Journal of Big Data

RESEARCH Open Access

Composition of weighted finite transducers in MapReduce

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

*Correspondence: bilal.el-ghadyry@univ-littoral.fr
[1] Univ. Littoral Côte d'Opale, UR 4491, LISIC, Laboratoire d'Informatique Signal et Image de la Côte d'Opale, 62100 Calais, France
Full list of author information is available at the end of the article

Abstract
Weighted finite-state transducers have been shown to be a general and efficient representation in many applications such as text and speech processing, computational biology, and machine learning. The composition of weighted finite-state transducers constitutes a fundamental and common operation between these applications. The NP-hardness of the composition computation problem presents a challenge that leads us to devise efficient algorithms on a large scale when considering more than two transducers. This paper describes a parallel computation of weighted finite transducers composition in MapReduce framework. To the best of our knowledge, this paper is the first to tackle this task using MapReduce methods. First, we analyze the communication cost of this problem using Afrati et al. model. Then, we propose three MapReduce methods based respectively on input alphabet mapping, state mapping, and hybrid mapping. Finally, intensive experiments on a wide range of weighted finite-state transducers are conducted to compare the proposed methods and show their efficiency for large-scale data.

Keywords: Finite transducers, MapReduce, Composition, Communication cost

Introduction
Weighted finite-state transducer (WFST) has been used in a wide range of applications such as digital image processing [1], speech recognition [2], large-scale statistical machine translation [3], cryptography [4], recently in computational biology [5] where pairwise rational kernels are computed for metabolic network prediction and many other applications [6–8]. Weighted finite-state transducers are finite-state machines in which each transition in addition to its input symbol is augmented with an output symbol from a possibly new alphabet, and carries some weight element of a semiring. Transducers can be used to define a mapping between two languages. In some cases, The weight represents the uncertainty or the variability of information. For example, weighted transducers introduced in speech recognition [9] are used to assign different pronunciations to the same word with different ranks or probabilities. In classification and learning, kernel methods, like support vector machines, are widely used [10]. In [11], Mohri et al. have introduced the theory of rational kernel. The computation of a rational kernel can be done efficiently using a fast algorithm for the composition of weighted transducers. Computing the composition of WFSTs is basically based on the

standard composition of unweighted finite-state transducers. It takes as input, two or more WFSTs $(\mathcal{T}_i)_{1 \le i \le n}$, and outputs the composed WFST $\mathcal{T}$ realizing the composition of all input WFSTs, such that the input alphabet of $\mathcal{T}_{i+1}$ coincides with the output alphabet of $\mathcal{T}_i$. The time complexity for computing this operation is shown to be $O\left(\prod_{i=1}^{n} |\mathcal{T}_i|\right)$ where $|\mathcal{T}_i|$ represents the number of states in $\mathcal{T}_i$ [11, 12] (i.e. if there are n input WFSTs, each having m states, the resulting WFST can reach the exponential bound of m^n states). The complexity issue of the composition computation leads us to devise efficient methods in large scale.

In this work, we tackle the problem of the composition of WFSTs in the MapReduce framework, which is introduced by Google as a simple parallel programming model [13]. MapReduce is considered as a single Instruction Multiple Data (SIMD) architecture [14] that can be easily implemented over the Hadoop Apache framework [15]. We also analyze the cost model for this problem following the optimization approach introduced by Afrati et al. [16]. The cost model includes the communication cost which is the amount of data transmitted during MapReduce computations and the replication rate which represents the number of key-value pairs generated by all the mapper functions, divided by the number of inputs. A growing number of papers deal with MapReduce algorithms for various problems [17–20]. Recently, Grahne et al. [21, 22] have implemented efficiently the intersection and the minimization operations of finite automata in MapReduce. To the best of our knowledge, this paper is the first approach for computing the composition of WFSTs in MapReduce. We propose three methods to perform this operation in large scale respectively based on states mapping, input alphabet mapping, and a hybrid input and output alphabets mapping.

The remainder of the paper is structured as follows. "MapReduce framework" section ncludes necessary technical definitions. "The composition of WFSTs in MapReduce" section is a reminder of the fundamental of MapReduce framework. "Methods" section presents a cost model analysis of the composition operation in MapReduce using Afrati et al. model. "Results and discussion" section presents and analyses three MapReduce methods that compute the composition of many WFSTs. Some comparative and extensive experiments are also conducted in this section to show the efficiency of our methods in large scale. "Conclusion" section concludes the paper.

Preliminaries

In this section, we introduce briefly the notion of weighted finite transducers. For further details in formal aspects of finite automata theory, we particularly recommend reading [23] or in French [24].

Semirings. A system $(\mathbb{K}, \oplus, \otimes, \underline{0}, \underline{1})$ is a semiring when $(\mathbb{K}, \oplus, \underline{0})$ is a commutative monoid with identity element 0; $(\mathbb{K}, \otimes, \underline{1})$ is a monoid with identity element $\underline{1}$); $\otimes$ distributes over $\oplus$; and $\underline{0}$ is an annihilator for $\otimes$: for all $a \in \mathbb{K}, a \otimes \underline{0} = \underline{0} \otimes a = \underline{0}$. Thus, a semiring is a ring that may lack negation. Some familiar semirings include the boolean semiring $(\mathbb{B}, \vee, \wedge, 0, 1)$, the tropical semiring $(\mathbb{R}_+ \cup \{\infty\}, min, +, \infty, 0)$, and the real semiring $(\mathbb{R}, +, \times, 0, 1)$ [25].

Weighted finite-state transducers. Let A and B be two alphabets. WFST Sometimes called $\mathbb{K}$-transducers over $A^* \times B^*$ are transducers endowed with weights in the semiring $\mathbb{K}$. The transition labels of a WFST are in $(A \cup \{\varepsilon\}) \times (B \cup \{\varepsilon\})$ (ε is the empty word).

In this work, we restrict the transition labels to be in $A \times B$. Formally, a WFST $\mathcal{T}$ is an 8-tuple $(A, B, Q, I, F, E, \lambda, \rho)$ where: A is the finite input alphabet of the transducer; B is the finite output alphabet; Q is a finite set of states; $I \subseteq Q$ the set of initial states; $F \subseteq Q$ the set of final states; $E \subseteq Q \times (A \times B) \times \mathbb{K} \times Q$ a finite set of transitions; $\lambda : I \to \mathbb{K}$ the initial weight function; and $\rho : F \to \mathbb{K}$ the final weight function mapping F to $\mathbb{K}$.

The size of $\mathcal{T}$, denoted by $|\mathcal{T}|$, is the number of its transitions. Given a transition $t \in E$, we denote by $s[t]$ its origin or start state, $d[t]$ its destination state or next state, and $w[t]$ its weight. A path $\pi = t_1 \ldots t_k$ is an element of E^* with consecutive transitions: $d[t_{i-1}] = s[t_i], i = 2, \ldots, k$. We extend s and d to paths by setting: $s[\pi] = s[t_1]$ and $d[\pi] = d[t_k]$. The weight function w can also be extended to paths by defining the weight of a path as the $\otimes$-product over the semiring $\mathbb{K}$ of the weights of its constituent transitions: $w[\pi] = w[t_1] \otimes \ldots \otimes w[t_k]$. Let $q, q' \in Q$, we denote by $P(q, q')$ the set of paths from q to q' and by $P(q, x, y, q')$ the set of paths from q to q' with input label $x \in A^*$ and output label $y \in B^*$. These definitions can be extended to subsets $R, R' \subseteq Q$, by: $P(R, x, y, R') = \bigcup_{q \in R, q' \in R'} P(q, x, y, q')$. A WFST $\mathcal{T}$ is regulated if the output weight associated by $\mathcal{T}$ to any pair of input-output string (x, y) by:

$$[\mathcal{T}](x, y) = \bigoplus_{\pi \in P(I,x,y,F)} \lambda[s[\pi]] \otimes w[\pi] \otimes \rho[d[\pi]]$$

is well-defined and in $\mathbb{K}$. We have $[\mathcal{T}](x, y) = 0$ when $P(I, x, y, F) = \emptyset$.

Composition of weighted finite transducers. Composition is a fundamental operation used to create complex weighted transducers from simpler ones. Let $\mathbb{K}$ be a commutative semiring and let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two WFSTs such that the input alphabet of $\mathcal{T}_2$ coincides with the output alphabet of $\mathcal{T}_1$. Assume that the infinite sum
$\bigoplus_z \mathcal{T}_1(x, z) \times \mathcal{T}_2(z, y)$ is well-defined and in $\mathbb{K}$ for all $(x, y) \in A^* \times B^*$. Then, the composition of $\mathcal{T}_1$ and $\mathcal{T}_2$ produce a WFST denoted by $\mathcal{T}_1 \circ \mathcal{T}_2$ and defined for all x, y by [11]:

$$[\mathcal{T}_1 \circ \mathcal{T}_2](x, y) = \bigoplus_z \mathcal{T}_1(x, z) \otimes \mathcal{T}_2(z, y)$$

There exists efficient composition algorithm for WFSTs [26]. States in the composition $\mathcal{T}_1 \circ \mathcal{T}_2$ of two WFSTs $\mathcal{T}_1$ and $\mathcal{T}_2$ are identified with pairs of a state of $\mathcal{T}_1$ and a state of $\mathcal{T}_2$. Its initial state is the pair of the initial states of $\mathcal{T}_1$ and $\mathcal{T}_2$. Its final states are pairs of a final state of $\mathcal{T}_1$ and a final state of $\mathcal{T}_2$. The following rule specifies how to derive a transition of $\mathcal{T}_1 \circ \mathcal{T}_2$ from appropriate transitions of $\mathcal{T}_1$ and $\mathcal{T}_2$:

$$(q_1, a, b, w_1, q_2) \text{ and } (q_1', b, c, w_2, q_2') \Longrightarrow \big((q_1, q_1'), a, c, w_1 \otimes w_2, (q_2, q_2')\big)$$

In the worst case, all transitions of $\mathcal{T}_1$ leaving a state q match all those of $\mathcal{T}_2$ leaving state q', thus the space and time complexity of composition is quadratic: $O(|\mathcal{T}_1||\mathcal{T}_2|)$. See [9] for a detailed presentation of the algorithm. The following Fig. 1 illustrates WFSTs composition.

In a general case, when considering the composition of many WFSTs, noted $(\mathcal{T}_i)_{1 \leq i \leq n}$, the space and time complexity of composition is $O\big(\prod_{i=1}^{n} |\mathcal{T}_i|\big)$. In this work, we propose a simple and efficient parallel methods to compute the composition of many WFSTs in MapReduce framework.

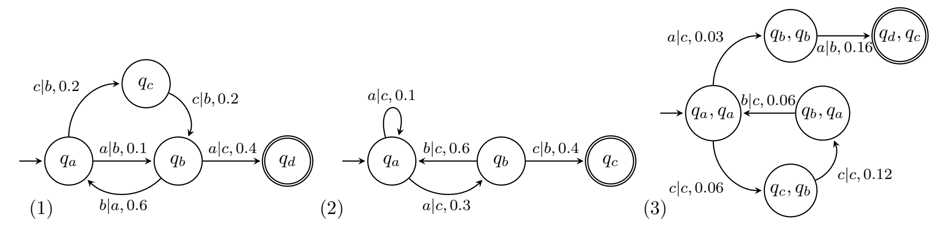

Fig. 1 (1) Weighted transducer $\mathcal{T}_1$ and (2) weighted transducer $\mathcal{T}_2$ over the tropical semiring. (3) Composition of $\mathcal{T}_1$ and $\mathcal{T}_2$

MapReduce framework

Big Data is a large and heterogeneous collection of datasets which makes it difficult to process using traditional data processing tools. Nowadays, the collected datasets come mostly from social networks and scientific applications. To overcome the computational and storage of Big data challenges, various solutions have been successfully proposed. Among the popular approaches, there is the famous Hadoop MapReduce framework [15].

In this section, we will focus on how distributed computing program works over the Hadoop MapReduce Model. First, the Hadoop framework and the MapReduce programming model will be briefly presented. Then, we describe how this system processes units of data $\langle key, value\rangle$ in parallel MapReduce approach.

Hadoop framework

The Apache Hadoop is one of the most popular open source framework for the clustered environment that allows reliable, scalable, and distributed storage. It also helps with the processing of large datasets through the simple programming models. It manages computer clusters built from single to thousands of machines, each offering local computation and storage. The failure of a node in the cluster is automatically managed by re-assigning its task to another node [15].

The project Apache Hadoop includes four fundamental modules: *Hadoop Common* or *Hadoop Core*, *HDFS*, *YARN* and *Hadoop MapReduce*. The following Fig. 2 schematizes these components.

- *Hadoop Common* or *Hadoop Core* provides essential services and basic processes such as abstraction of the underlying operating system and its file system. It also contains the necessary Java libraries and scripts required to start Hadoop. The Hadoop Common package also provides source code and documentation, as well as a contribution section that includes different projects from the Hadoop Community [15].
- *Hadoop Distributed File System (HDFS)* is a distributed file system developed by Apache Hadoop. It ensures high-throughput storage and access to application data on the community machines thus providing very high aggregate bandwidth across the cluster [15], high fault tolerance and native support of large data sets [27].

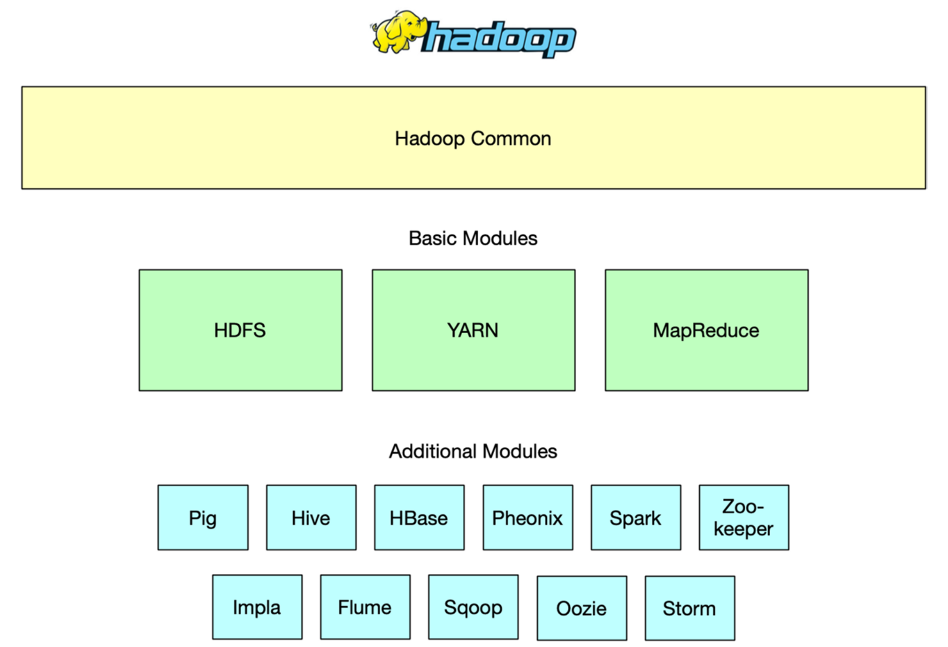

Fig. 2 Overview of Apache Hadoop. https://medium.com

- *Hadoop Yet Another Resource Negotiator (YARN)* is a platform to manage cluster resources and schedule tasks. It was added in the Hadoop 2.0 version to increase the capabilities by solving the limit of 4000 nodes and Hadoop's inability to perform fine-grained resource sharing between multiple computation frameworks [28].
- *Hadoop MapReduce* is an implementation of the MapReduce programming model based on YARN system for parallel processing of large data [29].

The MapReduce programming model

MapReduce is a programming model proposed by Google in 2004 [13] that provides parallel processing of large-scale data. It is easy to use and expresses a large variety of problems as MapReduce computation in a flexible way, which simplifies the data processing in large scale [13]. MapReduce programming model is a system to process the basic unit of information in a $\langle key, value\rangle$ pair, where key and value are two objects.

This computational model has three principal steps: Map, Shuffle and Reduce as schematized in Fig. 3.

During the map step, the model reads each $\langle key, value\rangle$ pair from a given input files. Then the Mapper operates on one pair at a time by calling the map function defined by the user, produces as output a finite multi-set of new $\langle key, value\rangle$ pairs, and determines new pair's sets through a hash function. This allows different machines to process the inputs of a different map in an easy parallel way. The shuffle step occurs automatically, it is done by Hadoop to manage the exchange of the intermediate data from the map task to the reduce task. One can be divide this step into three phases. The sort phase produces the set of intermediate keys received from the buffered mapper in a particular order. It assists the reducer

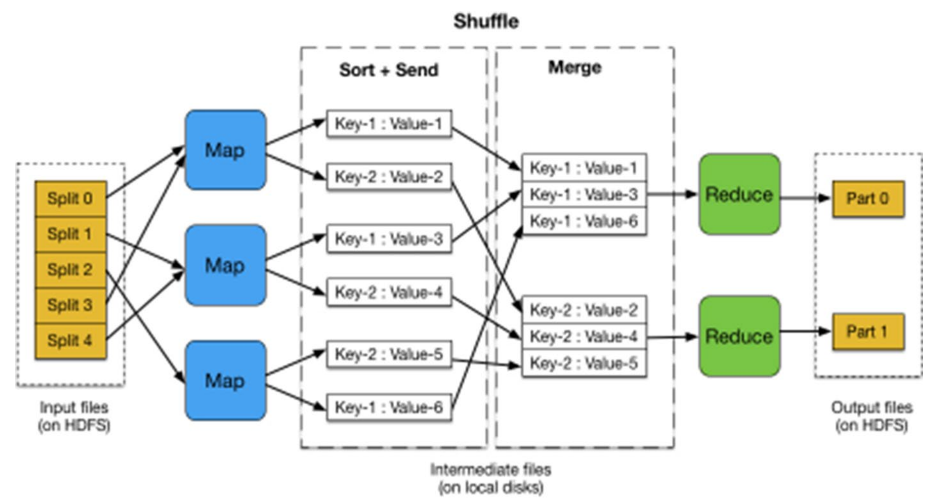

Fig. 3 The principal steps of MapReduce [30]

to know that a new reduce task should start when the next key in the sorted input data is different from the previous one. The merge phase group all intermediates input values having the same key *key* in one list and create the pair $(key, list\ of\ \langle value\rangle)$. The partitioner phase determines in which reducer a pair $(key, list\ of\ \langle value\rangle)$ will be sent. It is based on a hash function that associates and sent a pair to a reducer. In the last step, the reducers that receive the sorted $(key, list\ of\ \langle value\rangle)$ pairs can be executed simultaneously while operating on different keys, they reduce a set of intermediate values which share a key to a new smaller set of values. In our problem, each reducer emits zero, one or multiple outputs for each input $(key, list\ of\ \langle value\rangle)$ pairs.

The composition of WFSTs in MapReduce

In this section, we present three methods to perform WFSTs composition using MapReduce framework. We also study the communication cost based on Afrati et al. model [16] and analyze the replication rate for combining WFSTs.

Generic MapReduce algorithm for the composition of WFSTs

In the following, we present a general approach to perform the composition of WFSTs using a single round of MapReduce. Furthermore, we define and detail respectively the map and reduce functions.

The preprocessing phase in our algorithm store in a text file all transitions of WFSTs $(\mathcal{T}_i)_{1\leq i\leq n}$, each having m states. A transition t from a WFST $\mathcal{T}_i$ will be represented as a 4-tuple as follows : (t,type(s[t]),type(d[t]),index(t)), where *type*() is a function that maps a state to an element of the set {i (initial), f (final), if (initial and final), s (simple)} and $index(t) = i$ gives the WFST order index in the composition having the transition t.

```
input  : ⟨$key, value$⟩ pair, where $key$ represents an arbitrary instance identifier and $value$ is
         the 4-tuple form associated with the transition $t$.
output: collection of ⟨$k, t$⟩ pairs, where $k$ be an associated key with the transition $t$.
// create the transition t from the input value
1     $t \leftarrow$ getTransitionFrom($value$);
// generate the set of keys associated with the transition t
2     $setOfKeys \leftarrow$ getTransitionSetKeys($t$);
// replicate the transition t ;
3 foreach $k$ in $setOfKeys$ do
4     Emit($k$,$t$) ;
```

Algorithm 1: Map

The Map function produces a set of *key-value* pairs from each input record. In other words, the input transition is replicated and associated with all the keys generated from it based on the mapping method (line 2 of Algorithm 1). The intermediate outputs add a replication rate factor in the cost of MapReduce algorithm. The outputs from the Map function are fed into the Shuffle step. We recall that the Shuffle step occurs automatically in our implementation.

```
input  : ⟨$key, values$⟩ pair, where $values$ is a set of transitions having the same key $key$.
output: Transitions of the resulting composed WFST.
// group transitions w.r.t. their WFST index as a list of sets TransList
1 foreach transition $t$ in values do
2     add t to TransList[index(t)];
// join transitions sets TransList[i] from TransList using the cartesian
product
3 JoinedTransList = $\bigotimes_{i=1}^{n} TransList[i]$;
// compute the composition operation from JoinedTransList
4 foreach element $(t_1, \cdots, t_n)$ in JoinedTransList do
5     $t = t_1 \circ \cdots \circ t_n$;
6     Emit($null$,$t$) ;
```

Algorithm 2: Reduce

The Reduce function performs the composition of different transitions lists produced from the Shuffle step as follows: One transition is selected from each WFST such that the input symbol of WFST $\mathcal{T}_{i+1}$ coincides with the output symbol of the previous WFST $\mathcal{T}_i$. Moreover, the Reduce function group the transitions w.r.t. their WFST index in a list of sets (line 2 of Algorithm 2), and compute the Cartesian product of those sets (line 4 of Algorithm 2).

In the following, we will discuss the communication cost of the proposed algorithm according to Afrati et al. model [16].

The communication cost model

The communication cost model introduced by Afrati et al. [16] is powerful and simple. This model gives a good way to analyze problems and optimize the performance on 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 relevant 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 *list of* $\langle value \rangle$ associated with a key *key* that a reducer can receive. The global cost is the sum of the computation costs over each reducer processing all its associated values. 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 ith reducer. Let $|I|$ be the total number of different inputs, then the replication rate [16] is given by the expression $r = \sum_{i=1}^{p} q_i/|I|$.

Notice that limiting reducer size enables more parallelism. Small reducers size force us to redefine the notion of a key in order to allow more, smaller reducers and thus allow more parallelism with available nodes.

Lower bound on the replication rate

The replication rate is intended to model the communication cost, which is the total amount of information sent from the mappers to the reducers. The trade-off between reducer size q and replication rate r, is expressed through a function f, such that $r = f(q)$. The first task in designing a good MapReduce algorithm for a problem is to determine the function f, which gives us a lower bound of the replication rate r [16].

Let us now derive a tight upper bound, namely $g(q)$, on the number of outputs that can be produced by a reducer of size q for WFSTs composition. If there are n deterministic WFSTs $(\mathcal{T}_i)_{1 \leq i \leq n}$, each one having $\frac{|\mathcal{T}_i|}{|A_i|}$ transitions for each input alphabet. Let $\mathcal{T}$ be the result of the composition $\mathcal{T}_1 \circ \mathcal{T}_2 \circ \ldots \circ \mathcal{T}_n$. To compute a transition in $\mathcal{T}$, a reducer needs to receive n transitions, one from each WFST $\mathcal{T}_i$. Then, the WFST $\mathcal{T}$ can reach $\prod_{i=1}^{n} |\mathcal{T}_i|$ transitions. Assume that a reducer of size q receives $\dfrac{q}{n}$ transitions from each WFST $\mathcal{T}_i$ evenly distributed such that the input alphabet of $\mathcal{T}_{i+1}$ coincides with the output alphabet of $\mathcal{T}_i$. The following lemma gives us an upper bound on the output of a reducer.

Lemma 1 *When computing the composition* $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2 \circ \ldots \circ \mathcal{T}_n$ *a reducer of size* q *can produce no more than* $g(q) = \dfrac{q}{n}$ *outputs.*

From [16], one can compute a lower bound on the replication rate for the composition of WFSTs as a function of q using the following expression:

$$r \geq \frac{q \times |O|}{g(q) \times |I|}$$

where $|I|$ denote the input size, and $|O|$ denote the output size. The input size for our problem is the sum of transitions from all input WFSTs $\mathcal{T}_i$, that is $|I| = \sum_{i=1}^{n} |\mathcal{T}_i|$, and the size of the output is the size of $\mathcal{T}$ i.e. $|O| = |A_1| \times \frac{\prod_{i=1}^{n} |\mathcal{T}_i|}{\prod_{i=1}^{n} |A_i|}$.

Consequently, the lower bound on the replication rate for the composition of WFSTs will be as follows.

Proposition 1 *The replication rate r for the composition $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2 \circ \ldots \circ \mathcal{T}_n$ is*

$$r \geq \frac{n \times |A_1| \times \prod_{i=1}^{n} |\mathcal{T}_i|}{\sum_{i=1}^{n} |\mathcal{T}_i| \times \prod_{i=1}^{n} |A_i|}$$

Methods

This section includes the description of three mapping methods in order to design a suitable key format that maps a set of transitions to the same reducer. Explicitly, we will define the function `getTransitionFrom(t)` called in line 2 of Algorithm 1. In this section, we also present a formal analysis of the communication cost by computing an upper bound on the replication rate for each mapping method. Recall that we consider the composition of n WFSTs, each having m states.

States based mapping method

In our first method, for a transition $t \in E_i$ from a WFST $\mathcal{T}_i$, the map function generates a set of keys of the form $K_{state} = (i_1, i_2, \ldots, h(s[t]), \ldots, i_n)$, where h be a hash-function from Q_i to $\{1, \ldots, m\}$ and $i_j \in \{1, \ldots, m\}$. Consequently, the mappers produce the set of key-value pairs of the form $\langle (i_1, i_2, \ldots, h(s[t]), \ldots, i_n), t \rangle$. By way of explanation, suppose we have m^n reducers, then each transition is sent to m^{n-1} reducers.

The function $g(q)$ will be affected by the presence inside the same reducer of some transitions that cannot be combined. This give us a new upper bound on the number of outputs each reducer can produce, formally

$$g(q) = |A_1|$$

The following proposition gives an upper bound on the replication rate.

Proposition 2 *The replication rate r in the state-based mapping scheme is*

$$r \leq \frac{q \times \prod_{i=1}^{n} |\mathcal{T}_i|}{\sum_{i=1}^{n} |\mathcal{T}_i| \times \prod_{i=1}^{n} |A_i|}$$

Notice that from Propositions 1 and 2, the upper bound on replication rate exceeds the lower bound by a factor of $\frac{q}{n \times |A_1|}$. As a consequence, this mapping scheme is suitable when considering a small set of input alphabets.

Input alphabet based mapping

In the second method, transitions will be mapped to their input alphabet. Let us define the key $K_{input} = (j_1, j_2, \ldots, j_i, \ldots, j_n)$ associated with an input symbol a and let h_a be a hash-function with range $\{1, 2, \ldots, k\}$. One can associate a transition

$t = (s_i, a_i, b_i, w_i, d_i) \in E_i$ from the WFST $\mathcal{T}_i$ to a key K_{input} if there exists some $j_i = h_a(a_i)$. Thus, we have $\prod_{j=1}^{n} |A_j|$ available reducers and each transition from $\mathcal{T}_i$ is send to $\frac{\prod_{j=1}^{n} |A_j|}{|A_i|}$ reducers. Since the map task processes $\sum_{i=1}^{n} |\mathcal{T}_i|$ transitions and each WFST $\mathcal{T}_i$ have $\frac{|\mathcal{T}_i|}{|A_i|}$ transitions associated with an input symbol $c \in |A_i|$, the total number of transitions sent to each reducer is $n \times \frac{|\mathcal{T}_i|}{|A_i|}$ which can be approximated by $n \times m$. However, the function $g(q)$ is influenced by the presence of incompatible output and input symbols combination inside a reducer. This gives us a new upper bound on the number of outputs each reducer can produce, formally

$$g(q) = |Q_1|$$

Proposition 3 *The replication rate in the input alphabets based mapping scheme is*

$$r \leq \frac{q \times |A_1| \times \prod_{i=1}^{n} |\mathcal{T}_i|}{|Q_1| \times \sum_{i=1}^{n} |\mathcal{T}_i| \times \prod_{i=1}^{n} |A_i|}$$

From the Propositions 1 and 3, the upper bound for the replication rate overtake the theoretical lower bound by a factor of $\frac{q}{n \times |Q_1|}$. Therefore, the input alphabet based mapping is best suited for situations where the considered WFSTs sizes are small.

Hybrid mapping based on both input and output alphabets

In the last method, we propose a hybrid mapping based on both input and output alphabets. In other words, keys will be associated to a pair of input and output symbols. Formally, a transition $t \in E_i$ from a WFST $\mathcal{T}_i$ will be paired to a set of keys having the form $K_{hybrid} = (j_1, j_2, \ldots, h_a^i(t), h_b^o(t), \ldots, j_n)$, where $h_a^i()$ be an input symbol hash functions from $\bigcup_{i=1}^{n} A_i$ to $\{1, 2, \ldots, k\}$ and h_b^o be from be an output symbol hash functions from $\bigcup_{i=1}^{n} B_i$ to $\{1, 2, \ldots, k\}$. Transitions from $\mathcal{T}_1$ will be mapped according to their output symbols and those of $\mathcal{T}_n$ according to their input symbols. Consequently the number of reducers is k^n. However the function $g(q) = |Q_1|$.

Since transitions from $\mathcal{T}_2$ until $\mathcal{T}_{n-1}$ are sent to k^{n-2} reducers, the following Proposition holds.

Proposition 4 *The replication rate in the hybrid mapping method is strictly less than the replication rate in the input symbol mapping method.*

By comparing the Propositions 2, 3, and 4, we deduce that the upper bound of the replication rate in the hybrid mapping method is the closest one to the theoretical lower bound. Thus, this method is best suited for situations when the alphabet size is less than or equal to the number of states.

Results and discussion

This section includes extensive experiments to evaluate the efficiency and effectiveness of the proposed methods in term of communication cost and execution time for computing the composition of five WFSTs $\mathcal{T}_1 \circ \mathcal{T}_2 \circ \ldots \circ \mathcal{T}_5$. Experiments are conducted on a large variety of WFST data sets randomly generated using FAdo library [31] with various combinations of attributes including a number of states $|Q|$, input alphabet size $|A|$, and output alphabet size $|B|$.

Cluster configuration

Our experiments were run on Hadoop on the French scientific testbed Grid'5000 [32] at the site of Lille. We used for our experiments a cluster having 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 was 2.7.

Data generation method

We randomly generated a large variety of WFST data sets in two phases. First, we generated a set of deterministic finite automata using FAdo library [31], which is an open source project providing a set of tools for the symbolic manipulation of automata. FAdo is based on enumeration and generation of initially connected deterministic finite automata [33]. In the second phase, we implemented a function that randomly adds weights and some nondeterministic degree on the output symbols over transitions of the finite automata of the first phase with a uniform distribution. Our generation technique produces WFSTs based on two parameters: the number of states m and the input alphabet size k. Thus, one can define the transition density of the generated WFST as the ratio $\frac{|E|}{k \times m}$, the final state density as the ratio $\frac{|F|}{m}$ and consider a unique initial state as in [34].

Communication cost analysis

Let us evaluate the communication cost of the proposed methods on large scale WFST data sets. The communication cost is defined as the total number of key-value pairs transferred 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 gives us the relationship between the considered data set sizes and the communication cost:

Input size					**Communication cost (GB)**			**Output size**	
File size (kb)	$\sum_1^5 \|E_i\|$ $(\times 10^3)$	$\|Q_i\|$	$\|A_i\|$	$\|B_i\|$	**Hybrid mapping**	**Input alphabet mapping**	**States mapping**	$\|E\|$ $(\times 10^9)$	**Size (GB)**
132	6	75	16	16	10	22	10,699	38	1765
227	10	63	32	32	252	601	9032	31.8	1486
342	15	47	64	64	5896	14,402	4189	14.7	684
411	18	60	60	60	5465	13,327	13,327	46.7	2194

The obtained results show clearly that in general, the communication cost using the hybrid mapping method is minimal in all situations i.e. for all combinations of state sizes and alphabet sizes. This is due to the fact that the number of the transition copies sent to

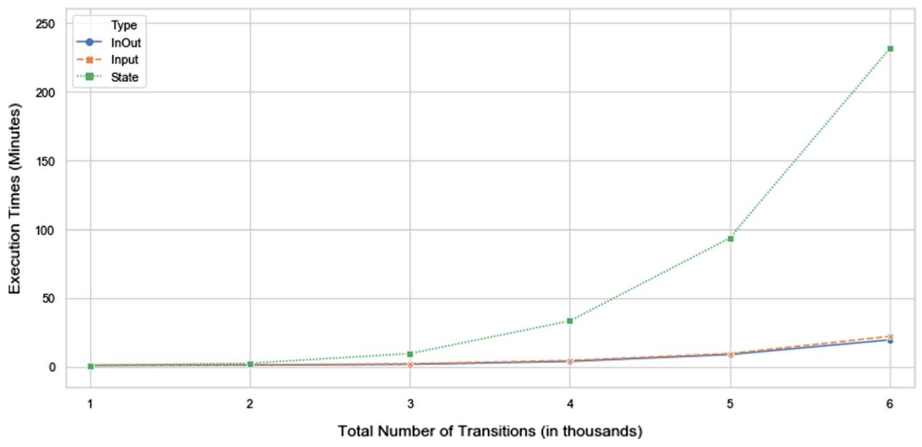

Fig. 4 Execution times of three methods for the alphabet size K = 16

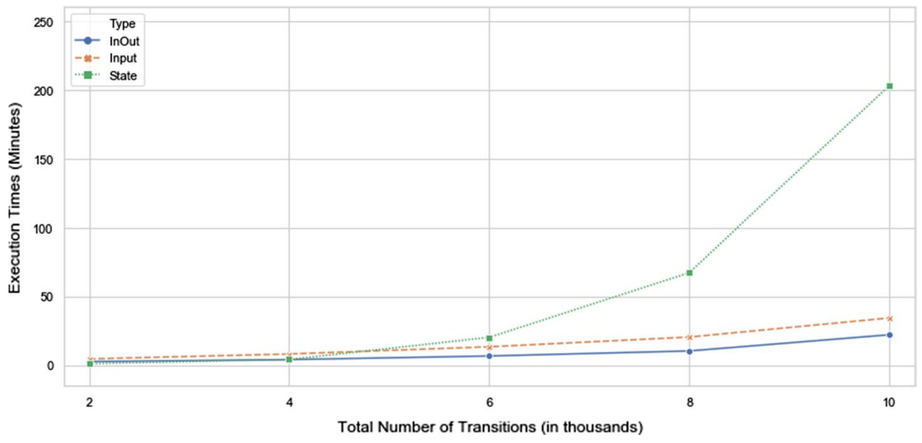

Fig. 5 Execution times of three methods for the alphabet size K = 32

the reducers with this method is less than the other ones as proved formally in Proposition 4. In some particular cases of WFSTs, when the state size is less than the alphabet size, the state based mapping has less communication cost. This coincides with the Propositions 2 and 3.

Computation cost analysis

The Computation cost is the time required to execute a MapReduce job. The graphs below, Figs. 4, 5, 6 and 7, show comparative results in term of the execution time of the three methods: states based mapping (State), input alphabet based mapping (Input), and the hybrid based mapping (InOut).

Figures 4, 5 and 6 present the execution time of the three proposed methods for different data sets sizes when varying the alphabet size to be 16, 32 or 64. In the three cases, the growth rate of hybrid and input alphabet mapping are close together and much less of states mapping. For example when K = 16, the growth rate of InOut,

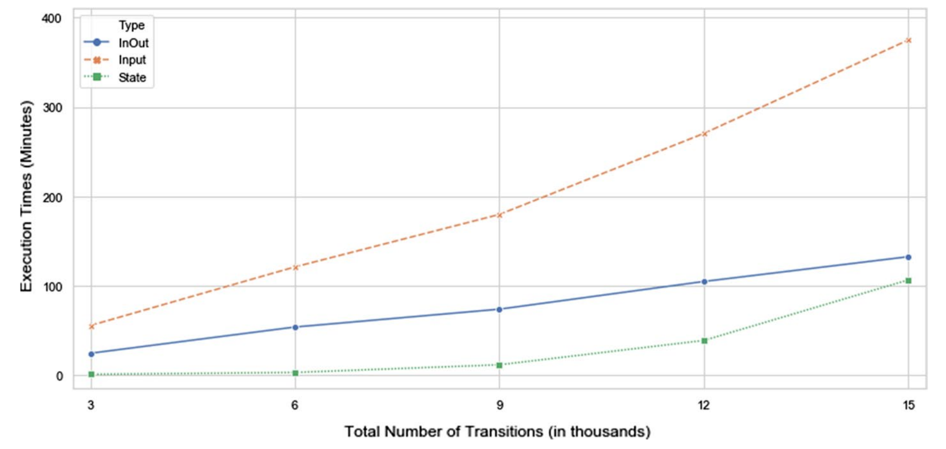

Fig. 6 Execution times of three methods for the alphabet size K = 64

Input and State are 21.06, 21.15 and 255.80, respectively. Figure 7 shows a comparison

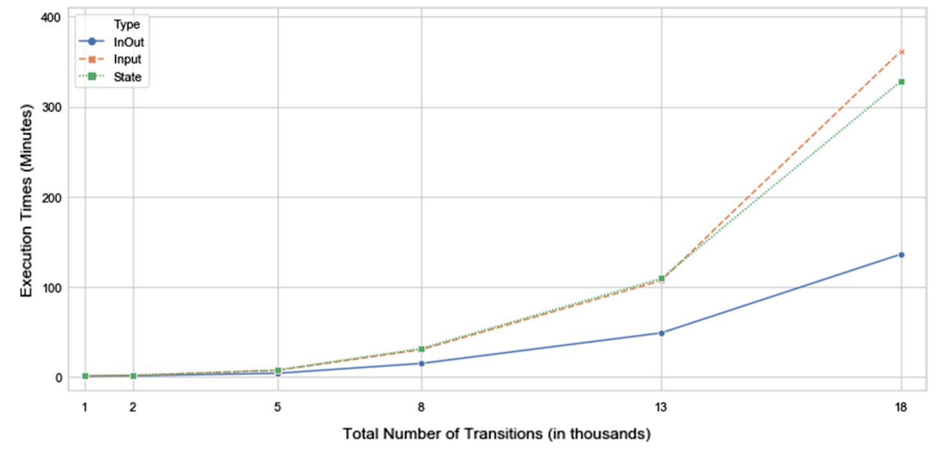

Fig. 7 Execution times of three methods when the number of states equals the alphabet size

of the three methods when the alphabet size equals the number of states. As foreseen, the hybrid method is clearly more efficient when the alphabet size is less than or equal to the number of states, and the execution time by the transition in this method is lower than two other methods.

Minimizing the replication rate decreases the time used by the mappers to replicate each transition, and avoid the existence of transitions that cannot be combined inside the same reducer. At the same time, it reduces the number of transitions that are assigned to a reducer. On the other hand, using an adequate number of reducers diminishes the waiting time a reducer spends to use a CPU.

Conclusion

In this paper, we presented a new parallel approach to compute the composition of WFSTs on a large scale in MapReduce framework. We described in detail three methods to carry out this task using a single round of MapReduce. Moreover, we analyze the communication and computation cost for each method. Finally, we evaluated the performance of the three methods on different data sets. The obtained results show that the best method in terms of the execution time is the one that minimizes the number of reducers and optimizes the inputs replication rate.

As a perspective, this work is considered as the first step to apply this method on real world problems. First target application is the design of a new distributed cryptosystem based on finite automata, the so-called finite automaton public key cryptosystem. In this application, the public key is a composition of $n + 1$ finite automata, and, the private key is the $n + 1$ weak inverse finite automata of them [4]. The second target application is in natural language processing to handle tasks such as translation between two languages, using one or multiple intermediate languages and for speech recognition. We also hope to extend and, test this method on a multi-nodes cluster environment with the GPU and OpenMP to accelerate the composition algorithm for large-scale computing.

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

Funding
Not applicable.

Availability of data and materials
Not applicable.

Ethics approval and consent to participate
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, 62100 Calais, France. [2] ANISSE Team, Department of Computer Science, Faculty of Sciences, Mohammed V University in Rabat, B.P. 1014 Rabat, Morocco.

Received: 20 October 2020 Accepted: 9 December 2020
Published online: 22 January 2021

References

1. Culik K II, Friš I. Weighted finite transducers in image processing. Discrete Appl Math. 1995;58(3):223–37.
2. Hofer J, Stemmer G. Optimizations to decoding of WFST models for automatic speech recognition. Google Patents. US Patent 10,127,902; 2018.
3. Blackwood G, De Gispert A, Brunning J, Byrne W. Large-scale statistical machine translation with weighted finite state transducers. In: Proceeding of the 2009 conference on finite-state methods and natural language processing; 2009. p. 39–49.
4. Tao R. Finite automata and application to cryptography. Berlin: Springer; 2008.
5. Roche-Lima A, Domaratzki M, Fristensky B. Pairwise rational kernels obtained by automaton operations. In: International conference on implementation and application of automata; 2014. p. 332–45.
6. Hellsten L, Roark B, Goyal P, Allauzen C, Beaufays F, Ouyang T, Riley M, Rybach D. Transliterated mobile keyboard input via weighted finite-state transducers. In: Proceedings of the 13th international conference on finite state methods and natural language processing (FSMNLP); 2017. p. 10–9.
7. Bellaouar S, Cherroun H, Nehar A, Ziadi D. Weighted automata sequence kernel. In: Proceedings of the 9th international conference on machine learning and computing; 2017. p. 48–55.

8. Huang R, OPARIN I. Applying neural network language models to weighted finite state transducers for automatic speech recognition. Google Patents. US Patent App; 2018. 10/049668.
9. Mohri M, Pereira F, Riley M. Speech recognition with weighted finite-state transducers. In: Springer handbook of speech processing. Berlin: Springer; 2008. p. 559–84.
10. Meng Z, Juang B-H. Minimum semantic error cost training of deep long short-term memory networks for topic spotting on conversational speech. In: INTERSPEECH; 2017. p. 2496–500.
11. Mohri M. Weighted automata algorithms. In: Handbook of weighted automata. Berlin: Springer; 2009. p. 213–54.
12. Wareham HT. The parameterized complexity of intersection and composition operations on sets of finite-state automata. In: International conference on implementation and application of automata; 2000. p. 302–10.
13. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. In: 6th symposium on operating system design and implementation (OSDI); 2004. p. 137–50.
14. Hennessy JL, Patterson DA. Computer architecture: a quantitative approach. Amsterdam: Elsevier; 2011.
15. Apache hadoop: Welcome to Apache Hadoop. http://hadoop.apache.org.
16. Sarma AD, Afrati FN, Salihoglu S, Ullman JD. Upper and lower bounds on the cost of a map-reduce computation. Proc VLDB Endow. 2013;6:277–88.
17. Bendre M, Manthalkar R. Time series decomposition and predictive analytics using mapreduce framework. Expert Syst Appl. 2019;116:108–20.
18. Dharayani R, Wibowo WC, Ruldeviyani Y, Gandhi A. Genomic anomaly searching with blast algorithm using mapreduce framework in big data platform. In: 2019 international workshop on big data and information security (IWBIS); 2019. p. 27–32.
19. Gao W, Zhao X, Gao Z, Zou J, Dou P, Kakadiaris IA. 3d face reconstruction from volumes of videos using a mapreduce framework. IEEE Access. 2019;7:165559–70.
20. Zhao C, Dong M, Ota K, Li J, Wu J. Edge-mapreduce-based intelligent information-centric IOV: cognitive route planning. IEEE Access. 2019;7:50549–60.
21. Grahne G, Harrafi S, Hedayati I, Moallemi A. Dfa minimization in map-reduce. In: Proceedings of the 3rd ACM SIGMOD workshop on algorithms and systems for mapreduce and beyond; 2016. p. 4.
22. Grahne G, Harrafi S, Moallemi A, Onet A. Computing NFA intersections in map-reduce. In: EDBT/ICDT workshops; 2015. p. 42–5.
23. Hopcroft JE, Ullman JD. Introduction to automata theory, languages and computation., Addison-Wesley series in computer scienceBoston: Addison-Wesley Publishing Company; 1979.
24. Sakarovitch J. Eléments de Théorie des Automates. Les Classiques de l'informatique. Vuibert; 2003.
25. Mohri M. Weighted finite-state transducer algorithms. an overview. In: Formal languages and applications. Berlin: Springer; 2004. p. 551–63.
26. Allauzen C, Mohri M. N-way composition of weighted finite-state transducers. Int J Found Comput Sci. 2009;20(04):613–27.
27. Borthakur D. The hadoop distributed file system: architecture and design. Hadoop Project Website. 2007;11.
28. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, et al. Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th annual symposium on cloud computing; 2013. p. 5.
29. Archana R, Hegadi RS, Manjunath T. A big data security using data masking methods. Indones J Electr Eng Comput Sci. 2017;7(2):449–56.
30. Lee S, Jo J-Y, Kim Y. Hadoop performance analysis model with deep data locality. Information. 2019;10(7):222.
31. Almeida A, Almeida M, Alves J, Moreira N, Reis R. Fado and guitar: tools for automata manipulation and visualization. In: International conference on implementation and application of automata; 2009. p. 65–74.
32. 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.
33. Tabakov D, Vardi MY. Experimental evaluation of classical automata constructions. In: International conference on logic for programming artificial intelligence and reasoning; 2005. p. 396–411.
34. Leslie T. Efficient approaches to subset construction. Ph.D thesis, University of Waterloo (Canada); 1995.

Publisher's Note