# Finite Automata Algorithms in Map-Reduce

Shahab Harrafi Saveh

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 2015

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

| | |
|---|---|
| By: | **Shahab Harrafi Saveh** |
| Entitled: | **Finite Automata Algorithms in Map-Reduce** |

and submitted in partial fulfillment of the requirements for the degree of

**Master of Computer Science**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

| | |
|---|---|
| **Eusebius J. Doedel** | Chair |
| **Lata Narayanan** | Examiner |
| **Nematollaah Shiri V.** | Examiner |
| | Examiner |
| **Gösta G. Grahne** | Supervisor |

Approved **Sudhir P. Mudur**
Chair of Department or Graduate Program Director

**April 15, 2015** **Amir Asif**

Rama Bhat, Ph.D.,ing., FEIC, FCSME, FASME, Interim Dean
Faculty of Engineering and Computer Science

# Abstract

Finite Automata Algorithms in Map-Reduce

Shahab Harrafi Saveh

In this thesis the intersection of several large nondeterministic finite automata (NFA's) as well as minimization of a large deterministic finite automaton (DFA) in map-reduce are studied. We have derived a lower bound on replication rate for computing NFA intersections and provided three concrete algorithms for the problem. Our investigation of the replication rate for each of all three algorithms shows where each algorithm could be applied through detailed experiments on large datasets of finite automata. Denoting $n$ the number of states in DFA $A$, we propose an algorithm to minimize $A$ in $n$ map-reduce rounds in the worst-case. Our experiments, however, indicate that the number of rounds, in practice, is much smaller than $n$ for all DFA's we examined. In other words, this algorithm converges in $d$ iterations by computing the equivalence classes of each state, where $d$ is the diameter of the input DFA.

# Acknowledgments

Above all, I would like to express my sincere gratitude to my supervisor, Prof. Gösta Grahne, without whom the success of this thesis would not be possible. His guidance, patience, and encouragement helped me greatly throughout this endeavor. On a more personal note, my gratitude goes to my family for their continued support. I would also like to give credit to our research team members, Dr. Adrian Onet, Mr. Ali Moallemi and Mr. Iraj Hedayati for their endless and generous advice throughout this process.

# Table of Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Abbreviations

| | |
|---|---|
| DBMS | Database Management System |
| DFA | Deterministic Finite Automata |
| EREW | Exclusive Read Exclusive Write |
| FA | Finite Automata |
| GFS | Google File System |
| HDFS | Hadoop Distributed File System |
| NFA | Nondeterministic Finite Automata |
| PRAM | Parallel Random Access Machine |
| RDBMS | Relational Database Management System |
| SQL | Structured Query Language |

# Chapter 1

# Introduction

## 1.1 Problem Statement

*Finite Automata* are one of the most useful and practical devices that model a plethora of computationally oriented phenomena. They are widely used to model a variety of hardware and software such as software for designing digital circuits' behavior, compiler's lexical analyzer component, pattern matching software and so on [12]. Having only a finite number of states, makes finite automata suitable devices to be implemented in a hardware or as a program such as those mentioned above.

Finite automata are used to describe a major class of formal languages called regular languages. More precisely, a language $L$ is said to be regular, if and only if there exists a finite automaton $A$, such that $L(A) = L$. A finite automaton has a set of states and transitions as a means to consume the input by moving from one state to another state.

There are essentially two types of finite automata:

- *Deterministic Finite Automata (DFA).* A finite automaton is deterministic, if it never has a choice in its moves. In other words, there is only one state to which a machine can move on an input symbol.

- *Nondeterministic Finite Automata (NFA).* As opposed to DFA's, an NFA is a machine whose states are allowed to have any set of transitions based on a particular input symbol. Additionally, it has the ability to move from its current state without reading any input by allowing a transition on $\epsilon$, the empty string. The NFA's with $\epsilon$-transitions are called $\epsilon$-NFA's. However we note that the NFA's considered in this study are $\epsilon$-free.

We shall describe later in this section two classical problems corresponding finite automata that are mainly considered in this thesis.

### 1.1.1 NFA Intersections

One of the advantages of NFA's is that they are closed under several operations, such as concatenation, intersection, difference, and homomorphic images. This makes NFA's ideally suited for a modular approach, for instance in the context of protocol design and web service composition. A simple, but illustrative example of an e-commerce application designed from components can be found in Chapter 2 in [12]. The salient operation here is the intersection of several finite state automata.

Problems relating to NFA's have been widely studied in the literature. One of the main issues for the NFA intersection problem is that the size of the output NFA is the product of the size of all input NFA's. There is not much hope for improvement, since testing for emptiness of the intersection of a set of languages represented by NFA's is known to be PSPACE-complete [16]. The most commonly used algorithm for computing the intersection NFA is to use the *Cartesian construct* for product automata. If there are $m$ input NFA's each having $n$ states, the product NFA will have $n^m$ states. It therefore would be important to come up with good distributed algorithms for the problem.

### 1.1.2 DFA Minimization

Finite automata help answering variety of questions corresponding regular languages. Given descriptions of regular languages, one of the critical question is whether they define the same language or not. To test the equivalence of regular languages, we generally make use of finite automata defining those languages. The fact that there might be an infinite number of finite automata accepting the same language, makes the answer to this question complicated. However, this is not of great concern, since there are methods to find a DFA with the minimum number of states. This minimal DFA which is essentially unique for a certain regular language, ensures the minimum computational cost for regular languages operations such as the one discussed above.

In addition to NFA intersections problem, we shall consider the problem of minimizing a DFA $A$ to find the minimal equivalent DFA accepting the same language. There are basically two classes of states than can be removed or merged:

- *Unreachable states.* The set of states to which there are no such paths from initial state. This type of states play absolutely no role in a DFA for any input string and can be simply removed.

- *Non-distinguishable states.* Two states $p$ and $q$ in a DFA $A$ are called non-distinguishable, if for every string $w$, either processing of $A$ on $w$ from $p$ and $q$ leads to an accepting state or ends up to a non-final state. In this thesis, we particularly consider merging this class of states.

## 1.2 Motivation and Objectives

Finite automata have long been established as a significant direction of research in several areas such as theory of computation. Being one of the simplest, yet practical computational modeling devices, they are broadly used both in theory and applications. On the other hand, the rapid increase in the amount of stored and shared information results in more and more sophisticated data models over the past few years. By considering these two growing demands, we conducted a research to discover the behavior of relatively large finite automata in the map-reduce environment. We concluded that problems relating to finite automata fit nicely into the map-reduce environment. The results of the first part of this study, motivated us to proceed with the same subject. To the best of our knowledge, in this research work, the NFA intersections and the DFA minimization algorithms are the first implementations in map-reduce. The main objective of this thesis is to show the methods of developing such efficient parallel algorithms using the minimum amount of resources.

## 1.3 Contributions

This thesis covers the results published by the author in [10] (being a joint work with Prof. Gösta Grahne, Ali Moallemi and Dr. Adrian Onet). We present the main ideas of this work including the algorithms for NFA intersections and the replication rate analysis in chapter 3.

Following this work, we propose a new method to minimize the output generated in the NFA intersections problem. Accordingly, in chapter 4, we discuss the DFA minimization technique and its correctness in map-reduce environment.

Furthermore, these works are supported by several experiments to confirm the optimization. Running the detailed experiments on different scenarios, not only validates our analysis, but also shows that these methods often work more efficiently, in practice. Consequently, the experimental results for both problems will be shown in Chapter 5.

## 1.4 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 provides the necessary background information and related work. Section 2.1 briefly introduces map-reduce and its open-source implementation, Hadoop. This chapter then explores some related and important contributions that have been recently made in map-reduce. Section 2.2 explains multiway joins problem in map-reduce. Section 2.3 addresses the map-reduce computation cost model and factors such as communication cost and replication rate. In the last section of this chapter, Section 2.4, we review some of the most significant works that have been previously done in the parallel DFA minimization domain.

In Chapter 3, we investigate the problem of implementing the Cartesian construct in map-reduce. After introducing the basic technical preliminaries and definitions in Section 3.1, we follow the optimization approach of Afrati et al. [3] and analyze the amount of communication required for computing the product NFA. We first derive a lower bound for the replication rate in the product computation in Section 3.2.

We then propose three algorithms for the product computation and analyze their behaviors in Section 3.3, thereby obtaining upper bounds for the replication rate.

In Chapter 4, we study the problem of minimizing a DFA in map-reduce. We begin with essential preliminaries in Section 4.1. Afterwards, we propose the algorithm in Section 4.2. Assuming $n$ be the number of states of a DFA, we show that this algorithm terminates in at most $n$ rounds in Section 4.3. The replication rate analysis is shown in Section 4.4 and the communication cost analysis is drawn in Section 4.5.

The experimental results for both problems are shown in Chapter 5 to validate the analysis of the previous two chapters. Moreover, the cluster configuration as well as the data generation methods are described in this chapter.

Finally, Chapter 6 is dedicated to conclusion and future work. In summary, the observations made during our research are drawn in this chapter. We also suggest a few ideas for future work.

# Chapter 2

# Background and Related Work

## 2.1 Map-Reduce and Hadoop

Today's modern applications generally deal with a huge amount of information, often called "big-data", that can no longer be processed and analyzed using traditional software and tools [13]. Due to the inevitable growth of big-data, organizations nowadays have difficulties storing, analyzing and basically handling such information. To overcome these challenges, new programming systems have been gradually developed during the last decade. One new such system consists of large clusters of commodity machines connected by network, the distributed file system running on top of commodity hardware, as well as its central heart called *Map-Reduce* [19].

### 2.1.1 Map-Reduce

Map-reduce was first introduced by Google as a parallel programming model [8] that can work over large clusters of commodity computers. Map-reduce provides a high-level framework for designing and implementing such parallelism. The main intention of its implementation at Google was in fact to perform massive matrix-vector multiplications required for the PageRank calculation.

The term PageRank refers to the algorithm that was first introduced in [6] as a means to evaluate the importance of each web page by calculating its quality ranking. Talking about web pages and the Internet in general, one may consider a very large graph including tens of billions of nodes and edges. In the adjacency matrix of this graph, the element $(i, j)$ is 1, if and only if there exists a hyperlink from page $i$ to page $j$. By means of the iterative calculation, the PageRank requires very large matrix-vector multiplications for its computation in each iteration. This computation simply is the computation of the fixedpoint of a matrix-vector multiplication. Providing algorithms for both matrix-vector and matrix-matrix multiplications, chapter 2 in [19] suggests that these problem can be nicely calculated by map-reduce, while it is worth pointing out that not every problem fits into map-reduce.

Map-reduce is a simple, yet powerful and highly scalable framework which delivers parallelism by design. A user can easily take advantage of parallelism in map-reduce environment without concerning any routine complexities of traditional distributed frameworks. This ease stems from several features such as automatic failure recovery, fault tolerance, load balancing and so on. Moreover, writing only two functions known as *map* for data distribution and *reduce* for task parallelization is the only concern to be taken care of by a programmer. Aforementioned variety of advantages, has brought a lot of attentions toward this framework.

During last decade, map-reduce has also attracted researchers and academia alongside the industry. As a result, a growing number of papers deal with map-reduce algorithms for various problems, for instance related to graphs [27, 18, 7, 21, 23], and related to relational joins [4, 14, 15, 20].

We shall now provide the programming model by describing the main tasks in map-reduce framework.

#### 2.1.1.1 Map Function

The map task also known as mapper function, transfers an input data into one or more intermediate key-value pairs. Based on the key in a key-value pair which is generated by the mapper, tuples are sent into the correspondingly defined reducer. As a result, each reducer receives a set of tuples of intermediate key-value pairs.

#### 2.1.1.2 Reducer Function

The reduce task also known as reducer function, deals with a set of key-value pairs as input. The output of the reducer is also tuples of form key-value pairs. In other words, it receives a key and a list of its values and applies a set of computations over them. The output of each reducer will be written in a single file on the distributed file system.

#### 2.1.1.3 Partitioner Function

The partitioner function receives the key-value pairs generated by the mapper and partitions them into $R$ pieces over reducers. This can be done using either the default hash function or the one provided by the user, e.g., $hash(key) \mod R$, where $R$ is the number of reduce tasks.

#### 2.1.1.4 Combiner Function

The combiner task is executed locally on the same machine on which a mapper is executed. It is useful having a combiner function, if a reduce function is associative and commutative. Thus, the key-value pairs are being grouped or aggregated, before transferring into the reducers. A combiner is typically implemented using the same code written for a reducer function.

### 2.1.2 Hadoop and HDFS

#### 2.1.2.1 Hadoop

Hadoop [1] is an open-source implementation of map-reduce framework. It was created by Doug Cutting in 2005. Further information such as definitions and a comprehensive guide about Hadoop can be found in [31].

Google File System was first introduced by Google in a paper published in 2003 [9]. This publication as well as introducing map-reduce in 2004 by Google, resulted in a huge impetus on other companies to implement such a parallel and distributed computing framework. Note that Google's GFS and its map-reduce implementation are proprietary systems developed in the company for its own use.

Followed by Google's work, Apache started the Hadoop project which is best known for open-source implementation of map-reduce and its distributed file system. As an open-source software framework, Hadoop offers several libraries that allow for distributed processing of large datasets across the cluster of commodity hardware using the map-reduce programming model [1]. It mainly consists of major modules such as Hadoop distributed file system and Hadoop map-reduce as well as variety of minor modules including Hive, Pig, etc.

Hive [29] is an open-source distributed data warehousing solution which manages data stored in HDFS. It is built on top of the Hadoop and provides a query language based on SQL - *HiveQL*, for querying the data.

Pig [22] is a platform for exploring and analyzing large datasets. It consists of an execution environment and a data flow language called Pig Latin.

#### 2.1.2.2 The Hadoop Distributed File System

In order to design a file system to store very large files, Hadoop developed the Hadoop Distributed File System (HDFS) [25] running on clusters of commodity hardware [31].

HDFS is operated on top of the hardware serves as a main file system used by map-reduce. The uploaded files on HDFS are partitioned into blocks called chunks, typically of size 64 MB, although the size parameter is configurable. The chunks are replicated several times (three by default) on different racks of the clusters for fault tolerance. This makes HDFS capable of recovering from media failure. The master node known as *name-node* maintains metadata and information about all files and directories on the file system which will be used by worker nodes called *data-node*. The files are being read for further analysis and processing using map-reduce, after they are uploaded on the HDFS. Map-reduce also writes and stores the output data of each job on HDFS.

## 2.2 Multiway Joins in Map-Reduce

Queries are used to perform standard operations such as data retrieval, modification, etc. in a database management system (DBMS). In a relational database management system (RDBMS), queries are usually written in a language called structured query language (SQL). One of the most useful and practical operator in a RDBMS is the natural join. Before getting into the multiway joins problem, let us consider computing the binary natural join by map-reduce in the following section.

### 2.2.1 Natural Join

Suppose we wish to compute the natural join of two relations $R(A, B)$ and $S(B, C)$ by map-reduce. The join result includes tuples that agree on their $B$ attribute. In order to produce the tuple $(a, b, c)$, a reducer needs to receive the tuples $(a, b)$ and $(b, c)$ of $R$ and $S$, respectively.

To do so, a mapper produces a key-value pair $(b, (R, a))$ for each input tuple $(a, b)$ of $R$ and generates a key-value pair $(b, (S, c))$ for each input tuple $(b, c)$ of $S$. This means intermediate key-value pairs are grouped and partitioned by their $B$-values. More precisely, a mapper sends those tuples with the same $B$-value to identical reducers, as they are hashed based on their keys.

A reducer then receives a key $b$ with a list of values in the form of $(R, a)$ or $(S, c)$. It outputs a tuple $(a, b, c)$, if the input list of values contains $(R, a)$ and $(b, c)$. Each reducer separately writes its own join results, if any, also as key-value pairs on the distributed file system in parallel. The key is irrelevant is this problem.

### 2.2.2 Cascading Multiway Joins

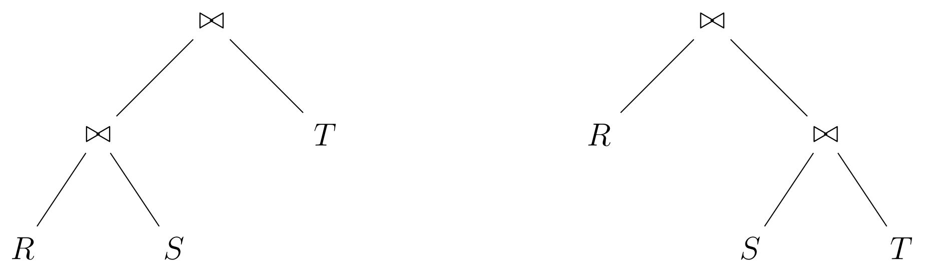
We shall now introduce the problem of computing multiway joins in map-reduce.

As discussed in [4, 19], there are basically two different viewpoints for computing this problem in map-reduce.

Let us consider computing the multiway joins of three relations $R$, $S$ and $T$ defined as $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. The first method, known as cascading, involves computing the multiway joins in two map-reduce rounds. The other approach is to come up with the result at once in a single map-reduce job which will be discussed later in this section.

To do so with the cascading method, the multiway joins problem is computed by the cascade of two natural joins, for this particular example. Figure 1(a) suggests computing $R \bowtie S$ first using $b$ buckets, whereas figure 1(b) describes starting by $S \bowtie T$ making use of $c$ buckets.

After the first round finishes, the output results of round one will be joined the by the third relation $T$ or $R$ in the second round using $c$ or $b$ reducers, respectively.



**(a)** $(R \bowtie S) \bowtie T$ **(b)** $R \bowtie (S \bowtie T)$

**Figure 1: Cascading multiway joins**

### 2.2.3 Multiway Joins using a Single Map-Reduce Job

The other method to take the multiway joins $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ is to compute it in a single map-reduce job by increasing the communication cost. In order to make this work, some of the tuples must be sent to more than one reducer. Also, the keys are in the form of pairs, rather than a single attribute.

Now, let us assume $k$ reducers are dedicated for this job. We define $k = b \cdot c$, where $b$ and $c$ are the number of buckets of $B$-values and $C$-values. The mapper function behaves differently with each tuple with respect to its relation. Using the above definition, the multiway joins problem can be computed as follows:

1. For any tuple $(b, c)$ of $S$, the mapper produces a key-value pair $\langle (h(b), h(c)), (S, (b, c)) \rangle$. It sends each $S$-tuple once to the reducer $(h(b), h(c))$.

2. For any tuple $(a, b)$ of $R$, the mapper produces key-value pairs $\langle (h(b), y), (R, (a, b)) \rangle$

for each of the $c$ possible values of $y$. It sends each $R$-tuple to the $c$ reducers.

3. For any tuple $(c, d)$ of $T$, the mapper produces key-value pairs $\langle (z, h(c)), (T, (c, d)) \rangle$ for each of the $b$ possible values of $z$. It sends each $T$-tuple to the $b$ reducers.

For every join output, there is at least one reducer that has received the required inputs from relations $R$, $S$ and $T$. Considering this fact, it is not difficult to compute the final join results in every reducer.

## 2.3 Map-Reduce Computation Cost

We shall now introduce a parameter that helps us modeling the map-reduce computation cost.

The *replication rate* is defined to be the number of key-value pairs generated by all the mapper functions, divided by the number of inputs [19]. Afrati et al. proposed a computation cost model in [3] to discover the lower bound on replication rate for problems in map-reduce. This model is based on a function of maximum number of inputs assigned to a reducer.

### 2.3.1 Mapping Schema

Let $q$ be the maximum size of a reducer. In other words, $q$ is the maximum number of inputs that can be sent to a reducer. A mapping schema is defined to be an assignment of inputs to every reducer considering the fact that no reducer can receive more than $q$ inputs. Also for every output of the problem, there should be at least one reducer that has the required inputs needed to generate that output [3].

Following the above definitions, the replication rate $r$ for an algorithm is the sum of all inputs sent to every reducer divided by the actual input size $I$. Also, let $q_i$ be the number of inputs assigned to the $i$th reducer and $p$ be the number of reducers. Thus, the replication rate $r$ is defined as:

$$r = \sum_{i=1}^{p} q_i \ / \ I \tag{1}$$

### 2.3.2 The Lower Bounds Recipe

The recipe for lower bounds offers a simple, yet powerful technique to find the lower bound on replication rate for a specific mapping schema.

As on the constraints of mapping schema, for every output there should be at least one reducer that covers that output, meaning that the reducer receives all the required inputs to generate the output. Now, let $g(q)$ be an upper bound, on the number of outputs a reducer of size $q$ can cover. Similarly, $g(q_i)$ is the number of outputs the $i$th reducer covers assuming it receives $q_i$ inputs. This can be simplified by a formula as follows:

$$\sum_{i=1}^{p} g(q_i) \geq |O| \tag{2}$$

Using equations 1 and 2, and also assuming $\frac{g(q_i)}{q_i}$ is monotonically increasing in $q_i$, we get the lower bound on replication defined as:

$$r = \frac{\sum_{i=1}^{p} q_i}{|I|} \geq \frac{q \times |O|}{g(q) \times |I|} \tag{3}$$

where $|I|$ is the input size, $|O|$ is the output size, $q$ is the reducer size, and $g(q)$ is the tight upper bound on the number of outputs a reducer of size $q$ can cover.

Taking advantage of this recipe, we found and analyzed the lower bound on replication rate for the NFA intersections problem. We also explored the upper bound for the algorithms which enabled us to compare those based on their replication rate.

## 2.4 A Short Review of Parallel DFA Minimization Algorithms

The DFA minimization problem have long been studied. John Hopcropf gave an efficient algorithm in 1971 for minimizing the number of states in a finite automaton or determining if two finite automata are equivalent [11]. The running time of his proposed algorithm was $O(kn \log(n))$ for $n$ states and some constant $k$ depending on the size of an input alphabet.

Since then there have been several investigations done on minimizing a DFA in order to improve the current solutions to this problem. Parallel DFA minimization has been further studied in [26, 24].

Bruce W. Watson in 1993 did a research on a taxonomy of finite automata minimization algorithms [30]. One the categories discussed in the paper is to find distinguishable states based on the equivalence class of every state.

Essentially, there are two different approaches for discovering the non-distinguishable states:

- *Bottom-up approach.* In this approach, the initial assumption is that every state is distinguishable from other states meaning that there are initially $n$ different equivalence classes, where $n$ is the number of states. In other words, every state belongs to its own equivalence class. Afterwards, during the refinement process, the equivalence classes are reduced and updated based on discovery of non-distinguishable states.

- *Top-down approach.* The basic assumption is this approach, however, is that all the states are equivalent or there might be at most two equivalence classes at the beginning: final states and non-final states. Gradually, more distinguishable states are discovered and thus the number of equivalence classes will be increased until there are no more new equivalence classes found.

The parallel algorithm in [24] proposed by Ravikumar and Xiong in 1996 intended to work on Exclusive Read Exclusive Write (EREW) Parallel Random Access Machine (PRAM) model (actual parallel machines) in which each memory cell can be read or written to by only one processor at a time.

The basic assumption of this algorithm was that all states are reachable. Otherwise, there should be a preprocessing step to remove unreachable states.

The original algorithm [24] from this paper is:

**Algorithm 1** Parallel DFA Minimization in SIMD Environment

**Input:** $DFA$ with $n$ states, $k$ inputs and 2 blocks $B_0$ and $B_1$.
**Output:** minimized $DFA$.

```
1: procedure ParallelMin($M$)
2:     repeat
3:         for $i = 0$ to $k - 1$ do                    ▷ Loop over a $k$-letter alphabet
4:             for $j = 0$ to $n - 1$ do                ▷ Do this loop in parallel
5:                 Get the *the block number* $B$ of state $q_j$;
6:                 Get the *the block number* $B'$ of the state $\delta(q_j, x_j)$, label the state $q_j$ with the pair $(B, B')$;
7:                 Re-arrange(e.g. by parallel sorting) the states into blocks so that the states with the same labels are contiguous;
8:             end for                                  ▷ End parallel for
9:             Assign to all states in the same block a new (unique) block number in parallel;
10:        end for
11:    until no new block produced
12: end procedure
```

As suggested by the author, the outer loop cannot be parallelized in this algorithm. However, as mentioned in this paper, there are no more than 10 loops based on the experiments done where the number of states $n$ ranged from $4,000$ to $1,000,000$. Also, there are three tasks to be done at each round:

1. The first task is to discover new equivalence classes based on an input tuples using the block number data $B$ of a state.

2. The second task is to update the block number of every state such that concatenate its block number with the block number it moves to for every alphabet symbol.

3. The last task is in fact the refinement task that is writing the updated block number of a state to disk.

We studied the DFA minimization problem in the map-reduce environment based on the equivalence class of every state using the top-down approach. In our work, there is only a single map-reduce job per round for discovering, updating and refining the equivalence classes at each step compared to the above parallel version. Later in Chapter 4, we will show that this decreases the replication rate and saves a great amount of communication cost. These two are important parameters that have to be considered in every map-reduce algorithm.

# Chapter 3

# Computing NFA Intersections in Map-Reduce

## 3.1 Preliminaries

In this section we introduce the basic technical preliminaries and definitions.

A *Nondeterministic Finite-state Automaton (NFA)* is a 5-tuple $A = (Q, \Sigma, \delta, s, F)$, where $Q$ is a finite set of states, $\Sigma$ is a finite set of alphabet symbols, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $s \in Q$ is the start state, and $F \subseteq Q$ is a set of final states. By $\Sigma^*$ we denote the set of all finite strings over $\Sigma$. Let $w = c_1 c_2 \ldots c_n$ where $c_i \in \Sigma$ be a string in $\Sigma^*$. An *accepting computation path* of $w$ in $A$ is a sequence $(s, c_1, q_1)(q_1, c_2, q_2) \ldots (q_{n-1}, c_n, f)$ of elements of $\delta$, where $s$ is the start state and

$f \in F$. The *language* accepted by $A$, denoted $L(A)$, is the set of all strings in $\Sigma^*$ for which there exists an accepting computation path in $A$. A language $L$ is regular if and only if there exists an NFA $A$ such that $L(A) = L$.

It is well-known that regular languages are closed under intersection. In particular, given NFA's $A_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$, an NFA $A$, such that $L(A) = L(A_1) \cap L(A_2)$ can be computed by the *Cartesian construct* $A = A_1 \otimes A_2$, where

$$A_1 \otimes A_2 = (Q_1 \times Q_2, \Sigma, \delta, (s_1, s_2), F_1 \times F_2),$$

and

$$\delta = \{((p_1, p_2), c, (q_1, q_2)) : (p_1, c, q_1) \in \delta_1, (p_2, c, q_2) \in \delta_2\}.$$

The $\otimes$ operation clearly is associative, and can be generalized to a polyadic operator $A_1 \otimes \cdots \otimes A_m$. The Cartesian construct amends itself easily to the map-reduce framework by having the mappers emit transitions $(p_i, c_i, q_i)$ from each NFA $A_i$, and the reducers output a transition $((p_1, \ldots, p_m), c, (q_1, \ldots, q_m))$ upon receiving inputs $(p_i, c_i, q_i)$, where $c = c_1 = \cdots = c_m$. The crucial question is how to distribute the transitions $(p_i, c_i, q_i)$ over the reducers. This is discussed in section 3.3.

## 3.2 Lower Bound on the Replication Rate

Recall that each mapper emits *key-value* pairs $(K, V)$, where $K$ determines the reducer that the pair is sent to. Each reducer receives and aggregates *key-value* lists of the form $(K, V_1, \dots V_q)$, where the $(K, V_i)$ pairs are emitted by the mappers. The largest list associated with one key is called the *reducer size*, and we will denote it by $q$. A small $q$-value ensures that the reducer can perform the aggregation in main memory, and also enables more parallelism. On the other hand, more parallelism usually increases the *replication rate*, which is the average number of key-value pairs that mappers create from one input. The replication rate is intended to model the *communication cost*, that 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 usually expressed through a function $f$, such that $r = f(q)$. The first task in designing a good map-reduce algorithm for a problem is to determine the function $f$, which gives us a lower bound of the replication rate $r$.

To start, we derive a tight upper bound, denoted $g(q)$, on the number of outputs that can be produced by a reducer of size $q$. We suppose that NFA $A_i$ has $|\delta_i|/k$ transitions for each of the $k$ alphabet symbols. To generate a transition for $A$, the reducer needs $m$ transitions, one from each NFA $A_i$. The intersection NFA $A$ has $\frac{|\delta_1| \times \cdots \times |\delta_m|}{k^m}$ transitions, for each alphabet symbol $c \in \Sigma$. As there are $k$ alphabet symbols, the total number of transitions will be $k \times \frac{|\delta_1| \times \cdots \times |\delta_m|}{k^m} = \frac{|\delta_1| \times \cdots \times |\delta_m|}{k^{m-1}}$.

It is known that the product of the elements in a partition with a fixed summation is maximum when the blocks of the partition have equal size. We therefore assume that input data is evenly distributed [2], so each reducer receives $q/m$ transitions from each NFA $A_i$. The proceeding gives us the following upper bound on the output of one reducer.

**Lemma 1** *In computing $A = A_1 \otimes \cdots \otimes A_m$ a reducer of size $q$ can cover no more than $g(q) = (q/m)^m$ outputs.*

Using Lemma 1, and the total number of transitions in $A$, we can get a lower bound on the replication rate as a function of $q$. As shown in [3] the lower bound is given by the expression

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

where $|I|$ is the size of input, and $|O|$ is the size of the output. The input size will be the sum of the size of the transition relation of all input NFA's, that is $|I| = |\delta_1| + \cdots + |\delta_m|$. As we saw above, the size of the output in terms of the number of transitions will be $|O| = \frac{|\delta_1| \times \cdots \times |\delta_m|}{k^{m-1}}$. This gives us the lower bound on replication rate for our problem as follows

**Proposition 1** *The replication rate $r$ for the Cartesian construct $A = A_1 \otimes \cdots \otimes A_m$ is*

$$r \;\geq\; \frac{q \times \frac{|\delta_1| \times \cdots \times |\delta_m|}{k^{m-1}}}{(q/m)^m \times (|\delta_1| + \cdots + |\delta_m|)}.$$

## 3.3 Algorithms for the Cartesian Construct

In this section we propose and analyze three different algorithms for computing $A = A_1 \otimes \cdots \otimes A_m$. Our algorithms compute $A$ in one map-reduce round, as opposed to an $m-1$ round cascade $(\ldots(A_1 \otimes A_2) \otimes \ldots) \otimes A_m$. Since the Cartesian construct shares features with the multiway join problem, and the latter has been shown to work more efficiently when done in one round, as opposed to a cascade [4, 14], we only consider the one-round version in this study.

We note that the main difference between the NFA intersection and the multiway join problem is that in the latter the only possibility for distributing the tuples is based on the value(s) of the join attribute(s) (corresponding to the alphabet symbols in $\Sigma$), whereas the NFA intersection problem we can also distribute the tuples of the transition relation based on the states they involve.

### 3.3.1 Mapping Based on States

Suppose we have $n^m$ reducers, where $n$ is the maximum number of transitions in any of the input NFA's. In our first algorithm the mappers produce keys of the form $(i_1, i_2, \ldots, i_m)$. Let $h$ be a hash-function with range $\{1, \ldots, n\}$. A transition $(p_i, c_i, q_i)$ from NFA $A_i$ is mapped as key-value pairs $(K, (p_i, c_i, q_i))$, where

$$K = (i_1, \ldots, i_{i-1}, h(p_i), i_{i+1}, \ldots, i_m).$$

for each $i_j \in \{1, \ldots, n\}$. In other words, each transition is sent to $n^{m-1}$ reducers.

In this method, the input and output sizes remain unchanged. However, the function $g(q)$ will be affected by presence of transitions with different alphabet symbols inside a single reducer. This gives us a new upper bound on the number of outputs each reducer can produce, namely $g(q) = k\,(q/mk)^m$. We thus have

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

$$r \leq \frac{q \times |\delta_1| \times \cdots \times |\delta_m|}{(q/m)^m \times (|\delta_1| + \cdots + |\delta_m|)}.$$

*If $n$ is the maximum number of transitions in any of the input NFA's, the upper bound on the replication rate becomes $r \leq (\frac{nm}{q})^{m-1}$.*

By comparing propositions 1 and 2, we observe that the upper bound for the replication rate obtained by mapping based on states exceeds the theoretical lower bound by a factor of $k^{m-1}$. We conclude that the state-based mapping approach is best suited for situations where the alphabet size is small, e.g., when the alphabet is binary.

## 3.3.2 Mapping Based on Alphabet Symbols

In our second algorithm, we have one reducer for each of the alphabet symbols. Thus, the number of reducers is equal to the alphabet size $k$. The mappers will send each transition $(p, c, q)$ to the reducer corresponding the alphabet symbol $c$. More precisely, from transition $(p_i, c, q_i)$ of NFA $A_i$ the mapper will generate the key-value pair $(h(c), (p_i, c, q_i))$. Here $h$ is a hash function with range $\{1, \ldots, k\}$. Thus each reducer will output transition $((p_1, \ldots, p_m), c, (q_1, \ldots, q_m))$, having received inputs $(p_i, c, q_i)$ for $i = 1, \ldots, m$.

The total number of transitions sent to all reducers is $\sum_{i=1}^{m} |\delta_i|$ which we approximate by $mn$, assuming that each $A_i$ has at most $n$ transitions. The replication rate is 1, since every transition is mapped to exactly one reducer. This algorithm works well when the alphabet size $k$ is large and the number of reducers is equal to the number of alphabet symbols. In summary:

**Proposition 3** *The replication rate in the alphabet-symbol based mapping scheme is 1, assuming that the number of reducers and alphabet symbols are the same.*

Obviously a replication rate of 1 is optimal. This matches the lower bound of Proposition 1, when observing that each reducer has to process $(nm)/k$ inputs, assuming that the alphabet symbols are uniformly distributed. Substituting $q = (nm)/k$ in the lower bound $(\frac{nm}{kq})^{m-1}$ of Proposition 1, gives $r \geq 1$.

### 3.3.3 Mapping Based on Both States and Alphabet Symbols

On one hand, if we map the transitions only based on the alphabet symbols, the algorithm does not allow for much parallelism if the alphabet $\Sigma$ is small. On the other hand, as we have observed, if the transitions are mapped based on states only, the replication rate, and consequently the communication cost, will be sharply increased $k^{m-1}$ times. We therefore consider a hybrid algorithm that maps transitions based on a combination of alphabet symbols and states. In the hybrid method we have a function $h_s$ that hashes states into $b_s$ buckets, and a function $h_a$ that hashes the alphabet symbols into $b_a$ buckets. A transition $(p_i, c_i, q_i)$ from $A_i$ is mapped to reducers $(i_1, \ldots, i_{i-1}, h_s(p_i), i_{i+1}, \ldots, i_m, h_a(c_i))$, for each $i_j \in \{1, \ldots, b_s\}$, and the total number of reducers will be $b_s^{m-1} \cdot b_a$.

To compute the replication rate in this method, we note the input and output sizes $|I|$ and $|O|$ remain unchanged. However, the function $g(q)$ will be affected by presence of transitions with different alphabet symbols inside a single reducer. We will now have $g(q) = \ell(q/m\ell)^m$, where $\ell$ is the average number of alphabet symbols received by a reducer, or equivalently, $\ell = k/b_a$. From this we can derive the replication rate.

**Proposition 4** *The replication rate $r$ in the hybrid mapping scheme is*

$$r \leq \frac{q \times \frac{|\delta_1| \times \cdots \times |\delta_m|}{k^{m-1}}}{(q/m)^m \times (|\delta_1| + \cdots + |\delta_m|)} \times \ell^{m-1}.$$

*Assuming that the maximum number of transitions in any of the input NFA's is $n$, we get $r \leq \left(\frac{nm\ell}{qk}\right)^{m-1}$.*

Note that if $b_a = 1$ then $\ell = k$ and there is no hashing on alphabet symbols, and as it can be seen, the replication rate will be equal to the replication rate of the first mapping schema. On the other hand, if $b_a = k$, that is if we hash fully on alphabet symbols, then $\ell = 1$ and as it can be seen, the replication rate will be equal to the replication rate of the second mapping schema.

# Chapter 4

# DFA Minimization in Map-Reduce

## 4.1 Preliminaries

A DFA is 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where $Q$ is a finite set of states, $\Sigma$ is a finite set of input symbols (alphabet), $\delta$ is the transition function $Q \times \Sigma \rightarrow Q$, $q_0 \in Q$ is the start state and $F \subseteq Q$ is a set of accept states. Additionally, we assume that $|Q| = n$, $|\delta| = m$ and $|\Sigma| = k$.

## 4.2 Map-Reduce Algorithm

In this section we propose an algorithm for minimizing a DFA using the map-reduce programming paradigm. As we know, DFA minimization inherently is an iterative problem and can be computed in several iterations where each step is a map-reduce job. We utilized the equivalence classes to find the non-distinguishable states in every round using the top-down approach, i.e. separating the states that are not equivalent. The process converges at some point where no more equivalence classes are generated.

Let us assume that a transition $\delta(p, a) = q$ in DFA $M$ is a tuple $t = (p, a, q, c, \Delta)$ in a relation $\delta$. $\Delta \in \{+1, -1\}$ indicates that a tuple is a transition or a reverse transition. More specifically, a tuple $(p, a, q, c, +1)$ shows that $\delta(p, a) = q$. However, a tuple $(p, a, q, c, -1)$ demonstrates that $\delta(q, a) = p$. Furthermore, we need to keep track of equivalence class of each state. To do so, we define $c$ to be an equivalence class for state $p$ in tuple $t$. We note that the equivalence class $c$ is represented by a bit vector.

### 4.2.1 Mapper Function

Suppose we have $n$ reducers where $n$ is the number of states in the input DFA. As we discussed earlier the mapper's input data are transitions in the form $t = (p, a, q, c, \Delta)$. The mappers send each transition to the reducer corresponding to the state $p$ and each reverse transition to the reducers corresponding to the state $p$ and state $q$. In other words, if $\delta(p, a) = q$, then $\Delta = +1$ and the mapper will generate a key-value pair $(K, V)$ from the input transition, where

$$K = h(p) \text{ and } V = (p, a, q, c, \Delta).$$

Otherwise, if $\delta(q,a) = p$, then $\Delta = -1$ and the mapper will generate $(K_1, V)$ and $(K_2, V)$, where

$$K_1 = h(p), K_2 = h(q) \text{ and } V = (p, a, q, c, \Delta).$$

In order to determine the equivalence class of state $p$, it is essential to know the equivalence class of state $q$, for which $\delta(p,a) = q$. This is why the mapper generates key-value pair $(K_2, V)$. On the other hand, since the equivalence class of the reverse transition needs to be updated, the mapper also generates key-value pair $(K_1, V)$. It is worth mentioning that because each self-loop transition carries the equivalence class of both source and destination states, there is no need to produce $(K_1, V)$ and $(K_2, V)$ key-value pairs. Clearly, it will decrease the communication cost by a factor $\frac{2}{3}\alpha$, where $\alpha$ is the probability of number of self-loop transitions.

Recall that the problem's input is just a set of transitions of initial DFA. Thus, in the first round, the mapper computes the reverse transitions from the input data, and then emits three key-value pairs $(K_1, V_1), (K_1, V_2)$ and $(K_2, V_2)$ per transition where

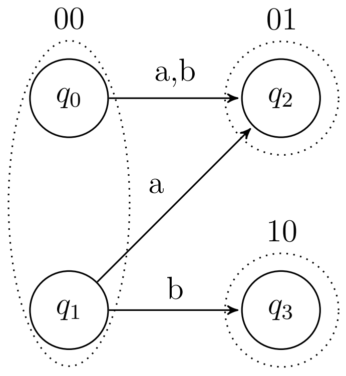
$$K_1 = h(p) \text{ and } K_2 = h(q),$$

$$V_1 = (p, a, q, c, +1) \text{ and } V_2 = (p, a, q, c, -1).$$

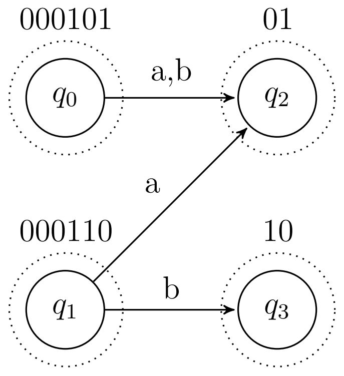
### 4.2.2 Reducer Function

There are two distinct tasks to be carried out inside the reducers, at each round. The first task is to determine equivalence class of every state based on the source and destination class of all its outgoing transitions. We present how to achieve this with the help of the following example. Figure 2(a) represents part of a DFA. States $q_0$ and $q_1$ belong to equivalence class 00, whereas states $q_2$ and $q_3$ belong to equivalence class 01 and 10, respectively. Considering $\delta(q_0, a) = q_2$ and $\delta(q_1, a) = q_2$, $q_0$ and $q_1$ are in the same equivalence class based on alphabet symbol $a$, say 0001. On the other hand, $\delta(q_0, b) = q_2$ and $\delta(q_1, b) = q_3$ imply that $q_0$ and $q_1$ should be in different equivalence classes 0001 and 0010 according to alphabet symbol $b$. Consequently, in order to distinguish classes of these two states, all alphabet symbols should be taken into account. In summary, $q_0$ belongs to class 000101 and $q_1$ belongs to class 000110. The updated classification for states $q_0$ and $q_1$ is shown in figure 2(b).

The above procedure is coded in lines 11 to 18 of algorithm 2. As discussed, there are three sorts of transitions in every reducer. The first loop updates equivalence class of each transition $t_i \in \delta$ using class of destination state. This can be obtained by $s$, the reverse transition of $t_i$ (line 16). Line 14 covers the self-loop transition. Afterwards, the equivalence class of state $p$ in round $r$ will be updated to $c_p^{r+1} = c_p^r \ c_{q_1}^r \ c_{q2}^r \ \ldots \ c_{q_k}^r$, where $c_{q_i}^r$ is the equivalence class of $q_i$ such that $q_i = \delta(p, a_i)$ in all transitions $t \in \delta$. Also, if there exists a state $d$ such that $\delta(d, a_i) = p$ for some $a_i \in \Sigma$, state $d$ will require the equivalence class of state $p$ in the next round. Thus, the class of $p$ has to be updated using $c_p^{r+1}$ in all reverse transitions that goes to state $p$.



**(a)** Before round $r$



**(b)** After round $r$

**Figure 2: Classification example**

## 4.3 Convergence

The algorithm will be stopped as soon as the equivalence classes of the states of the DFA converges. The result will be the minimal DFA.

Let $EQ$ be set of equivalence classes. Then $|EQ|$ would be the number of equivalence classes. The equivalence class of state $p$, $EQ_p$ is a bit-string of the form

$$EQ_0EQ_1 \dots EQ_w : \exists \delta(p, a) = q_i, 0 \leq i \leq |Q|, w = |\Sigma| \tag{4}$$

which indicates that state $p$ belongs to which class. As mentioned before,

$$|EQ_i| = |\Sigma|^r$$

where $r$ is map-reduce round number. Thus, each $EQ_i$ in (4) has length $|\Sigma|^{r-1}$.

In order to check if the algorithm has been converged or not, we check whether any new equivalence class has been generated or not. To do so, in each reducer we check the number of equivalence classes individually before and after the operation.

**Generate set of equivalence classes before reducer operations** Create set $EQ'$ consisting of substrings of $EQ_p$ each of which is of length $|\Sigma|^{r-2}$.

**Generate set of equivalence classes after reducer operations** Create set $EQ''$ consisting substrings of $EQ_p$ each of which is of length $|\Sigma|^{r-1}$.

**Check if any new class has been generated** If $|EQ'| \neq |EQ''|$, then $\exists q_i : \delta(p, a) = q_i$ and $q_i$ moved to new equivalence class and $C_{p_i} = false$.

The algorithm has converged if $\bigwedge_{0 \leq i \leq n} C_{p_i}$ is true.

**Algorithm 2** DFA Minimization Map-Reduce Algorithm

**Input:** A set of DFA transitions in the form of $t = (p, a, q, c, \Delta)$

**Output:** The minimal DFA

```
1: repeat

2:     function MAPPER(t)
3:         if $\Delta = +1$ then                                    ▷ $\delta(p, a) = q$
4:             Emit $(h(p), (p, a, q, c, +1))$
5:         else if $\Delta = -1$ and $p \neq q$ then        ▷ $\delta(q, a) = p$ and it is not a self-loop transition
6:             Emit $(h(q), (p, a, q, c, -1))$
7:             Emit $(h(p), (p, a, q, c, -1))$
8:         end if
9:     end function

10:    function REDUCER($\langle\, K, [t_1, t_2, \ldots, t_n]\, \rangle$)
11:        $c \leftarrow null$
12:        for all transition $\delta(p, a) = q$ do                ▷ $\Delta = +1$
13:            if $p_i = q_i$ then
14:                $c \leftarrow c\ c_i$
15:            else
16:                $c \leftarrow c\ c_s$ such that $s = {t_i}^{-1}$
17:            end if
18:        end for
19:        Update $c$ in all transitions $\delta(p, a) = q$
20:            Emit $(K, (p, a, q, c, +1))$
21:        Update $c$ in all transitions $\delta(q, a) = p$ and $q = K$
22:            Emit $(K, (p, a, q, c, -1))$
23:        if equivalence class of state $K$ changes then
24:            Emit $(K, true)$
25:        end if
26:    end function

27: until no new equivalence class produced

28: merge the non-distinguishable states and produce the minimal DFA
```

## 4.4 Replication Rate Analysis

In this section we analyze the replication rate of the DFA minimization algorithm. As discussed earlier in Section 4.2, the mapper function generates one key-value pair for every transition $\delta(p, a) = q$ and two key-value pairs for every reverse transition $\delta(q, a) = p$. More precisely, the mapper generates three key-value pairs for each transition and its reverse transition. This clearly shows that every two tuples is replicated exactly three times during the minimization process. As a result, the replication rate for this problem is $r = \frac{3}{2}$.

## 4.5 Communication Cost Analysis

In this section we analyze the communication cost of the DFA minimization algorithm. Recall that the communication cost is the input data plus the key-value pairs produced by the mapper function. We suppose that $|I| = 2m$, where $m$ is the number of transitions. Although every transition is replicated at most three times, each reducer emits at most two key-value pairs per transition at each round. Therefore, the communication cost is $|I| + \frac{3}{2}|I|$ per round. Simply, the total communication cost would be $\frac{5}{2}|I| \times d = 5md$, where d is the diameter of the graph of the input DFA.

# Chapter 5

# Experimental Results

In this chapter, we provide details of the experiments we conducted to validate the analysis of the previous two chapters.

## 5.1 Cluster Configuration

Our experiments were run on Hadoop on a 2-node, personal computer, cluster (8 cores per node running at 3.0 GHz and 24 GB memory in total). The number of reducers in the experiments was set to 128. The desktops were running Scientific Linux operating system with kernel version 6.0. The Hadoop version installed on both machine was 2.4.0.

## 5.2 NFA Intersections

### 5.2.1 Data Generation Method

We shall now briefly introduce the random model for classical automaton construction by Tabakov and Vardi [28]. Let $A = (Q, \Sigma, \delta, s, F)$ be an NFA. Then for each alphabet symbol $a \in \Sigma$, the random model generates a random directed graph $G_a$ on $Q$ with $k$ edges, corresponding to the transition $(p, a, q)$. The transition density is defined as the ratio $\frac{k}{|Q|}$. Similarly, the final state density is defined as the ratio $\frac{|F|}{|Q|}$. Finally, the transitions generated for every alphabet symbol are merged to form a final NFA.

Consequently, the NFA's were generated as labelled random graphs according to the above random model. The total number of transitions were determined by the *transition density*, that is, the ratio between the number of transitions and the number of states. In the data shown we used a transition density of 2.0 and a final state density of 0.2.

### 5.2.2 Experiments

We computed $A_1 \otimes A_2 \otimes A_3$, and varied the size of the NFA's and number of alphabet symbols.

In the experiments we compared the execution time obtained by hashing the input data based on states (States) and on both states and alphabet symbols (Hybrid).

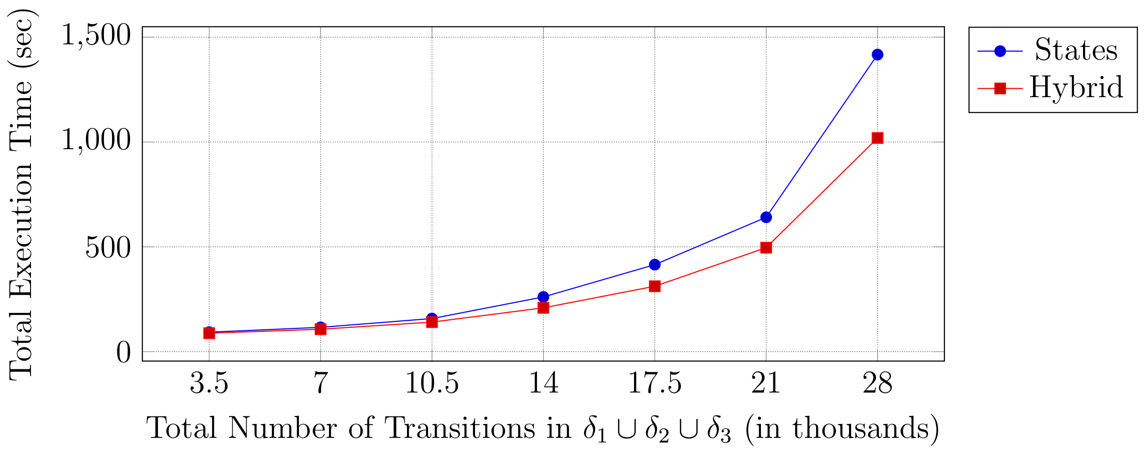


**Figure 3: Processing times of two methods for the alphabet size $k = 16$**

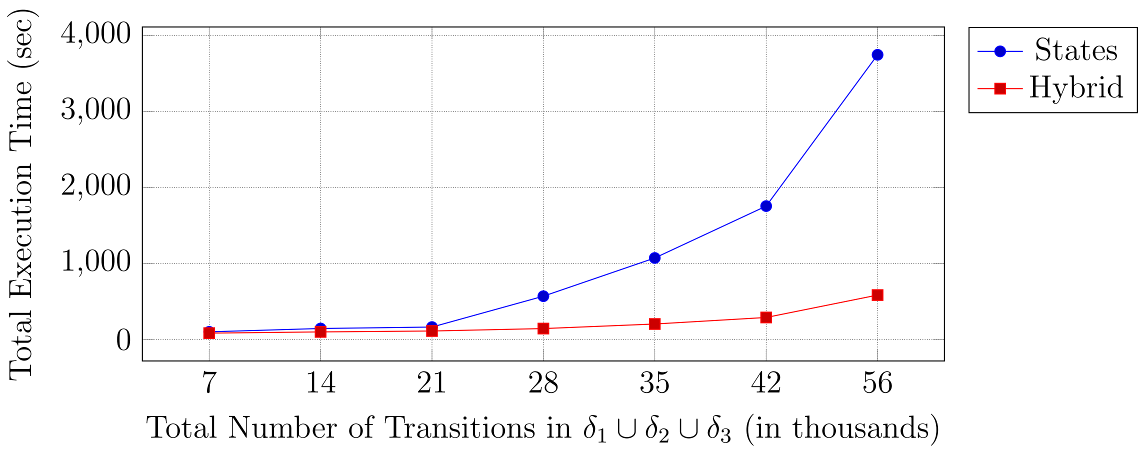


**Figure 4: Processing times of two methods for the alphabet size $k = 64$**

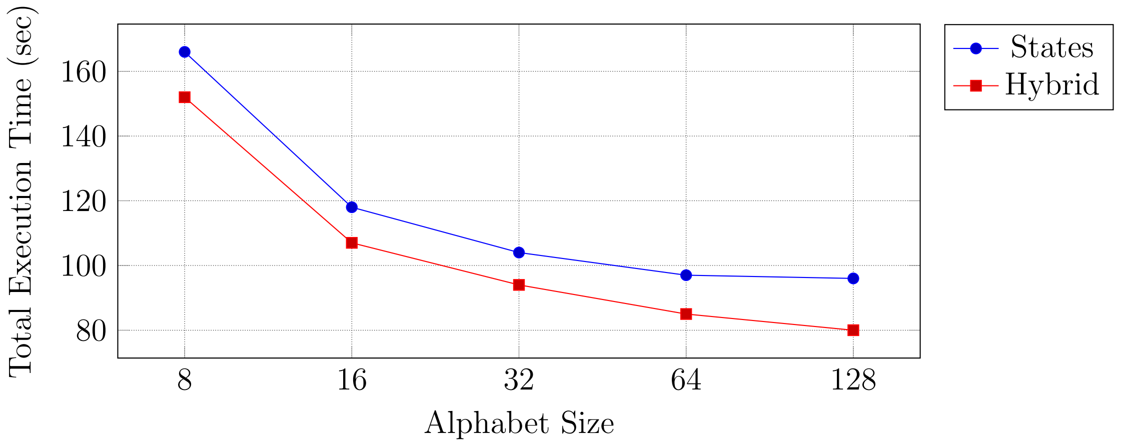


**Figure 5: Processing times of two methods with** $|\delta_1| + |\delta_2| + |\delta_3| = 7,000$

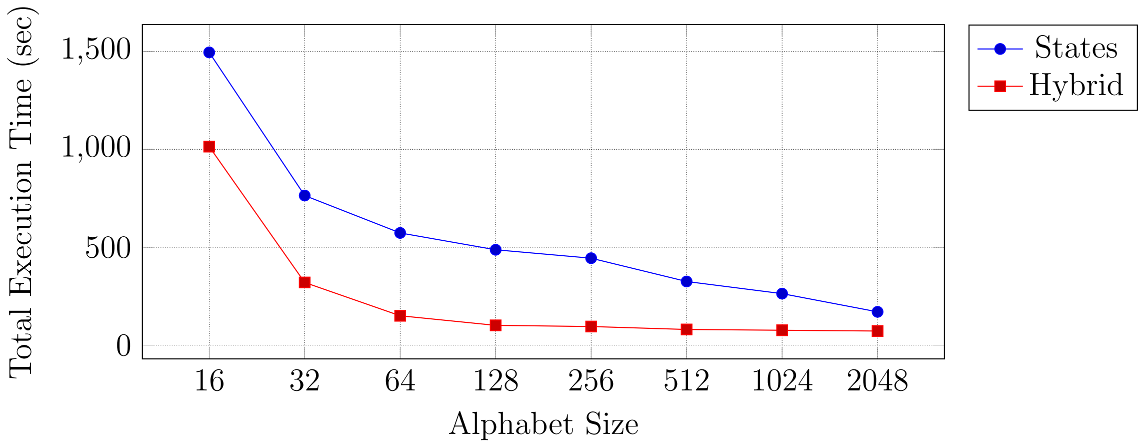


**Figure 6: Processing times of two methods with** $|\delta_1| + |\delta_2| + |\delta_3| = 28,000$

In Figure 3, we see the execution time for different data sizes with the alphabet size $k = 16$. Figure 4 shows the comparison of States and Hybrid methods, while the alphabet size $k = 64$. As expected, Hybrid method is clearly more efficient.

Figure 5 represents execution time of the hybrid method versus mapping based on states for different alphabet sizes when $|\delta_1| + |\delta_2| + |\delta_3| = 7,000$. This figure indicates while the hybrid method works faster, the execution time difference remains almost constant, since the alphabet size is less than or equal to the number of reducers.

Figure 6 represents execution time of the two methods for various alphabet sizes while $|\delta_1|+|\delta_2|+|\delta_3| = 28,000$. The figure shows that as the size of alphabet increases, the execution time of both algorithms get closer to each other. This is due to the fact that once the the size of the alphabet exceeds the number of reducers (128), in the hybrid method each reducer has to deal with several alphabet symbols, thus slowing down the computation inside the reducers.

## 5.3 DFA Minimization

For DFA minimization problem, we generated random DFA's with different sizes and ran the minimization algorithm over the cluster.

In the experiments we compared the execution time obtained by the DFA minimization implementation, the communication cost as well as the number of rounds.

In figure 7, we see the total execution time for DFA's with different number of states and a constant alphabet size $k = 2$. While there is a small difference among the execution time of DFA's with different number of states, it amazingly addresses the fact that the total number of rounds is an important factor. The diameter $d$ of all the DFA's in this figure was set to 7. It shows that the processing time taken to minimize a DFA depends on its diameter, regardless of its input size and communication cost. Table 1 shows the input size and the communication cost of 7 different datasets used in figure 7.

| Dataset | $\|Q\|$ | $\|I\|$ in KB | Comm. Cost in MB |
|---|---|---|---|
| 1 | 128 | 2.24 | 1.2 |
| 2 | 256 | 5.21 | 2.6 |
| 3 | 512 | 10.91 | 5.3 |
| 4 | 1024 | 21.94 | 10.6 |
| 5 | 2048 | 47.25 | 21.1 |
| 6 | 4096 | 98.6 | 42.6 |
| 7 | 8192 | 202.21 | 86 |

**Table 1: Sizes of dataset and communication cost**

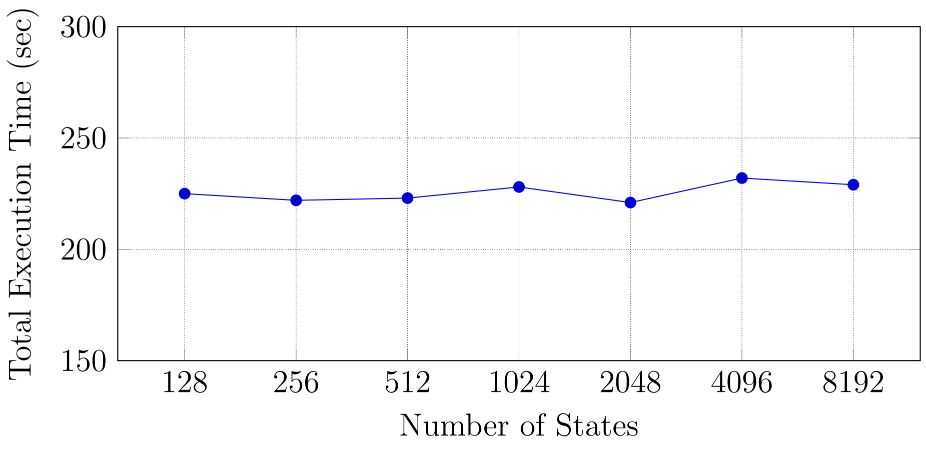


**Figure 7: Execution time vs. problem size**

Figure 8 shows the total execution time in seconds versus the number of rounds. Although the number of states for all DFA's was set to 1024, changes in a diameter has had a great impact on the execution time.

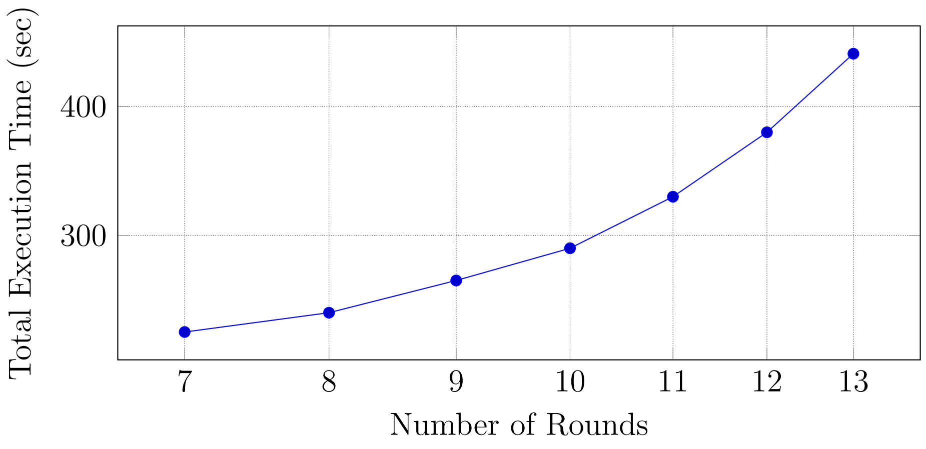


**Figure 8: Execution time vs. number of rounds**

As expected, it can be observed that the larger the diameter of a DFA is, the more time it takes to minimize this DFA. This is clearly because the minimization process terminates in $d$ rounds, where $d$ is the diameter of a graph of a DFA. Consequently, the number of rounds directly affects the execution time. Also, it is worth mentioning that every round of this process is a single map-reduce job compared with other parallel versions of this algorithm that have several jobs per round.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

In this research we proposed and studied methods for computing a product automaton using map-reduce. Additionally, our analysis and experimental results show that carefully optimizing the amount of inter-processor communication indeed pays off in improved processing time.

As a trivial solution to this problem, we showed that hashing only based on states is not always a suitable method, since it increases the replication rate. On the other hand, we introduced hashing based on alphabet symbols as an alternative to decrease the replication rate. Furthermore, we discussed that hashing based on alphabet symbols meets the lower bound on replication rate. However, in cases of dealing with huge datasets and small alphabet size, this hashing method may not necessarily provide expected parallelism.

Combining these two methods, we proposed a new flexible approach called hybrid method. Hashing based on both states and alphabet symbols, provides not only sufficient parallelism to reduce the size of sub-problems, but also allocates enough hash buckets to alphabet symbols to reduce the replication rate as much as possible.

Following the NFA intersections problem, we conducted another research on reducing the number of states in the product automaton, either by eliminating all or part of the useless states or by determinizing and minimizing the automaton.

The problem of DFA minimization inevitably brings iterations of map-reduce jobs. In this problem, not only maintaining the replication rate at its lowest level in each iteration is important, but also transferring data from one round to another is as well. Consequently, finding a suitable way to encode and transfer all required data between rounds became challenging.

We proposed an algorithm to find the minimal DFA that converges in at most linear in $|Q|$ number of rounds. This algorithm benefits from running each iteration of the problem using only one map-reduce job. In practice, the experimental results indicated that the actual number of iterations to terminate, is much smaller than the theoretical worst-case.

## 6.2 Future Work

In future work it will be interesting to investigate the optimal number of rounds in which the problem of DFA minimization terminates in map-reduce environment. Another extension is to apply more intensive preprocessing over the topology of the graph of input DFA in order to speed up the process and reduce the communication cost and replication rate.

We also suggest two interesting problems than can be studied alongside our research work. Being two challenging problems in theoretical computer science, the emptiness and infiniteness problem will be introduced later in the following sections.

### 6.2.1 Testing Emptiness of Regular Languages Intersections

As mentioned earlier in the first chapter, testing for emptiness of the intersection of a set of languages represented by finite automata is known to be PSPACE-complete. The comprehensive study on space and time complexity of this problem can be found in [17].

We shall now define the emptiness problem. The intersection of $m$ finite automata is an automaton $A$, that accepts the language $L$ containing all strings accepted by all $m$ automata. Clearly, the number of states in $A$ could be very large. This is because assuming each automaton has $n$ states, the product automaton will have $n^m$ states. Now, the question is if $A$ accepts any string at all. The emptiness problem of regular languages intersections tests whether there exist a string which is accepted by $A$. In other words, given a regular language $L = \bigcap_{i=1}^{m} L(A_i)$ ($A_i$ is a finite automaton), does $L$ contain any string? The complement of emptiness problem, known as the nonemptiness problem, could be reduced to the problem of finding any reachable final state from the initial state.

Obviously, finding the reachable states by computing the transitive closure of the corresponding graph of the product automaton, can be of a great help to solve this problem. An interesting investigation on implementation of transitive closure and recursive datalog on cluster can be found in [5]. However, computing transitive closure of a graph is known to be an iterative problem. Here, the important fact is to consider the trade-off between the number of iterations versus the communication cost to come up with an efficient approach to this problem.

### 6.2.2 Verifying Infiniteness of Regular Languages Intersections

Given a finite family of infinite regular languages, $\{L_i\}_{i=1}^{m}$, the problem of verifying infiniteness of regular languages intersections is whether the intersection of these languages, $L = \bigcap_{i=1}^{m} L_i$, also remains infinite. Let $A_i$ be the corresponding automata describing the language $L_i$, that is, $L(A_i) = L_i$. It is easy to see that this problem can be reduced to the problem of finding a cycle in the product automaton

$$A = A_1 \otimes A_2 \otimes \cdots \otimes A_m.$$

Clearly, this cycle must be reachable from the initial state and lead to a final state. In other words, let $n$ be the number of states in the intersection automaton. Existence of a string of length equal or larger than $n$ accepted by this automaton, ensures that the intersection language is infinite.

One might be interested in parallelizing the problem of verifying infiniteness of regular languages intersections considering its communication cost as the bottleneck in map-reduce environment.

# References

[1] Apache hadoop. Available at http://hadoop.apache.org/. 12

[2] Foto N. Afrati, Shlomi Dolev, Ephraim Korach, Shantanu Sharma, and Jeffrey D. Ullman. Assignment of different-sized inputs in mapreduce. In *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015.*, pages 28–37, 2015. 26

[3] Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *PVLDB*, 6(4):277–288, 2013. 6, 17, 18, 26

[4] Foto N. Afrati and Jeffrey D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011. 10, 15, 27

[5] Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *15th International Conference on Extending*

*Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 132–143, 2012. 51

[6] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998. 9

[7] Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-cover in map-reduce. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 231–240, 2010. 10

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. 9

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43, 2003. 12

[10] Gösta Grahne, Shahab Harrafi, Ali Moallemi, and Adrian 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.*, pages 42–45, 2015. 5

[11] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971. 19

[12] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. 1, 3

[13] IBM, Paul Zikopoulos, and Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011. 8

[14] Ben Kimmett, Alex Thomo, and S. Venkatesh. Three-way joins on mapreduce: An experimental study. In *IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications, Chania, Crete, Greece, July 7-9, 2014*, pages 227–232, 2014. 10, 27

[15] Ioannis K. Koumarelas, Athanasios Naskos, and Anastasios Gounaris. Binary theta-joins using mapreduce: Efficiency analysis and improvements. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014.*, pages 6–9, 2014. 10

[16] Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 254–266, 1977. 3

[17] Klaus-Jörn Lange and Peter Rossmanith. The emptiness problem for intersections of regular languages. In *Mathematical Foundations of Computer Science 1992, 17th International Symposium, MFCS'92, Prague, Czechoslovakia, August 24-28, 1992, Proceedings*, pages 346–354, 1992. 50

[18] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei 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)*, pages 85–94, 2011. 10

[19] J. Leskovec, A. Rajaraman, and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014. 8, 9, 15, 17

[20] Mahsa Mofidpoor, Nematollaah Shiri, and Thiruvengadam Radhakrishnan. Index-based join operations in hive. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 26–33, 2013. 10

[21] Gianmarco De Francisci Morales, Aristides Gionis, and Mauro Sozio. Social content matching in mapreduce. *PVLDB*, 4(7):460–469, 2011. 10

[22] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In

*Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110, 2008. 13

[23] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 50–61, 2013. 10

[24] Bala 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, 1996, Honolulu, Hawaii, USA*, pages 187–191, 1996. 20, 21

[25] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10, 2010. 13

[26] Y. N. Srikant. A parallel algorithm for the minimization of finite state automata. *International Journal of Computer Mathematics*, 32(1-2):1–11, 1990. 20

[27] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide*

*Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 607–614, 2011. 10

[28] Deian Tabakov and Moshe Y. Vardi. Experimental evaluation of classical automata constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, pages 396–411, 2005. 41

[29] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005, 2010. 12

[30] Bruce W. Watson. A taxonomy of finite automata minimization algorithms. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993. 20

[31] Tom White. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc, 3rd edition, May 2012. 12, 13