On Large Automata Processing: Towards A High Level Distributed Graph Language

1st Alpha Mouhamadou DIOP
LANI - Université Gaston Berger
Saint-Louis, Sénégal
diop.alpha-mouhamadou@ugb.edu.sn
0000-0002-6241-6787

2nd Cheikh BA
LANI - Université Gaston Berger
Saint-Louis, Sénégal
cheikh2.ba@ugb.edu.sn
0000-0002-4515-5044

***Abstract*—Large graphs or automata have their data that can't fit in a single machine, or may take unreasonable time to be processed. We implement with MapReduce and Giraph two algorithms for intersecting and minimizing large and distributed automata. We provide some comparative analysis, and the experiment results are depicted in figures. Our work experimentally validates our propositions as long as it shows that our choice, in comparison with MapReduce one, is not only more suitable for graph oriented algorithms, but also speeds the executions up. This work is one of the first steps of a long-term goal that consists in a high level distributed graph processing language.**

***Index Terms*—Big Data, Large Graphs and Automata, Distributed Computing, MapReduce, Bsp.**

I. Introduction

With the popularity and progress of computational technologies and social networks, data in the form of graph has become omnipresent, thus widely considered for modeling in many domains such as networks (computer, road, chemical, biological, and social networks), graph databases, linked data, knowledge bases, data mining, analytics, machine learning and business intelligence.

As an example, concerning social graphs, personalized ranking or popularity have to be calculated by Facebook, as well as finding communities, determining shared connections, and make propagation of advertisement for almost one billion users. Influential vertexes for up to one trillion web pages have to be determined by PageRank Google's algorithm. In the domain of road networks and communication, processing big graphs is unavoidable in order to determine routing transportation and maximum flow. Moreover, pathology graphs help in identifying some anomalies, and biology graphs are very important to understand interactions between proteins.

Our decade is without doubt the decade of digital universe, since almost all data are available in digital form; data are generated much more than before and the growth is exponential (data explosion). This is the consequence of the fact that the Internet is ubiquitous and each human is practically data generator. Furthermore, this state of affairs will be amplified in the future with the advent of IoT (Internet of Things), in which things (fridges, cars, watches, etc.) are also connected and thus lots of data will pass from users to servers, to devices and back.

When data become too large or complex to be processed by classical applications, a common solution is the use of many computers to distribute the storage of data and to process them in parallel. From-scratch solutions have been used in the past [1]–[3], but they have rapidly been replaced by higher level large-scale systems for graph processing. The most famous one is Hadoop [4], introduced by Google, and which has become the de facto standard for processing big data, with its parallel programming model called MapReduce [5]. Since the latter is not suitable for iterative graph algorithms, in-memory frameworks such as PowerGraph [6], Google's Pregel [7], GraphLab [8] and Spark/GraphX [9] are proposed to accelerate the execution of algorithms that are iterative. In this way, many classical graph problems, such as connected components, PageRank, shortest path, and minimum spanning tree, have been solved by using these platforms [10]–[19], and sometimes accompanied by comparative studies and experiments [20]–[24]. Nevertheless, their programing paradigms are not high level enough, and thus not very intuitive in many cases.

That being said, we can now introduce our concern. Our general and long term goal is to propose a high level language for graph-like structure processing, a sort of distributed graph language that hides, the most, the distributed aspect. A program with this language would be automatically or semi-automatically translated into a lower level one that we describe above. With this aim in mind, a first step of our goal was to find relevant programming building blocks or artefacts that characterize programs. Some artefacts concerning classical graphs solutions are already considered. But, for the sake of exhaustiveness, we decided to originally find and consider artefacts of another kind of graphs, namely automata. In this way, we proposed and studied solutions for automata distributed intersection [25], minimization [26] and determinization [27], [28]. The goal of the present work is to implement and evaluate in order to experimentally validate our first step propositions. To the very best of our knowledge, ours works are the only ones that use in-memory distributed platforms to address automata related problems.

The rest of this paper is organized as follows: Section II gives some related works and Section III briefly recalls some basic notions related to graph, automata and their distributed

processing. In Section IV we present the experiments we conducted in order to experimentally validate some of our propositions, as well as the associated analysis. Conclusions are drawn in Section V.

II. Related works

We talk about big data when data sets are so large or complex that they can't be dealt with by classical and conventional applications. The treatment concerns data storage, analysis, visualization, querying, sharing and so on. As we previously said, a common solution is the use of a many computers to distribute the storage of data and to process them in parallel. Obviously, very large automata are not outdone.

Here we will point out two families of solutions regarding large graphs distributed processing, more specially for specific graphs named automata. The first family is solutions built from scratch, in which user or designer has to be an expert in distributed systems in order to settle the infrastructure and manage it. The other family concerns the use of platforms with a higher level of abstraction hiding complexity of machines coordination, resources management and allocation. Our present work is placed in this second platform model.

A lot of algorithms in automata processing exist, namely very important ones such as automata intersection, determinization or minimization. For instance, when we consider the latter, which consists in transforming a deterministic finite automaton into a smaller and equivalent one, many low level or from scratch solutions have been proposed in the literature. Shared memory computers are considered in [1], [2] for parallel algorithms. These solutions are applicable for very tightly coupled parallel computers with shared Random Access Memory and intensive use of random access. Moreover, a 512-processor CM-5 supermachine is used in [1] for the minimization of a 525,000 states DFA. When we consider a very large DFA, a distributed disk storage may be necessary. In these circumstances, a disk-based and parallel algorithm is exhibited in [3]. A 29 computers cluster is used in order to obtain a mid-range DFA containing nearly two billion states and then goes on by obtaining the equivalent and minimal DFA containing not more than 800,000 states.

The solutions mentioned above are part of the first family or lower level solutions.

Not long ago, Hadoop distributed platform [4] is introduced by Google, and it rapidly became the de facto standard for big data processing. It has a parallel programming model called MapReduce [5]. The goal is to make easier parallel processing by providing two interfaces: *map* and *reduce*. The parallel processing is achieved by partitioning data across computers and running in parallel *map* and *reduce* routines on these partitions. In Hadoop, different machines are connected and the management complexity is hidden for the user, as if he is using one big computer. Ever since then, many MapReduce solutions for classical graph problems have be proposed [10]–[13]. When we consider big automata, authors in [10], [11] offer solutions for automata intersection and minimization respectively.

Even if MapReduce programming model can address a lot of classical graph algorithms, it has been noticed and recognized that MapReduce is not suitable for iterative or repetitive graph algorithms. This is due to excessive Input/Output with the Hadoop Distributed File System - HDFS - and data shuffling at each of the iterations. Since this model does not have a natural support for repetition, the only solution is to schedule consecutive rounds, which results in very significant overhead. This is the reason why some in-memory graph frameworks have been proposed in order to accelerate the running of repetitive graph algorithms. We can cite propositions such as Google's Pregel [7], PowerGraph [6], and Spark/GraphX [9]. The majority of these frameworks use a vertex-centric programming paradigm. For instance, when we consider Pregel, which is based on Bulk Synchronous Parallel (BSP [29]), each graph node or vertex may receive messages from its in-neighbors, updates its internal state, and sends messages to out-neighbors at every round. For many classical graph problems, such as connected components, pagerank, shortest path, and minimum spanning tree, platforms comparative studies and experiments have been done [20]–[24]. The platforms in question are PowerGraph [6], Pregel/Giraph [7], Spark/GraphX [9] and GraphLab [8].

However, the aforementioned frameworks were not yet exploited for the specific case of large automata. We then initiate the novel use of BSP model for automata intersection [25], minimization [26] and determinization [27], [28]. As we said earlier, the goal of our studies is to find new interesting artefacts for our long term goal. This led at least to the additional following contributions: (1) we speed up some algorithms due to the use of in-memory platforms, contrary to solutions based on MapReduce; (2) we optimize solutions by the use of more intuitive algorithms thanks to BSP programming model which is more suitable for graph-targeted algorithms; and (3) in some cases, this suitable model helps to avoid the production of useless data (some MapReduce algorithms cannot avoid this inconvenience).

In the present work, we then implement and evaluate some of our propositions in order to have an experimental validation.

III. Preliminaries

In this section, we briefly recall basic notions related to graph and automata. Then, we do the same for distributed and parallel platforms.

A. Graphs and Automata

1) Graph: A graph $G = (V, E)$ is a structure composed of a set V of vertices and a set $E = \{(u, v) \mid u, v \in V\}$ of edges connecting pairs of vertices. The number of vertices and the numbers of edges are denoted $|V|$ and $|E|$ respectively. In some situations, it may be important to assign a label σ to edges of the graph. Depending on field of study, this label can symbolize weight, distance, symbol, etc. A *labeled graph* is a graph $G = (V, E, \delta)$ with a label function $\delta : E \to \Sigma$ associated with the set E of edges, with Σ a set of labels. Furthermore, when we consider the edges features, many graph

topologies can be distinguished: *directed* or *undirected* graphs, *multigraphs*, *hypergraphs*, among others. In the present work, we are only interested in *directed* graphs. In an *undirected* graph, the edge (u, v) from vertex u to v is the same as (or has the same meaning as) the edge (v, u) from vertex v to u.

2) Automata: An automaton or Finite state Automaton (FSA) is a directed and labeled graph, with an *input vertex* (*state*) and a set of *output* (*final*) vertices. A non formal and brief presentation of *Deterministic* (DFA) and *Non-deterministic Finite state Automaton* (NFA) will be given here. They are the two kinds of FSA that we considered in our works.

A DFA - *Deterministic Finite state Automaton* - is a tuple $(\Sigma, V, v_i, \delta, F)$ that contains a set V of vertices (states). Between pairs of vertices, we may have directed and labeled edges or transitions. The set of all possible labels (symbols) is denoted by the alphabet Σ. From each vertex, at most one outgoing transition is labeled by a label of Σ. This transition (edge) is said to be *deterministic*. We have one *initial vertex* v_i and some vertices are said to be *accepting* or *final*. These vertices belong to $F \subseteq V$. In addition, the *transition function* δ of a DFA decides, from a current vertex, which vertex the system will move to after reading or considering a label. A word (sequence of labels) is said to be accepted by a DFA if its labels correspond to transitions from v_i to a final state. The *language* accepted by a FSA A is denoted by $L(A)$. It is the set of words accepted by A. Figure 1-**(a)** shows the DFA $A_{DFA} = (\Sigma = \{a, b\}, V = \{0, 1, 2\}, v_i = 0, \delta, F = \{2\})$ such that $\delta(0, a) = 1, \delta(1, a) = 1, \delta(1, b) = 2, \delta(2, a) = 1$ and $\delta(2, b) = 2$. $L(A_{DFA})$ is the set of words beginning with label a et ending with label b. In general, the input vertex is drawn with an incoming arrow, and final vetices are double-circled.

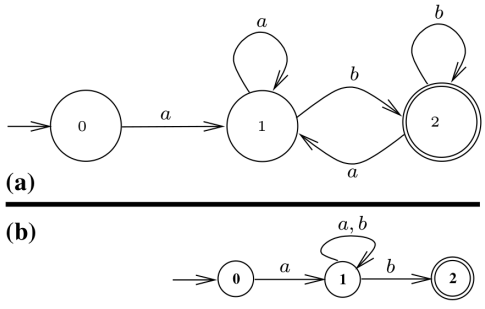


Fig. 1. **(a)** Deterministic Finite state Automaton A_{DFA}. **(b)** Non-deterministic Finite state Automaton A_{NFA}.

An NFA - *Non-deterministic Finite state Automaton* - is almost like a DFA, apart from, for a given vertex, we may have more than one outgoing transition (edge) with the same label. This transition is said to be *non-deterministic*. Figure 1-**(b)** gives the NFA $A_{NFA} = (\Sigma = \{a, b\}, V = \{0, 1, 2\}, v_i = 0, \delta, F = \{2\})$ such that $\delta(0, a) = \{1\}, \delta(1, a) = \{1\}$ and $\delta(1, b) = \{1, 2\}$. Only one transition (edge) makes A_{NFA} *non-deterministic*. It is the last one. In fact, from state 1, and for the label b, we have two outgoing transitions, and thus two target vertices 1 and 2. This NFA is *equivalent* to the DFA A_{DFA} depicted in Figure 1-**(a)** (that is, $L(A_{NFA}) = L(A_{DFA})$) since it also accepts all words that start with "a" and end with "b". The process that consists in transforming an NFA into an equivalent DFA is called *determinization*.

B. Parallel and Distributed computing

In this section, we will recall platforms and programming paradigms dedicated to parallel and distributed computing. We already talk about first, low level and from-scratch systems in related works (Section II). In this way, we'll only focus on higher level ones and briefly describe them.

1) MapReduce: MapReduce is definitely a well-known programming model when it is about processing large data. Nonetheless we will recall some features. MapReduce [5] is a Google's programming model that contains mainly two functions or routines, namely *map* and *reduce*. The user has to implement them. The signatures of these two functions are `map`: $\langle K, V\rangle \rightarrow \{\langle K', V'\rangle\}$ and `reduce`: $\langle K, \{V\}\rangle \rightarrow \{\langle K', V'\rangle\}$. The *Hadoop Distributed File System* (HDFS) ensures the storage of input data in a distributed manner, and each mapper has in charge a portion of these data. In each MapReduce round, key-value pairs $\langle K, V\rangle$ are output by mappers. These couples are automatically partitioned by the framework (*phase called shuffle*), depending on the values of K. Couples with the same value of K belong to the same group $\langle K, [V_1, \cdots, V_l]\rangle$, and this group will be received and processed by a same reducer.

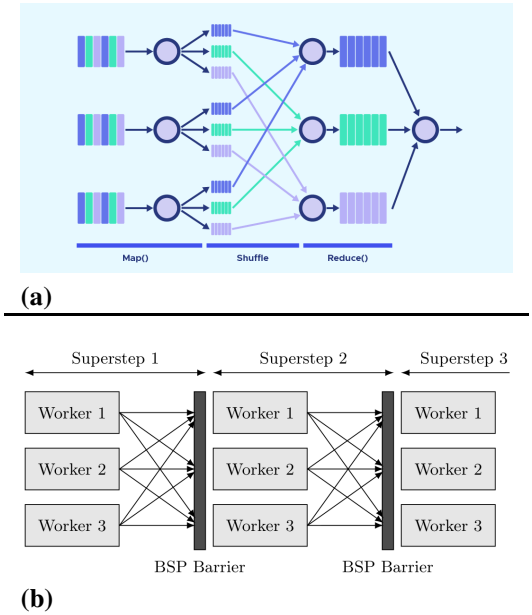


Fig. 2. **(a)** Process model of Mapreduce. **(b)** Basic computation model of BSP.

The process model of MapReduce is shown in Figure 2-**(a)**. Data (couples) with identical key are represented by rectangles of the same color. Mapper are represented by circles in the Map() phase. Seeing as we have three distinct keys (gray, green and blue rectangles), three reducers will only be needed, corresponding to the three different keys (the tree circles of the

phase of Reduce()). Data output by reducers are represented by rectangles in the Reduce() phase.

2) Memory-Based approaches: As we said earlier, MapReduce is not efficient for iterative algorithms. That is why several in-memory platforms have been proposed in order to speed up running of programs. A well-known solution is Spark [30] which can be nearly described as an "in-memory MapReduce". However, programming paradigms of MapReduce and Spark are also not suitable for graph-oriented problem. In this way, other in-memory platforms appeared that take into account the nature of graph, that is, consider the core notions of vertex and edge. We can cite for instance Spark/GraphX [9] or Pregel/Giraph [7] (based on BSP). Our work is based on this latter.

BSP (Bulk Synchronous Parallel [29]) is a parallel programming model with a message passing interface.It was proposed in order to achieve scalability by the parallelization of tasks across many workers. The model is defined as a combination of the following attributes: (i) several workers for processing computations, (ii) a router for exchanges of messages between these workers, and (iii) required *supersteps* in order to synchronize the components executions. In each *superstep*, every worker execute a task consisting in local processing, receiving and sending messages. Workers start the next superstep (barrier synchronization, Figure 2-**(b)**) just after the present one has been finished by all workers. A BSP program is a sequence of such supersteps.

One of the first implementations of BSP is Pregel [7]. It proposes a native interface expressly for programming algorithms on graph, and hides the details of communication between workers. Pregel uses a computing model said to be *"think like a vertex"*. The computation of graph is described in terms of what each graph node (vertex) has to compute; edges are communication channels between vertices. During a superstep, each active vertex run a function defined by the user and called `compute()`, may receive or send messages from or to some other vertices.The barrier of synchronization ensures that a message sent in a superstep will be available in the next superstep at its target vertices. A vertex can ask to become inactive during any superstep (function `voteToHalt()` call) and will wake up if it receives a message. When all vertices are inactive, and no other message is to be received, Pregel terminates. Remark that the implementation of Pregel provided by Google is not available publicly, but it exists other open source options, like Apache Giraph [31], that we use for our experiments.

IV. Experiments

In this section, we will present the experiments we conducted in order to experimentally validate some of our propositions. It is about automata intersection (Section IV-A) and automata minimization (Section IV-B).

For the experiments, we used a Hadoop cluster of four computers, with one *Namenode* and three *Datanodes*, running Linux Mint with Linux Kernels "5.*" and having 4 Go of memory. This cluster is rather modest but it was able to meet our goal. An overview of the characteristics is given below.

```
## Hadoop 2.4.0;  Giraph 1.1.0;  Openjdk 1.8.*
## NAMENODE ##    % Role of Master Node
   + 120 Go Hard Disk. 4 Go RAM.
   + Linux Mint 19.3 Cinnamon.
   + Linux Kernel 5.0.0-32-generic.
   + CPU Intel Core i5-3570 CPU @ 3.40GHz x 4
## DATANODES ##   % Role of Worker Nodes
   # DATA_NODE_1 & DATA_NODE_2
      + 500 Go Hard Disk. 4 Go RAM.
      + Linux Mint 20.1 Cinnamon.
      + Linux Kernel 5.4.0-58-generic.
      + CPU Intel Core i3 CPU 540 @ 3.07GHz x 2
   # DATA_NODE_3
      + 500 GoHard Disk. 4 Go RAM.
      + Linux Mint 20.1 Cinnamon.
      + Linux Kernel 5.4.0-58-generic.
      + CPU Intel Core i3-2120 CPU @ 3.30GHz x 2
```

All input data are stored in HDFS, as well as programs outputs. For iterative programs with MapReduce, we needed to write *shell (bash) scripts* in order to execute rounds one after the other, manage inputs of following rounds depending of previous outputs, and manage stop conditions.

A. Automata Intersection

Given m NFAs $A_1, \cdots, A_m$, their *intersection* is the automaton I that accepts only words accepted by all the m NFAs. In other words, the intersection is such that $L(I) = L(A_1) \cap \cdots \cap L(A_m)$. The intersection can be processed by the *Cartesian construct*. For instance, given two NFAs $A_1 = (\Sigma, V_1, v_{i1}, \delta_1, F_1)$ and $A_2 = (\Sigma, V_2, v_{i2}, \delta_2, F_2)$, an *intersection* A, such that $L(A_1) \cap L(A_2) = L(A)$, can be obtained by the *Cartesian construct* $A = A_1 \otimes A_2 = (\Sigma, V, v_i, \delta, F)$ such that $V = V_1 \times V_2$, $v_i = (v_{i1}, v_{i2})$, $F = F_1 \times F_2$ and $\delta : (V_1 \times V_2) \times \Sigma \rightarrow 2^{V_1 \times V_2}$, with $(w_1, w_2) \in \delta((v_1, v_2), \sigma)$ if and only if, for a given $\sigma \in \Sigma$, $w_1 \in \delta_1(v_1, \sigma)$ and $w_2 \in \delta_2(v_2, \sigma)$. This algorithm can be easily implemented when data are in a single computer (non distributed) and thus it will not be scalable. We focus our study on distributed solutions.

1) MapReduce-Based Solution: When we consider NFAs intersection $A_1 \otimes \cdots \otimes A_m$, 3 variants of a MapReduce solution that uses the Cartesian construct are presented in [10]. In our work, we consider and implement the mapping based on symbols. The MapReduce solution is done in such a way that there is one reducer for each of the alphabet labels or symbols. From edge (transition) (v_i, σ, w_i) of automaton A_i, a mapper will generate the key-value pair $\langle \sigma, (v_i, \sigma, w_i) \rangle$. Thus, after having received inputs $\{(v_i, \sigma, w_i)\}$ for $i = 1, \cdots, m$, the designated reducer will output transitions $\{((v_1, \cdots, v_m), \sigma, (w_1, \cdots, w_m))\}$.

Nevertheless, this solution is costly. For instance, if we have m input automata, with n states each, their method may output an NFA with around n^m vertices at least, and many of these vertices may be useless. A useless vertice is a vertex from which a final vertex can't be reached (dead vertex) or that can't be reached from the initial vertex. Deleting useless vertices

doesn't change the accepted language. For that matter, authors planned as perspective to *"investigate reducing the number of states in the product automaton, ..."* [10]. This aim is likely to be difficult or costly to meet with MapReduce model.

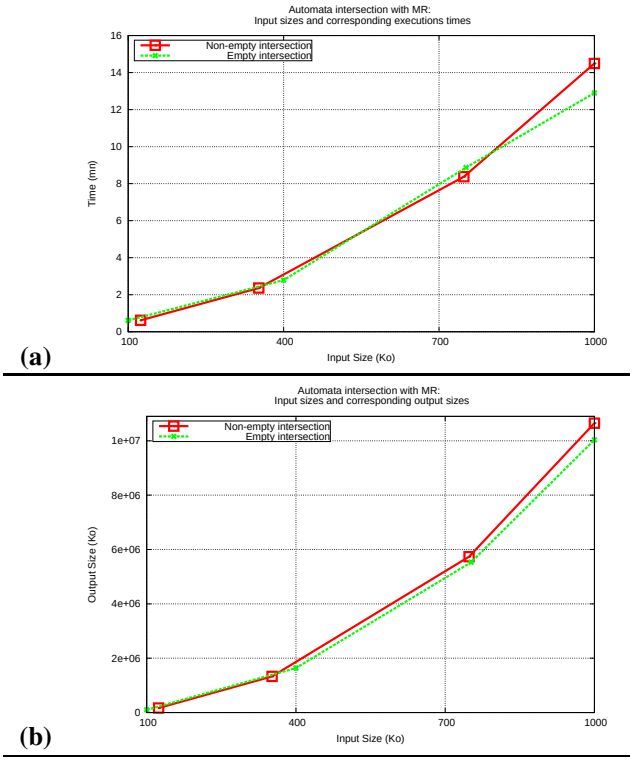


Fig. 3. Influence of input size on execution time and output size

We implement this solution on two automata with an alphabet of 4 symbols and by varying the autmata sizes. The results are depicted in Figure 3. Works in [10] only evaluate processing times for two methods by varying the alphabet size. We did the same (Figure 3-(a)) and our curve have the same look. But we also evaluate the influence of the size of input automata on output size (Figure 3-(b)).

The observations we can have are listed below:

1) An empty intersection is an intersection that leads to an empty language, that is, no word is accepted by both automata. In Figure 3, execution times are the same for empty and non-empty intersections. It is a similar state of affairs for their output sizes. This is a predictable result since the algorithm cannot avoid producing all vertices combinations, no matter if the result will be empty or not.
2) We have a growth worse than quadratic of the output size, thus as well as the execution time. We recall that if we have m input automata, with n states each, the algorithm may output an NFA with around n^m vertices at least, and many of these vertices may be useless. In our case, $m = 2$. In addition, storing states and transitions of the output automaton (intersection) costs more space than storing states and transitions of an input automaton since states of the product automaton, in our case, are pairs of input automata states. Finally, If we were varying the number of input automata, we would have an exponential growth for both execution time and output size.

TABLE I
DETAILS ON DATA CONFIGURATION RELATED TO EXPERIMENTS IN FIGURE 3.

Input size ⇓	Execution time	Number of states of 1 input FSA	Number of trans. of 1 input FSA	Output size	Number of trans. of output FSA
124 Ko	37.46 s	72	5,184	167.5 Mo	7,487,245
208 Ko	72.66 s	93	8,649	461.4 Mo	20,517,192
564 Ko	284.60 s	150	22,500	3.2 Go	137,102,413
748 Ko	502.44 s	172	29,584	5.6 Go	235,690,872
1 Mo	869.77 s	200	80,802	10.4Go	431,442,279

For the sake of completeness, we give in Table I some other details on our data configuration. The two input automata have the same configuration (number of states and transitions), but the nature of transitions are different (considering labels). Only non-empty intersection is considered since the algorithm has the same behavior as empty intersection.

2) Memory-Based Solution: In [25], we proposed a BSP-Based Approach for NFAs intersection, an in-memory approach in order to give a better alternative to the MapReduce solution proposed in [10], and whose experiments results are presented above. We recall the "compute()" function (Algorithm 1) that will be run during the system execution by every active vertex.

Unfortunately, we didn't implement Algorithm 1 for the pure and simple reason that our contribution in [25] in comparison to the works in [10] is more related to the space complexity than execution time. In fact since it is a one round MapReduce algorithm, the execution time would be almost the same as our Giraph algotihm with some supersteps, or less costly than our in-memory solution when the number of supersteps is large. Experiments would be better, even if we know that RAM access time is negligible. This will be illustrated by experiments presented in Section IV-B.

3) Analysis: The observations we can have, when we compare the MapReduce solution and our in-memory alternative are listed below:

- As we previously said, the main advantage of our solution in relation to the MapReduce proposition is the state complexity. Intrinsically, we cannot produce a vertex unreachable from the initial one, since Algorithm 1 starts

Algorithm 1 $compute(vertex\ v, messages\ m)$

if (PRODUCTION_SUPERSTEP) **then**
  if $(superstep = 0)$ **then**
    if $(v.isInitialSimpleState())$ **then**
      Let ${}^{\otimes}p_0 \leftarrow (v_{i1}, \cdots, v_{im})$;
      $createVertex({}^{\otimes}p_0)$;
      $sendMessage({}^{\otimes}p_0\,.\,\texttt{ID}, i_k)$
    end if
  else
    if $(m \neq \emptyset \wedge !v.isVisited())$ **then**
      Let ${}^{\otimes}p \leftarrow (p_1, \cdots, p_m)$;
      Let $\delta_i \leftarrow getEdges(p_i\,.\,\texttt{ID}, m)$
      Let ${}^{\otimes}\Sigma \leftarrow \bigcap_i \left\{\sigma \in \Sigma \mid \exists q_i \in Q_i : q_i \in \delta_i(p_i, \sigma)\right\}$;
      for each $\sigma \in {}^{\otimes}\Sigma$ **do**
        for each tuple $(q_1, \cdots, q_n)$ such that $q_i \in \delta_i(p_i, \sigma)$ **do**
          Let ${}^{\otimes}q \leftarrow (q_1, \cdots, q_m)$;
          $createVertex({}^{\otimes}q)$;
          $createEdge({}^{\otimes}p, \sigma, {}^{\otimes}q)$;
          $sendMessage({}^{\otimes}q\,.\,\texttt{ID}, q_i)$;
        end for
      end for
      $v.setVisited(TRUE)$;
    end if
  end if
else
  for each ${}^{\otimes}p\,.\,\texttt{ID}$ **in** m **do**
    $sendMessage(v.getEdges(), {}^{\otimes}p\,.\,\texttt{ID})$
  end for
end if
$v.voteToHalt()$;

by the initial vertex $(v_{i1}, \cdots, v_{im})$ and moves forward to the next new state only if a common label can be read.

1) The first consequence is that a lot of useless states will not be produced, contrary to the MapReduce solution.
2) The second consequence is, in the case of empty intersection, the whole process will stop as soon as the emptiness is detected; while MapReduce solution would continue be that as it may.

These positive points have been possible thanks to BSP model and the Giraph *"think like a vertex"* programming paradigm which is more suitable, intuitive and expressive in order to address graph algorithms.

B. Automata Minimization

A DFA M is said to be *minimal* if and only if all DFAs D accepting the same language ($L(M) = L(D)$) have at least as many vertices (states) as M. The process that finds the minimal DFA M from a DFA D is called *Minimization*. Based on a taxonomy given in [32], most of algorithms for the minimization of an automaton, like Hopcroft's [33] and Moore's [34], are based on the notion of equivalent classes regarding automaton states. There is one exception based on a alternating of *reversal* and *determinization*, and it is about Brzozowski's algorithm [35].

MapReduce implementations of Hopcroft's and Moore's algorithms for automaton minimization are described in [11], as well as experiments and analysis. We decided to focus only on Moore's algorithm (Algorithm 2), and then we proposed a memory-based alternative in [26], accompanied by a comparative study. In this section, we therefore present the comparative experiments we did on these two solutions.

Algorithm 2 An adaptation by [11] of the Moore's algorithm [34].

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

$i \leftarrow 0$
for all $v \in V$
  if $v \in F$ **then**
    $\pi_v^i \leftarrow 1$
  else
    $\pi_v^i \leftarrow 0$
  end if
end for
repeat
  $i \leftarrow i + 1$
  for all $v \in V$
    $\pi_v^i \leftarrow \pi_v^{i-1} \cdot \pi_{\delta(v,a_1)}^{i-1} \cdot \pi_{\delta(v,a_2)}^{i-1} \cdot \ldots \cdot \pi_{\delta(v,a_k)}^{i-1}$
  end for
until $|\pi^i| = |\pi^{i-1}|$

1) MapReduce-Based Solution: Moore's algorithm [11] is composed of a preprocessing step, and a number of MapReduce rounds. Due to the nature of MapReduce and to the fact that the algorithm is an iterative refinement of the initial equivalence class, authors had to create and maintain a data structure (set Δ) in order to help them transferring data from one round to another. In fact [7], MapReduce is fundamentally functional, thus expressing an algorithm on graph as a series of MapReduces needs passing the whole graph state from one round to the following, often requiring a lot of communication and corresponding overhead. In the present case, mappers will use part of Δ and output specific data going to reducers. The latter will then output new (identifiers of) equivalence classes and update Δ. Of course it would be much better if we could avoid using a non-intuitive set like Δ. Luckily, our memory-based solution [26] will not only avoid this extra data set, but also will speed the execution up.

Given a n-states automaton A, it is proved that in the worst case, $\equiv = \equiv_{n-2}$ [34]. So in the worst case, Moore's algorithm needs $(n-1)$ rounds.

2) Memory-Based Solution: Given an automaton $A = (\Sigma = \{a_1, \cdots, a_k\}, V, v_i, \delta, F)$, we propose in [26] a Giraph "`compute()`" function in order to minimize A. Our solution is inspired by the one of MapReduce (equivalence classes) and has the same programming style as Algorithm 1. We therefore don't put here this "`compute()`" function.

Our memory-based solution needs as many supersteps as 2 times the number of needed MapReduce rounds. In fact,

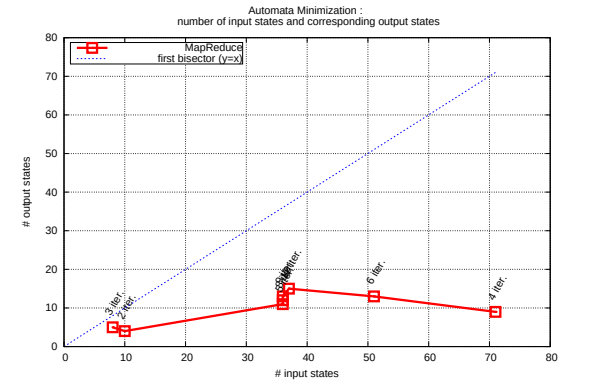
half of our supersteps are devoted to sending data. Despite this fact, our implementation is by a long way faster than the MapReduce's one when we consider speed ratio between disk and RAM accesses. We recall that MapReduce model is in pain due to excessive *"shuffle & sort"* and input/output with disk (HDFS) at each round, whereas Giraph is memory-based.

3) Analysis: We have implemented MapReduce Moore's algorithm proposed in [11], as well as our in-memory alternative presented in [26], and the comparative analysis is illustrated in Figures 4-**(a)**, 4-**(b)**, 4-**(c)** and 4-**(d)**. The experiments will be analyzed below.

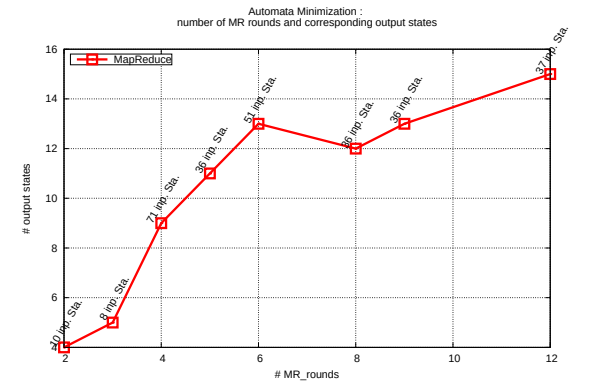
The curve in Figure 4-**(a)** shows the number of states of the input automaton and the corresponding number of states of the output (minimized) automaton. We could indifferently plot MapReduce or Giraph data since they are exactly the same: same input automaton, same algorithm, so we have the same output automaton. We draw the first bisector in order to emphasize that the input automaton is really reduced. In fact, all the curve is under this bisector and this means that the number of input automaton states is greater than the number of output automaton states. A first important remark is that (1) the curve is neither increasing nor decreasing, which means that the size of the output automaton is not proportional to the size of the input one. For instance, an automaton can be more reduced than a smaller one (a 71-states automaton is reduced to a 9-states one while a 37-states automaton is reduced to a 15-states one). In addition, we enrich the curve by adding the number of MapReduce rounds (iterations). For instance, the algorithm needs 4 iterations to reduce an input automaton from 71 states to 9 states. A second important remark is that (2) the number of required MapReduce rounds (or Giraph supersteps) is not proportional to the number of the automaton states, that is why the number of rounds is neither increasing nor decreasing depending on the number of automaton states. Remarks (1) and (2) indicate that the automaton reduction rate, as well as the number of required MapReduce rounds, depend on some other characteristics of the automaton, and not directly on its size.

The curve in Figure 4-**(b)** shows the number of MapReduce rounds and the corresponding number of states of the output (minimized) automaton. The curve is enriched by adding the numbers of input states, and this clearly shows that the automaton is really reduced. As for what we said concerning Figure 4-**(a)**, the number of output states is not proportional to the number of rounds (see the case of runs requiring 6 and 8 rounds).

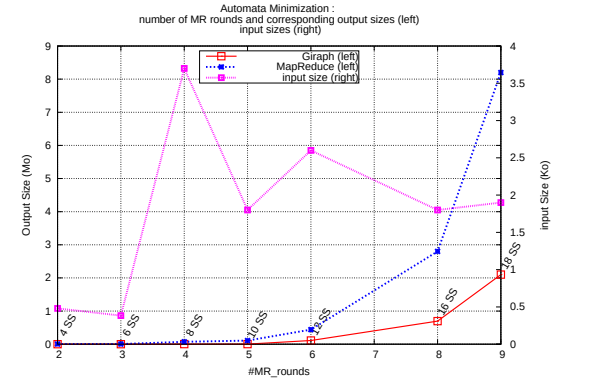
The curve in Figure 4-**(c)** shows principally the number of MapReduce rounds (as well as the number of Giraph supersteps) and the corresponding output sizes of the algorithms (left *y axis*). We recall that the number of required MapReduce rounds is two times the number of Giraph supersteps. The Giraph curve is then enriched by the number of required supersteps (SS). In addition, we also give the input size (right *y axis*) for each point of the curves. We can point out two observations. (1) Both outputs of MapReduce and Giraph exponentially increase at each round. The reason is



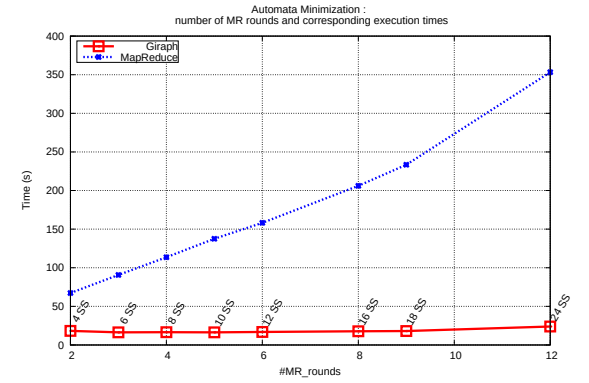
(a)



(b)



(c)



(d)

Fig. 4. Compratative minimization with MapReduce and Giraph.

that the algorithm we decided to implement (Algorithm 2, an adaptation by [11] of the Moore's algorithm [34]) uses strings of bits to identify states (equivalence classes) of the output automaton. In fact, the class identifiers π_p consist of $(k+1)$ of previous round, k being the size of the alphabet. That is why the output size has an exponential growth. We inherit this bit-string representation from the algorithm we decided to implement. (2) The growth of MapReduce outputs is by far faster than the ones of Giraph. For instance, when the Giraph output size is 2Mo, the one of MapReduce is more than 8 Mo (for times larger). And this ratio increases as a function of the number of rounds. The reason is that MapReduce solution uses a data structure Δ, and the latter also stores class identifiers π_p many times. We recall that they use Δ in order to help them transferring data from one round to another. Fortunately, our Giraph solution doesn't need such a data structure.

The curve in Figure 4-**(d)** shows the number of MapReduce rounds (as well as the number of Giraph supersteps) and the corresponding execution times of the two algorithms. Despite the fact that our in-memory solution needs two times the number of iterations of the one of MapReduce, it appears clearly that our in-memory solution is by far faster than MapReduce. For instance, for 12 MapReduce rounds (24 Giraph supersteps), MapReduce needs around 6 minutes while Giraph needs only 25 seconds. We recall that MapReduce model is in pain due to excessive *"shuffle & sort"* and input/output with disk (HDFS) at each round, whereas Giraph is memory-based.

V. Conclusion

In this work, we developed and studied some implementations in Map-Reduce and Giraph of two algorithms for intersecting NFAs and minimizing DFAs. In fact, we implement the NFAs intersection in MapReduce and compare it with our memory-based solution. We also implement, compare and analyze DFA minimization algorithms in MapReduce and giraph. We give information on our experiments environment and the results are depicted in some figures.

Generally, two important things have to be captured concerning the benefit of using in-memory and *"think like a vertex"* paradigm. First of all, the BSP paradigm is more suitable to express graph oriented algorithms. This is what allows us to avoid generating useless data in our intersection algorithm, and makes unnecessary the use and the maintenance of a quite counterintuitive data structure. Secondly, and obviously, the use of an in-memory platform has sped the execution up.

This work being done, the next step is to identify programming building blocks to use in a higher level language for distributed large graphs, considering that the distribution will be hidden at most.

References

[1] B. Ravikumar and X. Xiong, "A parallel algorithm for minimization of finite automata," in *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, April 15-19, Honolulu, USA*. IEEE Computer Society, 1996, pp. 187–191.

[2] A. Tewari, U. Srivastava, and P. Gupta, "A parallel DFA minimization algorithm," in *High Performance Computing - HiPC 2002, 9th International Conference, Bangalore, India, December 18-21*, ser. Lecture Notes in Computer Science, S. Sahni, V. K. Prasanna, and U. Shukla, Eds., vol. 2552. Springer, 2002, pp. 34–40.

[3] V. Slavici, D. Kunkle, G. Cooperman, and S. Linton, "Finding the minimal DFA of very large finite state automata with an application to token passing networks," *CoRR*, vol. abs/1103.5736, 2011.

[4] The Apache Software Foundation, "Apache hadoop," https://hadoop.apache.org/.

[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107113, Jan. 2008.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10*, C. Thekkath and A. Vahdat, Eds., 2012, pp. 17–30.

[7] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "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, June 6-10*. ACM, 2010, pp. 135–146.

[8] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[9] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8*, J. Flinn and H. Levy, Eds., 2014, pp. 599–613.

[10] G. Grahne, S. Harrafi, A. Moallemi, and A. Onet, "Computing NFA intersections in map-reduce," in *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015*, ser. CEUR Workshop Proceedings, P. M. Fischer, G. Alonso, M. Arenas, and F. Geerts, Eds., vol. 1330. CEUR-WS.org, 2015, pp. 42–45.

[11] G. Grahne, S. Harrafi, I. Hedayati, and A. Moallemi, "DFA minimization in map-reduce," in *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2016, San Francisco, CA, USA, July 1, 2016*, F. N. Afrati, J. Sroka, and J. Hidders, Eds. ACM, 2016, p. 4.

[12] S. Aridhi, P. Lacomme, L. Ren, and B. Vincent, "A mapreduce-based approach for shortest path problem in large-scale networks," *Eng. Appl. Artif. Intell.*, vol. 41, 2015.

[13] S. Lattanzi and V. S. Mirrokni, "Distributed graph algorithmics: Theory and practice," in *WSDM*, 2015, pp. 419–420. [Online]. Available: http://dl.acm.org/citation.cfm?id=2697043

[14] J. Cohen, "Graph twiddling in a mapreduce world," *Comput. Sci. Eng.*, vol. 11, no. 4, pp. 29–41, 2009.

[15] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii, "Filtering: a method for solving graph problems in mapreduce," in *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, R. Rajaraman and F. M. auf der Heide, Eds. ACM, 2011, pp. 85–94.

[16] S. N. Srirama, P. Jakovits, and E. Vainikko, "Adapting scientific computing problems to clouds using mapreduce," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 184–192, 2012.

[17] V. Slavici, "Scaling up scientific computations by using map-reduce-like control flow on numa architectures," Ph.D. dissertation, USA, 2013.

[18] L. Diop and C. Ba, "Parallelization of sequential pattern sampling," in *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*. IEEE, 2021, pp. 5882–5884. [Online]. Available: https://doi.org/10.1109/BigData52589.2021.9672071

[19] ——, "Parallélisation de l'échantillonnage de motifs séquentiels," in *Extraction et Gestion des Connaissances, EGC 2021, 25-29 Janvier 2021, Montpellier, France*, ser. RNTI, J. Azé and V. Lemaire, Eds., vol. E-37. Éditions RNTI, 2021, pp. 245–252. [Online]. Available: http://editions-rnti.fr/?inprocid=1002653

[20] X. Wang, L. Qin, L. Chang, Y. Zhang, D. Wen, and X. Lin, "Graph3s: A simple, speedy and scalable distributed graph processing system," *CoRR*, vol. abs/2003.00680, 2020. [Online]. Available: https://arxiv.org/abs/2003.00680

[21] K. Ammar and M. T. Özsu, "Experimental analysis of distributed graph systems," *PVLDB*, vol. 11, no. 10, pp. 1151–1164, 2018.

[22] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *PVLDB*, vol. 7, no. 12, pp. 1047–1058, 2014.

[23] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng, "Big graph analytics systems," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 2241–2243.

[24] J. Koch, C. L. Staudt, M. Vogel, and H. Meyerhenke, "An empirical comparison of big graph frameworks in the context of network analysis," *Social Netw. Analys. Mining*, vol. 6, no. 1, pp. 84:1–84:20, 2016.

[25] C. Ba and A. Gueye, "A BSP based approach for nfas intersection," in *Algorithms and Architectures for Parallel Processing - 20th International Conference, ICA3PP 2020, New York City, NY, USA, October 2-4, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, M. Qiu, Ed., vol. 12452. Springer, 2020, pp. 344–354. [Online]. Available: https://doi.org/10.1007/978-3-030-60245-1_24

[26] A. M. Diop and C. Ba, "A distributed memory-based minimization of large-scale automata," in *Research in Computer Science and Its Applications*, Y. Faye, A. Gueye, B. Gueye, D. Diongue, E. H. M. Nguer, and M. Ba, Eds. Cham: Springer International Publishing, 2021, pp. 3–14.

[27] C. BA and A. GUEYE, "On the distributed determinization of large nfas," in *2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*, Oct 2020, pp. 1–6.

[28] C. BA, "A comparative study of large automata distributed processing," in *2022 IEEE 16th International Conference on Application of Information and Communication Technologies (AICT)*, Oct 2022.

[29] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, S. D. Gribble and D. Katabi, Eds. USENIX Association, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[31] The Apache Software Foundation, "Apache giraph," https://giraph.apache.org/.

[32] B. Watson, "A taxonomy of finite automata minimization algorithms," 1993.

[33] J. E. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," Stanford, CA, USA, Tech. Rep., 1971.

[34] E. F. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton University Press, 1956, pp. 129–153.

[35] J. Brzozowski, "Canonical regular expressions and minimal state graphs for definite events," 1962.