A Distributed Memory-Based Minimization of Large-Scale Automata

Alpha Mouhamadou Diop(✉) and Cheikh Ba

LANI - Université Gaston Berger, Saint-Louis, Senegal
{diop.alpha-mouhamadou,cheikh2.ba}@ugb.edu.sn

Abstract. Big data are data that are not able to fit in only one computer, or the calculations are not able to fit in a single computer memory, or may take impractical time. Big graphs or automata are not outdone. We describe a memory-based distributed solution of such large-scale automata minimisation. In opposition to previous solutions, founded on platforms such as MapReduce, there is a twofold contribution of our method: (1) a speedup of algorithms by the use of a memory-based system and (2) an intuitive and suitable data structure for graph representation that will greatly facilitate graph programming. A practical example is provided with details on execution. Finally, an analysis of complexity is provided. The present work is a first step of a long term objective that targets a advanced language for large graphs programming, in which the distributed aspect is hidden as well as possible.

Keywords: Big data · Distributed computing · Automata minimizing

1 Introduction

Automata are a mathematical model of computation, a simple and powerful abstract machine modeling many problems in computer science. They have been studied long before the 1970s. Automata are closed under several operations and this advantage makes them suited for a modular approach in many contexts. Applications include natural language processing, signal processing, web services choreography and orchestration, compilers, among so many others.

Determinization, intersection, and minimisation are some examples of problems linked to automata and that have been largely studied in the literature. Since one of our main objectives is to have artifacts for distributed programming, we choose to tackle the minimisation of large-scale or big automata. When it is about very large graphs (classical graphs or automata), distributed methods such as disk based parallel processing [22,23], MapReduce model [3,8,12,13,16,17] and distributed memory-based system [4,15,24] are considered.

Presently we study the innovative use of Pregel [19], a model for large-scale graph processing, to implement automata minimisation. Unlike the work we compare ourselves [12], we speed up the whole process by the use of a memory-based distributed system, and we don't need to use a counterintuitive data structure.

Y. Faye et al. (Eds.): CNRIA 2021, LNICST 400, pp. 3–14, 2021.
https://doi.org/10.1007/978-3-030-90556-9_1

Our long term goal is a high level language for distributed graph, a sort of language that hides, as well as possible, the distributed nature of the graph. In this way, the present work is one of the first steps, and it provides interesting new artefacts in addition the ones that appear in starting works [5,6].

The remaining of our paper is structured as follows: Sect. 2 presents some related works and Sect. 3 gives technical definitions and some backgrounds, that is automata and their minimisation in MapReduce, which is the work we compare ourselves. In Sect. 4 we describe our proposition, as well as some important discussions. Conclusions are shown in Sect. 5.

2 Related Works

Big data are collections of data that are too large − in relation to their processing − to be handled by classic tools. It concerns data collecting, storage, analysis, visualizing, querying and so on. Using a lot of computers for a shared storage and a parallel processing is a usual solution. Large scale graphs and automata are, of course, not outdone.

In this section we will point out two families of methods. The first one is about solutions that are built from-scratch, that is, not only the infrastructure (cluster of computers), but also its coordination. The second family concerns methods with a high level abstraction so that coordination complexity is hidden. In this way, users don't need low tasks programming skill and can only focus on functional aspects. We place ourselves in this second family of methods.

When it comes to an automaton, an important algorithm consist in obtaining its unique minimal and deterministic version. Some parallel algorithms consider shared RAM computers, using the EREW PRAM models [21] for instance. The aforementioned algorithms are applicable for a cluster of tightly coupled parallel computers with shared and heavy use of RAM. In addition, authors in [21] used a 512-processor CM-5 supermachine to minimise a *Deterministic Finite state Automaton* (DFA) with more than 500,000 states. In the case of a very large DFA, a disk storage will probably be needed. In this context, authors in [22,23] propose a parallel disk-based solution with a cluster of around thirty commodity machines to produce the unique and minimal DFA with a state reduction of more than 90%.

As presented above, the aforementioned solutions are part of the first family of methods.

Not long ago, Hadoop distributed platform [2] has been proposed by Google and has become rapidly the standard for big data processing. Its model for parallel programming is called MapReduce [9]. The objective of MapReduce is to facilitate parallel processing through only two routines: *map* and *reduce*. Data are randomly partitioned over computers and a parallel processing is done by executing the *map* and *reduce* routines on partitions. In Hadoop, different computers are connected in such a way that the complexity is hidden to end users, as if he is working with a single supercomputer. From that moment, several graph problems have been tackled by using MapReduce [3,8,16,17]: shortest

path, graph twiddling, graph partitioning, minimum spanning trees, maximal matchings and so on. In the specific case of big automata, authors in [12,13] have proposed algorithms for NFAs intersection and DFA minimisation.

Even if MapReduce paradigm can be used for many graph concerns, it's well known that it is unsuitable for iterative graph algorithms. This a consequence of the immoderate input and output with HDFS, the Hadoop distributed file system, and data shuffling at all rounds. In this way, iterative algorithms can only be written by doing one job after another, because there is no natural support for iterative tasks in MapReduce. This often leads to considerable overhead. As a consequence, many graph processing frameworks using RAM are proposed: GraphLab [18], Spark/GraphX [11], PowerGraph [10], Google's Pregel [19] and Apache Giraph [1]. A vertex centric model is followed by the majority of these frameworks. For instance, in Pregel and Giraph frameworks, which are based on BSP [25], *Bulk Synchronous Parallel*, in each iteration, each graph node (vertex) may receive messages from some other ones, process a local task, and then may send messages to some other nodes.

Nonetheless, the particular case of automata is not very well exploited by the aforementioned frameworks. In this way, we propose the use of Pregel programming model to implement big DFA minimisation. Not only we enhance the process performance compared to the MapReduce solution in [12], but also we don't need to use and maintain their counterintuitive data structure.

3 Background and Terminology

3.1 Automata and Minimization

FSA, DFA and NFA: In this section we recall two kinds of automata or FSA (*Finite State Automata*), namely *Deterministic* and *Non-deterministic Finite state Automata*, respectively abbreviated by DFA and NFA.

A DFA consists in a finite set of states with labelled and directed edges between some pairs of states. Labels or letters come from a given alphabet. From each state, there is at most one edge labeled by a given letter. So, from a given state, a transition dictated by a given letter is *deterministic*. There is an *initial* or *start state* and also certain of the states are called *final* or *accepting*. A word w is accepted by the DFA if the letters of the word can be read through transitions from the start state to a accepting state. Formally, a DFA is a 5-tuple $A = (\Sigma, Q, q_s, \delta, F)$ such that the alphabet is Σ, the set of states is denoted by Q, the initial or start state is $q_s \in Q$, and the accepting or final states is the subset $F \subseteq Q$. The *transition function* is denoted by $\delta : Q \times \Sigma \rightarrow Q$, which decides, from a state and for a letter, in which state the system will move to.

An NFA is almost the same, with the difference that, from a state, we may have more than one edge with the same letter or label. The transition corresponding to the given letter is said to be *non-deterministic*. An NFA is formally defined by a 5-tuple $A = (\Sigma, Q, q_s, \delta, F)$. It differs from a DFA in the fact that $\delta : Q \times \Sigma \rightarrow 2^Q$, with 2^Q being the power set of Q. From a state and for a letter, an NFA can move to any one of the next states in a non-deterministic way.

The set of all words accepted by a FSA A defines the *language* accepted by A. It's denoted by $L(A)$. NFAs and DFAs are equivalent in terms of accepted languages. A language L^{reg} is *regular* if and only if there exists an FSA A such that $L^{reg} = L(A)$. Any NFA A can be transformed into a DFA A^D, such that $L(A^D) = L(A)$. This transformation is called *determinization*, and can be done by the *powerset construction* method. On the basis that a DFA transition function δ takes as arguments a state and a letter and returns a state, we denote by δ^* the extended transition function that takes a state p and a string $w = a_1 a_2 \cdots a_l$ and returns the unique state $q = \delta^*(p, w) = \delta(\delta(\cdots \delta(\delta(p, a_1), a_2) \cdots, a_{l-1}), a_l)$, which is the state that the automaton reaches when starting in state p and processing the sequence of symbols in w. Thus for a DFA A, the accepted language $L(A)$ can be defined as $\{w : \delta^*(q_s, w) \in F\}$. A DFA A is described as *minimal*, if all DFAs B that accept the same language ($L(A) = L(B)$) have at least as many states as A. There is a unique *minimal* and equivalent DFA for each regular language L^{reg}.

In Fig. 1 we depict two FSAs on $\Sigma = \{a, b\}$ and that accept words that start with "a" and end with "b": a NFA A (1-**a**) and its determinization A^D (1-**b**).

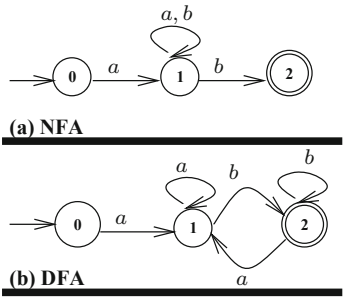


Fig. 1. An example of two FSAs.

Algorithm 1. Algorithm of Moore

Input: A DFA $A = (\{a_1, \cdots, a_k\}, Q, q_s, \delta, F)$
Output: $\pi = Q/_{\equiv}$

$i \leftarrow 0$
for all $p \in Q$ ▷ The initial partition
  if $p \in F$ **then**
    $\pi_p^i \leftarrow 1$
  else
    $\pi_p^i \leftarrow 0$
  end if
end for
repeat
  $i \leftarrow i + 1$
  for all $p \in Q$
    $\pi_p^i \leftarrow \pi_p^{i-1} \cdot \pi_{\delta(p,a_1)}^{i-1} \cdot \pi_{\delta(p,a_2)}^{i-1} \cdot \ldots \cdot \pi_{\delta(p,a_k)}^{i-1}$
  end for
until $|\pi^i| = |\pi^{i-1}|$

Minimization Algorithms: According to a taxonomy [26], the notion equivalent states is the one on which are based the majority of DFA minimisation algorithms. One exception is Brzozowski's algorithm [7] which is based on *determinization* and *reversal* of DFA. For a given DFA A, he showed that $(((A^R)^D)^R)^D$ is the minimal DFA for $L(A)$, knowing that the *reversal* of A is the NFA $A^R = (\Sigma, Q, F, \delta^R, \{q_s\})$, where $\delta^R = \{(p, a, q\}$ such that $(q, a, p) \in \delta$. In other words, if $w = a_1 \cdots a_l \in L(A)$, then $w^R = a_l \cdots a_1 \in L(A^R)$. However it is quite clear that this method is costly.

The other algorithms, like Moore's [20] and Hopcroft's [14], are based on states equivalence. Two states p and q are said to be *equivalent*, wich is denoted by $p \equiv q$, if for all strings w, it holds that $\delta^*(p,w) \in F \Leftrightarrow \delta^*(q,w) \in F$. The relation $\equiv$ is an equivalence relation, and the induced partition of the state space Q is denoted by $Q/_{\equiv}$. The *quotient* DFA $A/_{\equiv} = (\Sigma, Q/_{\equiv}, q_s/_{\equiv}, \gamma, F/_{\equiv})$, such that $\gamma(p/_{\equiv}, a) = \delta(p,a)/_{\equiv}$, is a *minimal* DFA, that is $L(A) = L(A/_{\equiv})$. One important property is that this equivalence relation $\equiv$ can iteratively be computed as a sequence $\equiv_0, \equiv_1, \cdots, \equiv_m = \equiv$, such that $p \equiv_0 q$ if $p \in F \Leftrightarrow q \in F$, and $p \equiv_{i+1} q$ if $p \equiv_i q$ and $\forall a \in \Sigma, \delta(p,a) \equiv_i \delta(q,a)$. It is known that this sequence converges in at most l iterations, l being the length of the longest simple path from the initial state to any final state.

In the following, only one states equivalence-based solution will be considered. It is about Moore's algorithm [20]. Authors in [12] proposed Algorithm 1 as an implementation of Moore's algorithm. Algorithm 1 calculates the partition $Q/_{\equiv}$ by iteratively refining the initial partition $\pi = \{F, Q \setminus F\}$. At the end of the computation, $\pi = Q/_{\equiv}$. π is considered as a mapping that assigns a string of bits π_p – as a way to identify partition blocks – to each state $p \in Q$. The number of blocks in π is denoted by $|\pi|$, and the value of π at the i^{th} iteration is denoted π^i.

3.2 DFA Minimization in MapReduce

Works in [12,13] are the unique contributions we know and that propose high level distributed platform to process automata related problems. In [12], authors describe MapReduce implementations of Moore's and Hopcroft's algorithms for DFA minimisation, as well as their analysis and experiments on several types of DFAs. As mentioned earlier, we will only focus on the algorithm of Moore, named *Moore-MR*.

We assume backgrounds on the MapReduce programming model [9], however we recall basic notions. This model is based on routines *map* and *reduce*, and the user has to implement them. The signature of `map` is $\langle K_1, V_1 \rangle \rightarrow \{\langle K_2, V_2 \rangle\}$ and the one of `reduce` is $\langle K_2, \{V_2\} \rangle \rightarrow \{\langle K_3, V_3 \rangle\}$. HDFS is used to store data, and each mapper task will handle a part of this input data. In each MapReduce round or iteration, all mappers emit a list of key-value couples $\langle K, V \rangle$. This list is then partitioned by the framework according to the values of K. All couples having the same value of K belong to the same partition $\langle K, [V_1, \cdots, V_l] \rangle$, which will be sent to the same reducer.

Moore-MR [12] consists in a preprocessing step, and several iterations of MapReduce.

Preprocessing: Since the algorithm is an repetitive refinement of the initial partition $\pi = \{F, Q \setminus F\}$, and due to the nature of MapReduce paradigm, authors have to build and maintain a set Δ from $A = (\{a_1, \cdots, a_k\}, Q, q_s, \delta, F)$. The set Δ consists in labeled transitions (p, a, q, π_p, D) such that: π_p is a string of bits representing the initial block of p, $D = +$ tells that the tuple represents a

transition, whereas $D = -$ tells that the tuple is a "dummy" transition that hold in its fourth position information of the initial block of state q. In addition, tuples $(r_i, a_{i_j}, p, \pi_p, -)$ are also in Δ, because state p new block will be required in the following round when it's come to update states $r_1, \cdots, r_m$ block annotation, where $(r_1, a_{i_1}, p), \cdots, (r_m, a_{i_m}, p)$ are transitions that lead to p.

***map* Routine:** Authors in [12] define ν as the number of reducers, and $h : Q \to \{0, \cdots, \nu - 1\}$ as a hash function. During iteration i, each mapper receives a part of Δ. For every tuple $(p, a, q, \pi_p^{i-1}, +)$, the mapper outputs key-value couple $\langle h(p), (p, a, q, \pi_p^{i-1}, +)\rangle$. And for every tuple $(p, a, q, \pi_p^{i-1}, -)$, the mapper outputs couples $\langle h(p), (p, a, q, \pi_p^{i-1}, -)\rangle$ and $\langle h(q), (p, a, q, \pi_p^{i-1}, -)\rangle$.

***reduce* Routine:** Every reducer $\rho \in \{0, \cdots, \nu - 1\}$ gets, for all $p \in Q$ such that $h(p) = \rho$, outgoing transitions $(p, a_1, q_1, \pi_p^{i-1}, +), \cdots, (p, a_k, q_k, \pi_p^{i-1}, +)$, along with "dummy" transitions $(p, a_1, q_1, \pi_{q_1}^{i-1}, -), \cdots, (p, a_k, q_k, \pi_{q_k}^{i-1}, -)$ and $(r_1, a_{i_1}, p, \pi_p^{i-1}, -), \cdots, (r_m, a_{i_m}, p, \pi_p^{i-1}, -)$.

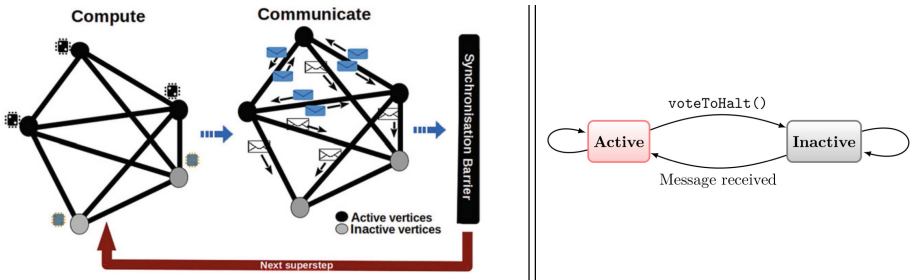
The reducer is now able to compute $\pi_p^i \leftarrow \pi_p^{i-1} \cdot \pi_{q_1}^{i-1} \cdot \pi_{q_2}^{i-1} \cdot \ldots \cdot \pi_{q_k}^{i-1}$ and write the new value π_p^i in the tuples $(p, a, q_j, \pi_p^{i-1}, +)$, for $j \in \{1, \cdots, k\}$, and $(r_j, a, p, \pi_p^{i-1}, -)$, for $j \in \{1, \cdots, m\}$, which it then returns. The reducer may return a "change" tuple $(p, true)$ as well, if the new value of π_p^i signifies the increase of the number of blocks in π_i. In this case, the algorithm need another iteration of MapReduce.

It goes without saying that it would be better if one could not have to use a structure like Δ, which is not very intuitive. Fortunately, our proposition presented in Sect. 4 will do without any extra data structure, in addition to the speedup of the whole process.

4 Our Memory-Based Approach

4.1 Pregel System

Pregel [19] is one of the first BSP (Bulk Synchronous Parallel, [25]) implementations that provides an high level API for programming graph algorithms. BSP is a model for parallel programming, and that uses MPI (message passing interface). It has been developed for scalability by parallelizing tasks over multiple computers. Apache Giraph [1] is an open-source alternative. Pregel computing paradigm is said to be "think like a vertex" as long as graph processing is done in terms of what each graph node or vertex has to process. Edges are communication means between vertices. During a superstep, a vertex can run "`compute()`", a single routine coded by the user, exchange messages with any other vertex and may change its state (active/inactive). BSP is based on synchronization barrier (Fig. 2-**a**) to ensures that all messages sent in a previous superstep will be received in the following superstep. Function `voteToHalt()` can be called by a vertex in order to be inactive during the following superstep. But, this vertex will be active if it receives a message. The Pregel process will end if at the beginning of a superstep all vertices are inactive. Figure 2-**b** shows state transition of a vertex.



(a) Basic model for graph processing **(b)** Vertex states transition

Fig. 2. About BSP model

4.2 Solution Inspired by the One of MapReduce

Given automaton $A = (\{a_1, \cdots, a_k\}, Q, q_s, \delta, F)$, we propose the `compute()` function (Algorithm 2) in order to minimise A. We remind that `compute()` will be executed by every active vertex during each superstep. And of course these vertices are distributed all over the memories of the distributed system (cluster of computers). Since the present work consider automata, Pregel's vertices represent automata states, and Pregel's edges represent automata transitions. Algorithm 2 consists in an alternation of two types of supersteps or rounds, thas is even $(0, 2, 4, \cdots)$ and odd $(1, 3, 5, \cdots)$ supersteps. New block identifiers are created in even supersteps. In order to create two rounds later the next block identifier of a state p, p has to ask for data to each of its target states q_i, to be sent in the next and odd round. Actually, odd rounds are dedicated to sending messages from target states q_i to soliciting state p.

At the start (line 2), that is the first superstep, all the automaton states are active and will divide up into the two initial blocks of the partition: the block identifier of a state p is 1 if p is final (line 4) and 0 otherwise (line 6). Apart from the first superstep, new block identifier π_p^{new} of a state p will be created depending on it previous block identifier π_p and the ones of its targets states π_{q_i} (line 13). For this purpose, state p asked for information, two steps earlier, from each target state q_i (line 16). Thus, this query is received and handled by each q_i in the previous (odd) round. In the end, p will get solicited data (through messages in M) in the present (even) round or superstep, and can extract block identifiers of each target state q_i (from line 9 to 12). The block identifier of p will be updated (line 14) in order to be sent, in the following (odd) superstep, to requesting states leading to p, that is, states for which p is a target state.

Finally, as said above, odd rounds are dedicated to providing data from target states q_i to each asking state p (line 20).

Concerning termination detection (line 24), we can be inspired by the MapReduce termination detection, or use the original Moore's algorithm termination detection. We'll consider the first one, and the second one will be discussed in Sect. 4.3.

Algorithm 2. *compute*(*vertex* p, *messages* M)

```
1: if (EVEN_SUPERSTEP) then
2:    if (superstep = 0) then
3:       if (p ∈ F) then
4:          π_p ← 1                                  {The initial bock identifier of p is 1}
5:       else
6:          π_p ← 0                                  {The initial bock identifier of p is 0}
7:       end if
8:    else
9:       for each a_i ∈ Σ do
10:         q_i ← δ(p, a_i)                         {q_i is the target state of p related to a_i}
11:         π_{q_i} ← M.get_π(q_i)                  {Get block identifiers of target states q_i}
12:      end for
13:      π_p^{new} ← π_p · π_{q_1} · π_{q_2} · . . . · π_{q_k}     {The new bock identifier π_p^i of p}
14:      π_p ← π_p^{new}                            {Updating identifier for the next round}
15:   end if
16:   sendMessage(p.ID, q_i);        {Asks for data (π_{q_i}) from each state q_i = δ(p, a_i)}
17: else
18:   {//ODD_SUPERSTEP}
19:   for each p.ID in M do
20:      sendMessage(π_p, p.ID)                     {to send message (π_p) to soliciting p}
21:   end for
22: end if
23: if (NO_NEW_BLOCK) then
24:   p.voteToHalt();
25: end if
```

The block identification proposed in [12] is sophisticated enough to "locally" detect the appearance of a new block. In this way, from a block identifier π_p, we can know if a new block was created for p. The block π_p is a bit-string consisting of $k+1$ of previous iteration. Having an increase of the different components signifies the creation of a new block. Algorithm 2 has to be completed by adding the following instruction: `"if (NEW_BLOCK) then sendToAggregator ('change')"` between lines 14 and 15. Pregel's *aggregators* enable "global" information exchange and will decide, in our case, to stop the whole process when no new block creation is detected, that is, the *aggregator* doesn't receive any *"change"* message from states. Example 1 gives running details of Algorithm 2 on a small automaton.

Example 1. Let us consider automaton $A = (\Sigma, Q, q_s, \delta, F)$, with $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_s = 0$, $F = \{3\}$ and δ presented in a state-transition table (Table 1-(a)) showing what states automaton A will move to, depending on present states and input symbols or letters.

In order to obtain A^{min}, the minimal version of automaton A, we give some details of the execution of Algorithm 2 in Table 1-(b). But only even supersteps are described. In superstep 0, states are divided up into the two initial blocks "0" and "1". State 0 belongs to block with identifier "0". Then, it will solicit data

(block identifiers) from its target states 0 and 3. In superstep 1, these states will send their block identifiers to all asking states. For instance, state 3 will send identifier "1" to state 0. In next superstep (2), state 0 will receive requested data, create the new block identifier and issue a message to indicate that a new block is detected. This process will continue until no new block creation is detected by any of the vertices. This is what happens in superstep 6 where each block identifier π_i^6 has the same number of different components as its previous identifier π_i^4. Finally, we remark that $\pi_0 = \pi_2$, and the state-transition table of A^{min} is given in Table 1-(c). □

Table 1. (a), (c) State-transition tables of A and A^{min}. **(b)** An example of a running.

(a)

δ	a	b
▶ 0	0	3
1	1	2
2	2	3
◀3	3	1

(c)

δ^{min}	a	b
▶ π_0	π_0	π_3
π_1	π_1	π_0
◀ π_3	π_3	π_1

(b)

Supersteps ⇓	π_0	π_1	π_2	π_3
0	$\pi_0^0 = 0$	$\pi_1^0 = 0$	$\pi_2^0 = 0$	$\pi_3^0 = 1$
2	$\pi_0^2 = 0.0.1$ Issues 'change'	$\pi_1^2 = 0.0.0$	$\pi_2^2 = 0.0.1$ Issues 'change'	$\pi_3^2 = 1.1.0$ Issues 'change'
4	$\pi_0^4 = 001.001.110$	$\pi_1^4 = 000.000.001$ Issues 'change'	$\pi_2^4 = 001.001.110$	$\pi_3^4 = 110.110.000$
6	$\pi_0^6 = 001001110.$ $001001110.$ 110110000	$\pi_1^6 = 000000001.$ $000000001.$ 001001110	$\pi_2^6 = 001001110.$ $001001110.$ 110110000	$\pi_3^6 = 110110000.$ $110110000.$ 000000001

4.3 Discussions

Termination Detection: As said in the previous section, we can be take our inspiration from the MapReduce termination detection, or use the original Moore's algorithm termination detection. The first kind of detection is used in Example 1. In fact, block identification proposed in [12] is sophisticated enough to "locally" detect the appearance of a new block, that is, from a local perspective or view. However it is not the case for the original termination detection for which we have to check if the number of blocks has changed from one round to another. If authors in [12] didn't find this kind of suitable block identification, they would be obliged to "globally" count, at each MapReduce round, the number of blocks. And for this purpose, and due to the nature of MapReduce paradigm, an extra round would be necessary after each functional or classic round.

Our solution (Algorithm 2) would not suffer from this concern since each vertex p just has to send its block identifier π_p to the *aggregator*; the latter will "globally" count the numbers of different block identifiers and decide whether the whole process will continue or not.

Comparative Complexities: In this section, an overview is given on complexities of the two different solutions.

Given a DFA A with n states, it is accepted that $\equiv = \equiv_{n-2}$ in the worst case [20], and thus, *Moore-MR* algorithm needs $(n-1)$ iterations in the worst case. Our Pregel solution needs $2 \times (n-1)$ supersteps. This is due to our odd supersteps devoted to sending data. Despite that, our solution is far and away faster then *Moore-MR* considering speed ratio between RAM and disk accesses. As a reminder, MapReduce model suffers from excessive input/output with HDFS (disk) and *"shuffle & sort"* at every iteration, whereas Pregel is memory-based.

Concerning communication cost, the one for *Moore-MR* is $O(k^2n^2 \log n)$ [12], specially due to their set Δ they have to maintain. Contrary to them, in each superstep, our solution causes $k \times n$ messages sending - when vertices ask for information or when data are delivered to requesting states - and the n messages sent by vertices to *aggregator*. We therefore have a cost of $O(kn^2)$ in all.

Generally, Pregel solution costs the product of the number of rounds or supersteps and the cost of a superstep. The latter is the sum of three costs: the longest execution of `compute()` function, the maximum exchange of messages, and the barrier of synchronization.

Implementation: We use Apache Giraph [1] to implement our solution. It runs on Hadoop platform and thus uses HDFS to read input data and write transformed graph (or output). Input is composed of two parts: a directed graph and a the "`compute()`" function.

We use a custom class to handle a custom read of the input graph (files) from the file system (HDFS). In these files, each vertex is represented in one line with the following structure:

```
[vrtxID, vrtxData, [[destID_1, edgeVal_1], ..., [destID_k, edgeVal_k]]]
```

`vrtxID` is the vertex ID representing a state. `vrtxData` is a data structure that will contain the initial and next block identifiers of the considered state. `destID_1` is a destination or target state ID, and `edgeVal_1` is the corresponding edge value, that is, the symbol of the corresponding transition.

At the end of the computation, the resulting automaton can be obtained by changing Q by π (the set of all blocks), q_s by π_{q_s}, F by $\{\pi_f \mid f \in F\}$ and each transition $(p, a, q) \in \delta$ by transition (π_p, a, π_q).

Perspectives: As long as our long term objectives is to find relevant programming artifacts, for a high level language for distributed graph, and with which and the distributed aspects are hidden at most, we have to find a distributed graph object, general enough to allow us to faithfully and automatically transcribe non-distributed graph algorithms. Clearly, when we consider the original Moore's algorithm, it starts with two initial blocks, and these blocks are refined (broken into sub-blocks) at every round. Unfortunately, this block object cannot be represented as a single vertex since it may not fit in a single computer RAM (one initial block can contains all the automaton vertices, for instance if all of them are final). A future work will consist of defining a logical structure over

the vertices, a kind of a graph distributed object, composed of several vertices and that may have its "`compute()`"-like function.

5 Conclusion

In this proposition, we have described and implemented a solution based on a cluster distributed memory to process big DFA minimisation. In fact, our proposition is based a model for big graph processing, namely Pregel/Giraph. Unlike the work we compare ourselves, we speed up the whole process by the use of a memory-based distributed system, and we don't need to use and maintain a counterintuitive data structure. In fact, Pregel offers an intuitive and suitable data structure for graph representation that greatly facilitates graph programming. A running example is given, as well as details on execution and complexity analysis. Finally, some important future works are described.

References

1. The Apache Software Foundation: Apache giraph. https://giraph.apache.org/
2. The Apache Software Foundation: Apache hadoop. https://hadoop.apache.org/
3. Aridhi, S., Lacomme, P., Ren, L., Vincent, B.: A mapreduce-based approach for shortest path problem in large-scale networks. Eng. Appl. Artif. Intell. **41**, 151–165 (2015)
4. Aridhi, S., Montresor, A., Velegrakis, Y.: BLADYG: a graph processing framework for large dynamic graphs. Big Data Res. **9**, 9–17 (2017)
5. Ba, C., Gueye, A.: On the distributed determinization of large NFAs. In: 2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT), pp. 1–6, October 2020
6. Ba, C., Gueye, A.: A BSP based approach for NFAs intersection. In: Qiu, M. (ed.) ICA3PP 2020. LNCS, vol. 12452, pp. 344–354. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60245-1_24
7. Brzozowski, J.: Canonical regular expressions and minimal state graphs for definite events (1962)
8. Cohen, J.: Graph twiddling in a mapreduce world. Comput. Sci. Eng. **11**(4), 29–41 (2009)
9. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
10. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: distributed graph-parallel computation on natural graphs. In: 10th USENIX OSDI 2012, Hollywood, CA, USA, 8–10 October 2012, pp. 17–30 (2012)
11. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: 11th USENIX OSDI 2014, Broomfield, CO, USA, 6–8 October 2014, pp. 599–613 (2014)
12. Grahne, G., Harrafi, S., Hedayati, I., Moallemi, A.: DFA minimization in mapreduce. In: Afrati, F.N., Sroka, J., Hidders, J. (eds.) Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2016, San Francisco, CA, USA, 1 July 2016, p. 4. ACM (2016)

13. Grahne, G., Harrafi, S., Moallemi, A., Onet, A.: Computing NFA intersections in map-reduce. In: Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference, Brussels, Belgium, 27 March 2015. CEUR Workshop Proceedings, vol. 1330, pp. 42–45 (2015)
14. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Technical Report, Stanford, CA, USA (1971)
15. Ko, S., Han, W.: Turbograph++: a scalable and fast graph analytics system. In: Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, 10–15 June 2018, pp. 395–410. ACM (2018)
16. Lattanzi, S., Mirrokni, V.S.: Distributed graph algorithmics: theory and practice. In: WSDM, pp. 419–420 (2015). http://dl.acm.org/citation.cfm?id=2697043
17. Lattanzi, S., Moseley, B., Suri, S., Vassilvitskii, S.: Filtering: a method for solving graph problems in mapreduce. In: Rajaraman, R., auf der Heide, F.M. (eds.) SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, 4–6 June 2011 (Co-located with FCRC 2011), pp. 85–94. ACM (2011)
18. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning in the cloud. PVLDB **5**(8), 716–727 (2012)
19. Malewicz, G., et al.: Pregel: a system for large-scale graph processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, 6–10 June 2010, pp. 135–146. ACM (2010)
20. Moore, E.F.: Gedanken-experiments on sequential machines. In: Shannon, C., McCarthy, J. (eds.) Automata Studies, pp. 129–153. Princeton University Press, Princeton, NJ (1956)
21. Ravikumar, B., Xiong, X.: A parallel algorithm for minimization of finite automata. In: Proceedings of IPPS 1996, The 10th International Parallel Processing Symposium, 15–19 April 1996, Honolulu, USA, pp. 187–191. IEEE Computer Society (1996)
22. Slavici, V., Kunkle, D., Cooperman, G., Linton, S.: Finding the minimal DFA of very large finite state automata with an application to token passing networks. CoRR abs/1103.5736 (2011)
23. Slavici, V., Kunkle, D., Cooperman, G., Linton, S.: An efficient programming model for memory-intensive recursive algorithms using parallel disks. In: International Symposium on Symbolic and Algebraic Computation, ISSAC 2012, Grenoble, France - 22–25 July 2012, pp. 327–334. ACM (2012)
24. Su, J., Chen, Q., Wang, Z., Ahmed, M.H.M., Li, Z.: Graphu: a unified vertex-centric parallel graph processing platform. In: 38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, 2–6 July 2018, pp. 1533–1536. IEEE Computer Society (2018)
25. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
26. Watson, B.: A taxonomy of finite automata minimization algorithms (1993)