, France). Pastel is made of 140 nodes, each of which features two dual-core AMD Opteron 2218 running at 2.6 GHz, 8 GiB of RAM and a GigaEthernet interconnection network. Griffon is made of 92 nodes, each of which features two quad-core Intel Xeon L5420 running at 2.5 GHz, 16 GiB of RAM and both GigaEthernet and 20G InfiniBand network interconnection networks. On these two clusters, the nodes were running a 64-bit Linux 3.2 kernel. The code was compiled using OCaml 4.01 and we used OpenMPI 1.8 with the OCamlMPI bindings.

C.2 Detailed Description of the Case Studies

Flip-flop4 is a 4-parameter dimension asynchronous flip-flop circuit, made of four complex logical gates, and constrained by a predefined environment. Parameters are timing delays in the gate traversal delays, as well as setup and hold values for the input signals in the environment. Depending on the values of the parameters, the system can have a very different behavior.

RCP is a parametric model of the IEEE 1394 root contention protocol, where nodes must elect a leader. The model is inspired by the TREX [ABS01] model from the literature.

Sched3-2, Sched3B-2, Sched3B-3 and Sched5 are parametric schedulability problems, where the goal is to find tiles where the system is robustly schedulable. Sched3-2, Sched3B-2 and Sched3B-3 are the same model, with a different number of parameters (2, 2 and 3 respectively), and a much larger D for Sched3B-2 and Sched3B-3, so as to test the scalability of our algorithms.

Finally, SiMoP is a parametric networked automation system, where several components communicate via a network bus [AS13].

C.3 Raw Experiments

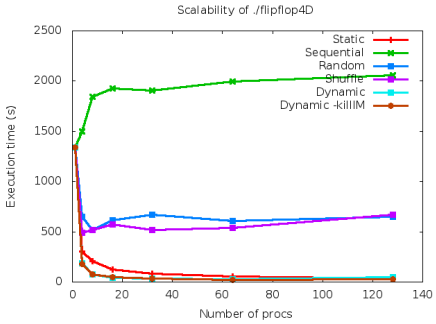
First, recall that Random is parameterized by the maximum number of attempts MAX before switching to a sequential enumeration. In all experiments, we used $MAX = 10$ (larger values did not significantly change the performances).

We give in Figs. 3 and 4 our experiments data under graphical forms. The full data (including all timings and the result of the cartography) are available at `www.imitator.fr/static/ICFEM15/`. In the graphics of Figs. 3 and 4, we give for each case study the execution time and the speedup for 4, 8, 16, 32, 64 and 128 nodes. Obviously, a low execution time is considered as good. The speedup is the execution time for an algorithm and a number of nodes N divided by the time needed for a perfect algorithm (*i.e.* the monolithic time divided by N). The speed-up is used to measure how the code scales, *i.e.* how much faster it runs as the number of nodes used for the computation increases. It is usually between 0

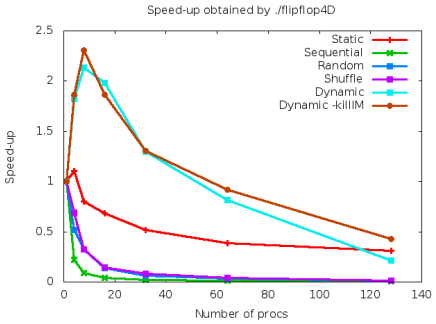
and 1; A high speedup (*i.e.* close to 1) is considered as good, while a value close to 0 denotes an inefficient algorithm (*i.e.* that does not scale).

Finally, we give in Table 3 an extended version of the data in Table 2. In addition to the information of Table 2, we add the following:

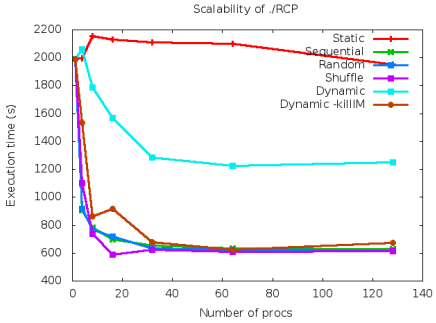
- the ratio at N_{max} w.r.t. the monolithic time, *i.e.* the execution time for an algorithm and a number of nodes N divided by the monolithic time and multiplied by 100 (of course, the smaller the better); note that a ratio greater than 100 means that the distributed algorithm is even slower than the monolithic one (which is the worst possible situation);
- the ratio at N_{max} w.r.t. the slowest distributed algorithm for any N, *i.e.* the execution time for an algorithm and a number of nodes N divided by the slowest distributed algorithm for any number of nodes and multiplied by 100 (again, of course, the smaller the better).



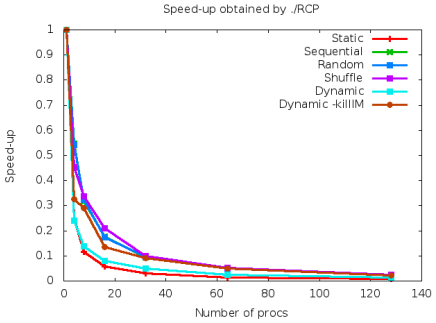
(a) Flip-flop4: execution time



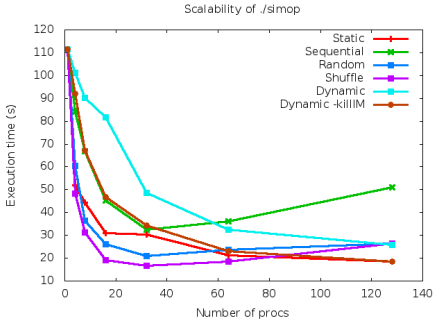
(b) Flip-flop4: speedup



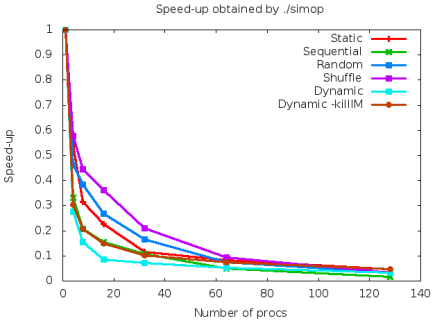
(c) RCP: execution time



(d) RCP: speedup

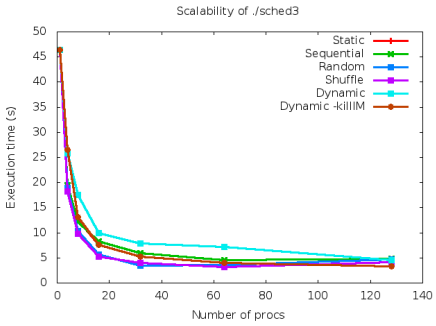


(e) SiMoP: execution time

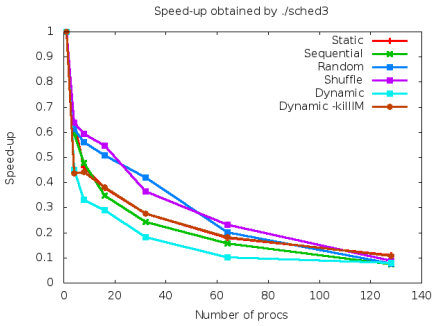


(f) SiMoP: speedup

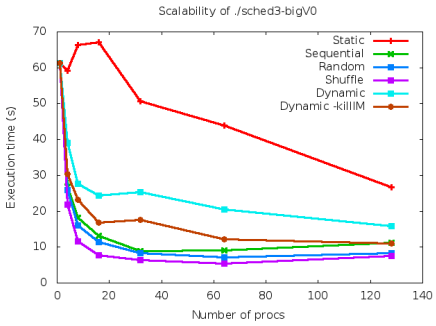
Fig. 3: Experiments: execution time and speedup (1/2)



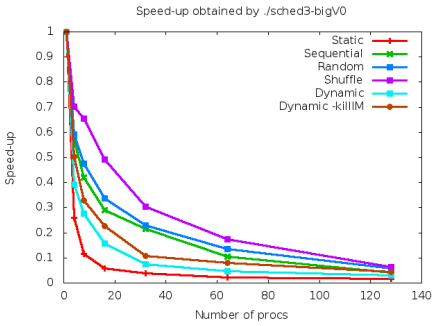
(a) Sched3-2: execution time



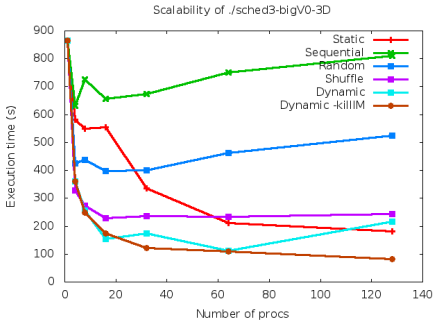
(b) Sched3-2: speedup



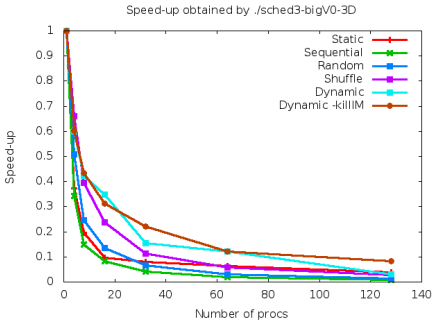
(c) Sched3B-2: execution time



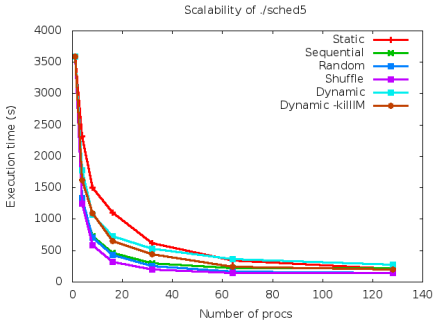
(d) Sched3B-2: speedup



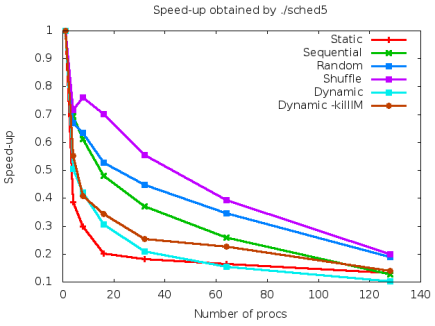
(e) Sched3B-3: execution time



(f) Sched3B-3: speedup



(g) Sched5: execution time



(h) Sched5: speedup

Fig. 4: Experiments: execution time and speedup (2/2)

Case study	Flip-flop4	RCP	Sched3-2	Sched3B-2	Sched3B-3	Sched5	SiMoP	**Average**
Model								
Clocks	5	6	13	13	13	21	8	
Parameters	4	2	2	2	3	2	2	
$\|D\|$	386400	3050	286	14746	530856	1681	10201	
Cartography								
# Tiles	190	19	59	71	378	177	48	
N_{max}	128	32	64	128	128	128	64	
N for speedup	128	19	59	71	128	128	48	
Monolithic	1341.0	1992.0	46.0	61.2	865.0	3593.0	111.6	
Execution time at N_{max}								
Static	33.0	2108.0	4.0	26.6	181.0	213.0	21.4	
Seq	2059.0	653.0	4.6	11.0	810.0	219.0	36.1	
Random	652.0	635.0	3.6	8.4	524.0	148.0	23.6	
Shuffle	670.0	624.0	3.1	7.6	243.0	140.0	18.7	
Subdomain	48.0	1286.0	7.2	15.8	217.0	273.0	32.4	
Subdomain + H	24.0	622.0	4.0	11.0	81.0	199.0	23.2	
Hybrid	24.0	624.0	3.1	7.6	81.0	140.0	18.7	
Ratio at N_{max} w.r.t. monolithic								
Static	2	106	9	43	21	6	19	29
Seq	154	33	10	18	94	6	32	49
Random	49	32	8	14	61	4	21	27
Shuffle	50	31	7	12	28	4	17	21
Subdomain	4	65	16	26	25	8	29	24
Subdomain + H	2	31	9	18	9	6	21	14
Hybrid	2	31	7	12	9	4	17	12
Ratio at N_{max} w.r.t. slowest distr								
Static	2	100	15	40	22	6	21	29
Seq	100	31	17	16	100	6	36	44
Random	32	30	14	13	65	4	23	26
Shuffle	33	30	12	11	30	4	18	20
Subdomain	2	61	27	24	27	8	32	26
Subdomain + H	1	30	15	16	10	6	23	14
Hybrid	1	30	12	11	10	4	18	12
Ratio at N_{max} w.r.t. slowest at N_{max}								
Static	2	100	56	100	22	78	59	60
Seq	100	31	64	41	100	80	100	74
Random	32	30	50	32	65	54	65	47
Shuffle	33	30	43	29	30	51	52	38
Subdomain	2	61	100	59	27	100	90	63
Subdomain + H	1	30	56	41	10	73	64	39
Hybrid	1	30	43	29	10	51	52	31
Speedup at N_{max}								
Static	32	5	19	3	4	13	11	12
Seq	1	16	17	8	1	13	6	9
Random	2	17	22	10	1	19	10	11
Shuffle	2	17	25	11	3	20	12	13
Subdomain	22	8	11	5	3	10	7	10
Subdomain + H	44	17	19	8	8	14	10	17
Hybrid	44	17	25	11	8	20	12	20

Table 3: Complete summary of experiments