Journal of Big Data

RESEARCH

Open Access

Efficient parallel derivation of short distinguishing sequences for nondeterministic finite state machines using MapReduce

Bilal Elghadyry[1,2*], Faissal Ouardi[2], Zineb Lotfi[2] and Sébastien Verel[1]

*Correspondence:
bilal.el-ghadyry@univ-littoral.fr
[2] ANISSE research Team, Department of Computer Science, Faculty of Sciences, Mohammed V University in Rabat, Rabat B.P. 1014, Morocco
Full list of author information is available at the end of the article

Abstract

Distinguishing sequences are widely used in finite state machine-based conformance testing to solve the state identification problem. In this paper, we address the scalability issue encountered while deriving distinguishing sequences from complete observable nondeterministic finite state machines by introducing a massively parallel MapReduce version of the well-known Exact Algorithm. To the best of our knowledge, this is the first study to tackle this task using the MapReduce approach. First, we give a concise overview of the well-known Exact Algorithm for deriving distinguishing sequences from nondeterministic finite state machines. Second, we propose a parallel algorithm for this problem using the MapReduce approach and analyze its communication cost using Afrati et al. model. Furthermore, we conduct a variety of intensive and comparative experiments on a wide range of finite state machine classes to demonstrate that our proposed solution is efficient and scalable.

Keywords: Conformance test, Finite state machines, Parallel algorithm, MapReduce framework

Introduction

Recent large-scale software systems, which contain thousands or even millions of interacting components of hardware and software, have come to perform more and more critical roles in many of society's most important infrastructures, including those used to support banking, health care, air traffic control, telephony, and many other sectors. Software systems are very large in scale: they consist of hundreds or thousands of computers and millions of lines of code and they perform complex tasks almost continuously. They raise a host of technical and nontechnical issues which can become critical and others of which arose recently as a result of the increases in scale and the degree of interconnection of software systems. Since software testing is an essential step in the software development process, we have to use efficient approaches to reduce both the cost and the time of testing.

Finite state machine and conformance testing

Due to their simplicity and ability to model complex systems, Finite State Machines (FSMs) are extensively used in several fields such as communication protocols [1], pattern matching [2], digital event reconstruction [3], smart contract [4], distributed testing [5, 6], genomics [7], and other reactive systems [8]. An FSM is a model which has a finite number of states, inputs, outputs, and a finite number of transitions each labeled by an input/output pair. Besides that, FSMs are the underlying models for formal description techniques, such as statecharts, Specification and Description Language (SDL) [9], Unified Modeling Language (UML) [10], programmable logic devices [11], and ethereum smart contracts [4].

Testing FSM is an indispensable part of system design and implementation to guarantee the right functioning of the modeled systems and find aspects of their behavior due to their simplicity and ability to model systems [12]. This is well studied in the FSM-based testing research area. However, we are basically missing some information about the black-box FSM Implementation Under Test (IUT), and, as a consequence, we need to recoup this information by trying experiments on this IUT [13]. The purpose of these experiments is to check whether the implementation of a model behaves in accordance with its specification, by applying checking sequences (CSs) to the IUT, observing the corresponding output responses, and drawing a conclusion about the IUT [14]. In other words, one needs to recognize the state of the IUT and bring the IUT to a particular state. The state recognition can be accomplished by using CSs like distinguishing sequences [15], Unique Input Output sequences [16, 17], Characterizing Sets (W-Set) [18], or synchronizing sequences (also known as reset sequences) [19], when such sequences exist. The motivation to study such sequences comes from different fields including robotics, bio-computing, propositional calculus, model-based testing, distributed testing, and many more [15, 19–28]. The literature contains many techniques that automatically generate CSs [21, 22, 29–34]. Most approaches consist, in principle, of three parts: initialization, state identification, and transition verification.

In this paper, we focus on the scalability problem of generating distinguishing sequences (DSs) from an FSM to resolve the state identification problem, which consists in finding an input sequence that produces different outputs for each initial state of an FSM. This problem was initially described in the seminal paper by Moore [35] in 1956, and in 1964 Hennie [36] provided the first FSM-based test generation algorithm that can be automated. One motivation is that many FSM-based test sequence generation techniques use DSs (see, for example, [30, 36–40]). It has been found that distinguishing sequences (DSs), where they exist, lead to shorter tests [13]. There are two well-known DSs, adaptive (ADS) and preset (PDS) [41] for different FSM classes (deterministic [13], nondeterministic [42], complete [43], partial [44], observable [45]). PDS is a single fixed input sequence that can be used to distinguish each state of the machine [46]. In ADS, the next input depends on the output of the current input. It is a rooted tree where each root to leaf path represents an input sequence specific to the state represented by the leaf. Throughout the paper, we refer to PDS when we write DS. To derive a shortest DS of a state's pair, Spitsyna et al. have designed the well-known Exact Algorithm (EA) [47]. It's based principally on two steps: For an FSM M, construct a truncated successor tree from the intersection of two initialized FSMs M/s_1 (*i.e.*, FSM M with the initial state s_1)

M/s_2 (*i.e.*, FSM M with the initial state s_2) and then derive a shortest DS from it. They suggested a method to analyze the separability relation between FSMs that can be used for deriving a shortest DS (if exists) of two given FSMs (or for two states of a given FSM) and show that for two states of an FSM, its upper bound becomes exponential.

When nondeterministic FSMs are considered, Alur *et al.* [48] have shown that the length of a DS for all states can reach the exponential bound. In addition, the complexity to decide if there exists a PDS is PSPACE-complete, and it is EXPTIME-complete to decide if there is an ADS.

To the best of our knowledge, the parallelization and scalability of deriving DSs from nondeterministic FSMs have not been thoroughly addressed using MapReduce framework. In this work, we focus on the design of an optimized MapReduce version of the Exact Algorithm, and experiments to prove its scalability.

Outline and contributions

Our study makes the following contributions:

- We present the first parallel MapReduce algorithm to efficiently derive a set of short distinguishing sequences for all pairs of states of a nondeterministic FSM.
- We provide a theoretical analysis of the communication cost of proposed methods in MapReduce model through a grounded theory.
- We evaluate the performance of the proposed algorithms through extensive experiments using a variety of large-scale FSMs datasets.

The remainder of the paper is structured as follows. Section 2 includes the necessary technical definitions and a brief introduction to the MapReduce computational model. Section 3 presents the related works as well as a survey of the Exact Algorithm for deriving DSs from nondeterministic FSMs. Section 4 presents and analyses the proposed MapReduce version of the Exact Algorithm to derive a set of shortest DSs for all pairs of states. Section 5 shows the efficiency and the scalability of the proposed methods by conducting extensive and comparative experiments on a variety of classes of nondeterministic FSMs, whereas section 6 covers the conclusion of the paper.

Preliminaries

In this section, we present basic concepts that are used throughout this paper and MapReduce framework in brief.

Finite state machine and distinguishing sequences

A *finite state machine* (FSM) is a 4-tuple $M = (S, I, O, E)$, where S is a finite set of states, I is a finite set of input symbols, called input alphabet, O is a finite set of output symbols, called output alphabet, and $E \subseteq S \times I \times O \times S$ is the set of transitions. Given a transition $t = (s, i, o, s') \in E$, we denote by $s[t]$ its origin or start state s, $d[t]$ its destination state or next state s', $I[t]$ its input symbol i, and $O[t]$ its output symbol o. Let s be a state in S, we denote by E_s the transitions subset having s as a start state *i.e.* for $t \in E_s$, $s[t] = s$. A FSM is nondeterministic if for a state $s \in S$ there exists $t, t' \in E$ such that $s[t] = s[t']$

and $I[t] = I[t']$.An FSM is called *complete* if for each pair $(s, i) \in S \times I$ there exists $(o, s') \in O \times S$ such that $(s, i, o, s') \in E$. Otherwise, the machine is called *partial.*

A complete nondeterministic FSM is *observable* if at each state, the machine has at most one transition under a given input/output pair. Otherwise, the machine is *non observable.*

An FSM is called *connected* if for each $s \in S$ there exists an input sequence that takes the FSM S from an arbitrary state to state s.Given an FSM $M = (S, I, O, E)$, a state s and an input i, the *i-successor* of state s contains each state s' for which there exists an output symbol $o \in O$ such that $(s, i, o, s') \in E$. Given a subset of states $S' \subseteq S$ and an input i, the set of states S' is the *i-successor* of S if S' is the union of the *i-successor* over all states of the set S.

Example 1

Let us consider the FSM M defined over the sets of inputs $I = \{a, b\}$, *outputs* $O = \{0, 1\}$, *and states* $S = \{s_0, s_1, s_2, s_3, s_4\}$, *schematized in Fig. 1. The a-successor of state* s_0 *is the set of states* $\{s_1, s_2\}$, *the a-successor of state* s_1 *is the set of states* $\{s_3, s_4\}$. *Thus, the a-successor of the set of states* $\{s_0, s_1\}$ *is the set* $\{s_1, s_2\} \cup \{s_3, s_4\} = \{s_1, s_2, s_3, s_4\}$.

We are interested in DS, which is an input sequence that produces different output sequences when starting from different states of an FSM. As mentioned previously, there exists two types of DSs: An input sequence x is considered as a *preset distinguishing sequence* (PDS) for FSM M if x is defined as an input sequence for states S and for any pair of distinct states $(s_1, s_2) \in S \times S$, x is a distinguishing sequence for s_1 and s_2. In the other side, an *Adaptive Distinguishing Sequence* (ADS) is a rooted tree T with exactly n leaves; the internal nodes are labeled with input symbols, the edges are labeled with output symbols, and the leaves are uniquely labeled with states of the FSM such that: a) edges descending from a common node have distinct output symbols, and b) for each leaf of T, if x and y are the input and output sequences respectively formed by the node and edge labels on the path from the root to the leaf labeled by some state s_i of the FSM then $(s_i, x, y_i, s') \in E$. Then, the input sequence x is considered as an adaptive distinguishing sequence of the state s_i. The length of such sequence is the depth of the tree T [12].

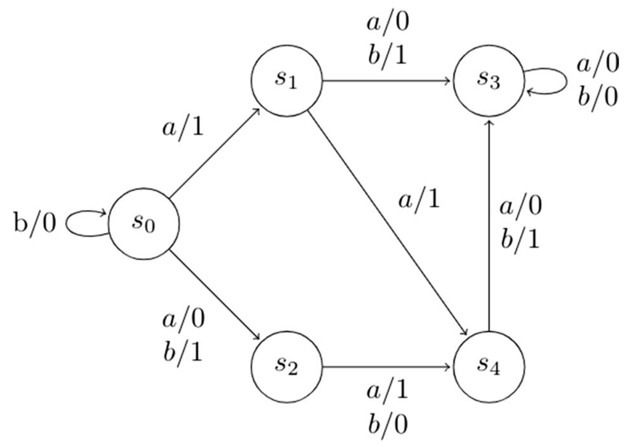


Fig. 1 Finite State Machine *M*

MapReduce model of computation

Works based on MapReduce were recently introduced as an optimal parallel model to compute the intersection [49] and the composition [50] operations of FSMs. In this work, we implement the first MapReduce version of the EA to derive a shortest DS for each pair of states if exists from a large-scale complete observable nondeterministic FSM.

MapReduce [51] is considered as one of the most prominent programming models for processing scalable problems. Nowadays, Hadoop [52] offers the most popular open-source framework written in Java for implementing MapReduce algorithms. Authored by Apache Software Foundation, the Hadoop project includes modules enabling reliable and scalable distributed computing. It presents several advantages, such as scalability, flexibility, cost-effectiveness, organized architecture, and resilience to failure. Recently, several Hadoop-based platforms have been proposed as efficient and flexible solutions for computational storage with the strategy of processing data close to where they reside [53–55]. Among them, we cite the lineage-aware data management (LDM) that exploits the data locality to decrease the network footprint [56, 57]. MapReduce algorithm consists of three major phases: Map, Shuffle and Reduce. Each phase runs several tasks in a completely parallel manner. The Map phase is responsible for filtering and transforming input records into intermediate records, the Shuffle phase occurs automatically, it is done by Hadoop to manage the exchange of the intermediate data from the map phase to the reduce phase, while the Reduce phase is in charge of summarizing the outputs of the previous phase.

When designing a parallel MapReduce algorithm, it is essential to propose an optimal one that offers the best trade-off between parallelism and communication cost in a MapReduce computation. To do this, we analyze the communication and the computation costs of the proposed methods using Afrati et al. theoretical model [58].

Related work

Parallel approaches for deriving distinguishing sequences

A variety of studies have recently focused on the use of parallel processing techniques in order to derive CSs in a large-scale context: UIO sequences [17], harmonized state identifiers, and characterizing sets [18], synchronizing sequences [59, 60]. For DSs generation, Hierons and Türker [61] and El-Fakih et al. [62] introduced independently parallel multi-threading implementations of the EA over the Central Processing Unit (CPU) and Graphics Processing Unit (GPU) architectures [63]. They conducted extensive experiments when considering a large variety of FSM classes, using different CPU-GPU architectures and workloads. The obtained results show that their approaches are sufficiently efficient in large-scale data, and the execution time for deriving DSs from nondeterministic FSMs increases exponentially *w.r.t.* different parameters such as: the degree of nondeterminism, the number of transitions, and the input alphabet size to the output alphabet size ratio.

In order to reduce the execution time of constructing successors' table of all pairs of states for a given nondeterministic FSM, El-fakih et al. [62] proposed different multi-threading parallel approaches based on multicore CPU and GPU architecture. They considered two options: Thrust software platforms and GPU implementations using the CUDA platform. They also proposed and evaluated a Network of Workstations solution (NoWs) based on Divisible Load Theory. They conducted their experiments on the class of nondeterministic FSMs with a large number of input and output symbols.

These experiments bring out the difference between the proposed algorithms in terms of speedup and execution time.

In [61], Hierons and Türker considered the partial observable nondeterministic FSMs and studied the scalability issue while constructing preset and adaptive distinguishing sequences (PDS and ADS) for all states. They proposed an ADS generation algorithm that can process inputs up to 2048 times better than the existing ADS construction algorithm and a PDS generation algorithm that can process inputs up to 8 times better than the existing PDS generation algorithm. Their approach is based on the available parallelism in a GPU computing model, called *the thin thread strategy*. it utilizes global device memory and so maximizes the number of threads in order to maximize parallelism. The results of their experiments are good and indicate that the proposed algorithm can derive DSs from observable partial nondeterministic FSMs with 32000 states in an acceptable amount of time.

Overview of the exact algorithm

In this section, we present a concise overview of the well-known Exact Algorithm [47] that derives shortest distinguishing sequence for a pair of states, if it exists, of a complete observable nondeterministic FSM.

From an FSM M, we will consider two FSMs with different initial state M/s_1 and M/s_2. The EA will be applied to a single FSM M to derive a DS of two states s_1 and s_2 and we note that the state pair order doesn't make a distinction *i.e.* $(s_0, s_1) = (s_1, s_0)$. In order to derive a shortest distinguishing sequence (when it exists), EA is implemented using the Breadth-First Search (BFS) method that explores the search successor tree level by level.

Algorithm 1: Exact Algorithm (EA)

```
input  : Two different states $s_k$ and $s_l$ of a complete observable nondeterministic
         $M = (S, I, O, E)$.
output: A shortest DS of states $s_k$ and $s_l$ (if it exists) or the message that states $s_k$ and $s_l$
         are non-separable.
1  Intersection step
   /* Derive the successor tree Tree from $M/s_k \cap M/s_l$                          */
2  foreach transition $(t, t') \in E \times E$ do
3      if $I[t] = I[t'] \&\& O[t] = O[t']$ then
4          insert(Tree, $((s[t], s[t']), I[t], (d[t], d[t']))$)
5  if Tree is a complete FSM then
6      The states $s_k$ and $s_l$ are non-separable.
7  else
8      derive a truncated successor tree of $M/s_k \cap M/s_l$.
9  Derivation step
   /* An intermediate node of the $j^{th}$ level labeled with a subset P of states of the
   intersection                                                                     */
   /* A current node Current, at the $p^{th}$ level, labeled with the subset P of state
   pairs                                                                            */
10 if there exists an input i such that each pair of the set P has no i-successors
11 or there exists a node at a $j^{th}$ level, $j < p$, labeled with subset R of states such that $P \supseteq R$,
12 or for some pair $(s, t)$ of the set P and some output o, the I/O sequence io takes the FSM
     from states s and t to the same state then
13     The current node is claimed as a leaf node
14 if none of the paths of the truncated tree derived is terminated then
15     return $s_k$ and $s_l$ are non separable.
16 else
17     a shortest sequence $\alpha i$ where $\alpha$ labels the path from the root of the tree to a leaf, is a
        shortest distinguishing sequence of $s_k$ and $s_l$, return $\alpha i$.
```

The EA is divided into two major steps: the intersection step and the derivation step. In the first step, we compute the intersection $M/s_1 \cap M/s_2$. Then, if this intersection is a partial (non-complete) FSM, we derive a truncated successor tree *Tree*. Otherwise, we return the message that the two states are non-distinguishable. In the second step, we derive from the truncated successor tree *Tree* a short DS of the state pair of the root node, if it exists, using BFS method. The root of this tree, which is at the 0th level, is the initial state (s_k, s_l) of the intersection; the nodes of the tree are labeled with subsets of states of the intersection. Given already derived j tree levels, $j \geq 0$, an internal node of the jth level labeled with a subset P of states of the intersection, and an input i, there is an outgoing edge from this internal node labeled with i to the node labeled with the subset of the i-successors of pairs of states of the subset P. A current node *Current*, at the pth level, $p \geq 0$, labeled with the subset P of state pairs, is claimed as a *leafnode* if one of the conditions in line 10, 11 or 12 of Algorithm 1 holds. Next, if no leaf node exists following the condition of line 10, then the states pair of the root node are non-separable. Otherwise, if there is a leaf node labeled with the subset P of states such that for some input i, each state of the set P has no i-successors, then derive a shortest DS αi where α labels the path from the root node of the tree to the leaf node.

The number of different possible subsets of pairs of states in an FSM having n states is $2^{\frac{n^2}{2}}$ (the number of possible subsets P in Algorithm 1). Hence the worst-case time complexity of this step of EA is in $O(2^{\frac{n^2}{2}})$. In the second step of EA, derivation of a DS, if it exists, from the previously derived truncated successor tree is performed using the classical Breadth-First Search method as recalled in Algorithm 1. Then, the worst-case time complexity of the EA is shown to be in $O(2^{\frac{n^2}{2}} \times \frac{n^2}{2} \times |I| \times |O|)$.

When considering two FSMs having respectively n and m states, it's shown in [47] that the length of the shortest DS is at most 2^{mn-1} and this upper bound can be reached. As a consequence, the upper bound for a single FSM becomes 2^{n^2-1}.

However, according to the conducted experiments in [47] there exists a large class of FSMs with n and m states such that the length of the shortest DS is less than mn and less than n^2 when considering one FSM. The experiments also show that the existence of a DS of two FSMs significantly depends on the degree of nondeterminism in the FSMs [47].

Example 2

Let us consider the FSM M from Example 1. and apply the EA to derive a DS between the state s_0 and the state s_1. Figure 2 shows the successor tree derived from step 1 of the EA. The nodes of this successor tree are labeled from n_1 to n_7.

As the intersection of M/s_0 and M/s_1 is not a complete FSM, the root node n_1 associated with the state pair (s_0, s_1) is at the level $j = 0$. The successors of n_1 are nodes n_2 and n_3. Then, for the level $j = 1$, we obtain the state pairs in nodes n_4 and n_5 as successors of n_2, and the successors of n_3 are the state pairs in nodes n_6 and n_7. The node n_4 has a pair with repeated state (s_3, s_3), so, we do not consider n_4 for further exploration. Node n_6 is labeled with the empty set, *i.e.*, there are no successors for n_3 under any input symbol.

Therefore, the input sequence "*ba*" which starts from the root and leads to the node labeled with the empty set is a short DS for the FSMs M/s_0 and M/s_1.

Efficient MapReduce algorithm for deriving short distinguishing sequences from FSMs

In this section, we present and analyze our parallel version of the EA using MapReduce framework to extract a set of short distinguishing sequences from complete observable nondeterministic FSM. Our method outperforms the previous parallel approaches in the sense that it efficiently provides a short distinguishing sequence for each pair of states of a large FSM.

Framework overview

The proposed solution consists of two MapReduce steps namely: the intersection step and the derivation of short distinguishing sequences step or the derivation step for short.

Figure 3 illustrates the workflow of our solution, which receives a large FSM as input and produces a set of short distinguishing sequences for each pair of states using two MapReduce algorithms. Initially, an input FSM $M = (S, I, O, E)$ is pre-processed to produce a text file where every line (*value*) represents a transition $t = (s[t], I[t], O[t], d[t])$. This file is the input of the intersection step. MapReduce framework is composed essentially of a map function which performs filtering and sorting, and a reduce function which performs a summary operation. The map function produces a set $\langle key, value\rangle$ pairs. In our case, for a transition t, it generates a set of associated keys. Following that, the set of pairs produced by map function are grouped and sent to reduce function. This last receives transitions having the same key and computes their intersection *i.e.* for two transitions t_i and t_j having the same *input*/*output*, the result will be an edge in the so-called truncated successor tree $(\{(s[t_i], s[t_j])\}, I[t_i], \{(d[t_i], d[t_j])\})$, else $(\{(s[t_i], s[t_j])\}, I[t_i], \{\})$. In the second step, we developed an iterative MapReduce algorithm to derive a short distinguishing sequence

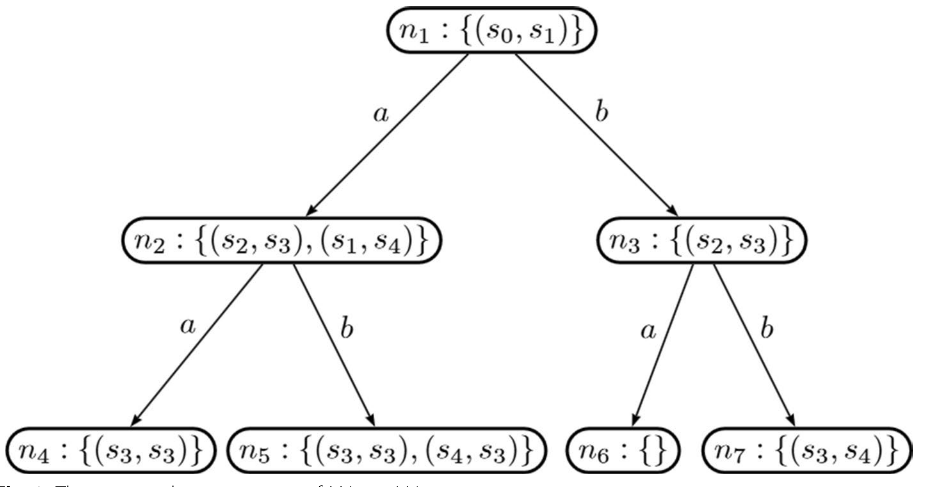


Fig. 2 The truncated successor tree of $M/s_0 \cap M/s_1$

if exists. The input of this step is the output of the previous intersection step. The map function of this step takes an edge of the truncated successor tree (n_s, l, n_d) where n_s denotes the source node, l is the label, and n_d is the destination node and produces a set of associated keys. For a given edge, if its destination node is empty, the associated key will be its source node, else each states' pair in its destination node becomes a new associated key. Next, the reduce function receives the set of edges having the same key and divides the set of associated edges into two subsets, the first one contains all edges having an empty destination node, the other one contains the rest of the edges. Then, the cartesian product of these two subsets will be performed *i.e.* for two edges e and e', if the source node of e' is a subset of the set of pairs in the destination node e, then the resulting edge will be $(n_s(e), l(e)l(e'), n_d(e) \setminus n_s(e'))$. This process is equivalent to a BFS in the truncated successor tree of the EA, when we concatenate edges label to construct DSs if exist. Figure 4 shows the derivation step of the successor tree presented in Fig. 2. The first round of the derivation step performs the successor tree received from the intersection step and maps its edges to different reducers following the previously described mapping schema. The first round of the derivation step performs the successor tree received from the intersection step and maps its edges to different reducers following some mapping schema. For example, from Fig. 4, the pair (s_2, s_3) in the node n_3 maps all edges having the forms $((s_2, s_3), *, \perp)$, $((s_2, s_3), *, \{\})$ or $(*, *, (s_2, s_3))$ to a given reducer. Then, the cartesian product between edges having the form $(*, *, (s_2, s_3))$ and edges having the form $((s_2, s_3), *, \{\})$ or $((s_2, s_3), *, \perp)$ is computed inside this reducer to produce compact edges having the from $(*, **, \{\})$ or $(*, **, \perp)$. So, the reducer mapped by (s_2, s_3) produces the edge $((s_0, s_1), ba, \{\})$ from two edges $((s_0, s_1), b, (s_2, s_3))$ and $((s_2, s_3), a, \{\})$. Then, this edge will be mapped, in the second round, to the reducer associated with the key (s_0, s_1). The same process will be repeated iteratively in the next MapReduce rounds of the derivation step until the stop condition is true.

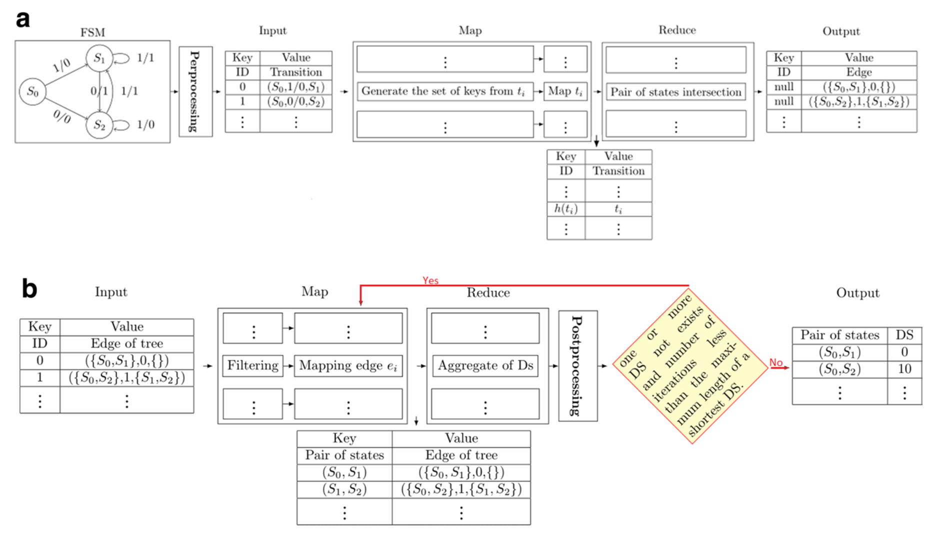

a
FSM
S_0
S_1
S_2
1/0
0/0
1/1
0/1
1/1
1/0
Perprocessing
Input
Key | Value
ID | Transition
0 | $(S_0,1/0,S_1)$
1 | $(S_0,0/0,S_2)$
Map
Generate the set of keys from t_i
Map t_i
Key | Value
ID | Transition
$h(t_i)$ | t_i
Reduce
Pair of states intersection
Output
Key | Value
ID | Edge
null | $(\{S_0,S_1\},0,\{\})$
null | $(\{S_0,S_2\},1,\{S_1,S_2\})$
b
Input
Key | Value
ID | Edge of tree
0 | $(\{S_0,S_1\},0,\{\})$
1 | $(\{S_0,S_2\},1,\{S_1,S_2\})$
Map
Filtering
Mapping edge e_i
Key | Value
Pair of states | Edge of tree
(S_0,S_1) | $(\{S_0,S_1\},0,\{\})$
(S_1,S_2) | $(\{S_0,S_2\},1,\{S_1,S_2\})$
Reduce
Aggregate of Ds
Postprocessing
one or more DS not exists and number of iterations less than the maximum length of a shortest DS.
Yes
No
Output
Pair of states | DS
(S_0,S_1) | 0
(S_0,S_2) | 10

Fig. 3 An overview of the parallel Exact Algorithm using MapReduce

Finally, if a DS not exists and the number of iterations is less than the maximum bound, we repeat the derivation step by considering the output as an input of the next iteration, else the final output is the set of pairs and their short DSs if they exist.

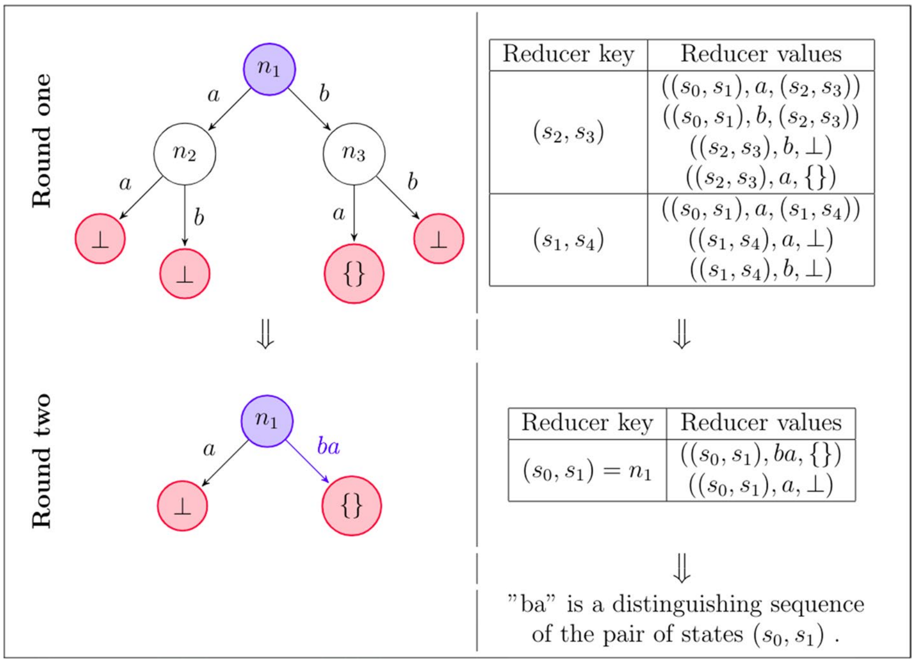
In the next section, we present the communication cost in a MapReduce framework of this problem using Afrati et al. model [58].

Communication cost analysis

The communication cost model introduced by Afrati et al. [58] gives a good way to analyze problems and optimizes the performance of any distributed computing environment by explicitly studying an inherent trade-off between communication cost and parallelism degree. By applying this model in a MapReduce framework, we can determine the best algorithm for a problem by analyzing the trade-off between reducer size and communication cost in a single round of MapReduce computation. There are two parameters that represent the trade-off involved in designing a good MapReduce algorithm: the first one is the reducer size, denoted by q, which represents the size of the largest list of values associated with a key that a reducer can receive. The second parameter is the amount of communication between the map step and the reduce step. The communication cost, denoted by r, is defined as the average number of key-value pairs that the mappers create from each input.

Formally, suppose that we have p reducers and $q_i \leq q$ inputs are assigned to the i^{th} reducer. Let $|In|$ be the total number of different inputs, then the replication rate is given by the expression $r = \sum_{i=1}^{p} q_i/|In|$ [58].

From [58], we compute a lower bound on the replication rate for the intersection of FSMs as a function of q using the following expression:



Reducer key	Reducer values
(s_2, s_3)	$((s_0, s_1), a, (s_2, s_3))$ $((s_0, s_1), b, (s_2, s_3))$ $((s_2, s_3), b, \perp)$ $((s_2, s_3), a, \{\})$
(s_1, s_4)	$((s_0, s_1), a, (s_1, s_4))$ $((s_1, s_4), a, \perp)$ $((s_1, s_4), b, \perp)$

Reducer key	Reducer values
$(s_0, s_1) = n_1$	$((s_0, s_1), ba, \{\})$ $((s_0, s_1), a, \perp)$

Fig. 4 The derivation step of $M/s_0 \cap M/s_1$

$$r \geq \frac{q \times |Out|}{g(q) \times |In|}$$

where $|In|$ denotes the input size, $|Out|$ denotes the output size, and $g(q)$ the number of outputs that can be produced by a reducer of size q. Since, we consider a complete observable nondeterministic FSM, we have $|In| = |E|$ and $|Out| = \frac{n(n-1)}{2} \times |I|$. Thus

Proposition 1 *The lower bound on the replication rate is*

$$r \geq \frac{n(n-1)}{2} \times \frac{q \times |I|}{g(q) \times |E|}$$

It is worth noting that limiting the reducer size enables more parallelism. Small reducers' size forces us to redefine the notion of a key in order to allow more, smaller reducers, and thus allow more parallelism using the available nodes.

MapReduce algorithm for the intersection step

Let us present the MapReduce implementation of the intersection step using a modified version of the algorithms proposed in [49]. Notice that our approach produces a truncated successor tree, also called successor table, for all pairs of states of a complete observable nondeterministic FSM. The conducted experiments in [62] show that when deriving distinguishing sequences, the construction time of the successor tree takes 96% of the whole EA's time. That is why three methods will be presented later in this section for the construction of the truncated successor tree.

The Algorithm 2 below contains the definitions of the map and reduce functions of the intersection step. The map function produces a set of keys based on a defined schema from the input FSM transitions. The reduce function performs, inside reducers, the intersection of the received transitions from the mapper tasks.

Algorithm 2: Intersection step in MapReduce

```
input  : $M = (S, I, O, E)$, a complete observable nondeterministic FSM.
output: Tree, a Truncated successor tree of all pairwise states.
1  Function Map($key$, $values$):
       Data: $\langle key, value \rangle$ pair, where $key$ represents an arbitrary instance identifier and value is
             a transition $t \in E$.
       Result: Collection of $\langle k, t \rangle$ pairs, where $k$ is a key associated with the transition $t$.
       /* create the transition t from the input value */
2      $t$ = getTransitionFrom($value$)
       /* generate the set of keys associated with the transition t */
3      $keys$ = getKeysFromTransition($t$)
4      foreach $k$ in $keys$ do
5          Emit ($k$,$t$)

6  Function Reduce($key$, $values$):
       Data: $\langle key, values \rangle$ pair, where values is a set of transitions having the same key $key$.
       Result: Derived part of truncated successor tree.
       /* Compute the intersection of the set of transitions in values and add it to
          the truncated successor tree Tree */
7      foreach $(t, t') \in values \times values$ do
8          if $I[t] = I[t']$ and $O[t] = O[t']$ then
9              add the edge $((s[t], s[t']), I[t], (d[t], d[t']))$ to Tree
```

The three proposed mapping schema emit a transition to a set of reducers w.r.t. a key defined from some hash functions. Our mapping methods are based respectively: on states, input alphabet symbols, and both states and input alphabet symbols.

Formally, lets $M = (S, I, O, E)$ be a complete observable nondeterministic FSM having n states and $t = (s, i, o, d)$ be a transition in E. A mapper produces a set of keys from the transition t based on some hash function h. This hash function is integrated in the definition of these keys as a part of the sub-function `getKeysFromTransition()` in Line 3 of Algorithm 2. Let us explain in more details the three mapping methods by designing the sub-function `getKeysFromTransition()`.

Mapping based on states

In the first mapping method, from a transition $t \in E$, the mappers produce a set of key-value pairs having the form $\langle key, t\rangle$, where $key = \langle h_S(s[t]), s\rangle$, for all $s \in S$ such that $s[t] \neq s$ and h_S be a hash function defined from S to $\{1, \cdots, n\}$. In this case, we have $\frac{n(n-1)}{2}$ reducers.

In this method, the function $g(q)$, which is the number of outputs that can be produced by a reducer of size q, can be affected by the presence of transitions with different alphabet symbols inside the same reducer. Formally, since we consider a complete observable nondeterministic FSM, one has $q \leq 2 \times (|I| + |I| \times |O|)$ and $g(q) = |I|$. Thus, the following proposition gives the upper bound on the replication rate for this method.

Proposition 2 *The replication rate r in the state-based mapping scheme is* $r \leq (n-1)$.

Mapping based on input alphabets

In the second method, we have one reducer for each of the input alphabets. Thus, the number of reducers is equal to the input alphabet size $|I|$. The mappers will send each transition t to the reducer corresponding to its input symbol $I[t]$. More precisely, from a transition $t \in E$, the mappers produce a set of key-value pairs having the form $\langle key, t\rangle$, where $key = h_{In}(I[t])$ such that h_{In} be a hash function defined from I to $\{1, \cdots, |I|\}$. We will now have $g(q) = \frac{n(n-1)}{2}$, where $q \leq n + n \times |O|$. Assuming that the alphabet symbols are uniformly distributed, we have

Proposition 3 *The replication rate in the input alphabets based mapping scheme is optimal and equal to* 1.

Mapping based on both states and input alphabets

In the last method, we propose a hybrid mapping between first and second method. In other words, keys will be based on the states and input alphabets in the same time. Then, we consider the key form $key = (s[t], s, I[t])$, where $s \in S$ such that $s \neq s[t]$. The number of reducers, in this case, is equal to $\frac{n(n-1)}{2} \times |I|$, the reducer size $q \leq 2 \times |O|$, and each reducer will produce no more than one edge of the truncated successor tree. Thus, we can deduce an upper bound of the replication rate in the following proposition.

Proposition 4 *The replication rate in the hybrid mapping method is* $r \leq (n-1)$.

Theorem 1 *Algorithm* 2 *correctly computes the successor tree for all pairs of states of an FSM* $M = (S, I, O, E)$ *in single MapReduce round using at least* $\frac{n(n-1)}{2} \times \frac{q \times |I|}{g(q) \times |E|}$ *communication, where* $n = |S|$.

Proof

In the map function of Algorithm 2, `getTransitionFrom(t)` *returns the set of keys associated with the transition t w.r.t. a given mapping method. Then, it sends the transition t to all reducers indexed by these keys. In order to ensure that the algorithm correctly constructs the successor tree, it is necessary to have the property (*): all the transitions with the same input and output symbols inside the same reducer. Then, the reducer function computes their pairwise intersection to extend the successor tree. Using the proposed mapping methods, we have:*

- for the mapping based on states, a reducer $\langle s_i, s_j \rangle$ receives from the mappers all the transitions starting from the state s_i or s_j; as a consequence, all outgoing transitions from state s_i or state s_j are inside this reducer. Then the property (*) is verified using this mapping method.
- for the mapping based on input alphabets, a reducer receives from the mappers transitions having the same input symbol c; Then inside a reducer, we have also all transitions having the same input and output symbols.
- for the hybrid mapping, a reducer $\langle s_i, s_j, c \rangle$ receives from the mappers transitions having the same input symbol c and starting from the state s_i or s_j; then the property (*) is obviously verified.

The Proof for the communication complexity follows from Proposition 1. $\square$

MapReduce algorithm for the derivation step

In this step, multiple MapReduce rounds are used to derive a set of shortest distinguishing sequences for each pair of states of an observable nondeterministic FSM. In each round, the mappers run in parallel and produce a collection of pairs $\langle key, edge \rangle$, while the reducers trait this collection and derive a set of short DSs if they exist.

```
Algorithm 3: Derivation step in MapReduce
input  : The truncated successor tree of an FSM M.
output: A shortest DS for each pair of states.
1  Function Map($key$, $values$):
       Data: $\langle key, value\rangle$ pair, where $key$ represents an arbitrary instance identifier and $value$ is
             an edge from the truncated successor tree.
       Result: Collection of $\langle k, edge\rangle$ pairs, where $k$ be an associated key with the edge $edge$.
       /* create the edge edge from the input value */
2      $edge$ = getEdgeFrom ($value$)
3      if $n_d$ ($edge$) is empty then
4          $k = n_s$ ($edge$)
5          Emit ($k$,$edge$)
6      else
7          $keysList = n_d$ ($edge$)
           /* keysList is a set of pairs of states */
8          foreach $k$ in $keysList$ do
9              Emit ($k$,$edge$)
10 Function Reduce($key$, $values$):
       Data: $\langle key, values\rangle$ pair, where $values$ is a set of edges having the same key $key$.
       Result: Truncated successor tree minus the leaves.
       /* Split values into two disjoined subsets of edges $values_1$ and $values_2$ */
11     $values_1 = \{e \in values \mid n_d(e)$ is not empty$\}$
12     $values_2 = values \setminus values_1$
13     foreach ($e_1$,$e_2$) in $values_1 \times values_2$ do
           /* construct a common suffix of a set distinguishing sequences */
14         $l(e_1) = l(e_1)l(e_2)$
15         $n_d$ ($e_1$)= $n_d$ ($e_1$)$\setminus$ $n_s$ ($e_2$)
16     Emit (null,$e_1$)
```

In this step, we derive a shortest DS for each pair of states. It received from the intersection step $\frac{n(n-1)}{2} \times |I|$ truncated successor tree edges and produces $\frac{n(n-1)}{2}$ pairs of states with their DS if exists and if not we mention the "*not found*" notation. To that end, we use a single mapping method based on states. Each map function takes as input an edge e from the truncated successor tree and produces a set of $\langle key, e\rangle$ pairs. A key key is the pair of states in the source node $n_s(e)$ if the destination node $n_d(e)$ is empty; otherwise, it is the set of state pairs in the destination node of the edge e.

Let us compute the replication rate in each MapReduce round of the derivation step based on Algorithm 3. We have $\frac{n(n-1)}{2}$ available reducers and each reducer cannot contain more than $q \leq (\frac{n(n-1)}{2}) \times |I|$ edges. The number of outputs that can be produced is $g(q) = 1$ in the last iteration, then the following proposition holds.

Proposition 5 *The replication rate r in each MapReduce round of the derivation step is* $r \leq \frac{n(n-1)}{2}$

Theorem 2 *Algorithm* 3 *correctly derives a short DS, if it exists, for all pair of states of an FSM* $M = (S, I, O, E)$ *in maximum mn MapReduce rounds, where* $m = |E|$ *and* $n = |S|$.

Proof

It is obvious to see that a DS is the label of a path from the root node to a leaf node indexed by the empty set {} *in the successor tree. During each MapReduce round of Algorithm 3, the successor tree Tree is compacted by level until the stop condition. Without loss of generality, let us consider a leaf node* ne_k *indexed by the empty set which is located* *at the kth level in the successor tree, and let prec(n) be the set of predecessor nodes of the node n. In each MapReduce round, the node* ne_k *replaces all nodes located at the level* $k-1$*, belonging to the set* $prec(ne_k) = \{ne^1_{k-1}, \cdots, ne^l_{k-1}\}$*. As a consequence the successor tree is compacted in the following way: a label x of an edge* (ne^i_{k-1}, x, ne_k) *is concatenated with all labels of the set of edges* $\{(n, y, ne^i_{k-1} \mid n \in prec(ne^i_{k-1})\}$*, for all* $1 \leq i \leq l$*, to produce the set of edges* $\{(n, yx, ne_k \mid n \in prec(ne^1_{k-1})\}$*. The number of MapReduce rounds in Algorithm 3 is related to the stop condition which is true when the root node of the successor tree is reached and a set of DSs is derived, if exists, in less than mn iterations.* □

Implementation and experimental results

This section includes extensive experiments with the above described methods to evaluate their efficiency and effectiveness in terms of the communication cost and the execution time.

The experiments are conducted on randomly generated complete observable nondeterministic FSMs which cover a varying number of states, input and output alphabets sizes, degree of nondeterminism, and range. We run five different experiments then we calculate and depict the average of the obtained results in the corresponding figures. Finally, we compare the proposed methods in terms of the communication cost and the execution time required in MapReduce framework to derive the truncated successor tree and extract a short distinguishing sequence for each pair of states if exists.

Cluster configuration

Our experiments were run on Hadoop on the French scientific testbed Grid'5000 [64] at the site of Lille. We used for our experiments a cluster composed of 15 nodes, 30 CPUs, 300 cores. Each node is a machine equipped with two Intel Xeon E5-2630 v4 with 10-cores processors, 256 GB of main memory, and two disk drives (HDD) at 300 GB. The machines are connected by 10 Gbps Ethernet network and run 64-bit Debian 9. The Hadoop version installed on all machines is 2.7.

Data generation method

We randomly generated a large variety of FSMs data sets in two phases based on different combinations of the input alphabet size $|I|$, the output alphabet size $|O|$, the number of states $|S|$, the degree of nondeterminism D and the range R. First, we randomly generated complete deterministic finite automata using the method described in Abbadingo One competition (see http://abbadingo.cs.nuim.ie). Then, we randomly selected $\{(D \times |S| \times |I|)\}$ transitions where $|D|$ equals 20, 40, 60 or 80. Finally, we add to the obtained FSM nondeterministic the observability property, by generating randomly $(\frac{R \times |O|}{100})$ number of replications for each selected transition where the observability range R is 20-30, 40-50, 60-70 or 80-90. Table 1 summarizes all the datasets used in our experiments along with their respective properties. In order to obtain accurate results, we have generated five samples $S^1_i, \cdots, S^5_i$ for each dataset S_i in Table 1. Then, the results of different experiments are the mean of results obtained from these samples.

Communication cost analysis

The communication cost is equal to the total number of key-value pairs sent from the map phase to the reduce phase. It can be optimized by minimizing the replication rate parameter *i.e.* the number of input copies sent to the reducers.

The following table summarizes the relationship between the FSM size and the communication cost in the MapReduce algorithm for different datasets:

Since we have introduced three mapping methods in the intersection step of our approach, we present the communication cost for the considered datasets in Fig. 5.

In Fig. 5, the obtained results show clearly that the mapping based on input alphabet outperforms the other mapping methods in terms of the communication cost. This is due to the fact that the number of the transition copies sent to the reducers using this method is less than the other ones as proved formally in Proposition 4. In some particular cases of FSMs, when the number of states is less than the input alphabet size, the state-based mapping has less communication cost. This coincides with the results of Propositions 2 and 3.

Table 1 Datasets used in different experiments

Parameter	Dataset	Size (MB)	$\|E\|$ ($\times 10^6$)	$\|S\|$	$\|I\|$	$\|O\|$	D (%)	R (%)
Number of transitions	S_1	14.24	1	143	143	143	70	50
	S_2	29.86	2	180	180	180	70	50
	S_3	44.74	3	205	205	205	70	50
	S_4	59.94	4	225	225	225	70	50
	S_5	76.22	5	243	243	243	70	50
Degree of nondeterminism	S_6	13.26	0.896	200	200	200	20	50
	S_7	26.88	1.816	200	200	200	40	50
	S_8	38.61	2.608	200	200	200	60	50
	S_9	51.28	3.464	200	200	200	80	50
Input alphabet size	S_{10}	17.41	1.203	250	50	250	70	50
	S_{11}	35.82	2.458	250	100	250	70	50
	S_{12}	49.99	3.345	250	150	250	70	50
	S_{13}	78.06	5.160	250	200	250	70	50
	S_{14}	93.64	6.144	250	250	250	70	50
Range	S_{15}	14.51	0.980	200	200	200	50	20
	S_{16}	27.51	1.860	200	200	200	50	40
	S_{17}	38.19	2.580	200	200	200	50	60
	S_{18}	50.01	3.380	200	200	200	50	80
Number of states	S_{19}	4.62	0.367	100	100	100	70	50
	S_{20}	10.82	0.790	200	100	100	70	50
	S_{21}	17.53	1.248	300	100	100	70	50
	S_{22}	21.68	1.524	400	100	100	70	50
	S_{23}	29.83	2.080	500	100	100	70	50
	S_{24}	31.73	2.202	600	100	100	70	50
	S_{25}	40.69	2.813	700	100	100	70	50
	S_{26}	44.19	3.048	800	100	100	70	50
	S_{27}	49.82	3.429	900	100	100	70	50
	S_{28}	58.51	4.020	1000	100	100	70	50

Computation cost analysis

The computation cost is the time required to execute a MapReduce job. The graphs below present comparative results in terms of the execution time of the proposed methods when varying different parameters such as the number of states $|S|$, the input alphabets size $|I|$, the output alphabet size $|O|$, the degree of nondeterminism D, the range R, and the total number of transitions. We note that we provide a real execution time without any inference computation rules.

The Figs. 6, 7, 8, 9 and 10 show increasing curves that present the execution time of the proposed methods for the intersection and the derivation steps for different data sets when varying the input alphabet size in Fig. 6, the number of states in Fig. 7, the degree of nondeterminism in Fig. 8, the range in Fig. 9 and the number of transitions in Fig. 10. The input alphabet method is more efficient when the alphabet size is less than or equal to the number of available reducers. The change of the FSM parameters has little effect on the performance of the symbol-based method because each reducer receives only useful transitions, so the only influence is the increase of the input size (i.e. number of transitions). However, if we have a large number of resources, it involves a waste of them, and each reducer can contain multiple transitions having the same input alphabet from different states. The hybrid mapping method is more parallel compared to the other methods, for example, such that each reducer process a few numbers of transitions which leads to a global reduced running time. However, this method takes a lot of time in the mapping phase when replicating the transitions. Otherwise, this may require a large number of reducers compared to available resources. Therefore, a set of reducers has to wait, which implies a rise in the execution time. The states mapping method is the weakest one because it has an important replication rate and so a large number of reducers. Besides, the transitions' intersection is performed inside a reducer which is defined from the associated key. In the derivation step, we propose only one mapping method, which is based on states. The results show that the execution time is nearly linear for all considered parameters. According to the experiments results, minimizing the replication rate decreases the time used by the mappers to replicate each transition, and avoids read/write of the large intermediate results. In the same time, it reduces the number of transitions that are assigned to a reducer. On the other hand, using the adequate number of reducers diminishes the waiting time that a reducer spends to use a CPU. Therefore, we get an optimal parallel MapReduce scheme to produce a set of short distinguishing sequences for a complete observable nondeterministic FSM.

Comparative study

We performed a set of experiments to evaluate the efficiency and scalability of the proposed methods in comparison with the state-of-the-art approach. We used the speedup metric, which is defined as how much faster the parallel method is in comparison to the sequential method, to evaluate the performance of our MapReduce methods with a multi-threads based approach [65]. We conducted experiments in a single node from the cluster described above, a node is a machine equipped with 2 Intel Xeon E5-2630 v4 with 10 cores per processor, 2 threads per core, 256 GB of main memory, and two disk drives (HDD) at 300 GB. For each experiment, we run different methods five times on 40 threads to determine which one gives the best performance and analyze the effect of the

Table 2 The communication cost of the intersection step and the derivation step for some datasets

Dataset	Intersection Output Size=Derivation Input Size (MB)	Derivation step			
		Derivation phase		Postprocessing phase	
		Communication Cost (GB)	Output Size (GB)	Communication Cost (GB)	Output Size (GB)
S_1	138	4.8	188	211.4	1
S_2	357	15.7	793.8	869	1.5
S_3	603	29.8	1738.7	1882.5	2
S_4	882	47.9	3078.5	3308.6	2.4
S_5	1205	70.7	4930.4	5270	3

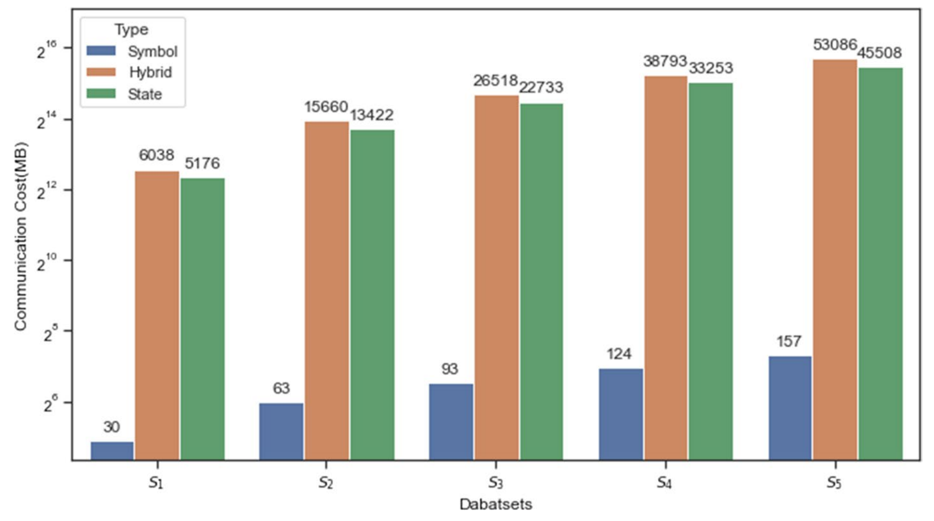
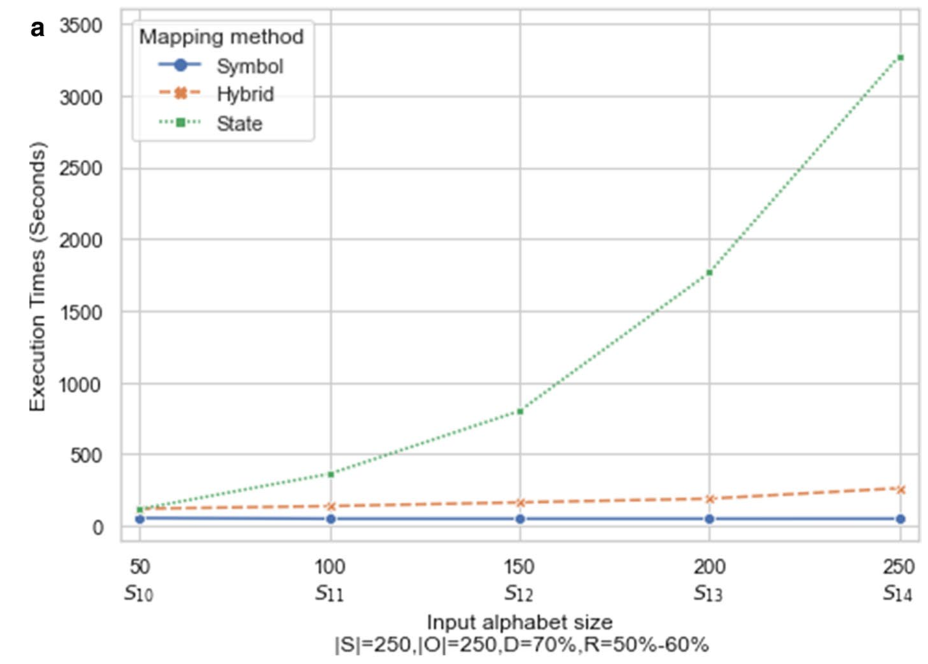


Fig. 5 Communication cost of the three proposed methods in the intersection step

number of transitions and range nondeterminism of the FSM on the speedup test. The comparison of the three MapReduce methods proposed previously in the intersection step of our solution with a multi-threads parallel implementation based on OpenMP of the sequential Exact Algorithm (EA) step on a multicore CPU [66] using OMP4J [67]. OMP4J is an open-source implementation of OpenMP and is used as a preprocessor for Java. The experiments confirm previous results, in Fig. 11 depicts the speedup for different datasets from Table 2. It can be seen that the mapping method based on input alphabets (Symbol) is the best and effective for different datasets. Figure 12 shows the speedup according to the ranges of nondeterminism R 50–60, 60–70, 70–80consistent performance gains that the speedup of the alphabets-based method is better than the three other methods under different circumstances. This performance is due to the low cost of the replication rate between map and reduce, and each reducer receives only useful transitions. Besides, the performance of the Symbol method is faster than the OpenMP-based method, and the speedup for this method grows exponentially as the number of transitions and range nondeterminism of the FSM increases.



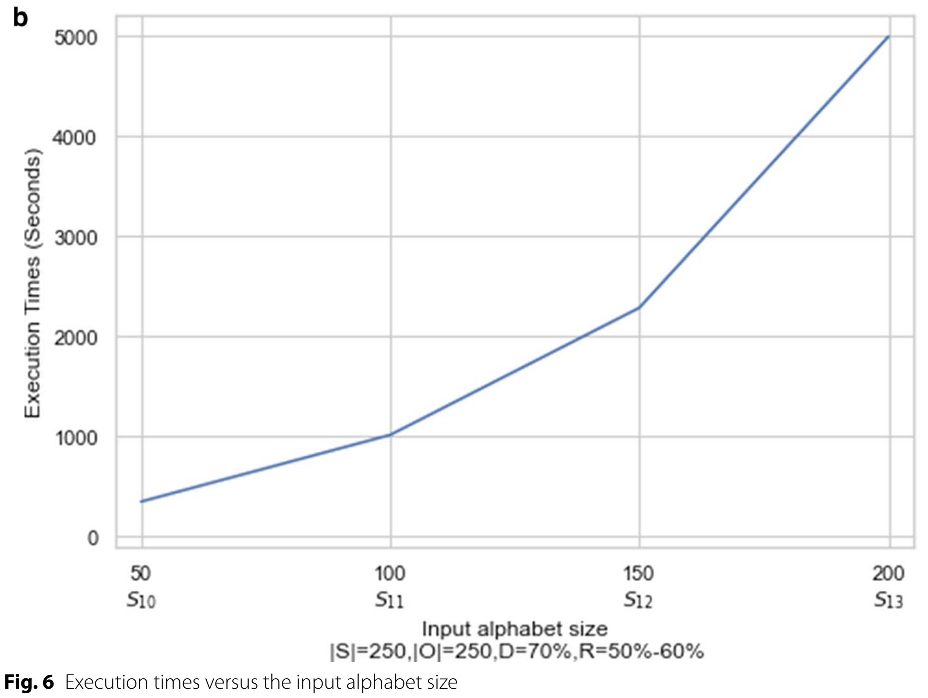


Fig. 6 Execution times versus the input alphabet size

We have obtained satisfactory results proving that the proposed approach for deriving short DSs from nondeterministic FSMs is well adapted in a large-scale context. However, MapReduce is not designed for iterative processing, and the data has to be written onto the disk after every iteration, thus making the disk I/O a huge bottleneck. To combine the results from different nodes after every iteration presents a significant challenge due to the complex network structure. To overcome this limitation

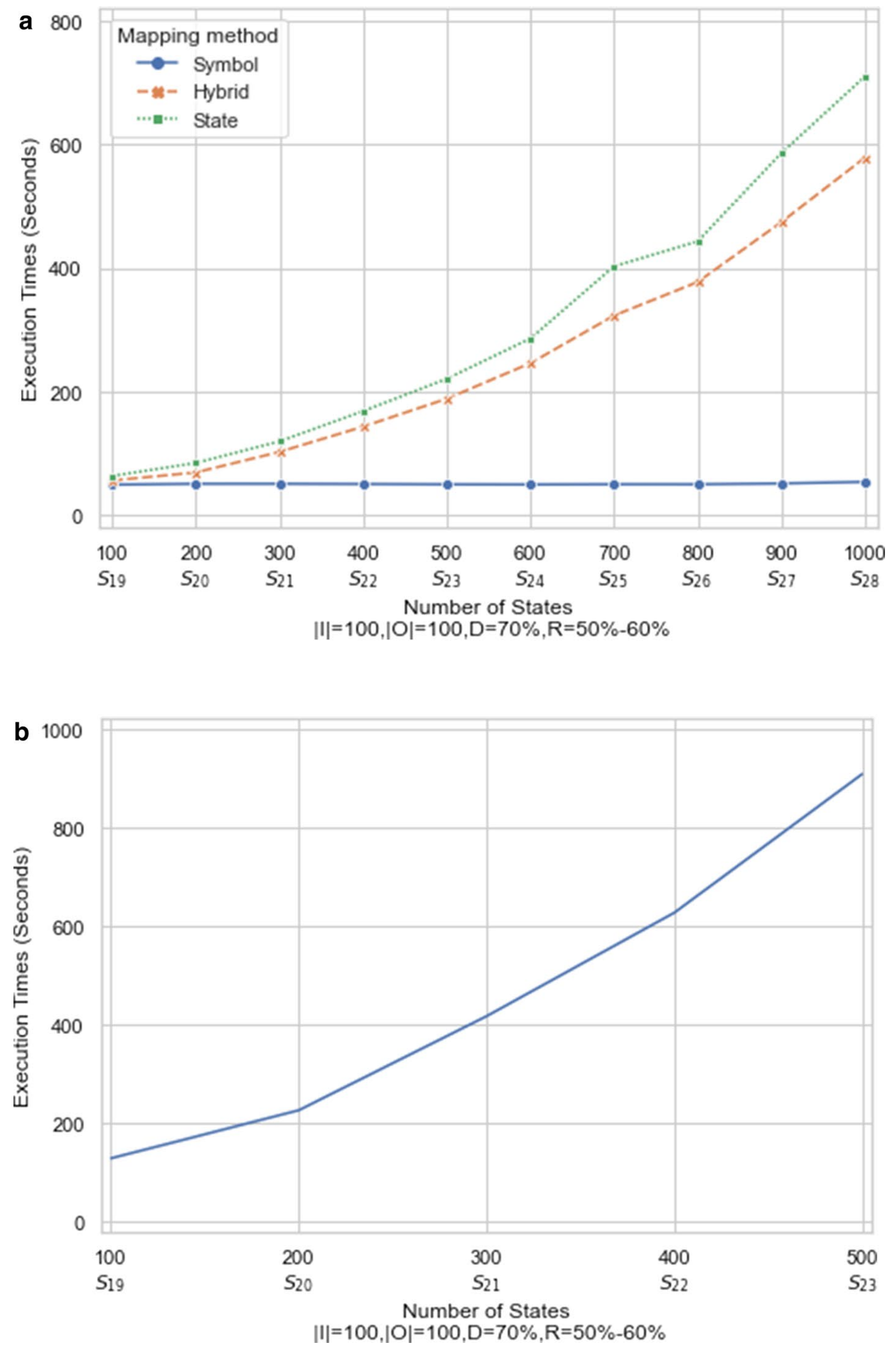


Fig. 7 Execution times versus the number of states

in the derivation step, we can use I2 MapReduce framework [68], which introduces a MapReduce Bipartite Graph model to represent iterative and incremental computations, which contains a loop between mappers and reducers.

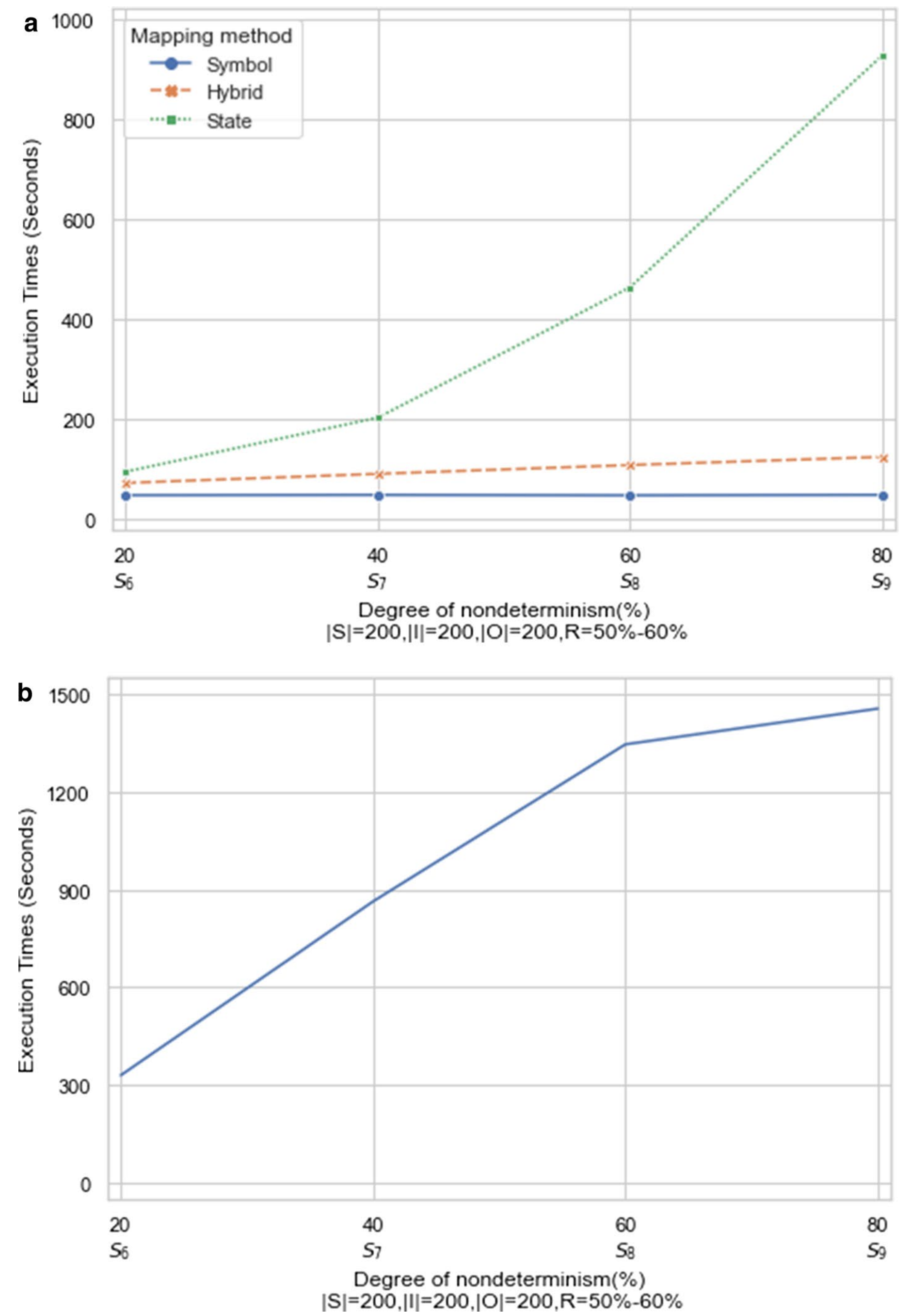


Fig. 8 Execution times for different degrees of nondeterminism

Conclusions and future works

FSMs are widely used in various application domains, such as pattern matching, communication protocols, logical devices synthesis, and other reactive systems. In FSM-based testing, distinguishing sequences are derived from a model and then applied to a machine or a black-box Implementation Under Test (IUT) to solve the state identification problem. A lot of work has been on testing from deterministic FSMs but much

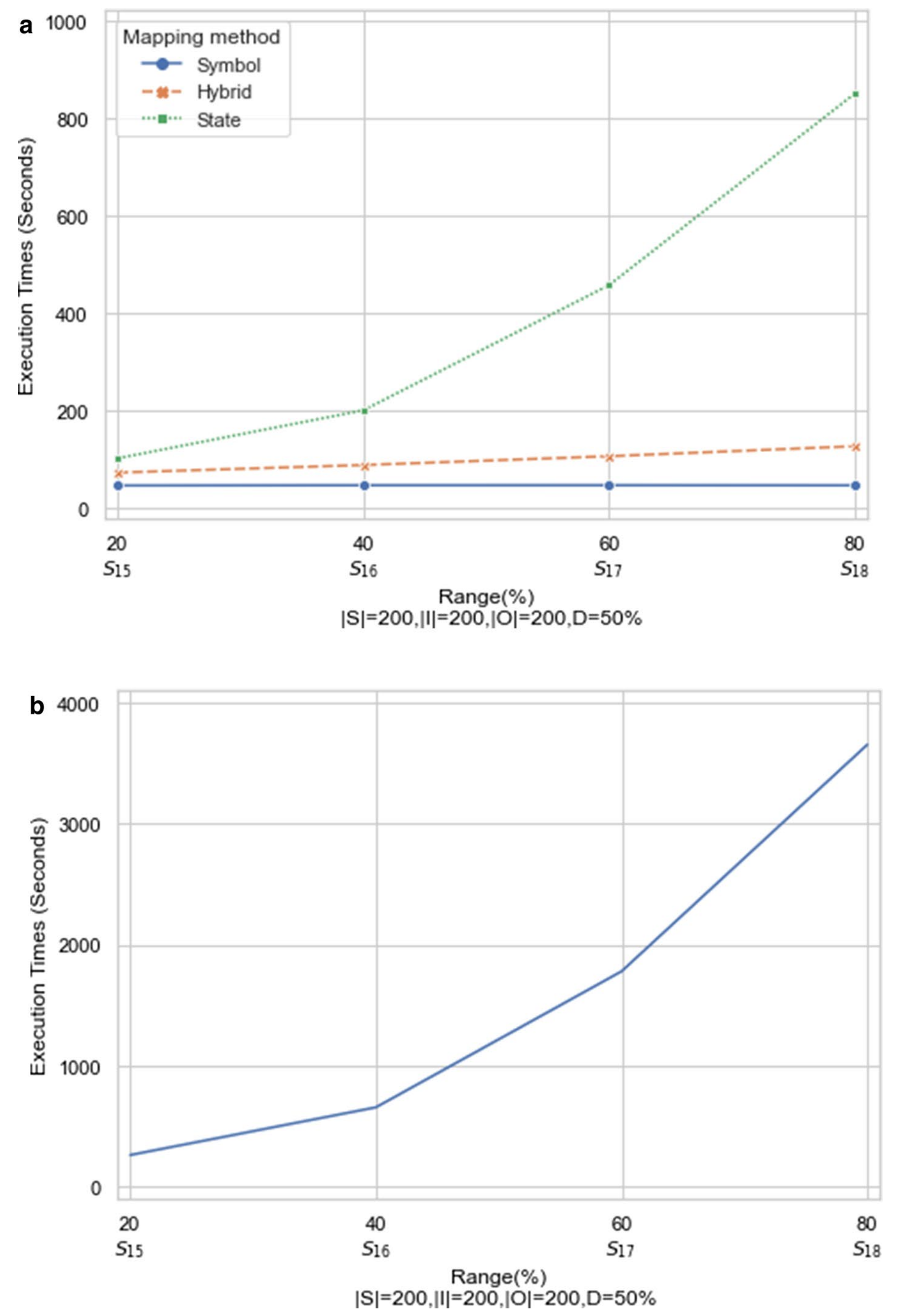


Fig. 9 Execution times for different ranges

of FSMs specifications of real-life systems are nondeterministic, which introduces the scalability challenge: the shortest DS can reach the exponential bound.

In this work, we addressed the scalability issue encountered while deriving distinguishing sequences from complete observable nondeterministic finite state machines

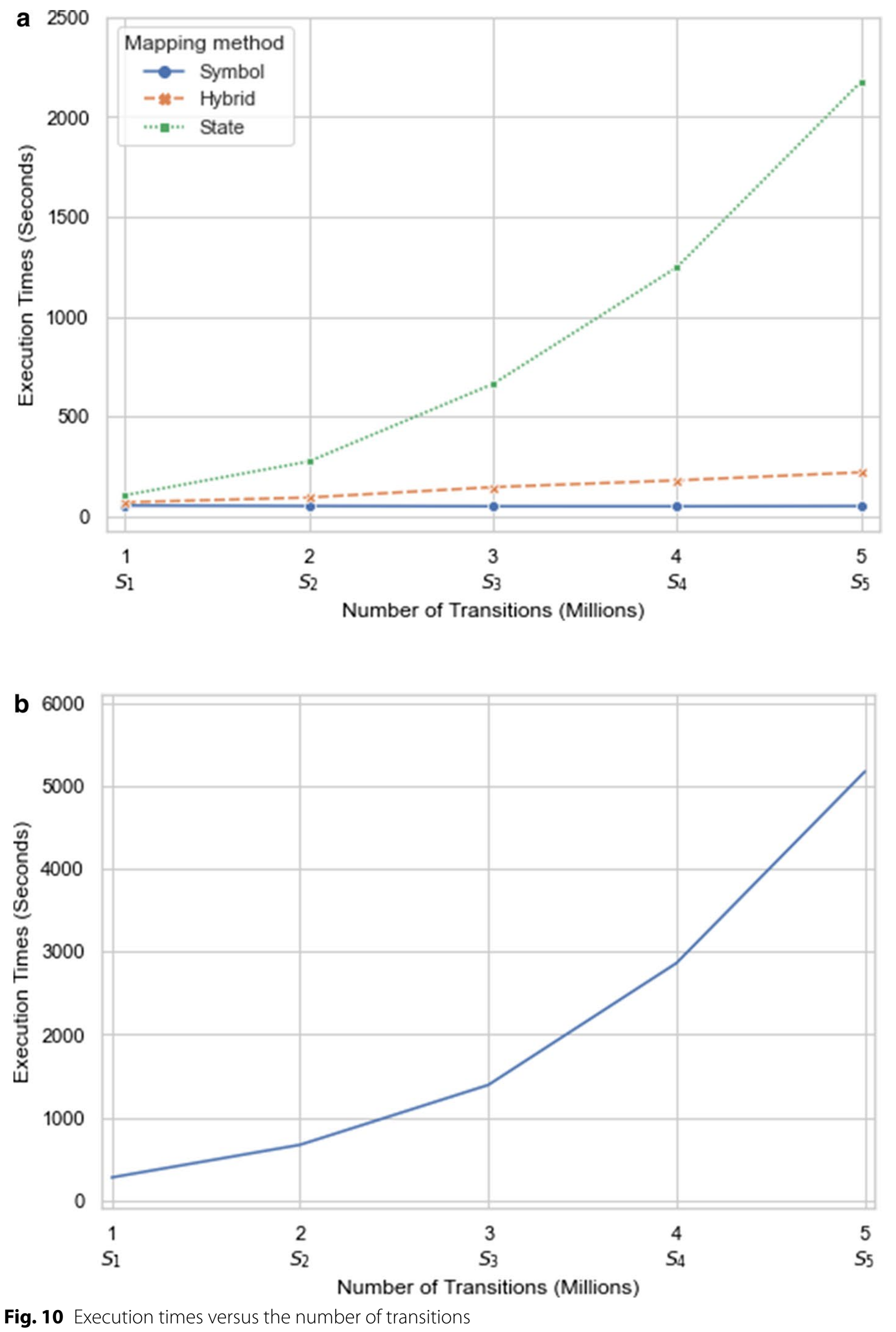


Fig. 10 Execution times versus the number of transitions

(FSMs) by introducing a massively parallel MapReduce version of the well-known Exact Algorithm, with experiments showing that this scaled much better than a classical generation algorithm. Our approach is based on two MapReduce steps: the intersection step and the derivation of short distinguishing sequences step. In the first step, we have proposed three MapReduce methods based respectively on a mapping

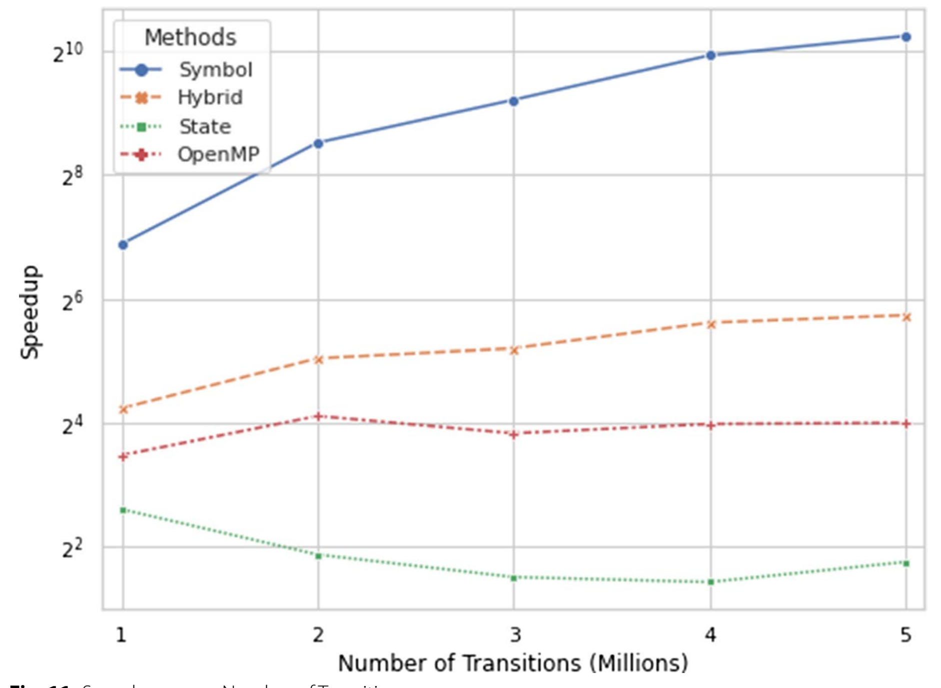


Fig. 11 Speedup versus Number of Transitions

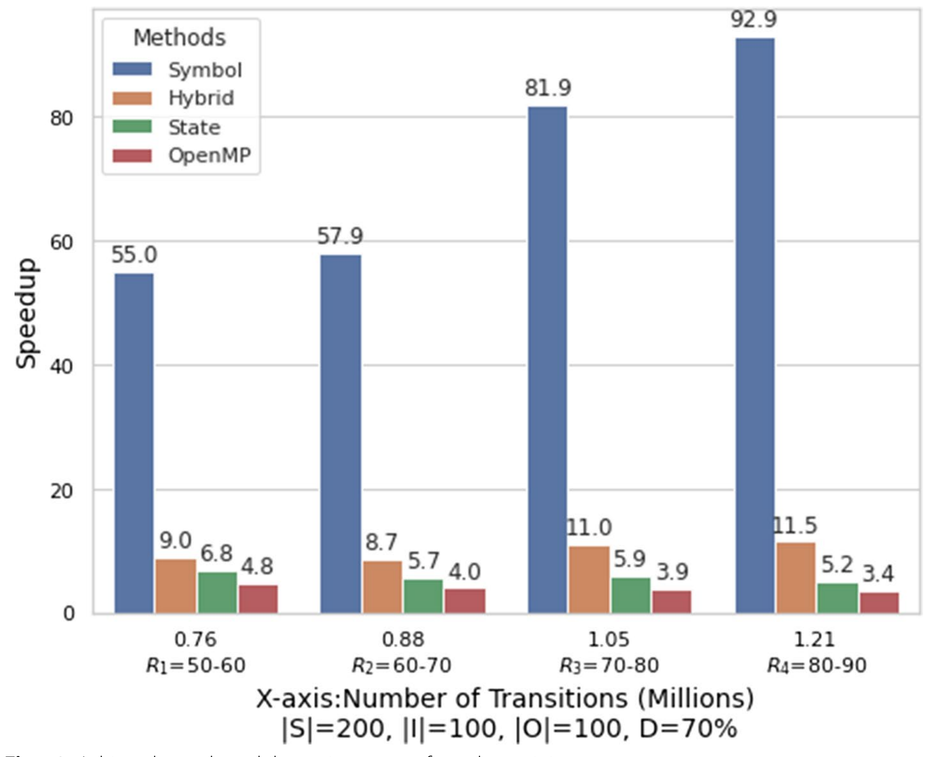


Fig. 12 Achieved speedup while varying range of nondeterminism

based on states, a mapping based on input alphabets, and a hybrid mapping based on both states and input alphabets. The introduction of three methods is justified by the fact that the required time of this step takes about 96% of the whole EA's time. In the second step, an iterative MapReduce algorithm is introduced to derive a set of short distinguishing sequences. For both steps, we have analyzed the communication cost using Afrati et al. model that offers a formal aspect of an inherent trade-off between communication cost and parallelism degree in a distributed computing environment. We performed experiments with randomly generated FSMs, the implementations are assessed with respect to communication cost, execution time, and speedup. During the experiments, we compared the results of the proposed algorithms with multiple threads based algorithm in terms of speedup and found that the proposed algorithm is efficient and much more scalable: Our MapReduce algorithm was able to process FSMs having 5 million transitions, which is up to 150 times larger than the existing PDS generation algorithms and with a speedup 2^6 more than OpenMP based intersection algorithm.

One particular line of future work is to investigate our parallel MapReduce-based algorithm for deriving other checking sequences used in test generation such as UIO sequences [17], characterizing sets, harmonized state identifiers [18] and synchronizing sequences [19]. Finally, there would also be value in additional experiments with FSMs from the industry.

Abbreviations
FSM:: Finite state machine; SDL:: Specification and Description Language; UML:: Unified Modeling Language; IUT:: Implementation Under Test; DS:: Distinguishing sequence; ADS:: Adaptive distinguishing sequence; PDS:: Peset distinguishing sequence; EA:: Exact Algorithm; CPU:: Central Processing Unit; GPU:: Graphics Processing Unit; NoWs:: Network of Workstations solution; BFS:: Breadth-First Search.

Authors' contributions
All authors read and approved the final manuscript.

Funding
Université du Littoral Côte d'Opale (ULCO).

Availability of data and materials
Not applicable.

Declarations

Ethics approval and consent to participatet
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Univ. Littoral Côte d'Opale, UR 4491, LISIC, Laboratoire d'Informatique Signal et Image de la Côte d'Opale, Calais 62100, France. [2]ANISSE research Team, Department of Computer Science, Faculty of Sciences, Mohammed V University in Rabat, Rabat B.P. 1014, Morocco.

Received: 21 July 2021 Accepted: 31 October 2021
Published online: 20 November 2021

References
1. Aho AV, Dahbura AT, Lee D, Uyar MU. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours," protocol specification, testing and verification, vol. viii. Testing, and verification VIII, Atlantic City, NorthHolland; 1988. p. 75–86.

2. Knuth DE, Morris JH Jr, Pratt VR. Fast pattern matching in strings. SIAM J Comput. 1977;6(2):323–50.
3. Gladyshev P, Patel A. Finite state machine approach to digital event reconstruction. Digit Investig. 2004;1(2):130–49.
4. Mavridou A, Laszka A. Designing secure ethereum smart contracts: a finite state machine based approach. In: International conference on financial cryptography and data security. Springer; 2018. p. 523–540.
5. Hsaini S, Azzouzi S, Charaf MEH. A temporal based approach for mapreduce distributed testing. Int J Parallel Emergent Distrib Syst. 2021;36(4):293–311.
6. Hierons RM, Türker UC. Distinguishing sequences for distributed testing: preset distinguishing sequences. Comput J. 2017;60(1):110–25.
7. Thomas N, Heather J, Ndifon W, Shawe-Taylor J, Chain B. Decombinator: a tool for fast, efficient gene assignment in t-cell receptor sequences using a finite state machine. Bioinformatics. 2013;29(5):542–50.
8. Alur R, Henzinger TA. Computer-aided verification: an introduction to model building and model checking for concurrent systems. Draft. 1998. www-cad.eecs.berkeley.edu/~tah/CavBook.
9. Rockstrom A, Saracco R. Sdl-ccitt specification and description language. IEEE Trans Commun. 1982;30(6):1310–8.
10. McUmber WE, Cheng BH. A general framework for formalizing uml with formal languages. In: Proceedings of the 23rd international conference on software engineering. ICSE 2001. IEEE; 2001. p. 433–442.
11. Czerwinski R, Kania D. Finite state machine logic synthesis for complex programmable logic devices. Springer; 2013.
12. Lee D, Yannakakis M. Principles and methods of testing finite state machines-a survey. Proc IEEE. 1996;84(8):1090–123.
13. Dorofeeva R, El-Fakih K, Maag S, Cavalli AR, Yevtushenko N. Fsm-based conformance testing methods: a survey annotated with experimental evaluation. Inf Softw Technol. 2010;52(12):1286–97.
14. Lai R. A survey of communication protocol testing. J Syst Softw. 2002;62(1):21–46.
15. Türker UC, Ünlüyurt T, Yenigün H. Effective algorithms for constructing minimum cost adaptive distinguishing sequences. Inf Softw Technol. 2016;74:69–85.
16. Aho AV, Dahbura AT, Lee D, Uyar MU. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. IEEE Trans commun. 1991;39(11):1604–15.
17. Hierons RM, Türker UC. Parallel algorithms for testing finite state machines: generating UIO sequences. IEEE Trans Softw Eng. 2016;42(11):1077–91.
18. Hierons RM, Türker UC. Parallel algorithms for generating harmonised state identifiers and characterising sets. IEEE Trans Comput. 2016;65(11):3370–83.
19. Jourdan G-V, Ural H, Yenigün H. Reduced checking sequences using unreliable reset. Inf Process Lett. 2015;115(5):532–5.
20. Benenson Y, Paz-Elizur T, Adar R, Keinan E, Livneh Z, Shapiro E. Programmable and autonomous computing machine made of biomolecules. Nature. 2001;414(6862):430–4.
21. Boute RT. Distinguishing sets for optimal state identification in checking experiments. IEEE Trans Comput. 1974;100(8):874–7.
22. Chow TS. Testing software design modeled by finite-state machines. IEEE Trans Softw Eng. 1978;3:178–87.
23. Eppstein D. Reset sequences for monotonic automata. SIAM J Comput. 1990;19(3):500–10.
24. Hierons RM, Ural H. Uio sequence based checking sequences for distributed test architectures. Inf Softw Technol. 2003;45(12):793–803.
25. Natarajan BK. An algorithmic approach to the automated design of parts orienters. In: 27th annual symposium on foundations of computer science (sfcs 1986). IEEE; 1986. pp. 132–142.
26. Hierons R. Using a minimal number of resets when testing from a finite state machine. Inf Process Lett. 2004;6(90):287–92.
27. Türker UC, Yenigün H. Complexities of some problems related to synchronizing, non-synchronizing and monotonic automata. Int J Found Comput Sci. 2015;26(01):99–121.
28. Vasilevskii M. Failure diagnosis of automata. Cybernetics. 1973;9(4):653–65.
29. Moore E. Gedanken-experiments. In: Shannon C, McCarthy J, editors. Automata studies. Princeton University Press; 1956.
30. Hierons RM, Ural H. Optimizing the length of checking sequences. IEEE Trans Comput. 2006;55(5):618–29.
31. Jourdan G-V, Ural H, Yenigün H, Zhang JC. Lower bounds on lengths of checking sequences. Form Asp Comput. 2010;22(6):667–79.
32. Lee D, Yannakakis M. Testing finite-state machines: state identification and verification. IEEE Trans Comput. 1994;43(3):306–20.
33. Simao A, Petrenko A. Checking completeness of tests for finite state machines. IEEE Trans Comput. 2010;59(8):1023–32.
34. Simao A, Petrenko A, Yevtushenko N. On reducing test length for fsms with extra states. Softw Test Verif Reliab. 2012;22(6):435–54.
35. Moore EP. Gedanken-experiments. J. Automata Studies: Princeton University Press; 1956.
36. Hennine F. Fault detecting experiments for sequential circuits. IEEE; 1964. p. 95–110. .
37. Ural H, Zhu K. Optimal length test sequence generation using distinguishing sequences. IEEE/ACM Trans Netw. 1993;1(3):358–71.
38. Gonenc G. A method for the design of fault detection experiments. IEEE Trans Comput. 1970;100(6):551–8.
39. Hierons RM, Jourdan G-V, Ural H, Yenigun H. Checking sequence construction using adaptive and preset distinguishing sequences. In: 2009 seventh IEEE international conference on software engineering and formal methods. IEEE; 2009. p. 157–166.
40. Ural H, Wu X, Zhang F. On minimizing the lengths of checking sequences. IEEE Trans Comput. 1997;46(1):93–9.
41. Kushik N, El-Fakih K, Yevtushenko N. Preset and adaptive homing experiments for nondeterministic finite state machines. In: International conference on implementation and application of automata. Springer; 2011. p. 215–224.
42. Hierons RM. Adaptive testing of a deterministic implementation against a nondeterministic finite state machine. Comput J. 1998;41(5):349–55.
43. Starke PH. Abstract automata. 1972.
44. Petrenko A, Yevtushenko N. Testing from partial deterministic fsm specifications. IEEE Trans Comput. 2005;54(9):1154–65.
45. Alur R, Dill DL. A theory of timed automata. Theor Comput Sci. 1994;126(2):183–235.
46. Güniçen C, İnan K, Türker UC, Yenigün H. The relation between preset distinguishing sequences and synchronizing sequences. Form Asp Comput. 2014;26(6):1153–67.

47. Spitsyna N, El-Fakih K, Yevtushenko N. Studying the separability relation between finite state machines. Softw Test Verif Reliab. 2007;17(4):227–41.
48. Alur R, Courcoubetis C, Yannakakis M. Distinguishing tests for nondeterministic and probabilistic machines. In: Proceedings of the twenty-seventh annual ACM symposium on theory of computing. 1995; p. 363–372.
49. Grahne G, Harrafi S, Moallemi A, Onet A. Computing NFA intersections in map-reduce. In: EDBT/ICDT workshops. 2015; p. 42–45.
50. Elghadyry B, Ouardi F, Verel S. Composition of weighted finite transducers in mapreduce. J Big Data. 2021;8(1):1–15.
51. Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. 2004.
52. Foundation AS. Welcome to apache hadoop. 2018. http://hadoop.apache.org/.
53. Mishra P, Mishra M, Somani AK. Applications of hadoop ecosystems tools. In: NoSQL: database for Storage and Retrieval of Data in Cloud. Chapman and Hall/CRC; 2017. p. 159–176. .
54. Torabzadehkashi M, Rezaei S, HeydariGorji A, Bobarshad H, Alves V, Bagherzadeh N. Computational storage: an efficient and scalable platform for big data and HPC applications. J Big Data. 2019;6(1):1–29.
55. Kishani M, Tahoori M, Asadi H. Dependability analysis of data storage systems in presence of soft errors. IEEE Trans Reliab. 2019;68(1):201–15.
56. Mishra P. Host managed storage solutions for big data. 2018.
57. Mishra P, Somani AK. Ldm: lineage-aware data management in multi-tier storage systems. In: Future of information and communication conference. Springer; 2019. p. 683–707.
58. Afrati FN, Sarma AD, Salihoglu S, Ullman JD. Upper and lower bounds on the cost of a map-reduce computation. Proc VLDB Endow. 2013;6(4).
59. Türker UC. Parallel brute-force algorithm for deriving reset sequences from deterministic incomplete finite automata. Turk J Electr Eng Comput Sci. 2019;27(5):3544–56.
60. Karahoda S, Erenay OT, Kaya K, Türker UC, Yenigün H. Multicore and manycore parallelization of cheap synchronizing sequence heuristics. J Parallel Distrib Comput. 2020;140:13–24.
61. Hierons RM, Türker UC. Parallel algorithms for generating distinguishing sequences for observable non-deterministic fsms. ACM Trans Softw Eng Methodol (TOSEM). 2017;26(1):1–34.
62. El-Fakih K, Barlas G, Ali M, Yevtushenko N. Parallel algorithms for reducing derivation time of distinguishing experiments for nondeterministic finite state machines. Int J Parallel Emergent Distrib Syst. 2018;33(2):197–210.
63. Pospichal P, Jaros J, Schwarz J. Parallel genetic algorithm on the cuda architecture. In: European conference on the applications of evolutionary computation. 2010; p. 442–451.
64. Bolze R, Cappello F, Caron E, Daydé M, Desprez F, Jeannot E, Jégou Y, Lanteri S, Leduc J, Melab N, et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. Int J High Perform Comput Appl. 2006;20(4):481–94.
65. Kang SJ, Lee SY, Lee KM. Performance comparison of OpenMP, MPI, and MapReduce in practical problems. Adv Multimed. 2015; 2015.
66. Haddad AR, El-Fakih K, Barlas G. Parallel implementation for deriving preset distinguishing experiments of nondeterministic finite state machines. In: 2017 7th international conference on modeling, simulation, and applied optimization (ICMSAO). 2017; p. 1–6.
67. Bělohlávek P, Steinhauser A. Omp4j - openmp for java. PhD thesis, Univerzita Karlova, Matematicko-fyzikální fakulta 2015. http://www.omp4j.org/.
68. Zhang Y, Chen S. i2mapreduce: incremental iterative mapreduce. In: Proceedings of the 2nd international workshop on cloud intelligence. 2013; p. 1–4.