# DFA Minimization Algorithms in Map-Reduce

Iraj Hedayati Somarin

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

January 2016

# Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Iraj Hedayati Somarin**

Entitled: **DFA Minimization 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:

**Rajagopalan Jayakumar** Chair

**Brigitte Jaumard** Examiner

**Hovhannes A. Harutyunyan** Examiner

Examiner

**Gösta K. Grahne** Supervisor

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

**January 11, 2016**

Amir Asif, Ph.D., Dean
Faculty of Engineering and Computer Science

# Abstract

DFA Minimization Algorithms in Map-Reduce

Iraj Hedayati Somarin

Map-Reduce has been a highly popular parallel-distributed programming model. In this thesis, we study the problem of minimizing Deterministic Finite State Automata (DFA). We focus our attention on two well-known (serial) algorithms, namely the algorithms of Moore (1956) and of Hopcroft (1971). The central cost-parameter in Map-Reduce is that of *communication cost* i.e., the amount of data that has to be communicated between the processes. Using techniques from Communication Complexity we derive an $O(kn \log n)$ lower bound and $O(kn^3 \log n)$ upper bound for the problem, where $n$ is the number of states in the DFA to be minimized, and $k$ is the size of its alphabet. We then develop Map-Reduce versions of both Moore's and Hopcroft's algorithms, and show that their communication cost is $O(kn^2(\log n + \log k))$. Both methods have been implemented and tested on large DFA, with 131,072 states. The experiments verify our theoretical analysis, and also reveal that Hopcroft's algorithm – considered superior in the sequential framework – is very sensitive to skew in the topology of the graph of the DFA, whereas Moore's algorithm handles skew without major efficiency loss.

# Acknowledgments

Above all, the researcher's honest and sincere appreciation extends to his supervisor, Prof. Gösta Grahne, for his constant encouragement and creative and comprehensive guidance until this work came to existence.

On a more personal note, his gratitude goes to love of his life for her boundless support and patience. The researcher also wishes to give credit to research team members, Mr. Shahab Harrafi and Mr. Ali Moallemi, for their generous and constructive comments, suggestions and foresights throughout this process.

# Table of Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Abbreviations

| | |
|---|---|
| CC | Communication Complexity |
| CRCW | Common Read Common Write |
| CREW | Common Read Exclusive Write |
| DCC | Deterministic Communication Complexity |
| DFA | Deterministic Finite Automata |
| DFS | Distributed File System |
| ERCW | Exclusive Read Common Write |
| EREW | Exclusive Read Exclusive Write |
| FA | Finite Automata |
| HDFS | Hadoop Distributed File System |
| MIMD | Multiple Instructions Multiple Data |

| | |
|---|---|
| MISD | Multiple Instructions Single Data |
| NFA | Non-deterministic Finite Automata |
| PHF | Perfect Hashing Function |
| PPHF | Parallel Perfect Hashing Function |
| PRAM | Parallel Random Access Machine |
| SIMD | Single Instruction Multiple Data |
| SISD | Single Instruction Single Data |

# Chapter 1

# Introduction

*Finite Automata (FA)* are one of the most robust machines for modelling discrete phenomena with a wide range of applications playing one of the two roles: *models* or *descriptors* [36]. *Deterministic Finite Automaton (DFA)* is a variant of FA which plays a significant role in pattern matching, compiler and hardware design, lexical analysis, protocol verification, and software model testing problems among the others. Although for any specific language there could be a wide range of accepting DFA, it is well-known that there is just one unique minimal DFA for this language. As computing very large DFA is time consuming, a good practice is DFA minimization. Nowadays, in presence of exponentially growing amount of data and data-models as well, applying serial algorithms to this matter becomes almost impossible.

In this thesis, we will discuss some of the most efficient minimization methods and

propose parallel- distributed algorithms on Map-Reduce model.

## 1.1 DFA Minimization

FA are represented by directed graphs where states are the vertices and each transition represented by a labeled link. FA are intensely studied in Computer Science since the beginnings of automata theory. In a DFA, all the states need an outgoing transition for each alphabet symbol. If DFA gets a finite sequence of symbols (string) as input, it performs a special calculation of the automaton and will accept/reject it [20]. DFA minimization is the process of discovering an equivalent DFA to given one with minimum number of states.

There are basically two classes of states that can be removed or merged:

- *Unreachable states.* The set of states to which there are no paths from the initial state. This type of states plays 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 distinguishable, if there is a string $w$ where, processing of $A$ on $w$ from $p$ leads to an accepting state and from $q$ ends up to a non-final state. If they are not distinguishable, $p$ and $q$ are called non-distinguishable.

DFA minimization has long been studied since the late 1950's and there are a

variety of algorithms utilizing different approaches [37]. On the other hand, data is growing and consequently, nowadays, we are dealing with DFA's with *petastates* [1] for which DFA minimization process becomes not only time-consuming but also almost impossible. It is well known that one of the most powerful methods to handle large size of data is Map-Reduce model where we first send part of data appropriate to a processing unit to one or more machines (*Mapping*) and do the processing simultaneously (*Reducing*) [14].

As mentioned, there are variety of algorithms introduced in field of DFA minimization. However, Finite Automata processing is very difficult to parallelize because of tight dependencies between successive loops making it hard to distribute loops over processing units and also consequently in each iteration there is a few computations which are very input-dependent and unpredictable. When it comes to processing finite automata in Map-Reduce environment, the most important part of algorithm design is reducing number of rounds as well as replication rate and communication cost.

## 1.2 Big-Data Processing in Map-Reduce Model

It is more than trivial that amount of data is unlimited including data from past, present and what will happen in future as long as could be observed by humankind.

[1] *petastate* DFA is a DFA with more than $10^{15}$ states

Fortunately, data storage technology is growing exponentially. This leads to increase in amount of data to be processed as well. It can be observed from amount of data that individual organizations, in a narrowed field of study, are dealing with. For instance, the followings are facing to petabyte [2] data:

- Since 20 years ago, NASA ECS project archive [10]
- Since 13 years ago, High Energy Physics [5]
- Around 5 years ago, Astronomy [8].

In order to give the possibility of processing this amount of data, Map-Reduce model is introduced to organize data in smaller chunks and process each of them separately. This will be discussed more in details in 2.2.

## 1.3 Motivation and Contributions

DFA Minimization is a fundamental field of study in computer science because of the exhibited power for modeling variety of problems. On the other hand, data explosion happened in computer industry results complexity in running data intensive programs. The lack of proper work on performing DFA minimization in parallel motivated us to conduct this research to propose algorithms on Map-Reduce model mainly by using equivalence relation and analyzing the performance of different approaches.

[2] Petabyte is equal to $2^{50}$ byte

Besides proposing two distinct algorithms, we found that the graph topology of DFA has a direct effect on the running performance and consuming resources. In addition, the upper and lower bound on communication cost for minimizing a DFA in parallel environments is calculated. Also, it has been discovered that there is not a universal and comprehensive cost model and new measures is suggested in this thesis. These works are supported by experiments running on DFA with more than $100,000$ states with different topologies.

## 1.4 Thesis Organization

The rest of this thesis is organized as follows.

In Chapter 2, we review the background information necessary for this research. DFA minimization approaches in serial and parallel are represented in Section 2.1. Section 2.2 will briefly introduce the Map-Reduce model and its open source implementation called Hadoop. We look into contributions in processing FA using Map-Reduce model, in Section 2.3 and finally the work done in cost model in parallel environment is investigated in Section 2.4.

In Chapter 3, we study the DFA minimization problem in Map-Reduce model and its cost measures. We propose an enhancement to the only known implementation of Moore's algorithm in Map-Reduce model [18] in Section 3.1. Additionally, a new algorithm based on Hopcroft's DFA minimization method [19] is proposed in the

Map-Reduce model in Section 3.2. Section 3.4 is about the cost measures for DFA minimization in Map-Reduce. The upper and lower bound on communication cost is derived from Yao's two parties model [41] in Section 3.4.1. In addition, following Afrati et al. [2], the lower bound for data replicating over processing units is studied in Section 3.4.2. The communication cost for the algorithms are presented in Sections 3.4.3 and 3.4.4.

Afterwards, Chapter 4 will demonstrate the performance of each algorithm in terms of execution time and required space. After reviewing the underlying environment in Section 4.1, the methods for generating sample data sets are described in Section 4.2. In Section 4.3 the results gathered from experiments are shown and discussed.

Finally, Chapter 5 is assigned for conclusion and future works including the reflection of study conducted in this thesis.

# Chapter 2

# Background and Related Work

## 2.1 Finite State Automata and their Minimization

An FA 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 $w = a_1 a_2 \dots a_n$ where $a_i \in \Sigma$, be a string in $\Sigma^*$. An *accepting computation path* of $w$ in $A$ is a sequence $(s, a_1, q_1), (q_1, a_2, q_2), \dots, (q_{n-1}, a_n, q_f)$ of tuples (elements) of $\delta$, where $q_f \in F$. 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 FA $A$ such that $L(A) = L$.

An FA $A = (Q, \Sigma, \delta, s, F)$ is said to be *deterministic* if for all $p \in Q$ and $a \in \Sigma$

there is a $q \in Q$, such that $(q, a, p) \in \delta$. Otherwise the FA is *non-deterministic*. Deterministic FA's are called DFA's, and non-deterministic ones are called NFA's. By the well known subset construction, any NFA $A$ can be turned into a DFA $A^D$, such that $L(A^D) = L(A)$. For a DFA $A = (Q, \Sigma, \delta, s, F)$, we also write $\delta$ in the functional format, i.e. $\delta(p, a) = q$ iff $(p, a, q) \in \delta$. In this thesis, both notation is used whenever it is necessary. For a state $p \in Q$, and string $w = a_1 a_2 \dots a_n \in \Sigma^*$, we denote by $\hat{\delta}(p, w)$ the unique state $\delta(\delta(\cdots \delta(\delta(p, a_1), a_2) \cdots, a_{n-1}), a_n)$.

A DFA $A$ is said to be *minimal*, if all DFA's $B$, such that $L(A) = L(B)$, have at least as many states as $A$. For each regular language $L$, there is a unique (up to isomorphism of their graph representations) minimal DFA that accepts $L$.

The DFA minimization problem has been studied since 1950s. A taxonomy of DFA minimization algorithms was created by B. W. Watson in 1993 [37]. The taxonomy is illustrated in Figure 1. Most of the algorithms are based on the notion of equivalent states (to be defined). The sole exception is Brzozowski's algorithm [42], which is based on *reversal* and determinization of automata. Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA. Then the reversal of $A$ is the NFA $A^R = (Q, \Sigma, \delta^R, F, \{s\})$, where $\delta^R = \{(p, a, q) : (q, a, p) \in \delta\}$. Brzozowski's showed in 1962 that if $A$ is a DFA, then $(((A^R)^D)^R)^D$ is a minimal DFA for $L(A)$. This rather surprising result does not however yield a practical minimization algorithm since there is a potential double exponential blow-up in the number of states, due to the two determinization steps.

The rest of the algorithms is based on equivalence of states. Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA, and $p, q \in Q$. The $p$ is *equivalent* with $q$, denoted $p \equiv q$, if for all strings $w \in \Sigma^*$, it holds that $\hat{\delta}(p, w) \in F$ if and only if $\hat{\delta}(q, w) \in F$. The *quotient* DFA $A/_{\equiv} = (Q/_{\equiv}, \Sigma, \gamma, s/_{\equiv}, F/_{\equiv})$ where $\gamma(p/_{\equiv}, a) = \delta(p, a)/_{\equiv}$, is then a minimal DFA, such that $L(A/_{\equiv}) = L(A)$. Note that $Q/_{\equiv}$ is a *partition* of the state-space $Q$. An important observation is that $\equiv$ can be computed iteratively as $\bigvee_{i=0}^{\infty} \equiv_i$, where $p \equiv_0 q$ if $p \in F \Leftrightarrow q \in F$, and $p \equiv_{i+1} q$, if $p \equiv_i q$ and $\delta(p, a) \in F \Leftrightarrow \delta(q, a) \in F$, for all $a \in \Sigma$. This means that $Q/_{\equiv}$ can be computed iteratively, each step refining $Q/_{\equiv_i}$ to $Q/_{\equiv_{i+1}}$.

Here is a brief description of each node under the Equivalence of States tree:

- Equivalence Relation: Find distinguishable states based on the equivalence class of every state.
  - *Bottom-up approach.* This approach starts out with a partition of $n$ blocks where $n$ is number of states. Each block contains one and only one state. Afterwards, the classes are iteratively merged and updated based on discovering more equivalent states. After each iteration, intermediate results can be used to make a smaller DFA [38].
  - *Top-down approach.* In this classical methodology, we initially divide states in two class of equivalency called final and non-final. During minimization process, this partition is refined gradually by finding new distinguishable

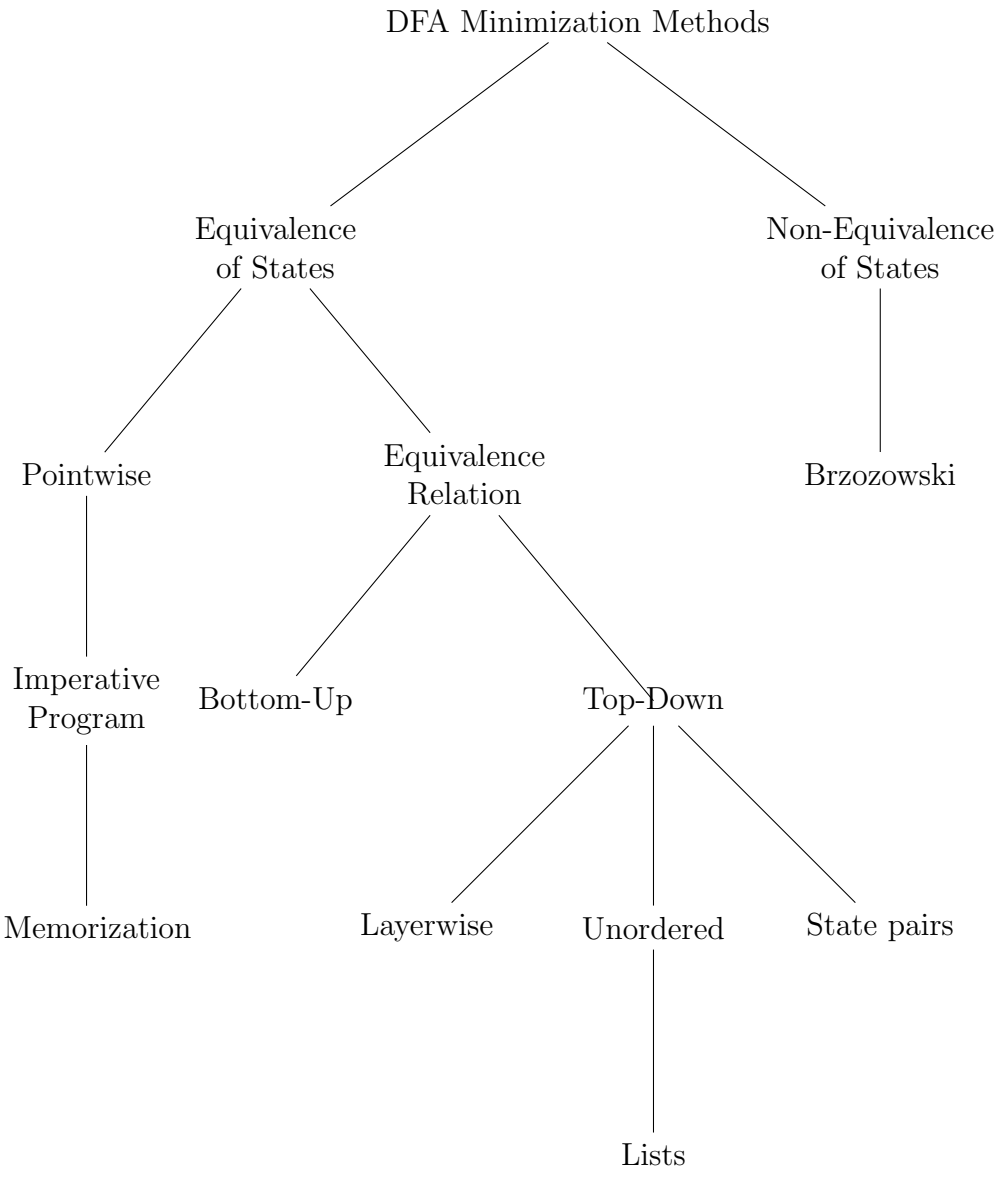


**Figure 1: The forest expressing the family of finite automata minimization algorithms and their relations. As it can be seen, Brzozowski algorithm for minimization is completely unrelated to others and is not connected to this tree.**

states. This approach is similar to the Coarsest Partitioning problem [20]. The input to the Coarsest Partition problem is a set $S$ of $n$ elements and a partition $\pi$ over them as well as one or more function $f_i : S \to S$. The problem is to form a new partition $\pi'$ where each member of $\pi'$ is a subset of one of the $\pi$ members and $f_i(s) = f_i(t)$ whenever $s$ and $t$ belong to the same block of the new partition. The final partition of minimization process contains equivalence classes each of which represents one state in minimized DFA. Under this category we can talk about the following methods:

* **Layer-wise**: this algorithm is called *Layer-Wise* as it computes the equivalency of two states for strings of length $i$ [40] [27] [9] [35]. **Moore**'s algorithm [27] will be discussed later on in Section 2.1.1.
* **Un-ordered**: this method called Un-Ordered because it does not follow normal orders of processing like *Layer-wise*. This family of algorithms have a free choice of states at each round to calculate partition refinement. The best known running time of DFA minimization algorithm [19] utilizes a list of blocks to which others should be compared. This algorithm is so called *Hopcroft's DFA Minimization Algorithm* that will be discussed later in Section 2.1.2.
* **State pair**s: This algorithm presented in [20] in which every two pairs

will be compared to each other and if there exist an alphabet symbol they go to different classes with, they are distinguishable.

- Point-wise: this method is mainly introduced in [32] for functional programming purpose. The main idea is that two states are equivalent unless it is shown otherwise. This matter can be achieved recursively. To accomplish finding equivalent states, it mainly requires a data structure. Basically we can employ global variables. However in real world applications it is almost impossible to achieve that. Derived methods such as *Imperative Program* and *Memorization* are placed under *Point-wise* sub-tree.

From above discussion, despite all differences, one may observe that a common property for all methods under taxonomy tree is being iterative. Each tries to refine partition of a DFA at each iteration until no more refinement can be applied. We may call each iteration *a round in DFA minimization algorithm* or shortly *round* in this thesis.

### 2.1.1 Moore's Algorithm

The earliest iterative algorithm was proposed by Moore in 1956 [27]. The algorithm computes the partition $Q/_{\equiv}$ by iteratively refining the initial partition $\pi = \{F, Q \setminus F\}$. In the algorithm, partitions are recorded by a function $\pi : Q \to \{0,1\}^{\log n}$.

Algorithm 1 exhibits Moore's algorithm. The main data structure employed here is an array *block* with size of $n = |Q|$ which keeps equivalence class for every state. The partitions are labeled by bit strings. The initial partition is $\pi = \{F, Q \setminus F\}$ and $block(p)$ is 1 if $p \in F$ and 0 otherwise(line 1 to 6). At each iteration, we refine the current partition based on the outgoing transitions of the states for all alphabet (lines 8 to 10). Note that the symbol "$\cdot$" on line 9 of the algorithm denotes concatenation (of strings). Based on discovered *block*, lines 11 to 13 of algorithm are responsible to refine every block of $\pi$ using similarity ($\sim$) relation. Two states $p$ and $q$ are similar if and only if $block(p) = block(q)$. By replacing $B$ by $B/_{\sim}$, it will refine $\pi$ consequently. This algorithm will stop whenever new partition is equal to previous one ($\pi'$), e.g. no refinement is happened. The minimal automaton can now be obtained by replacing each transition $(p, a, q) \in \delta$ by $(block(p), a, block(q))$, and then removing duplicate tuples.

As an example, consider DFA in Figure 2. Initial partition is $\pi = \{0, 1\}$. Running loop from line 9 yields to the marking shown in Table 1.

**Table 1: A sample partition refinement for Moore's algorithm**

| Block 1 | Block 0 |
|---|---|
| $1 \rightarrow 0\ 1$ | $2 \rightarrow 1\ 1$ |
| $3 \rightarrow 1\ 0$ | $6 \rightarrow 0\ 0$ |
| $4 \rightarrow 1\ 0$ | |
| $5 \rightarrow 1\ 0$ | |

**Algorithm 1** Moore's DFA minimization algorithm

**Input:** DFA $A = (Q, \{a_1, \ldots, a_k\}, \delta, s, F)$
**Output:** $\pi : Q \rightarrow \{0, 1\}^*$, where $\pi_p = \pi_q$ iff $p \equiv q$

1: **for all** $p \in Q$ **do**
2: **if** $p \in F$ **then** $block(p) \leftarrow 1$
3: **else** $block(p) \leftarrow 0$
4: **end if**
5: **end for**
6: **repeat**
7: $\pi' \leftarrow \pi$
8: **for all** $p \in Q$ **do**
9: $block(p) \leftarrow block(p) \cdot block(\delta(p, a_1)) \cdot block(\delta(p, a_2)) \cdot \ldots \cdot block(\delta(p, a_k))$
10: **end for**
11: **for all** $B \in \pi$ **do**
12: Replace $B$ in $\pi$ by $B/_\sim$ where $p \sim q$ if $block(p) = block(q)$
13: **end for**
14: **until** $\pi = \pi'$

Obviously, block 1 is divided to $\{1\}$ and $\{3, 4, 5\}$. On the other hand, block 0 is divided to $\{2\}$ and $\{6\}$. Now using above marking, new partition can be constructed as: $\pi = \{0, 1, 2, 3\}$. After the next iteration, the algorithm cannot generate new blocks and it stops. The minimized DFA is illustrated in Figure 3.

The number of iterations required for this algorithm is called *depth* of Moore's

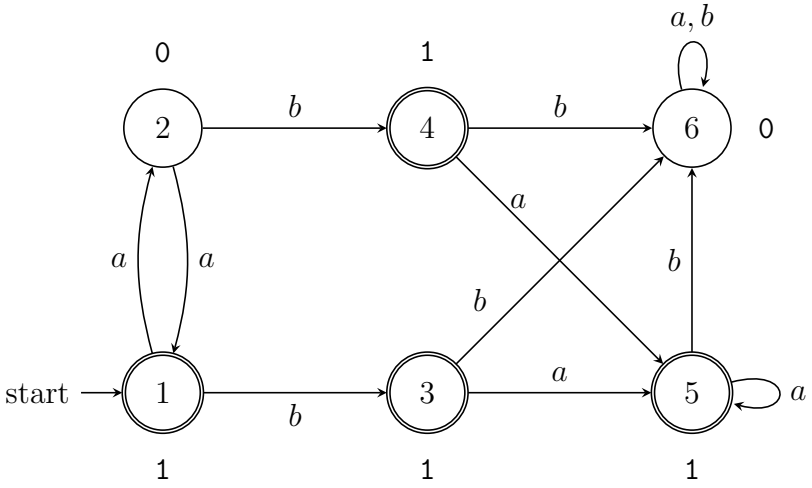


**Figure 2: An example DFA**

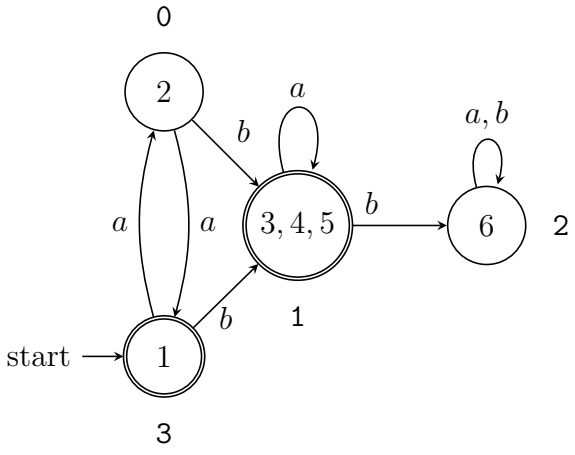


**Figure 3: The minimized version of DFA in Figure 2**

algorithm [6]. This factor strictly relies on the language recognized by automaton not topology of DFA. Lines 8 to 10 of Algorithm 1 iterates $O(kn)$ where $n$ is the number of states and $k$ is size of alphabet. Besides, second loop at lines 11 to 13 can be implemented using Radix Sort having running time of $O(kn)$. Denoting the number of rounds as $r$, running time of Moore's algorithm is $O(rkn)$. Although the worst case is obtained for $r = n - 2$ in family of *slow automata* (described in 2.1.2.2), a detailed study in [3] [13] shows that there are only a few automata for which $r$ is greater than $\log n$.

### 2.1.2 Hopcroft's Algorithm

As it can be seen in Figure 1, under un-ordered DFA minimization method, there should be a list to control the next step of the process. In ordered methods such as Moore's algorithm, the topology of DFA is not considered. In these algorithms, all the states are being processed. By doing a pre-processing on DFA, we can avoid these unnecessary work. The only known method using lists, has been introduced by John Hopcroft [19].

Hopcroft gave an efficient algorithm in 1971 for minimizing the number of states in a finite automaton [19]. The running time of his algorithm is $O(kn \log n)$ for $n$ states and $k$ input alphabet symbols. Here, the original algorithm by Hopcroft is described in more details.

The algorithm computes the partition $\pi$ generated by $\equiv$. We start with some definitions. Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA, $P, R \in \pi$ and $a \in \Sigma$. Then $\langle P, a \rangle$ is called a *splitter*, and

$$\begin{aligned} R \div \langle P, a \rangle &= \{R_1, R_2\}, \text{ where} \\ R_1 &= \{q \in R : \delta(q, a) \in P\} \\ R_2 &= \{q \in R : \delta(q, a) \notin P\}. \end{aligned}$$

If either of $R_1$ or $R_2$ is empty, we set $R \div \langle P, a \rangle = R$. Otherwise $\langle P, a \rangle$ is said to *split*

$R$. Furthermore, for $P \subset Q$ and $a \in \Sigma$, we define

$$a(P) = \{p \in P : \delta(q, a) = p \text{ for some } q \in Q\}. \tag{1}$$

That is, $a(P)$ consists of those states in $P$ that have an incoming transition labelled $a$. As initial partition $\pi$, contains a final and a non-final block, labelled as 1 and 2 respectively. Then, for all $a$ in $\Sigma$ and $P \in \{1, 2\}$, construct:

$$a(P) = \{q | q \in P \wedge \delta^{-1}(q, a) \neq \emptyset\} \tag{2}$$

where $P \in \pi$ and $\delta^{-1}(q, a)$ is set of states from which there is a transition labelled $a$ to $q$.

Now using Equation 2, we can define a set of *splitters*.

$$\mathcal{Q} = \{\langle P, a\rangle | a \in \Sigma, P \in \pi = \{1, 2\}\} \tag{3}$$

where,

$$P = \begin{cases} 1 & \text{if } |a(1)| \leq |a(2)| \\ 2 & \text{otherwise} \end{cases}$$

The above mentioned steps about initializing required data structure are shown in line 1 to 7 of Algorithm 2. The algorithm will stop as soon as the list $\mathcal{Q}$ becomes

empty. At each round , it picks a pair $\langle P, a\rangle$ from $\mathcal{Q}$ (line 9), and finds the blocks which has to be partitioned with respect to $\langle P, a\rangle$. Once partition is refined (lines 11 and 12), it is necessary to apply Equation 2 again in order to update data structures regarding new $\pi$. On the other hand, each time the block $R$ is split, corresponding *splitters* have to be added to $\mathcal{Q}$. For every alphabet symbol $a$, if $\langle R, a\rangle$ we previously in $\mathcal{Q}$, it has to be replaced by $\langle R_1, a\rangle$ and $\langle R_2, a\rangle$. Otherwise, we only insert one of $\langle R_1, a\rangle$ or $\langle R_2, a\rangle$, namely the one where the block $R_i$ has fewer incoming transitions labelled $a$ by extending Equation 3 to Equation 4 (lines 17 to 19).

$$\forall a \in \Sigma, \mathcal{Q} = \begin{cases} \mathcal{Q} \cup \{\langle R_1, a\rangle\} & \text{if } |a(R_1)| \leq |a(R_2)| \\ \mathcal{Q} \cup \{\langle R_2, a\rangle\} & \text{otherwise} \end{cases} \tag{4}$$

The minimal automaton equivalent with $A$ is obtained by choosing a representative $p_i$ for each equivalence class $P_i$, then replacing each $(p, a, q)$ in $\delta$ with $(p_i, a, q_j)$ where $p \in P_i$ and $q \in P_j$, and finally removing duplicates.

#### 2.1.2.1 Complexity of Hopcroft's Algorithm

Hopcroft's work published in 1971 and his proof of complexity is difficult to understand. David Gries gave a better explanation of algorithm complexity in [17] two years later which we tried to simplify here.

**Algorithm 2** Hopcroft's DFA minimization algorithm

**Input:** $A = (Q, \Sigma, \delta, s, F)$
**Output:** $\pi = Q/_{\equiv}$

```
1: π ← {F, Q \ F}                                   ▷ The initial partition
2: 𝒬 ← ∅                                            ▷ Queue set
3: for all a ∈ Σ do
4:     if |a(F)| < |a(Q \ F)| then  Add ⟨F, a⟩ to 𝒬
5:     else Add ⟨(Q \ F), a⟩ to 𝒬.
6:     end if
7: end for
8: while 𝒬 ≠ ∅ do
9:     Pick and delete a splitter ⟨P, a⟩ from 𝒬.
10:    for each R ∈ π which is split by ⟨P, a⟩ do
11:        {R1, R2} ← R ÷ ⟨P, a⟩.
12:        Replace R in π by {R1, R2}.
13:        for all b ∈ Σ do
14:            if ⟨R, b⟩ ∈ 𝒬 then
15:                Replace ⟨R, b⟩ in 𝒬 by ⟨R1, b⟩ and ⟨R2, b⟩
16:            else
17:                if |b(R1)| < |b(R2)| then  add ⟨R1, b⟩ to 𝒬
18:                else  add ⟨R2, b⟩ to 𝒬.
19:                end if
20:            end if
21:        end for
22:    end for
23: end while
```

**Lemma 1** *The number of iterations in the main loop is* $2kn$. *[19]*

*Proof (following Gries [17]).* The number of iterations is equal to the number of pairs we put in the list. Considering the fact that for each newly created block $R$ we put $\langle R, a\rangle$ for all $a \in \Sigma$, we just need to show that $2n$ blocks can be generated. Just think of a binary tree with a dummy root node. This node has two children; one set of final states and another set of non-final states. Every time we do split, a node will

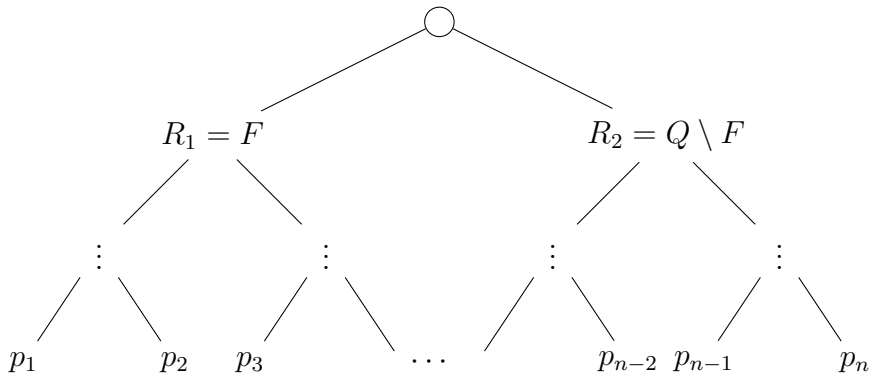


**Figure 4: The tree exposing procedure of re-partitioning in Hopcroft's algorithm**

be broken in two. The number of leaves for this tree would be equal to $n$ in case that DFA is already minimal. Hence, it can have at most $2n$ nodes. Figure 4 shows the structure of this binary tree.□

The operations within the lines 9 to 22, are divided in two sections: find blocks which is split by $\langle P, a\rangle$ and split these blocks. Using Lemma 2 and Lemma 3, we show that amortized analysis of detection is $O(kn \log n)$ and refinement section (lines 11 to 21) is also $O(kn \log n)$ using Lemma 3.

In order to do an amortized complexity analysis we use the following descriptors introduced by Hopcroft and also used by Gries:

- Let's define a variable $C$, the number of states in all splitters added to $\mathcal{Q}$. Let $\langle P, a\rangle$ be the splitting pair dequeued at line 9. So $C = C + |P|$ where $|P|$ is the number of states in $P$.

- In addition, $K = \{P | \langle P, a\rangle \in \mathcal{Q}\}$ and $\bar{K} = \{P | \langle P, a\rangle \notin \mathcal{Q}\}$. We define $T$ as a function of $n$ and $\mathcal{Q}$ as follows,

$$T = n \log n - \sum_{P_i \in K} |P_i| \log |P_i| - \sum_{P_i \notin K} |P_i| \log \frac{|P_i|}{2} \tag{5}$$

**Lemma 2** *Determine the blocks splitting with respect to $\langle P, a\rangle$ takes $O(kn \log n)$. [19]*

*Proof. (Gries [17])* Determine if a block has to be split with respect to $\langle P, a\rangle$ requires $O(1)$ using $\delta^{-1}(p, a)$ where $p \in P$. Hence, the complexity of this operation is less than or equal to the number of states in splitters. As explained in the proof of Lemma 1, it is only required to show that $C \leq T = O(n \log n)$ for the singleton alphabet $\Sigma = \{a\}$. The initial state holds as $C = 0$. Obviously, value of $C$ and $T$ changes at line 9 where we pick a pair from $\mathcal{Q}$ and within loop at line 10 where we split a block.

Whenever a pair $\langle P, a\rangle$ is removed from $\mathcal{Q}$, $\hat{C} = C + |P|$ where $\hat{C}$ holds new value of $C$. Using Equation 5 as well as the fact that before line 9 $P \in K$,

$$\hat{K} = K - P \text{ and } \hat{\bar{K}} = \bar{K} + P \rightarrow \hat{T} = T + |P| \log |P| - |P| \log \tfrac{|P|}{2} = T + |P|$$

which holds $\hat{C} \leq \hat{T}$.

On the other hand, after splitting a block, the algorithm will add pairs to $\mathcal{Q}$ or delete from. Suppose block $R$ is split to $R_1$ and $R_2$. Thus $|R| = |R_1| + |R_2|$ where

without loss of generality $|R_1| \leq |R_2|$. First consider the situation that $\langle R, a\rangle$ was in the list. In this situation, the algorithm removes $\langle R, a\rangle$ and adds $\langle R_1, a\rangle$ and $\langle R_2, a\rangle$ back to the $\mathcal{Q}$. Then:

- $\hat{T} = T + |R|\log|R| - |R_1|\log|R_1| - |R_2|\log|R_2|$
- $|R_1|\log|R_1| + |R_2|\log|R_2| \leq (|R_1| + |R_2|)\log|R_2| = |R|\log|R_2| < |R|\log|R|$

$\therefore \hat{T} > T$

Now suppose that $\langle R, a\rangle \notin \mathcal{Q}$, then $\langle R_1, a\rangle$ has to be added into $\mathcal{Q}$. Hence:

$$\hat{T} = T + |R|\log\tfrac{|R|}{2} - |R_1|\log|R_1| - |R_2|\log\tfrac{|R_2|}{2}$$

Which again $\hat{T} > T$. Therefore $\hat{C} < \hat{T} = O(n\log n)$ in either cases.□

The last step is to find number of times a pair $\langle P, a\rangle$ consisting a specific $p \in P$ can be in $\mathcal{Q}$.

**Lemma 3** *The total number of states need to move to new block because of split operation is bounded by* $O(kn\log n)$ *[19]*

*Proof. (Gries [17])* Suppose $p, q \in Q$, $q \in P$ and $\delta(p, a) = q$ for some arbitrary $a \in \Sigma$. When $\langle P, a\rangle$ is picked as splitter, $p$ has to be moved to new block. We show that the next time a splitter $\langle P_1, a\rangle$ where $q \in P_1$ is added to $\mathcal{Q}$, $|P_1| \leq \frac{|P|}{2}$. As a result, the number of times that $p$ has to be moved from one block to another one

because of split is bounded to $O(k \log n)$. If $\langle P, a \rangle$ is picked and if later $P$ itself splits into $P_1$ and $P_2$, and w.l.g. $|P_1| \leq \frac{|P|}{2} \leq |P_2|$, then only $\langle |P_1|, a \rangle$ will be put back into $\mathcal{Q}$ at line 17. In addition, $|P| \leq n/2$ together with above mentioned fact exposes that $p$ can not be moved with respect to transition $(p, a, q)$ more than $\log n$ times. There are $n$ state each of which having $k$ transitions. This concludes that the total number of states need to move to new block because of split is bounded by $O(kn \log n)$ □

Considering Lemma 1, 2 and 3, amortized complexity of running time of Hopcroft's algorithm is $O(kn \log n)$.

#### 2.1.2.2 Slow Automata

As proved earlier by Lemma 1, the main loop will run at most $kn$ times. This is in the worst case, and happens when each newly generated block has a constant number of states and at each iteration only one block can be split. One of the famous *slow automaton* demonstrated in Figure 5. These types of automata are slow for most of the DFA minimization algorithms.

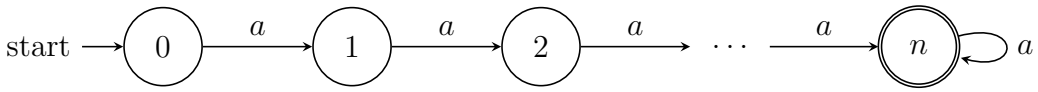


**Figure 5: An example of slow automata for Hopcroft and Moore's algorithms**

### 2.1.3 Parallel Algorithms

In the presence of massive computations in terms of time and space, parallel computing is a handful tool. The Parallel Random Access Memory (PRAM) is used to model parallel algorithmic performance such as time and space complexity. Whenever multiple parties are co-operating to solve a problem beside time and space issues we face shared data operation conflicts as well as communication. The later will be discussed in Section 2.4.1. Here we outline the data conflicts briefly.

There are two basic types of data operations: Read and Write. In the abstract PRAM model there are four strategies to address the matter.

CRCW (Common Read Common Write) Each data unit can be read/written by multiple parties

CREW (Common Read Exclusive Write) Several parties can Read a data unit simultaneously while just one can write at a time

ERCW (Exclusive Read Common Write) Multiple parties can write simultaneously while only one has read access. This model is never used.

EREW (Exclusive Read Exclusive Write) Both read and write are exclusive and only one can be done at a time.

In order to refer above mentioned strategies used in parallel algorithm design, it will be denoted by a combination of conflict resolution strategy and PRAM itself e.g.

EREW-PRAM. The complexity of parallel algorithms in PRAM model is expressed in both space and time. Obviously time has a direct relation to the number of processing units assigned for solving the problem. To compare two different algorithms, work complexity is defined as $O(T(n)\cdot P(n))$ where $T(n)$ is required time for each processing unit in presence of input size $n$ and $P(n)$ is the number of processing units.

Although DFA minimization is inherently an iterative problem and hard to perform in parallel environment[12], especially for *slow automata*, efforts have been done in this area. DFA minimization has been broadly studied on numerous parallel computation models. An exceptionally straightforward algorithm is suggested by Srikant [29], while the best cost-effective algorithm is presented by Jaja and Ryu [21] when the size of alphabet is 1. The algorithm described using CRCW-PRAM has time complexity of $O(\log n)$ requires $O(n^6)$ processors. Having one alphabet makes this a one function coarsest partitioning problem [11] which can be solved by transitive closure. As focus of this thesis is on general DFA's without any restrictive assumptions, these algorithms will not be discussed here.

#### 2.1.3.1 CRCW-PRAM DFA Minimization Algorithm

A clear, simple and efficient algorithm on CRCW-PRAM is from [33] with time complexity $O(kn\log n)$ using $O(\frac{n}{\log n})$ processors where $k$ is the number of alphabet symbols and $n$ is the number of states. This algorithm uses the Moore's method to

assign new block numbers in parallel. This is illustrated in Algorithm 3. States and blocks are represented by a number. The shared data structure in this algorithm is an array $block[0 \dots n]$, where the block number of state $p$ is stored in $block[p]$.

**Algorithm 3** CRCW-PRAM algorithm for DFA minimization [33]

```
1:  Initialize block array
2:  while Number of blocks is changing do
3:      for i = 0 to k − 1 do
4:          for j = 1 to ⌈n / log n⌉ do                    ▷ Parallel loop
5:              for m = (j − 1) log n to j log n − 1 do
6:                  b1 = block[q_m]
7:                  b2 = block[δ(q_m, a_i)]
8:                  label state q_m with (b1, b2)
9:              end for
10:         end for
11:         Refine group numbers using parallel hashing
12:     end for
13: end while
```

As it can be seen, main loop iterates in parallel until there are no more splits in partition which as it mentioned in section 2.1.1, is $n$. Inside the loop, for every alphabet $a$, all states will get new labels based on previous blocks they belonged to and the block with $a$-transition they go.

Considering the fact that if we represent blocks using consecutive integer numbers, values can not exceed $n$. Thus, block numbers can be stored using $\log n$ bits. However, in line 8, size of these markers is being doubled. If iteration takes $i$ rounds for minimization, at round $i$, it is $2^i \log n$. In order to prevent exponential space requirements for block numbers in line 11 of algorithm, they will be shrink to $O(n)$

using a *Parallel Perfect Hashing Function* (PPHF).

The *Perfect Hashing Function* (PHF) used in above mentioned algorithm comes from [26] whose function is mapping a number from set of integers $S$ from interval $[1, m]$ to a set of integers $R$ from interval $[1, n]$ where $m >> n$ but $|S| \leq |R|$. First, it finds values present in $S$ and counts them in parallel. Then the function tries to compute partial sums. Using later results, it can now map from $S$ to $R$ in parallel.

Despite simplicity and efficiency of this algorithm with work time $O(kn^2)$, it has a section in parallel hashing called partial summation which has to be done in serial and execution of algorithm is opposed to CRCW-PRAM model.

#### 2.1.3.2 EREW-PRAM DFA Minimization Algorithm

Algorithm 4 describes the EREW-PRAM algorithm by Ravikumar and Xiong in 1996 [28]. The basic assumption of this algorithm is that all states are reachable. Otherwise, there should be a pre-processing step to remove unreachable states.

As claimed by author, the outer loop cannot be parallelized in this algorithm. This algorithm fits in the Layer-wise family and uses Moore's technique for minimization. Instead of perfect hashing function introduced in CRCW-PRAM algorithm, parallel sorting is being used here. Additionally, assigning new block number is done in line 8 of Algorithm 4 in serial again.

**Algorithm 4** EREW-PRAM algorithm for DFA minimization [28]

**Input:** DFA $A = (Q, \Sigma, \delta, s, F)$
**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:                  Label q_j ∈ Q with B_{q_j} B_{δ(q_j,a_i)};
6:                  Re-arrange(e.g. by parallel sorting) the states into blocks so that
    the states with the same labels are contiguous;
7:              end for                              ▷ End parallel for
8:              Assign a unique number to all states in the same block
9:          end for
10:     until no new block produced
11: end procedure
```

The cost of above mentioned algorithm is $O(kn^2)$. The main loop iterates at most $n$ times. The number of iterations for the loop at line 3 is $k$. Algorithm requires $n$ processors which leads to total work complexity of $O(kn^2)$.

## 2.2 Map-Reduce and Hadoop

In the past decade, we experienced exponentially growing amount of data which leads to introducing the "Big-Data" concept into computer science. Big-data is the amount of information that is stored but not possible to be processed utilizing traditional algorithms and infrastructures [43]. Caused by unavoidable aggregation of data, computer scientists have put an effort to introduce new processing models to address enterprise concerns in this regards. One of the most famous models is called *Map-Reduce* (MR) which divides data into smaller chunks, does operations and as a result reduces the amount of data. Underneath this model is situated a robust infrastructure consisting of a variety of technologies in order to handle data distribution, fault tolerance, and job management. This enables separating algorithm design from low level technical details.

### 2.2.1 Map-Reduce

Due to massive data mining and processing in one of the largest company working with unimaginable amount of data, Map-Reduce was born in Google as a parallel-distributed programming model working on a cluster of computers [14]. It has been called parallel as tasks are done by dedicating multiple processing units in a parallel environment and distributed as data is laid over separate storages. Hadoop [1] is an open-source implementation of Map-Reduce framework and mainly developed

by Apache Software Foundation. An abstract outline of Map-Reduce framework is illustrated in Figure 6. Data is distributed among cluster of data nodes which are listed using a *namenode* server. Additionally, there is a cluster of compute nodes to run client jobs in parallel. The job scheduler on the other hand schedules jobs among workers. The Execution System is responsible for taking client requests and giving the result back. The proprietary file system of Hadoop called HDFS (Hadoop Distributed File System) enables access to chunk of files for worker nodes in a transparent way. It means compute nodes do not have any information about data storage. The detailed architecture and functionality of Hadoop is beyond this work. Further details can be found in [39].

Providing such reliable, easy to use, abstract and scalable environment made Map-Reduce the most favourite framework not only for organizations but also for computer scientist to propose and express their ideas without concerning about infrastructure complexity and problems.

We shall now provide the programming model by describing the main tasks in the Map-Reduce framework.

#### 2.2.1.1 Map Function

A Map-Reduce job starts by mapping input data to one or more specific reducer to process. Each mapper gets a chunk of files and for each unit of data produces a list of

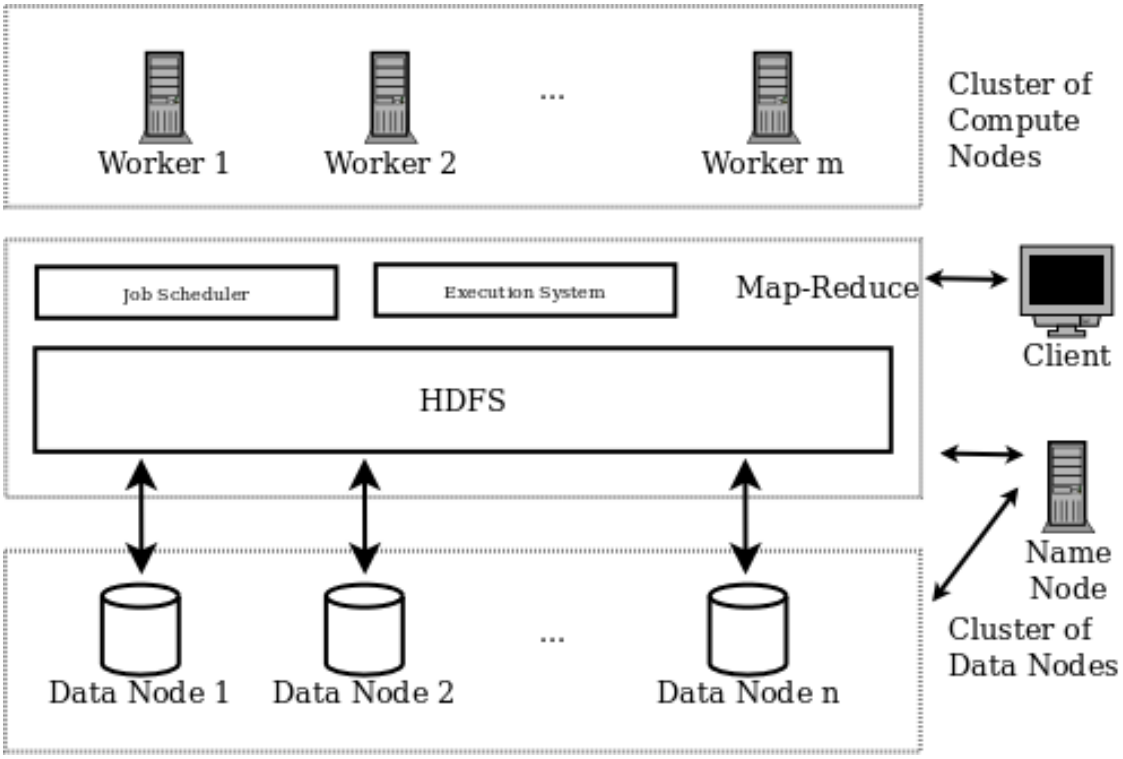


**Figure 6: Map-Reduce architecture**

$\langle key, value \rangle$ pairs based on a *mapping schema* (Section 2.2.2). Pairs with same keys form a list and would be present in the same reducer. The mapping schema is the most important part of a Map-Reduce algorithm which directly affects correctness, time and space complexity. Figure 7 shows how the mapping schema works. Note that a record can be present in different lists and the amount of replication can be independent from all other input records.

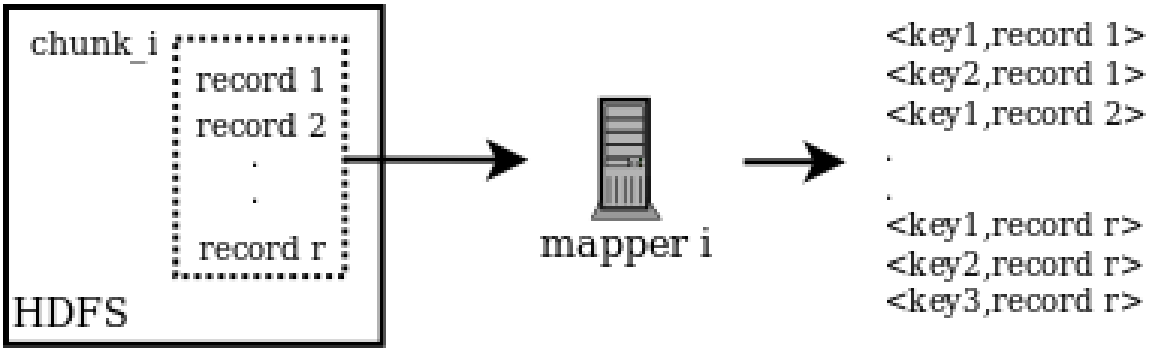


**Figure 7: Mapping in Map-Reduce**

#### 2.2.1.2 Partitioner Function

Although defining mappers and reducers are mandatory for a Map-Reduce job which are main parts of Map-Reduce algorithms, there are some auxiliary functions which are not included in high-level abstract algorithms. However, being informed about existence of them helps us design the conceptual algorithms.

The partitioner function receives pairs of key-value generated by the mapper and partitions them over reducers. This can be done using either a default hash function or the one provided by the user to send a record based on key properties.

#### 2.2.1.3 Reducer Function

Input for the reducer function is the output of mappers. As a result, each reducer gets a key associated to list of records as $\langle key_i, \langle record_{i_1}, record_{i_2}, \ldots, record_{i_r} \rangle \rangle$. The output of reducer is also pairs of $\langle key, value \rangle$ which again could generate multiple values with the same keys. The reducer function defined by the algorithm designer is

applied in parallel to the input list of values within each reducer. The output of each reducer will be written on the *Distributed File System* (DFS).

### 2.2.2 Mapping Schema

As we studied, every input element can be mapped to one or more reducers in the mapping part of Map-Reduce model. Although in most algorithms, we consider unlimited capacity in terms of memory and computation for reducers, in the real world the reducer workers have limitations. Let us denote this limit for maximum input size as $\rho$. A mapping schema is defined to be an assignment of inputs to every reducer with respect to $\rho$. It is also trivial that for every output of the problem at least one reducer has to cover it. It means that reducers can have overlap output. [2].

Having in mind this definition, the *replication rate* $\mathcal{R}$ for an algorithm is the sum of all inputs sent to every reducer divided by the actual input size $|I|$. Furthermore, let $\rho_i$ be the size of input sent to reducer $i$ and denote the number of reducers by $\eta$. Hence,

$$\mathcal{R} = \frac{\sum_{i=1}^{\eta} \rho_i}{|I|}. \tag{6}$$

## 2.3 Automata Algorithms in Map-Reduce

Graphs and automata take a large portion of Big-Data nowadays as part of social networks, web, software models, etc. It is a serious concern for organizations to process and interpret stored data. In this section, two algorithms for processing FA using Map-Reduce model are represented.

### 2.3.1 NFA Intersection

NFA are one of the powerful tools in automata theory having a great benefit in being closed under concatenation, intersection, difference, and homomorphic images. This makes NFA interesting in design and study modular approaches such as web service composition and protocol design.

In [16], authors have studied NFA intersection using Map-Reduce model broadly. They proposed three algorithms with different mapping schema as well as introducing the lower bound on replication rate computing NFA intersection using Map-Reduce data processing model.

Consider automata $A_1 = (Q_1, \Sigma_1, \delta_1, s_1, F_1)$ and $A_2 = (Q_2, \Sigma_2, \delta_2, s_2, F_2)$ as two NFAs accepting $L(A_1)$ and $L(A_2)$ languages. Intersection of these two NFA can be computed as

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

where $L(A) = L(A_1) \cap L(A_2)$. Using associative property of $\otimes$, we can compute intersection of $m$ NFA as $A_1 \otimes \ldots \otimes A_m$.

The lower bound for replication rate (which will be discussed later in section 2.4.2) for NFA intersection is:

$$\mathcal{R} \geq \frac{\rho \times \frac{|\delta_1| \times \cdots \times |\delta_m|}{k^{m-1}}}{(\rho/m)^m \times (|\delta_1| + \cdots + |\delta_m|)} \tag{7}$$

where $\rho$ is maximum capacity of a reducer and $k = |\Sigma|$ is number of alphabets.

**Mapping based on states**

The basic assumption is that there are $n^m$ reducers where $m$ denotes number of NFA and $n$ is maximum transitions in any. In first approach, mappers emit key-value pairs as $\langle K, (p_i, c_i, q_i)\rangle$ where $(p_i, c_i, q_i)$ is a transition and key is for all $i_j \in \{1, \ldots, n\}$:

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

Note that $h$ is a function as $h : Q \mapsto \{1, \ldots, n\}$. Replication rate for this method is $\mathcal{R} \leq (\frac{nm}{\rho})^{m-1}$.

**Mapping based on alphabet**

As the second approach, it is been assumed that there is one reducer for each

alphabet symbol. So the mappers emit key-value pair for every transition from automata $A_i$ as:

$$\langle h(c), (p_i, c, q_i)\rangle.$$

Every reducer will output transition as $\langle (p_1, \ldots, p_m), c, (q_1, \ldots, q_m)\rangle$. Replication rate is $\mathcal{R} = 1$.

**Mapping based on both alphabet and state**

In this method, we have two hashing function $h_s : p_i \mapsto \{1, \ldots, b_s\}$ and $h_a : a_j \mapsto \{1, \ldots, b_a\}$. For all $i_j \in \{1, \ldots, b_s\}$, we will map each transition to reducers $\langle i_1, \ldots, i_{i-1}, h_s(p_i), i_{i+1}, \ldots, i_m, h_a(c_i)\rangle$. It has been assumed that total number of reducers are $b_s^{m-1} \cdot b_a$. Replication rate for this method is $\mathcal{R} \leq (\frac{nml}{\rho^k})^{m-1}$ where $l$ is average number of alphabet symbols received by a reducer.

### 2.3.2 DFA Minimization (Moore-MR)

To the best of our knowledge, the only algorithm for DFA minimization proposed on Map-Reduce model is published in [18]. Author used method of Moore's DFA minimization for distributing data among reducer and compute minimal DFA.

Algorithm is introduced as a sequence of Map-Reduce jobs which output of one job is input of next consecutive one. Mappers will emit pair of key-values as:

$$\langle h(p), (p, a, q, c, \Delta)\rangle$$

where $\Delta \in \{0, 1\}$ indicates if value is a transition (0) or a reverse transition (1). The reverse transition for $(p, a, q)$ is defined as $(q, a, p)$. Assume that there are $n$ reducers (number of reducers is equal to number of states), transitions belonging to each state will be present in one reducer called $h(p)$. In each reducer, new equivalence class is computed as

$$P^i_{p_j} = P^{i-1}_p P^{i-1}_{q_{j1}} \dots P^{i-1}_{q_{jk}}$$

having $k$ alphabet where $P^i_p$ denotes block of state $p$ in round $i$ represented as a bit string. In order to calculate the classes for each state, it is required to have class of target state $q$ in each transition. This can be achieved if each reverse transition also be sent to its target (which is actually source of the original transition). In above mentioned procedure, we just update class of source states in original transitions. Therefore, we need to send each reverse transition to its source (which in this case is target of the original transition). This procedure requires replication rate $\frac{3}{2}$ if the input is the union of both transitions and their reverses. The Algorithm will stop if there is no new block generated within the partition. Decision of convergence is being made as a parallel voting system where each reducer is responsible for one state ad votes to continue if number of adjacent blocks is changed. Same as in Moore's algorithm, it requires $O(n)$ rounds to generate the minimal DFA.

Beside simplicity of the algorithm, it suffers from exploding amount of data. Consider that algorithm will take $r$ rounds to finish. Size of class identifiers at round $i$

would be $k^i$. This directly will affect communication cost despite having replication rate $O(1)$. An improvement for this algorithm will be proposed later in 3.1.

## 2.4 Cost Model

We now introduce a parameter that helps us modelling the Map-Reduce cost respecting both computation and communications. First, we will discuss a general and basic method to find the minimum required communication to solve a function in parallel. Then after introducing a lower bound recipe for determining replication rate and communication cost for Map-Reduce algorithm having one round was first suggested in [2], a new enhanced version of Map-Reduce cost model from [34] will be discussed.

### 2.4.1 Communication Complexity

In parallel computing environment multiple parties are engaged to accomplish either one single or multiple tasks, based on one of the computer architectures in Flynn's taxonomy [15]:

SISD Single Instruction Single Data which is a serial computer running single instruction on one instance of data stream.

SIMD Single Instruction Multiple Data which is an array of processors computing multiple streams of data. Obviously, data should be naturally parallelized.

MISD Multiple Instructions Single Data which multiple processing units are working on the same data stream. This model is more about integrity, consistency and fault tolerance. All parties should be agree about the result.

MIMD Multiple Instructions Multiple Data, In this model, multiple processing units are working on multiple streams of data at the same time. This model is the most common one nowadays and Distributed Systems are a generalization of this model.

Based on the selected model, most of the time it is required that processing units communicate data to fulfil their tasks. An algorithm designed to run on one of the parallel or distributed computing models, should be analysed regarding to the amount of communication done in the system. Considering the fact that in Map-Reduce model, after mapping data to associated reducers, one pre-defined function will be applied on each list of key-value pairs, this will be equivalent to SIMD definition of Flynn's model which also can extended to MIMD.

*Communication Complexity* (CC) is an area has been studied for a long time. It has been used for proving lower bounds in many areas. In this thesis Yao's two-party CC model [41] described and extended by Kushilevitz [23] is employed which is best match to SIMD model. In Yao's model, there are two parties (Alice and Bob) wishing to evaluate a Boolean function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$. Here, $x \in \{0,1\}^n$ is input in Alice's hand and Bob just got $y \in \{0,1\}^n$. The aim of this model is to

determine the amount of communication and so we do not care how parties will do the computation. Although there are different methods for CC, we use deterministic one which would gives more robust analysis to this problem.

The communication protocol $\mathcal{P}$ is responsible to decide which party is allowed to which decides about who, when and what is sent. In addition, the protocol decides whenever communication shall be terminated and when it does, what is the final result of $f$. Also, if $\mathcal{E}_{\mathcal{P}}(x, y) = \{m_1, \ldots, m_k\}$ contains messages exchanged between Alice and Bob during execution of protocol $\mathcal{P}$ over input $x$ and $y$, then we can denote $|\mathcal{E}_{\mathcal{P}}(x, y)| = \Sigma_{i=1}^{k} |m_i|$. Now we will define *Deterministic Communication Complexity* (DCC) of $\mathcal{P}$ as

$$\mathcal{D}(\mathcal{P}) = \max_{(x,y)\in\{0,1\}^n\times\{0,1\}^n} |\mathcal{E}_{\mathcal{P}}(x, y)| \tag{8}$$

DCC of a function $f$ can be defined as

$$\mathcal{D}(f) = \min_{\mathcal{P}:\mathcal{P} \text{ computes } f} \mathcal{D}(\mathcal{P}) \tag{9}$$

The worst case for any function $f : \{0, 1\}^n \times \{0, 1\}^n \to \{0, 1\}$ is one party Alice sends all its input to Bob and Bob will send the result back. Hence,

$$\forall f \ , \mathcal{D}(f) \leq n + 1 \tag{10}$$

is an upper bound. In order to find the minimum DCC, first we define *rectangles*.

**Definition 1** *A rectangle is a subset of* $\{0,1\}^n \times \{0,1\}^n$ *of the form* $A \times B$*, where each of* $A$ *and* $B$ *is a subset of* $\{0,1\}^n$*. A rectangle* $R = A \times B$ *is called* $f$*-monochromatic if for every* $x \in A$ *and* $y \in B$ *the value of* $f(x,y)$ *is the same. [23]*

Here we define $C^{\mathcal{P}}(f)$ as the minimum number of $f$-monochromatic rectangles in protocol $\mathcal{P}$ that partition the space of inputs, $\{0,1\}^n \times \{0,1\}^n$.

**Lemma 4** *For every function* $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$*,*

$$\mathcal{D}(f) \geq \log_2 C^{\mathcal{P}}(f) \tag{11}$$

*proof.* Every protocol $\mathcal{P}$ will partition input space of $\{0,1\}^n \times \{0,1\}^n$ into $f$-monochromatic rectangles. Each message would lead to two new states. Thus, the number of rectangles which equals to the number of possible communications ($C^{\mathcal{P}}(f)$) is at most $2^{D(\mathcal{P})}$. From Equations 8 and Equation 9, we know that $D(f) \leq D(\mathcal{P})$. Hence, $C^{\mathcal{P}}(f) \leq 2^{D(f)}$ and it is equivalent to say $\mathcal{D}(f) \geq \log_2 C^{\mathcal{P}}(f)$ $\square$

In Lemma 4 we found that in order to calculate DCC, it is required to find $C^{\mathcal{P}}(f)$. In [25], the *fooling set* method was explicitly used to find lower bound in VLSI problems.

**Definition 2** *A set of input pairs* $S \subseteq X \times Y$ *is called a fooling set (of size* $\ell$*) with*

*respect to* $f$ *if there exists* $z \in \{0,1\}$ *such that*

1. $\forall (x,y) \in S, f(x,y) = z$

2. *For any two pairs* $(x_1,y_2)$ *and* $(x_2,y_1)$ *either* $f(x_1,y_2) \neq z$ *or* $f(x_2,y_1) \neq z$

**Lemma 5** *If there exists a fooling set of size* $\ell$ *with respect to* $f$ *then*

$$\mathcal{D}(f) \geq log_2 \ell \tag{12}$$

*proof.* Suppose that $\mathcal{D}(f) < \ell$. Then, from Lemma 4 we have $C^{\mathcal{P}}(f) < \ell$. Thus there should be two pairs in fooling set $\{x_i, y_i\}$ and $\{x_j, y_j\}$ which belong to the same rectangle $A \times B$. This means that $x_i, x_j \in A$ and $y_i, y_j \in B$. Thus $(x_i, y_j), (x_j, y_i) \in A \times B$. By the definition of fooling set, $f(x_i, y_i) = f(x_j, y_j) = b$ while at least either $f(x_i, y_j) \neq b$ or $f(x_j, y_i) \neq b$. This implies that the rectangle $A \times B$ is not $f$-monochromatic [23] $\square$.

Now consider that Alice and Bob are willing to solve more than one function. We define the notation of $\mathcal{D}(f,g)$ if two parties are engaged to get the result of $f(x_f, y_f)$ and $g(x_g, y_g)$. The relation between $\mathcal{D}(f,g)$ and $\mathcal{D}(f)$ and $\mathcal{D}(g)$ is an open problem [23]. We know that $\mathcal{D}(f,g)$ can not be greater than $\mathcal{D}(f) + \mathcal{D}(g)$ but we don't know that if it is smaller or not. And if it is smaller, how much? Hence, if we have $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ we can say,

$$\mathcal{D}(\mathcal{F}) = m(\max_{\forall f_i \in \mathcal{F}} \mathcal{D}(f_i)). \tag{13}$$

The minimum communication complexity for DFA minimization has not been determined yet. Though we know that it is in NL [11][12]. However, in Section 3.4, we will propose an analysis for finding minimum compulsory amount of data to be exchanged between multiple parties for minimizing DFAs using lower bound recipe and extended Map-Reduce CC model which will be discussed more in this section.

### 2.4.2 The Lower Bound Recipe for Replication Rate

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 [24]. Afrati *et al.* proposed a cost model in [2] to discover the lower bound on the replication rate for problems in map-reduce.

Recall the mapping schema from Section 2.2.2, a mapper is free to send each input to more than one reducer. This is called replication rate. The replication rate will directly affect effectiveness of the algorithm. On the other hand, for some problems, there is a lower bound for replication rate as well. Here, in this section, the recipe from [2] to find this lower bound is presented.

Regarding the notion that every output shall be produced by at least one reducer, this particular reducer should receive all required inputs to compute it. Denote $g(\rho)$

as an upper bound on the number of outputs reducer with capacity of $\rho$ can cover. Thus, $g(\rho_i)$ is the number of outputs that reducer $i$ covers. This can be simplified by a formula as follows:

$$\sum_{i=1}^{\eta} g(\rho_i) \geq |O| \tag{14}$$

Using equations 6 and 14, as well as assuming $\frac{g(\rho_i)}{\rho_i}$ is monotonically increasing in $\rho_i$, we get the lower bound on replication defined as:

$$\mathcal{R} = \frac{\sum_{i=1}^{\eta} \rho_i}{|I|} \geq \frac{\rho \times |O|}{g(\rho) \times |I|} \tag{15}$$

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

This simple but powerful recipe helps to find the minimum replication rate required for a problem. However, this would not be a complete model to compare Map-Reduce algorithms. An extended model described in next section will give a better measure for Map-Reduce solutions.

### 2.4.3 Computational Complexity of Map-Reduce

Above mentioned methods in evaluating complexity of Map-Reduce algorithms are not sufficient. Standard CC does not apply in this context directly because amount of

communication is more than linear. Additionally, lower bound recipe is just working on one round algorithms. In order to remedy these issues, György Turán in [34] has proposed a formal definition of Map-Reduce model using Turing Machine to analyse Map-Reduce algorithms and compare them not only with each other, but also with other parallel and distributed computing models.

Time-space tradeoffs are studied in [7] by relating TIME($T(n)$) and SPACE($S(n)$). $TISP(T(n), S(n))$ are problems which can be decided by a Turing Machine where $T(n)$ is time complexity and $S(n)$ is required space in presence of input size $n$. As it is been discussed, Map-Reduce algorithms consist two major sections: *mapper* and *reducer*. In above mentioned paper, each of them is a separate run of the same Turing machine $M(m, r, n, \rho)$ where $m$ is a flag denoting whether run map or reduce function, $r$ is round number, $n$ is total input size and $\rho$ as machine input size . Moreover, it requires $R$ rounds of Map-Reduce run to accomplish the task. On the other hand, let us denote a mapper as $\alpha$ and reducer as $\beta$, both polynomial time-space Turing machines. Then we can model a Map-Reduce run as series of mappers and reducers $\alpha_1, \beta_1, \alpha_2, \beta_2, \ldots$. Authors in [34] defined uniform deterministic Map-Reduce Complexity model (MRC) as described in definition 3.

**Definition 3** *A language $L$ is said to be in $MRC[f(n), g(n)]$ if there is a constant $0 < c < 1$, an $O(n^c)$-space and $O(g(n))$-time Turing machine $M(m, r, n, \rho)$, and an $R = O(f(n))$, such that for all $x \in \{0, 1\}^n$, the following holds,*

1. *Letting* $\alpha_r = M(1, r, n, -)$, $\beta_r = M(0, r, n, -)$, *the MRC machine* $M_R = (\alpha_1, \beta_1, \ldots, \alpha_R, \beta_R)$ *accepts* $x$ *if and only if* $x \in L$

2. *Each* $\alpha_r$ *outputs* $O(n^c)$ *distinct keys*

The first parameter $c$ is related to the replication rate. Recall Equation 6, if $c = 1$,

$$\mathcal{R} = \frac{\sum_{i=0}^{\eta} |I|}{|I|} = \eta$$

which means we are sending whole data to all reducers. This is also an upper bound for replication rate. On the other hand if $c = 0$ then we have $n^c = 1$ means every input record would be sent once and $\mathcal{R} = 1$. Number of rounds $R = f(n)$ however has a serious impact, as initializing a Map-Reduce job in practice requires an initialization overhead mainly containing huge amount of I/O read and write as well as network communications. After that, although $g(n)$ as mentioned before, has the least effect on evaluation of Map-Reduce algorithms, yet shall be counted as a complexity parameter. Last but not least, is input size $\rho$ to each Turing machine. As it can be combination of intermediate data, total input and subset of total input.

Later in Chapter 3 and 4, we will use this model to compare our algorithms.

# Chapter 3

# DFA Minimization Algorithms in Map-Reduce

In this chapter, DFA minimization algorithms in Map-Reduce model are presented. First we discuss the Moore-MR algorithm introduced in Section 2.3.2 in more detail. Then after, an algorithm for PPHF in Map-Reduce model (PPHF-MR) will be proposed. Finally, an improved version of Moore-MR, called Moore-MR-PPHF will be constructed by applying PPHF-MR. Then a new naive algorithm called Hopcroft-MR, based on Hopcroft's algorithm, will be proposed. Right after, an enhanced version of Hopcroft-MR, called Hopcroft-MR-PAR is represented. At the end of this chapter, communication and computation costs will be analysed.

## 3.1 Moore's DFA Minimization in Map-Reduce (Moore-MR-PPHF)

Moore's algorithm (Section 2.1.1) is one of the most simple and straightforward methods for minimizing DFA and such it can be easily developed in parallel environments. In this section, the enhanced version of the algorithm described in Section 2.3.2 (Moore-MR) from [18] will be introduced.

The Moore-MR algorithm consists of a pre-processing stage, and one or more rounds of map and reduce functions. Additionally, each reducer which is associated to transitions of one state, will check whether the number of equivalence classes has been changed or not. Based on this parallel voting system, algorithm will decide about convergence and finish the process. The final minimized DFA can be generated from output of the last round (Algorithm 5).

**Pre-processing:** Let the automaton be $A = (Q, \{a_1, \ldots, a_k\}, \delta, s, F)$. We first build a set $\Delta$ from $\delta$. This set will consist of annotated transitions of the form $(p, a, q, \pi_p, D)$, where $\pi_p$ is a bit-string representing the initial block where the state $p$ belongs, $D = +$ indicates that the tuple represents a transition (an outgoing edge), and $D = -$ indicates that the tuple is a "dummy" transition carrying in its fourth position the information of the initial block of the aforementioned state $q$ (now occuring in the

first position). More specifically, for each $(p, a, q) \in \delta$ we insert into $\Delta$

$$\begin{cases} (p, a, q, \mathtt{1}, +) \text{ and } (q, a, p, \mathtt{1}, -) & \text{when } p, q \in F \\ (p, a, q, \mathtt{1}, +) \text{ and } (q, a, p, \mathtt{0}, -) & \text{when } p \in F, q \in Q \setminus F \\ (p, a, q, \mathtt{0}, +) \text{ and } (q, a, p, \mathtt{1}, -) & \text{when } p \in Q \setminus F, q \in F \\ (p, a, q, \mathtt{0}, +) \text{ and } (q, a, p, \mathtt{0}, -) & \text{when } p, q \in Q \setminus F. \end{cases}$$

Recall that Moore's algorithm is an iterative refinement of the initial partition $\pi = \{F, Q \setminus F\}$. In Moore-MR, each reducer will be responsible for one or more states $p \in Q$. Since there is no global data structure, we need to annotate each state $p$ with the block it currently belongs to. This annotation is kept in all transitions $(p, a_i, q_i), i = 1, \ldots, k$. Hence tuples $(p, a_i, q_i, \pi_p, +)$ are in $\Delta$. In order to update class of equivalency for $p$, the reducer also needs to know the current block of all the above states $q_i$, which is why tuples $(q_i, a_i, p, \pi_{q_i}, -)$ are in $\Delta$. Furthermore, the new block of state $p$ will be needed in the next round when updating the block annotation of states $r_1, \ldots, r_m$, where $(r_i, a_{i_j}, p)$ are all transitions leading to $p$. Thus tuples $(p, a_{i_j}, r_i, \pi_p, -)$ are in $\Delta$. Also we assume that the number of reducers available for Moore-MR is equal to the number of states $(n)$.

**Map function:** Let $h : Q \to \{1, \ldots, n\}$ be a hashing function. Each mapper gets a

chunk of $\Delta$ as input, and for each tuple $(p, a, q, \pi_p, D)$ emits

$$\begin{cases} \langle h(p), (p, a, q, \pi_p, D)\rangle & \text{when } D = + \\ \langle h(p), (p, a, q, \pi_p, D)\rangle \text{ and } \langle h(q), (p, a, q, \pi_p, D)\rangle & \text{when } D = - \end{cases}$$

**Reducer function:** Each reducer is denoted by an integer number in range of $[1, n]$ and for all $p \in Q$, reducer $h(p)$ receives the list of outgoing transitions

$$(p, a_1, q_1, \pi_p, +), \ldots, (p, a_k, q_k, \pi_p, +)$$

as well as the dummy for its outgoing transitions

$$(q_1, a_1, p, \pi_{q_1}, -), \ldots, (q_k, a_k, p, \pi_{q_k}, -),$$

and the dummy for its incoming transitions $(p, a_{1_j}, r_1, \pi_p, -), \ldots, (p, a_{m_j}, r_m, \pi_p, -)$ where $r_1, \ldots, r_m$ are all the states with transitions into $p$. The reducer computes the new equivalence class of state $p$ in line 16 to 25 of the algorithm as

$$\pi_p^{r+1} \leftarrow \pi_p^r \cdot \pi^r_{\delta(p,a_1)} \cdot \ldots \pi^r_{\delta(p,a_k)} \tag{16}$$

and then in line 26 to 29, writes the new value $\pi_p^{r+1}$ in all the above tuples. At the end, in line 31, it will emit a *true* record to vote YES for continuing the algorithm if

there is any new class of equivalency around this state.

The proof of correctness is well described in [18]. But the cost analysis lacks considering exponentially growing amount of data. The size of equivalence class labels, starting from 1 bit for original DFA, is:

$$|\pi_p^0| = 1 \text{ and } |\pi_p^r| = (k+1) \cdot |\pi^{r-1}| \ \rightarrow |\pi_p^r| = (k+1)^r \tag{17}$$

By denoting a transition as $t$ and number of storage units to store a transition record as $|t|$, the communication cost of this algorithm claimed to be $\frac{5}{2}(2kn|t|)R = 5kn|t|r$ where $R \leq n$ is number of rounds. The constant $\frac{5}{2}$ comes from the replication rate $\frac{3}{2}$ plus the output of each round which is equal to writing down all the transitions and dummy transitions once. However, the size of each transition varies at each iteration. Let's denote the size of transition at iteration $r$ as $|t^r|$. Then $|t^r| = |\langle p, a, q, \pi_p^r, D\rangle| = \log n + \log k + \log n + |\pi_p^r| + 1 = 2\log n + \log k + |\pi_p^r| + 1$. Now we can rewrite the communication cost as:

$$CC = 5kn \sum_{r=1}^{R} t^r = 5kn \sum_{r=1}^{R} (2\log n + \log k + |\pi_p^r| + 1)$$

which by Equation 17, it will be:

$$CC = 5kn(R(2\log n + \log k + 1) + \sum_{r=1}^{R} (k+1)^r) < 5kn\frac{k^R - 1}{k-1} = O(k^n n) \tag{18}$$

**Algorithm 5** Moore's algorithm in Map-Reduce (Moore-MR)

**Input:** $\Delta$ as union of $\delta$ and dummy transitions
**Output:** Updated $\delta$ exposing the minimal DFA

```
1: repeat

2:     function MAPPER(Δ)
3:         for all t ∈ Δ do
4:             if D = + then
5:                 Emit (⟨h(p), t⟩
6:             else
7:                 if p ≠ q then
8:                     Emit ⟨h(q), t⟩
9:                     Emit ⟨h(p), t⟩
10:                end if
11:            end if
12:        end for
13:    end function

14:    function REDUCER(⟨ K, [t_1, t_2, ..., t_j] ⟩)
15:        π̂ ← null
16:        for all t_i do
17:            if D = + then
18:                if p_i = q_i then
19:                    π̂ ← π̂ π_{p_i}
20:                else
21:                    s ← find dummy transition of t_i from the input list
22:                    π̂ ← π̂ π_{s.p}
23:                end if
24:            end if
25:        end for
26:        Update π_p in all transitions t_i ∈ [t_1, t_2, ..., t_j] by π̂
27:            Emit ⟨K, t_i⟩
28:        Update π_q in all dummy transitions t_i ∈ [t_1, t_2, ..., t_j] where q = K by π̂
29:            Emit ⟨K, t_i⟩
30:        if the number of equivalence classes around state p = K changes then
31:            Emit (⟨p, true⟩
32:        end if
33:    end function

34: until no new equivalence class produced
```

In order to avoid this data explosion, a new job at each iteration is introduced in next section to change the labels using PPHF.

### 3.1.1 PPHF in Map-Reduce (PPHF-MR)

Most parallel algorithms, as mentioned in 2.1.3, use PPHF. The function $PHF : S \rightarrow R$ where $S \subset S'$, and $|S| = |R| << |S'|$ is a one-to-one and total function has the following property:

$$\forall x_1, x_2 \in S \ (\text{PHF}(x_1) = \text{PHF}(x_2) \rightarrow x_1 = x_2)$$

In order to apply PPHF in Map-Reduce (PPHF-MR), let $S = \pi^r$ and from Equation 17, $S' = \{0, 1, \ldots, n^{(k+1)}\}$. On the other hand, let $R = n^2$. Now we can denote PPHF-MR as,

$$\textit{PPHF-MR} : \pi^r \rightarrow n^2$$

The idea is that the mapper in PPHF job will map transitions based on equivalence block label, $h(\pi_p)$, and all with the same classes will get a unique number. However, recall that $\pi_p >> n$ and it will cause transitions from two class of equivalency meet each other at the same reducer due to nature of hashing function. In order to avoid

this conflict of labels, each reducer $i$ has a range of labels from

$$[i \cdot n, (i+1)n - 1] \tag{19}$$

where $0 \leq i \leq n-1$. So each reducer has a range of distinct $n$ numbers:

$0 : [0, n-1]$

$1 : [n, 2n-1]$

$\vdots$

$n-1 : [n^2 - n, n^2 - 1]$

One may observe for the very first rounds of minimization process, a number of transitions will map to a few reducers which overflows reducer capacity. This is due to the fact that the partition has few blocks. Let's denote a threshold $T$ as the computation capacity of the node running the master program for computing a certain number of groups in memory. Assume that we have a homogeneous cluster. Obviously,

$$\rho > k|t| \rightarrow T = k|t| > |\pi^r|$$

where $k$ is the number of alphabet symbols and $|t|$ is size of a transition. As a result, if number of blocks in partition $\pi$ is beyond of threshold $T$, PPHF-MR shall be executed. Otherwise, we will extend Map-Reduce model as follows.

**Extending Map-Reduce model.** Recall the structure of Map-Reduce illustrated in Figure 6. As discussed before, *namenode* is responsible for keeping index of files and directories on HDFS. The information about the folders and where the files are stored can be retrieved using an API call to *namenode*. Here this property can be employed to implement in-memory PHF. Suppose that we store transitions associated to the states within a block in same directory and all the directories in one upper level directory, called *DFA*. Now querying list of sub-directories under *DFA*, it will get back a list of labels for every block. Despite the order of block labels in the list, we will associate index of retrieved label as new label name. This method follows exactly definition of PHF. Adding PPHF-MR at the end of each iteration in line 33 of Algorithm 5, block labels are mapped into range $[0 - n^2]$ and we call this algorithm Moore-MR-PPHF.

### 3.1.2 An Example of Running Moore-MR-PPHF

In order to have better understanding of Moore's algorithm in Map-Reduce and PPHF-MR, a sample DFA will be minimized.

Consider DFA from Figure 8 where $Q = \{1, 2, 3, 4, 5, 6\}$, $\Sigma = \{a, b\}$, $F = \{1, 3, 4, 5\}$ and $\pi^0 = \{0, 1\}$. The input contains transitions as well as dummy transitions:

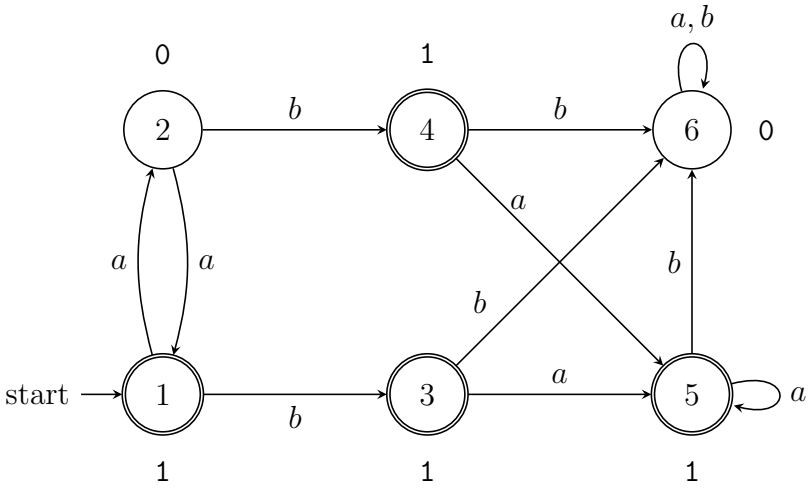


**Figure 8: An example DFA**

$$\begin{aligned}\Delta \quad = \{ \quad & \langle 1, a, 2, \mathtt{1}, + \rangle, \langle 1, b, 3, \mathtt{1}, + \rangle, \langle 2, a, 1, \mathtt{0}, + \rangle, \langle 2, b, 4, \mathtt{0}, + \rangle, \langle 3, a, 5, \mathtt{1}, + \rangle, \\ & \langle 3, b, 6, \mathtt{1}, + \rangle, \langle 4, a, 5, \mathtt{1}, + \rangle, \langle 4, b, 6, \mathtt{1}, + \rangle, \langle 5, a, 5, \mathtt{1}, + \rangle, \langle 5, b, 6, \mathtt{1}, + \rangle, \\ & \langle 6, a, 6, \mathtt{0}, + \rangle, \langle 6, b, 6, \mathtt{0}, + \rangle, \\ & \langle 2, a, 1, \mathtt{0}, - \rangle, \langle 3, b, 1, \mathtt{1}, - \rangle, \langle 1, a, 2, \mathtt{1}, - \rangle, \langle 4, b, 2, \mathtt{1}, - \rangle, \langle 5, a, 3, \mathtt{1}, - \rangle, \\ & \langle 6, b, 3, \mathtt{0}, - \rangle, \langle 5, a, 4, \mathtt{1}, - \rangle, \langle 6, b, 4, \mathtt{0}, - \rangle, \langle 6, b, 5, \mathtt{0}, - \rangle \}\end{aligned}$$

It can be seen there is no need to have dummy transitions for self loop because the transition carries equivalence classes for both sides. Additionally, let us define hashing function $h$ as: $h(1) = 0$, $h(2) = 1$, $h(3) = 2$, $h(4) = 3$, $h(5) = 4$ and $h(6) = 5$. By applying mapper function to the above set of records we get distribution of them

all over the reducers as shown in Table 2. Note that mapper is a function of source state which leads to having same destination for the rest of algorithm.

**Table 2: Distribution of records among the reducers in first round of Moore-MR-PPHF with input DFA illustrated in Figure 8**

| Reducer 0 | Reducer 1 | Reducer 2 | Reducer 3 | Reducer 4 | Reducer 5 |
|---|---|---|---|---|---|
| $\langle 1, a, 2, \mathtt{1}, + \rangle$ | $\langle 2, a, 1, \mathtt{0}, + \rangle$ | $\langle 3, a, 5, \mathtt{1}, + \rangle$ | $\langle 4, a, 5, \mathtt{1}, + \rangle$ | $\langle 5, a, 5, \mathtt{1}, + \rangle$ | $\langle 6, a, 6, \mathtt{0}, + \rangle$ |
| $\langle 1, b, 3, \mathtt{1}, + \rangle$ | $\langle 2, b, 4, \mathtt{0}, + \rangle$ | $\langle 3, b, 6, \mathtt{1}, + \rangle$ | $\langle 4, b, 6, \mathtt{1}, + \rangle$ | $\langle 5, b, 6, \mathtt{1}, + \rangle$ | $\langle 6, b, 6, \mathtt{0}, + \rangle$ |
| $\langle 2, a, 1, \mathtt{0}, - \rangle$ | $\langle 2, a, 1, \mathtt{0}, - \rangle$ | $\langle 3, b, 1, \mathtt{1}, - \rangle$ | $\langle 4, b, 2, \mathtt{1}, - \rangle$ | $\langle 5, a, 3, \mathtt{1}, - \rangle$ | $\langle 6, b, 3, \mathtt{0}, - \rangle$ |
| $\langle 3, b, 1, \mathtt{1}, - \rangle$ | $\langle 1, a, 2, \mathtt{1}, - \rangle$ | $\langle 5, a, 3, \mathtt{1}, - \rangle$ | $\langle 5, a, 4, \mathtt{1}, - \rangle$ | $\langle 5, a, 4, \mathtt{1}, - \rangle$ | $\langle 6, b, 4, \mathtt{0}, - \rangle$ |
| $\langle 1, a, 2, \mathtt{1}, - \rangle$ | $\langle 4, b, 2, \mathtt{1}, - \rangle$ | $\langle 6, b, 3, \mathtt{0}, - \rangle$ | $\langle 6, b, 4, \mathtt{0}, - \rangle$ | $\langle 6, b, 5, \mathtt{0}, - \rangle$ | $\langle 6, b, 5, \mathtt{0}, - \rangle$ |

After executing reducer and finding new labels, $\pi^1$ looks like:

$[\mathtt{000}] = \{6\}$, $[\mathtt{011}] = \{2\}$, $[\mathtt{101}] = \{1\}$, $[\mathtt{110}] = \{3, 4, 5\}$

As it can be seen $\pi^1 \subset \{\mathtt{0},\mathtt{1}\}^3$ and $n = 6 \in \{0, 1\}^3$. So there is no need to apply PPHF-MR. On the other hand, for convergence, all the reducers can see the changes in $\pi^0$ by looking at adjacent states. It is because of for the round 0, they just was aware of equivalence class of themselves and after first round, they become aware of adjacent states. As a result, algorithm have to run at least one round. DFA after the first round looks like Figure 9.

By running second round partition $\pi^2$ is as follows:

$[\mathtt{000\ 000\ 000}] = \{6\}$, $[\mathtt{011\ 101\ 110}] = \{2\}$, $[\mathtt{101\ 011\ 110}] = \{1\}$,

$[\mathtt{110\ 110\ 000}] = \{3, 4, 5\}$

Considering convergence, all states except 6 which has all transitions as self-loops, in the first round had seen two different classes $\mathtt{0}$ and $\mathtt{1}$. After second iterations, $3, 4, 5$

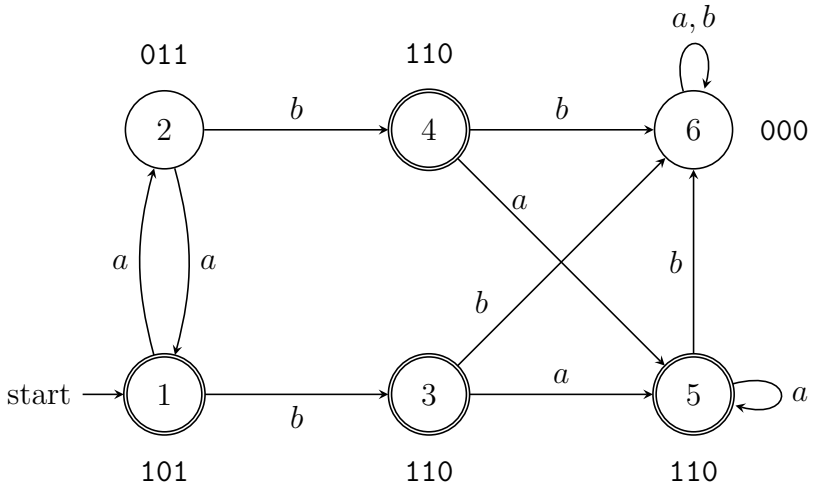


**Figure 9: New partition after the first round of Moore-MR-PPHF execution**

see two classes of 110 and 000, so from their point of view nothing is changed. But 2 observes different classes of 011, 101 and 110 which means a new class is generated. Then algorithm has to continue for the next round. But before starting new round, observe that $\pi^2 \subset \{0,1\}^9$ and it is required to run PPHF-MR to shrink number of bits. Based on PPHF mapper we have reducer arrangement as demonstrated in Table 3 by using hash function as $h(\pi_p) = \pi_p$ **mod** $n$ (note that class labels are not shown for simplicity).

Now using Equation 19, [011 101 110] will get label $4 \cdot 6 = 24 =$ [11000], [101 011 110] will get label $2 \cdot 6 = 12 =$ [01100], [000 000 000] will get label $0 \cdot 6 = 0 =$ [00000], and finally [110 110 000] will get label $0 \cdot 6 + 1 = 1 =$ [00001]. The result DFA looks like Figure 10.

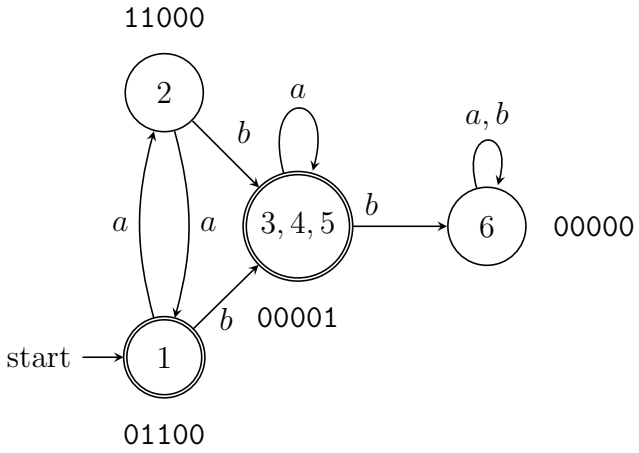


**Figure 10: Minimized version of DFA in Figure 9**

**Table 3: Data distribution across reducers in PPHF-MR**

| Reducer 0 | Reducer 1 | Reducer 2 | Reducer 3 | Reducer 4 | Reducer 5 |
|---|---|---|---|---|---|
| $\langle 3, a, 5, -, + \rangle$ | | $\langle 1, a, b, -, + \rangle$ | | $\langle 2, a, 1, -, - \rangle$ | |
| $\langle 4, b, 6, -, + \rangle$ | | $\langle 1, a, b, -, - \rangle$ | | $\langle 2, b, 4, -, + \rangle$ | |
| $\langle 5, a, 5, -, + \rangle$ | | $\langle 1, b, 3, -, + \rangle$ | | $\langle 2, a, 1, -, + \rangle$ | |
| $\langle 3, b, a, -, - \rangle$ | | | | | |
| $\langle 3, b, 6, -, + \rangle$ | | | | | |
| $\langle 5, a, 4, -, - \rangle$ | | | | | |
| $\langle 5, a, 3, -, - \rangle$ | | | | | |
| $\langle 4, b, 2, -, - \rangle$ | | | | | |
| $\langle 6, b, 3, -, - \rangle$ | | | | | |
| $\langle 6, b, 4, -, - \rangle$ | | | | | |
| $\langle 6, b, 5, -, - \rangle$ | | | | | |
| $\langle 5, b, 6, -, + \rangle$ | | | | | |
| $\langle 4, a, 5, -, + \rangle$ | | | | | |
| $\langle 6, a, 6, -, + \rangle$ | | | | | |
| $\langle 6, b, 6, -, + \rangle$ | | | | | |

## 3.2 Hopcroft's DFA Minimization algorithm in Map-Reduce (Hopcroft-MR)

This section proposes Hopcroft-MR, a Map-Reduce version of Hopcroft's algorithm. Algorithm 6 is the master program of Hopcroft-MR. Denote that members of a tuple can be accessed using the *dot* operator, i.e, if $t = (p, a, q)$, then state $p$ of this tuple is denoted by $t.p$. The input of the algorithm is $\delta$ classified in groups $[\mathrm{i}] \in \pi^0 = \{0, 1\}$ as follows:

$[1] = \{t | t \in \delta, t.p \in F\}$

$[0] = \{t | t \in \delta, t.p \notin F\}$

and the output is the minimized DFA. A transition $t$ as $\delta(p, a) = q$ is defined in form of $\langle p, a, q, \pi_p, \pi_q \rangle$.

Table 4 introduces data structures used in Hopcroft-MR that will be described later on in more detail.

**Table 4: Data structures used in Hopcroft-MR**

| Tuple Name | Data Structure | Description |
|---|---|---|
| Transition tuple | $\langle p, a, q, \pi_p, \pi_q \rangle$ | $(p, a, q) \in \delta$, $p \in \pi_p$ and $q \in \pi_q$ |
| Block tuple | $\langle a, q, \pi_q \rangle$ | $q \in \pi_q$, $\exists p \in Q$ and $(p, a, q) \in \delta$ |
| Splitter tuple | $\langle B_\mathrm{i}, a \rangle$ | |
| Update tuple | $\langle p, \pi_p, \pi_p^{\mathsf{new}} \rangle$ | $p$ belongs to $\pi_p$ and has to move to $\pi_p^{\mathsf{new}}$ |

**Algorithm 6** Hopcroft's algorithm in Map-Reduce (Hopcroft-MR)

**Input:** $A = (Q, \Sigma, \delta, s, F)$
**Output:** $\{(p, a, q, \pi_p) : (p, a, q) \in \delta \text{ and } \pi_p = p/_{\equiv}\}$

```
1: PREPROC(A)
2: size ← 2
3: for all a ∈ Σ do
4:     if |blocks[a, π0]| ≤ |blocks[a, π1]| then
5:         ENQUEUE(QUE, ⟨π0, a⟩)
6:     else
7:         ENQUEUE(QUE, ⟨π1, a⟩)
8:     end if
9: end for
10: while QUE ≠ ∅ do
11:     ⟨Bi, b⟩ ←DEQUEUE(QUE)
12:     PARTITIONDETECT(Δ, blocks[b, Bi], size)
13:     ——————
14:     Build set New
15:     BLOCKUPDATE(Δ, blocks, New)
16:     PARALLELPERFECTHASHINGFUNCTION(Δ, blocks)
17:     for all Group Bj divided to Bk and Bj and ∀a ∈ Σ do
18:         if ⟨Bj, a⟩ ∈ L then
19:             ENQUEUE(⟨Bk, a⟩)
20:         else
21:             ENQUEUE(QUE,min(⟨Bk, a⟩, ⟨Bj, a⟩))
22:         end if
23:     end for
24: end while
```

**Pre-processing and data structures:** Let $A = (Q, \Sigma, \delta, s, F)$. For each transition $(p, a, q) \in \delta$ we will in a set $\Delta$ have a

- *transition tuple* $\langle p, a, q, \pi_p, \pi_q \rangle$,

where $\pi_p$ is the current block to which state $p$ belongs, and $\pi_q$ is the current block of state of $q$. We use positive integers to name the blocks, so $\pi$ is to be construed as a

mapping from $Q$ to the positive integers. Initially, $\pi_p = \mathtt{1}$ for $p \in F$, and $\pi_p = \mathtt{0}$ for $p \in Q \setminus F$. The set $\Delta$ will initially be chunked over DFS. The file system contains an array $\mathsf{blocks}[a, \mathtt{i}]$, where an entry contains, for current block $B_{\mathtt{i}}$, and alphabet $a$,

- *block tuples* $\langle a, q, \pi_q \rangle$,

where $\pi_q = \mathtt{i}$. A block tuple indicates that state $q$ currently belongs to block $B_{\mathtt{i}}$, and that there is a transition labelled $a$ incident upon state $q$. The file system also includes

- a global variable $\mathsf{size}$,

initially having the value 2 assigned in line 2 of master program, indicating the number of blocks in the current partition. The value of $\mathsf{size}$ can be queried by extended model of Map-Reduce described in Section 3.1.1. Finally, the queue of splitters is maintained by the master node, and it is denoted $\mathsf{QUE}$. As blocks are represented by positive integers, we denote

- *splitter tuples* with $\langle B_{\mathtt{i}}, a \rangle$,

where $B_{\mathtt{i}}$ is the $i$:th block.

The pre-processing job responsible for initializing *block tuples* and $\mathsf{blocks}$ array, is presented in Algorithm 7. Each mapper handles a chunk of $\Delta$, and for each tuple $\langle p, a, q, \pi_p, \pi_q \rangle$ in its chunk, it emits $\langle t.q, t \rangle$. Within the reducer, $\delta^{-1}(b)$ for every

alphabet symbol is created consisting all the states having transition labelled $b$ to $q$. A block tuple $\langle a, q, \pi_q \rangle$ will be added to the array entry $\mathsf{blocks}[a, \pi_q]$ if $\delta^{-1}(b)$ is not empty. Using $\mathsf{blocks}$ array generated in Pre-Processing job, master program (lines 3 to 9) initializes $\mathsf{QUE}$.

After pre-processing and initialization process, the master program iterates over lines 11 to 23 by picking a splitter and refining the partition. After updating blocks, in line 16, PPHF function described in Section 3.1.1 prevents growing size of block numbers. The list of splitters ($\mathsf{QUE}$) is updated by following Formula 4 in lines 17 to 23. Note that split is done by moving part of states from one block to a new block. So, $B_{\mathsf{j}}$ will be divided into $B_{\mathsf{j}}$ and $B_{\mathsf{k}}$. The algorithm terminates whenever $\mathsf{QUE}$ is empty.

**Algorithm 7** Pre-Processing Job for Hopcroft-MR

**Input:** $\Delta$
**Output:** Set of *block tuples* constructing $\mathsf{blocks}$ array

```
1: function MAPPER(Δ)
2:     for all ∀t ∈ Δ do
3:         Emit ⟨t.q, t⟩
4:     end for
5: end function

6: function REDUCER(⟨K, [t_1, t_2, ..., t_m]⟩)
7:     for all b ∈ Σ do
8:         δ^{-1}(b) ← {t.p | t.a = b}
9:         if δ^{-1}(b) ≠ ∅ then
10:            Emit ⟨a, B_q, q⟩
11:        end if
12:    end for
13: end function
```

**Function** PARTITIONDETECT($\Delta, \mathsf{blocks}[b, B_\mathsf{i}], \mathsf{size}$) consists of one round of map and reduce demonstrated in Algorithm 8 as follows:

**Map function:** The mapper reads a chunk of the current set $\Delta$, either the initial one or a chunk of new tuples from the previous round. Let $h$ be a mapping from the states to reducers, as before. For each transition tuple $t$ the mapper emits key-value pair $\langle h(t.q), t\rangle$ along with $\mathsf{size}$ in line 3. The mapper also gets a chunk of $\mathsf{blocks}[b, B_\mathsf{i}]$ where splitter $\langle B_\mathsf{i}, b\rangle$ is picked in line 11 of master program and sends *block tuple* $\langle b, q, \pi_q\rangle$ where $\pi_q = B_\mathsf{i}$, to reducer $h(q)$ in line 6.

**Reduce function:** Reducer $h(q)$ receives variable $\mathsf{size}$, and transition tuples $\langle p_1, b, q, \pi_{p_1}, \pi_q\rangle, \ldots, \langle p_m, b, q, \pi_{p_m}, \pi_q\rangle$, where $p_1, \ldots, p_m$ are the states from which there is a $b$-transition to state $q$. Additionally, if $\mathsf{blocks}[b, \pi_q]$ contains a block tuple $\langle b, \pi_q, q\rangle$, this tuple is also sent to reducer $h(q)$.

If reducer $h(q)$ does not receive a block tuple, it does nothing. If it does receive block tuple $\langle b, \pi_q, q\rangle$ it means that all states $p_1, \ldots, p_m$ need to be moved to a new block. In this case the reducer $h(q)$ calculates $\pi^{\mathtt{new}}_{p_j} = \pi_{p_j} + \mathsf{size}$ at line 12, and outputs update tuples $\langle p_j, \pi_{p_j}, \pi^{\mathtt{new}}_{p_j}\rangle$ at line 13, for $j = 1, \ldots, m$. The reducers will in phase 2 output

- *update tuples* $\langle p, \pi_p, \pi^{\mathtt{new}}_p\rangle$

with the meaning that state $p$ belonged to block $\pi_p$, and should now be moved to block $\pi_p^{\texttt{new}}$.

Before updating the blocks of the states, the master program checks for each block $B_{\texttt{i}}$ whether the reducers in the previous phase outputted update tuples $\langle p, \pi_p, \pi_p^{\texttt{new}} \rangle$ for *all* or only for *some* states $p$, where $\pi_p = B_{\texttt{i}}$. Only in the latter case are the update tuples inserted in set NEW.

**Algorithm 8** Partition Detection Job for Hopcroft-MR

**Input:** $\Delta$, blocks$[b, B_{\texttt{i}}]$, size
**Output:** *Update Tuples*

```
1: function MAPPER(Δ, blocks[b, B_i], size)
2:     for all ∀t ∈ Δ do
3:         Emit ⟨q, [t, size]⟩
4:     end for
5:     for all block tuple ⟨b, π_q, q⟩ in blocks[b, B_i] do
6:         Emit ⟨q, ⟨b, π_q, q⟩⟩
7:     end for
8: end function

9: function REDUCER(⟨K, [t_1, t_2, ..., t_m, size, block_tuple]⟩)
10:    if block_tuple ≠ ∅ then
11:        for all t_i do
12:            π_p^new ← t_i.π_p + size
13:            Emit ⟨t_i.p, t_i.π_p, π_p^new⟩
14:        end for
15:    end if
16: end function
```

**Function** BLOCKUPDATE($\Delta$, blocks, New). Here we do one round of map and reduce illustrated in Algorithm 9 as follows:

**Map function.** The mappers are assigned a chunk of New, a chunk of $\Delta$, and a chunk of blocks. For each update tuple $\langle p, \pi_p, \pi_p^{\texttt{new}} \rangle$ it emits key-value pair $\langle h(p), \langle p, \pi_p, \pi_p^{\texttt{new}} \rangle \rangle$. It also emits $\langle h(p), \langle p, a, q, \pi_p, \pi_q \rangle \rangle$ for each transition tuple in its chunk of $\Delta$. The mapper as well emits $\langle h(p), \langle a, \pi_p, p \rangle \rangle$ for every block tuple from $\mathsf{blocks}[a, \pi_p]$.

**Reduce function.** Reducer $h(p)$, if it receives tuple $\langle p, \pi_p, \pi_p^{\texttt{new}} \rangle$, updates $\pi_p$ in all of its transitions. The reducer also updates records from `blocks` for the next iteration of DFA minimization process.

**Algorithm 9** Block Update for Hopcroft-MR

**Input:** $\Delta$, blocks, New
**Output:**

```
1: function MAPPER(Δ, blocks, New)
2:     ∀t ∈ Δ Emit ⟨t.p, t⟩
3:     ∀⟨a, π_p, p⟩ ∈ blocks Emit ⟨p, ⟨a, π_p, p⟩⟩
4:     ∀⟨p, π_p, π_p^new⟩ ∈ New Emit ⟨p, ⟨p, π_p, π_p^new⟩⟩
5: end function

6: function REDUCER(⟨K, [t_1, t_2, ..., t_m, size, block_tuples, update_tuple]⟩)
7:     if update_tuple ≠ null then
8:         Update π_p in transitions and block tuples
9:     end if
10: end function
```

### 3.2.1 Correctness

In this section, we prove the correctness of the Hopcroft-MR.

**Theorem 1 (Termination)** *The algorithm terminates when the queue is empty.*

*Proof.* Initially, a splitter is enqueued either in line 5 or 7 of Algorithm 6 for every alphabet. During the main loop of master program, first we dequeue a splitter. A new splitter will be added at line 19 or 21 if there is a split. Considering the fact that the number of blocks in any partition of DFA can be no bigger than the number of states, thus the algorithm will terminate eventually □

**Theorem 2** *When algorithm terminates, $p \not\equiv q$ if and only if $p \in \pi_p$ and $q \in \pi_q$ and $\pi_p \neq \pi_q$.*

*Proof.* Suppose $p, q \in B_{\mathsf{i}}$ and $p \not\equiv q$ and there exists $a \in \Sigma$ where $p$ and $q$ are distinguishable because of $\delta(p, a)$ and $\delta(q, a)$. As $p$ and $q$ are distinguishable states, they are in one of two following situations:

1.

$$\delta(p, a) \in B_{\mathsf{j}}, \delta(q, a) \in B_{\mathsf{k}} \text{ and } \mathsf{j} \neq \mathsf{k} \tag{20}$$

2.

$$\delta(p, a), \delta(q, a) \in B_{\mathsf{l}} \tag{21}$$

First, following Equation 21, there exists $w \in \Sigma^*$ where $\hat{\delta}(p, w) \in B_{\mathsf{j}'}$ and $\hat{\delta}(q, w) \in B_{\mathsf{k}'}$ where $B_{\mathsf{j}'} \neq B_{\mathsf{k}'}$. Here, $w$ is the shortest word with this property. In this case there is a word $w'$ where $w = w'a$ and $\hat{\delta}(p, w'), \hat{\delta}(q, w') \in B_{\mathsf{l}'}$ and $\delta(\hat{\delta}(p, w'), a) \not\equiv \delta(\hat{\delta}(q, w'), a)$. Replacing $\hat{\delta}(p, w')$ by $p$ and $\delta(t, w')$ by $q$ gets back to Equation 20.

Consider the point when $\delta(p, a)$ and $\delta(q, a)$ were partitioned. At this point, without loss of generality, in reducer $h(\hat{\delta}(p, aa'))$ where $a'$ can be equal to $a$ itself, we do have the transition $\langle \delta(p, a), a', \hat{\delta}(p, aa') \rangle$, as well as $\langle a', \pi_{\hat{\delta}(q,aa')}, \hat{\delta}(p, aa') \rangle$. In this case reducer will decide to emit the partitioning flag and puts $\delta(p, a)$ in new generated block. After the second round, the master program decides to enqueue at least one of the new blocks. Through one of the further rounds, this block will be dequeued and the block containing $p$ and $q$ is partitioned and $p$ and $q$ can not be in the same block. This is a contradiction for $p, q \in B_{\mathsf{i}}$.

Evaluating the other direction of the theorem, the block label of states (transitions) is only updated in the refinement job and in the presence of an update tuple which says state $p$ is moved to new block. Based on the fact that in the DFA there is a transition per alphabet, this record can be generated just in one reducer $h(q)$ where $\delta(p, a) = q$. Reducer $h(q)$ will emit the update tuple if and only if it received block tuple stating that $q$ belongs to a block $B_{\mathsf{i}}$ where $\langle B_{\mathsf{i}}, a \rangle$ is splitter pair for this

iteration. So,

$$p \in B_1 \wedge q \in B_2 \wedge B_1 \neq B_2 \rightarrow \exists a \in \Sigma,\ (\delta(p,a) \in B_1' \wedge \delta(q,a) \in B_2') \rightarrow p \not\equiv q$$

□

Updating records should keep consistency and integrity of data structure and DFA records. We proved that updating group numbers respect equivalency class of states. But to make sure that algorithm is and remains correct, it is necessary to know that updates do not violate integrity of auxiliary data structure. Specially *block tuples* which has a critical role in partition detection job.

**Theorem 3** *Suppose $p \equiv_r q$ and "block tuples" for $p$ and $q$ are in* $\mathsf{blocks}[a, B_\mathsf{i}]$. *If $B_\mathsf{i}$ splits in round $r+1$ to $B_\mathsf{i}$ and $B_\mathsf{j}$, $p \in B_\mathsf{i}$ and $q \in B_\mathsf{j}$, then $\langle a, B_\mathsf{i}, p\rangle \in \mathsf{blocks}[a, B_\mathsf{i}]$ and $\langle a, B_\mathsf{j}, q\rangle \in \mathsf{blocks}[a, B_\mathsf{j}]$.*

*proof.* Recall Equation 2 and the format of *block tuple* records,

$$(p \in B_\mathsf{i} \wedge (\exists p' \in Q\ : \delta(p', a) = p)) \rightarrow \langle a, B_\mathsf{i}, p\rangle \in \mathsf{blocks}[a, B_\mathsf{i}]$$

This holds initially when the records are generated in pre-processing and the *block tuple* record for state $p$ is emitted if $\delta^{-1}(p, a) \neq \emptyset$. Suppose after round $r$, there are records $\langle a, B_\mathsf{i}, p\rangle$ and $\langle a, B_\mathsf{i}, p\rangle$ in $\mathsf{blocks}[a, B_\mathsf{i}]$. Now, if in round $r+1$, the detection

job decided to refine $B_i$, the *update tuple* $\langle q, B_i, B_j \rangle$ is generated and by mapper of refinement job will be sent to reducer $h(q)$. Then, the *block tuple* record for $q$ is updated to $\langle a, B_j, q \rangle$. However, $\langle a, B_i, p \rangle$ remains untouched. This means $p \in a(B_i)$ and $q \in a(B_j)$ $\square$.

## 3.3 Enhanced Hopcroft-MR (Hopcroft-MR-PAR)

In previous section we employed a list of splitter as introduced in Hopcroft's original algorithm. However, at each iteration we picked one splitter and refined the partition if there was any split. Then we added more splitters. However, it is possible to process all the splitters in QUE in parallel. In this section, by applying some changes in the master program of Hopcroft-MR (Algorithm 6), the detection job (Algorithm 8), and its refinement job (Algorithm 9), an enhanced version called Hopcroft-MR-PAR will be proposed.

### 3.3.1 Splitters and QUE

Recall in line 11 of Algorithm 6, one splitter is picked from the queue at each iteration. In the enhanced version, we remove this line and put the whole QUE as the input of PartitionDetect job. Furthermore, in line 18, if block $B_i$ is divided into $B_i$ and $B_j$, for every alphabet symbol $a$, we first check if there is already a splitter $\langle B_i, a \rangle$

in QUE or not. But in Hopcroft-MR-PAR, the whole queue is deleted and there is no need to check it. On the other hand, as there are multiple splitters processing in detection job, it is highly possible that one block divide into more than two blocks. To make Hopcroft-MR-PAR consistent with Hopcroft-MR, let us first examine an example. Consider block 1 in Figure 11 with four states $(p_1, p_2, p_3, p_4)$ inside as part of a DFA with singleton set of alphabet $\Sigma = \{a\}$. In addition, each state has one and only one incoming transition labelled $a$. Before processing round $r$, there are 3 splitters in queue,

$$\mathsf{QUE} = \{\langle 2, a\rangle, \langle 3, a\rangle, \langle 4, a\rangle\}$$

Lets take $\langle 2, a\rangle$ out. The block 1 will split into $\{p_1, p_2\}$ and $\{p_3, p_4\}$. Then let us put new splitters to another queue $\mathsf{QUE}'$. As splitter $\langle 1, a\rangle$ is not in QUE and size of both newly generated blocks are the same, we put just $\langle 1, a\rangle$ into $\mathsf{QUE}'$. Then we pick $\langle 3, a\rangle$ out. Again, block 1 will split into $\{p_3\}$ and $\{p_4\}$. Now we already have $\langle 1, a\rangle$ splitter. So it is required to put $\langle 6, a\rangle$ into $\mathsf{QUE}'$. By picking $\langle 4, a\rangle$ nothing will change as block 1 only has one member. We need to make sure that by processing all the splitters in QUE and getting newly generated blocks, QUE for next iteration is same as QUE' in this example or at least exhibits the same functionality. Our focus is on states inside original block 1. After split it into $\{1, 5, 6\}$, it is possible to add a splitter for every blocks except the biggest one. Consequently, $\mathsf{QUE} = \mathsf{QUE}' = \{\langle 6, a\rangle, \langle 1, a\rangle\}$.

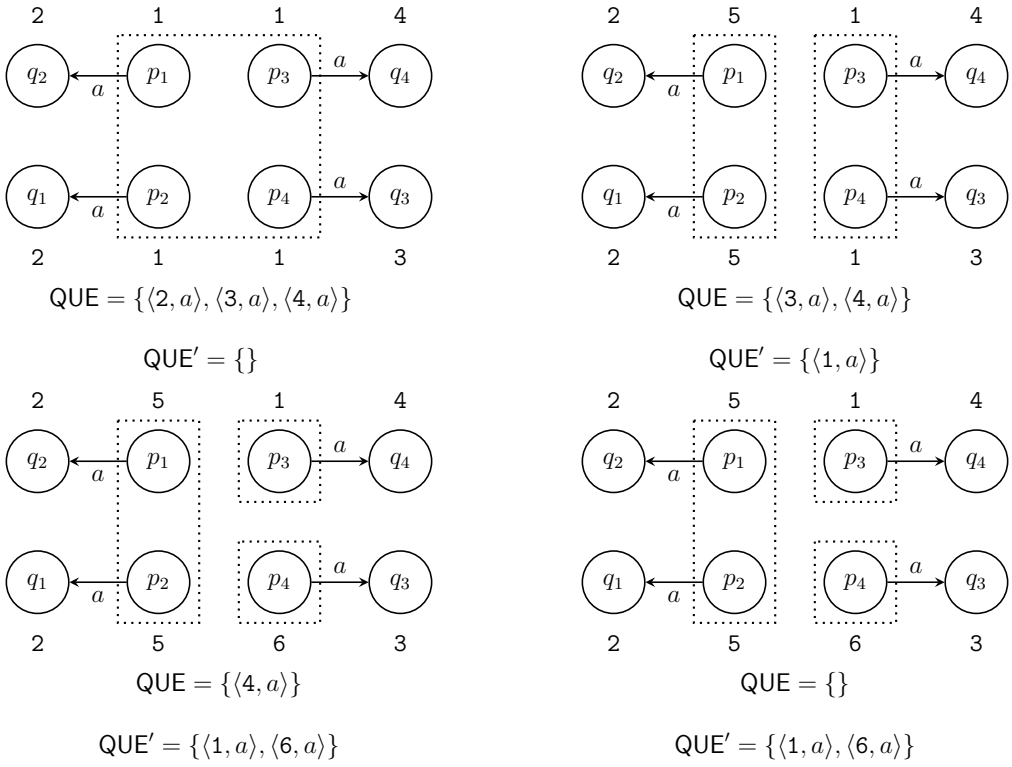


**Figure 11: Processing multiple splitters from QUE**

### 3.3.2 Detection

The mapper function of detection job remains the same as before. Inside the reducers, there could be none or multiple *block tuples* based on the splitters picked from QUE. We need to change the way of generating the new block number.

Recall from line 12 of Algorithm 8 where $\pi_p^{\mathsf{new}} = \pi_p + \mathsf{size}$ and the fact that the splitters can have different block-alphabet pairs. To reflect these two facts in new block numbers and distinguish states within a block by splitters, a bit array denoted as A all elements initialized to 0 is employed. Now suppose a set of splitters $S = \{\langle \pi_q, a_1 \rangle, \ldots, \langle \pi_q, a_m \rangle\}$ are present in reducer $h(q)$ where $0 \leq m \leq k$ and state $p$ has a set of transitions to state $q$. The value of A for state $p$ is defined as follows:

$$\mathsf{A}_p[i] = 1 \leftrightarrow (\langle \pi_q, a_i \rangle \in S \wedge (p, a_i, q) \in \delta)$$

Now line 12 of Algorithm 8 can be replaced by,

$$\pi_p^{\mathsf{new}} = \pi_p + \mathsf{size} \cdot \mathsf{A}$$

Note that, two states $p, p' \in B_{\mathtt{k}}$ can be marked in different reducers and splitters $\langle B_{\mathtt{i}}, a_i \rangle$ and $\langle B_{\mathtt{j}}, a_i \rangle$. In order to differentiate these two changes, we attach $B_{\mathtt{i}}$ and $B_{\mathtt{j}}$ to the new block numbers.

### 3.3.3 Update Blocks

The block update job in Algorithm 9 is designed to apply changes outlined in detection job. By using an enhanced version of the detection job, it is possible to have different new block numbers present in reducer $h(p)$. The new block number can be calculated by applying logical AND on A bit arrays and concatenating $B_i$s attached to the update records. Although it may generate large bit strings, applying PPHF-MR will map them to range of $[0, n^2]$.

## 3.4 Cost Measures for DFA Minimization in Map-Reduce

Taking into account the amount of data that needs to be processed in DFA minimization algorithms in Big-Data context, it is crucial to analyse the cost model precisely. Although computation analysis is very important, the amount of data replicated to reducers is the most important part of the complexity analysis in Map-Reduce algorithms [34]. On top of that, the number of rounds for iterated Map-Reduce algorithms is an effective factor on the cost [22]. In this chapter, we try to find the minimum communication for finding minimal DFA using equivalence class. For this matter, the communication complexity model described in Section 2.4.1 is utilized. Additionally we will investigate minimum replication rate for this problem, using the lower-bound

recipe from Section 2.4.2. At the end, by extending complexity model from 2.4.3, the cost for all algorithms are examined. It is worth to mention that the analysis is done in this thesis is about algorithms which are under *Top-Down* sub-tree in Figure 1.

### 3.4.1 Minimum Communication Cost

In this section we will find maximum communication cost of minimizing a DFA in parallel environment as well as the lower bound for communication complexity using two parties model.

Recall Yao's two-party model from Section 2.4.1. We need to define a simple function of equivalency (EQ). Having two inputs $x, y \in \{0,1\}^m$, $z = EQ(x,y)$ is 1 if $x = y$ and 0 otherwise. The maximum communication needed to evaluate this function is sending all the $m$ bits of $x$ by Alice to Bob and Bob returns one bit result of evaluation back to Alice. Thus $\mathcal{D} \leq m + 1$. Moreover, the *fooling set* for EQ is:

$$S = \{(x,x) | x \in \{0,1\}^m\}$$

Counting $z = 0$, we have $|S| = m + 1$. Hence $\mathcal{D}(EQ(x,y)) \geq \log_2(m+1) \geq \log_2 m$.

**Lemma 6** *Maximum communication cost for* $p \equiv_i q$ *is* $k \log n$

*Proof.* Recall definition of equivalency,

$$p \equiv_i q \leftrightarrow (\pi_p^{i-1} = \pi_q^{i-1} \wedge \forall a \in \Sigma\ (\pi_{\delta(p,a)}^{i-1} = \pi_{\delta(q,a)}^{i-1}))$$

Each equivalence class can be represented by maximum $m = \log n$ bits. Thus we can define the following functions:

1. $f_0 : \{0,1\}^m \times \{0,1\}^m \to \{0,1\}$ for $\pi_p^{i-1} = \pi_q^{i-1}$
2. $\forall a \in \Sigma,\ f_a : \{0,1\}^m \times \{0,1\}^m \to \{0,1\}$ for $\pi_{\delta(p,a)}^{i-1} = \pi_{\delta(q,a)}^{i-1}$

Each above mentioned function is equivalent to $EQ$. Hence there are $k+1$ functions. So, using Equation 10 and 13 :

$$\mathcal{D}(p \equiv_i q) = (k+1)(n'+1) = (k+1)(\log n + 1) = O(k \log n) \tag{22}$$

□

**Theorem 4** *The upper bound for DFA minimization in Map-Reduce model is* $O(kn^3 \log n)$.

*Proof.* DFA minimization has $n$ rounds in worst case as it is discussed in Section 2.1.2. At every iteration, again in worst case, it is required to check equivalency of all pairs which is $\frac{n(n-1)}{2} = O(n^2)$. Using Lemma 6, each pair requires at most $O(k \log n)$

bits to evaluate the function. So upper bound for DFA minimization DCC would be $O(kn^3 \log n)$.□

From now on we can talk about lower bound on DCC for above mentioned problem $p \equiv_i q$.

**Theorem 5** *The lower bound on DCC for DFA minimization in Map-Reduce model is* $O(kn \log n)$.

*Proof.* Lemma 3, shows that during splitting process, $O(kn \log n)$ states will move from one block to new one. Now suppose *Alice* has $p$ and *Bob* has $q$. *Bob* will inform *Alice* to move $p$ from its current block to a new one. To carry out this task, *Bob* has to send a bit to *Alice*. So, minimum number of bits required to be communicated is $O(kn \log n)$ □

### 3.4.2 Lower Bound on Replication Rate

Replication Rate has a direct effect on communication cost for Map-Reduce algorithms. Using lower-bound recipe introduced in Section 2.4.2, we will find minimum required replication rate. Recall Equation 15,

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

The input data for DFA minimization algorithm ($I$) is the transitions of original DFA. Hence $|I| = kn|t|$ where $|t|$ is size of transition in bits. On the other hand, the output ($O$) is updated transitions from input exposing minimal DFA and $|O| = kn|t|$. Clearly the size of transitions are different between input and output. This is due to at the beginning we have two blocks can be represented by one bit. However, for outputs the number of bits needed to store block numbers is $\log n^2 = 2 \log n$ by applying PPHF algorithm (Section 3.1.1). We can assume transition size in input is equal to output. Moreover, if a reducer gets $\rho$ amount of transitions, it can compute and update at most only those transitions not anymore. So, maximum output from a reducer is $g(\rho) = \rho$. Now having these facts, we can compute lower bound in replication for DFA minimization as follow:

$$\mathcal{R} \geq \frac{\rho \times |O|}{g(\rho) \times |I|} = \frac{\rho \times kn\tau}{\rho \times kn\tau} = 1 \tag{23}$$

### 3.4.3 Communication Complexity of Moore-MR-PPHF

In this section we examine replication rate and communication cost of Moore-MR-PPHF.

#### 3.4.3.1 Replication Rate

Recall Algorithm 5 for computing minimal DFA in Map-Reduce using Moore's method, the input data are not only transitions from original DFA, but also dummy ones. We will emit each transition once at and each dummy transition twice. This makes the replication rate,

$$\mathcal{R}_{\text{Moore-MR}} = \frac{3}{2}$$

This replication rate is almost equal to the lower bound.

#### 3.4.3.2 Communication Cost

Communication cost as discussed before is amount of data transmitted during Map-Reduce computations. Input records (transitions) for each round is in form of $t = \langle p, a, q, \pi_p, D \rangle$ which makes the size of transition $|t| = O(\log n + \log k)$. Note that for first job it is $2\log n + \log k + 2$ due to the size of $\pi$ equal to two and can be represented by 1 bit. In addition, note that using PPHF, block labels are in the range of $[1, n^2]$ and we need $2\log n$ bits for that.

The communication cost is equal to the number of rounds times communications in each round, which is replication rate times input plus output. Considering the maximum number of rounds is $R = n$, we can say

$$CC = O(R(\mathcal{R}|I| + |O|)) = O(n(\mathcal{R}|I| + |O|)) \tag{24}$$

The output of each round is:

- One record per transition
- One record per dummy transition
- One record for convergence voting

Third item is about if one state detects any of other $k$ states it has a transition to, has moved to a new block. Lemma 3 shows that there could be at most $n \log n$ state moving to new block during DFA minimization procedure each of which can force at most $k$ states to issue the voting record in size of 1. This will be applied to $CC$ separately. As a conclusion,

$$|O| = n(k|t| + k|t|) = O(kn(\log n + \log k))$$

Now the communication cost of Moor's algorithm in Map-Reduce can be computed as follows,

$$\begin{aligned} CC &= O(n(\mathcal{R}|I| + |O|)) + O(kn \log n) \\ &= O(n(\frac{3}{2}kn(\log n + \log k) + kn(\log n + \log k)) + kn \log n) \\ &= O(n(kn(\log n + \log k))) \\ &= O(kn^2(\log n + \log k)) \end{aligned} \tag{25}$$

Amount of required communication in PPHF is sending all transitions in mapper and sending all updated transitions back in reducer and it is $2kn|t|$ which is much less than $CC$ from Equation 25.

### 3.4.4 Communication Complexity of Hopcroft-MR

Calculating the cost for Hopcroft-MR algorithm due to auxiliary generated information is a little bit complicated than Moore-MR-PPHF. First of all, from Algorithm 6, we know that there are 4 different jobs, one of which is Pre-Processing. This job will not be included in computation because of (1) replication rate is 1 (2) during asymptotic analysis, in presence of iterative procedure, the amount of communication for pre-processing job is much less than the iterative part.

#### 3.4.4.1 Replication Rate

There are 3 different jobs in Hopcroft-MR except PPHF-MR. We saw in previous section that its replication rate is one.

1. Pre-Processing: The mapper emits every transition record one in line 1 of Algorithm 7 once.

2. Detection: The mapper emits transitions in line 3 once and *block tuple* once in line 6 og Algorithm 8.

3. Block Update: The mapper emits every record once in Algorithm 9.

Above analysis shows that the replication rate for Hopcroft-MR is $\mathcal{R} = 1$. It is exactly equal to minimum replicate rate we found in section 3.4.2.

#### 3.4.4.2 Communication Cost

The main communication cost is related to three jobs in the loop. We calculate each of them separately:

- **Partition Detection.** Using the splitter $\langle P, a \rangle$, every transition in $P$ is emitting once at round $r$. Besides, mapper sends every *block tuple* record in $\mathsf{blocks}[P, a]$. According to Lemma 2 the total number of states in all splitters we may put in queue is bounded by $n \log n$. So these records can not be more than $C = O(n \log n)$ each of which requires $2 \log n + \log k$ bits. On the other hand, reducers would send a record for every state which has to be moved to new block. According to Lemma 3, it is $O(n \log n)$. Every *update tuple* is in form of $\langle p, \pi_p, \pi_p^{\mathsf{new}} \rangle$ and its size is $\log n + 2 \cdot 2 \log n = 5 \log n$. Thus the total communication cost for detection job is:

$$CC_1 =$$

$$O(n \log n \cdot (|t| + \log n) + n \log n \cdot (\log k + \log n) + n \log n \cdot \log n) =$$

$$O(n \log n (\log n + \log k))$$

- **Update Blocks** For the update job, we need to send all the transitions, *block tuples* as well as *update tuples* to the reducers. Number of *update tuples* is $O(n \log n)$ in whole DFA minimization process. Moreover, the number of *block tuples* is at most $n$. According to Lemma 1 the number of iterations in Hopcroft's method is equal to $R = 2kn$. Moreover, we found in Section 3.4.4.1 that replication rate for every job in Hopcroft-MR is equal to $\mathcal{R} = 1$. Now, using Equation 24, the amount of communication in mapper side is equal to $CC^{\mathsf{mapper}} = O(R\mathcal{R}|I|)$. Thus we have,

$$CC_2^{\mathsf{mapper}} = O(R\mathcal{R}|I|)$$

$$O(kn \cdot (kn \cdot (\log k + \log n) + n \cdot (\log k + \log n))) + O(n \log n \cdot \log n) =$$

$$O((kn)^2(\log n + \log k)) + O(n \log n^2) =$$

$$O((kn)^2(\log n + \log k))$$

In reducer side, it will write in output every transitions and *update tuple*s, either it is updated or not. So, the total communication cost for the reducer task in update job is $CC^{\mathsf{reducer}} = O(R|O|)$

$$CC_2^{\mathsf{reducer}} =$$

$$O(kn \cdot (kn \cdot (\log k + \log n) + n \cdot (\log k + \log n))) =$$

$$O((kn)^2(\log n + \log k))$$

Thus,

$$CC_2 = CC_2^{\mathsf{mapper}} + CC_2^{\mathsf{reducer}} =$$

$$O((kn)^2(\log n + \log k) + (kn)^2(\log n + \log k)) =$$

$$O((kn)^2(\log n + \log k))$$

- **PPHF** As it has been computed in previous section,

$$CC_3 = O(kn^2(\log n + \log k))$$

Hence we can conclude communication cost for minimizing DFA using Hopcroft-MR is:

$$CC = CC_1 + CC_2 + CC_3 =$$

$$O(n \log n(\log n + \log k) + (kn)^2(\log n + \log k) + kn^2(\log n + \log k)) = \quad (26)$$

$$O((kn)^2(\log n + \log k))$$

### 3.4.5 Communication Complexity of Hopcroft-MR-PAR

As shown in Section 3.3, mapping schema of Hopcroft-MR-PAR is same as Hopcroft-MR, $\mathcal{R} = 1$. Moreover, in Hopcroft-MR-PAR we send the whole $\mathsf{QUE}$ in the detection job though the total number of *block tuples*, *update tuples*, and states within *splitters*

follow the Lemma 2 and 3. However, number of bits in *update tuples* is now equal to $\log n + 2\log n + (k + 2\log n + c2\log n)$. Recall the *update tuples* in form of $\langle p, \pi_p, \pi_p^{\mathsf{new}} \rangle$. We need $\log n$ bits to store $p$ and $2\log n$ bits to store $\pi_p$ (note that PPHF-MR maps block labels to range $[0 - n^2)$). The $\pi_p^{\mathsf{new}}$ contains $\pi_p$ along with $k$-bits $\mathsf{A}$ and block labels for those we have splitter. If $c$ bits of $\mathsf{A}$ is set to 1, then we need to attach $c$ block labels to new block number. As a result the communication cost for detection job is,

$$CC_1 = O(n \log n(\log n + k))$$

**Lemma 7** *Number of rounds for Hopcroft-MR-PAR is* $O(n)$

*Proof.* Using Lemma 1, the maximum number of iterations for Hopcroft's algorithm is $O(kn)$ where $k$ is the number of alphabet symbols and $n$ is the number of states. The worst case is slow automata where at each iteration block $R$ is split into $R$ and $R_1$ where w.l.g $|R_1| = 1$. In this case, we add $k$ splitters $\langle R_1, a_i \rangle$ into $\mathsf{QUE}$ where $i = 1, 2, ..., k$. In Hopcroft-MR-PAR, we process the whole $\mathsf{QUE}$. As a result, all the $k$ splitters are taken out and $\mathsf{QUE}$ is empty for the next round. So, the number of iteration of Hopcroft-MR-PAR is equal to $R = n$.

**Update Blocks.** Recall communication cost for mapper as $CC = O(R(\mathcal{R}|I| + |O|))$ from 3.4.4.2. As mentioned before $\mathcal{R} = 1$ and also input size and output size for Hopcroft-MR-PAR is equal to Hopcroft-MR's. However, by Lemma 7, the number of

rounds in worst case for Hopcroft-MR-PAR is equal to

$$R_{\text{Hopcroft-MR-PAR}} = O(n) = R_{\text{Hopcroft-MR}}/k$$

So,

$$CC_2 = O(R_{\text{Hopcroft-MR-PAR}}(\mathcal{R}|I+|O|)) = \frac{O((kn)^2(\log n + \log k))}{k} = O(kn^2(\log n+\log k))$$

The last job is PPHF-MR to shrink block labels. It has the same complexity as Hopcroft-MR and is equal to:

$$CC_3 = O(kn^2(\log n + \log k))$$

Using above mentioned factors, the communication cost for Hopcroft-MR-PAR is:

$$CC = CC_1 + CC_2 + CC_3 =$$

$$O(n \log n(\log n + k)) + O(kn^2(\log n + \log k)) + O(kn^2(\log n + \log k)) =$$

$$O(kn^2(\log n + \log k))$$

**Table 5: Comparison of complexity measures for algorithms**

| | Replication Rate | Communication Cost | Sensitive to Skewness |
|---|---|---|---|
| Lower Bound | 1 | $O(kn \log n)$ | – |
| Moore-MR 2.3.2 [18] | $\frac{3}{2}$ [18] | $O(nk^n)$ | No |
| Moore-MR-PPHF 3.1 | $\frac{3}{2}$ | $O(kn^2(\log n + \log k))$ | No |
| Hopcroft-MR 3.2 | 1 | $O((kn)^2(\log n + \log k))$ | Yes |
| Hopcroft-MR-PAR 3.2 | 1 | $O(kn^2(\log n + \log k))$ | Yes |

### 3.4.6 Comparison of Algorithms

In this section we will compare the algorithms as illustrated in Table 5. Three measures is employed here; *Replication Rate*, *Communication Cost*, and *Sensitivity to Skewness*. All the algorithm are analyzed using MRC[$f(n)$,$g(n)$] (2.4.3) and all are in MRC[$O(n)$,$O(n)$] with $c = 0$.

Sometimes, the mapping schema can not distribute the input data evenly among the reducers. This phenomenon can cause flooding some reducers while others are starving and causes a phenomenon called *the curse of last reducer* [30] where

> 99% of the computation finishes quickly, but the remaining
> 1% takes a disproportionately longer amount of time.

The other drawback of *Skewness* is that it makes it impossible to run an algorithm on some special input data. That is the reason why a new measure has been added to the comparison table to measure if an algorithm is sensitive to skewness or not.

In section 3.4.2 we found that the lower bound on *Replication Rate* is 1 which all the algorithms are almost equal to. On the other hand, as all the algorithms are under family of *Top-Down* minimization approach, they are iterative and number of required rounds is $O(n)$ except for Hopcroft-MR which is $O(kn)$.

The second column shows that the amount of communication in bits required for each algorithm bounded from bottom to $O(kn \log n)$. The basic algorithm from [18] (Moore-MR), requires $O(nk^n)$ bits for computing minimized DFA while by applying PPHF-MR it becomes $O(kn^2(\log n + \log k))$ (Moore-MR-PPHF). On the other hand, the Hopcroft-MR algorithm requires $O((kn)^2(\log n + \log k))$ bits of communication. By processing the splitters in QUE in Hopcroft-MR-PAR, we reach to the same communication complexity as Moore-MR. However, it would require less actual computations in reducers because not all the reducers are engaged in dividing partitions.

The last parameter to consider is *Sensitivity to Skewness*. In Moore-MR and Moore-MR-PPHF, the mapping schema is based on the source state of a transition. The number of the records (transitions) associated to each state is determined and equal to $k$, the size of the alphabet. Correspondingly, data has been distributed evenly among the reducers and there is no *Skewness*. However, in Hopcroft-MR and Hopcroft-MR-PAR, mapping schema for detection job is based on the reverse transitions and is not deterministic. This may cause sending amount of transitions

to a reducer that may not be tolerated in terms of space.

Recall MRC from Section 2.4.3 as a class of computational complexity. Although both time and space are included in the model along with number of iterations and replication rate, it may not reflect cover to very important factors affecting algorithms in Map-Reduce:

- *Sensitivity to Skewness*
- *Horizontal Growth of Data* [4]

We already discussed about the skewness. In some algorithms like Moore-MR, in order to accomplish the task, there is required to put additional data in input records. However, if this operation repeats over several iterations, the amount of data will go beyond of space capacity. As the main focus of Map-Reduce model is to address data intensive problems, these two measures are extremely critical to be reflected in complexity class of MRC.

# Chapter 4

# Experimental Results

During this chapter we will examine these two approaches by implementing Map-Reduce algorithms on Hadoop 2.7.1 on a cluster of two machines.

## 4.1 Cluster Configuration

These experiments were run on a 2-node cluster running Hadoop 2.7.1 with specifications demonstrated in Table 6.

**Table 6: Experimental Hadoop cluster configuration**

| | CPU | Memory | Operating System | Hadoop |
|---|---|---|---|---|
| Node 1 | Intel®Core™i5 760 2.80GHz | 8 GB | Linux kernel 3.10.0 | 2.7.1 |
| Node 2 | Intel®Core™i5 760 2.80GHz | 8 GB | Linux kernel 3.10.0 | 2.7.1 |

## 4.2 Data Generation Methods

Before exposing the results, we shall briefly discuss about sample data we ran algorithms on. States, alphabets and block labels are all represented by numbers.

For these experiments we have three different types of DFA. First a circular DFA introduced in original paper of Hopcroft. Having $Q = \{0, 1, 2, \ldots, n-1\}$ and $\Sigma = \{1, 2, \ldots, k\}$, for all transitions $(p, a, q) \in \delta$, we will first compute $q$ as follows:

$$q = \begin{cases} p + a, & \text{if } a \leq \lceil \frac{k}{2} \rceil \\ p - \lfloor \frac{a}{2} \rfloor, & \text{otherwise.} \end{cases}$$

Then we will refine it as follows.

$$q = \begin{cases} q, & \text{if } q \geq 0 \\ n + q. & \text{otherwise} \end{cases}$$

Start state is $s = 0$ and $p \in F$ if $k|p$.

Note that the state index starts from 0 while alphabet index starts from 1. The minimized DFA has $|\Sigma| = k$ states. A sample generated DFA with $|Q| = 8$ and $|\Sigma| = 4$, is illustrated in Figure 12 and its minimal DFA is Figure 13.

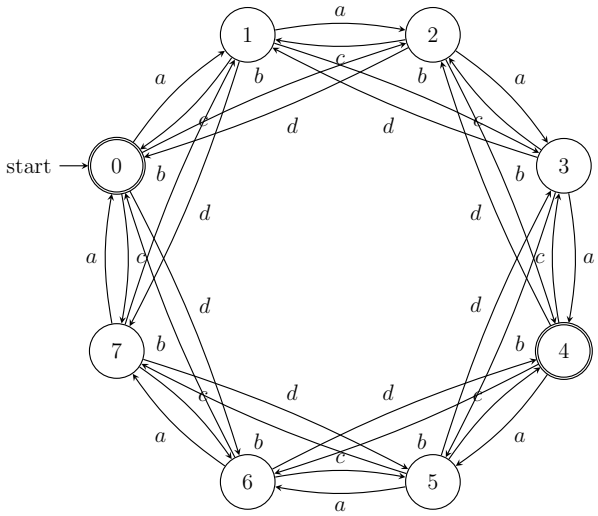


**Figure 12: A sample circular DFA**

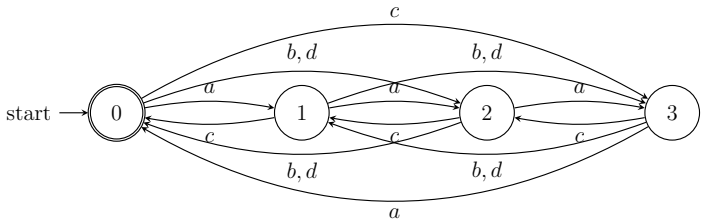


**Figure 13: Minimal DFA equivalent to the DFA in Figure 12**

The other method for generating a DFA is making a random DFA. However, the experimental results in [31] revealed that randomly generated DFA are covering universal language ($\Sigma^*$). Because of that, we replicated these DFA $k$ times each of which is reachable by one of the alphabet symbols from a start state. A sample DFA generated by this method having $|\Sigma| = 2$ from Figure 8 is illustrated in Figure 14. The minimal DFA, shown in Figure 15, is equal to the minimal of replicated-random DFA plus additional start state.

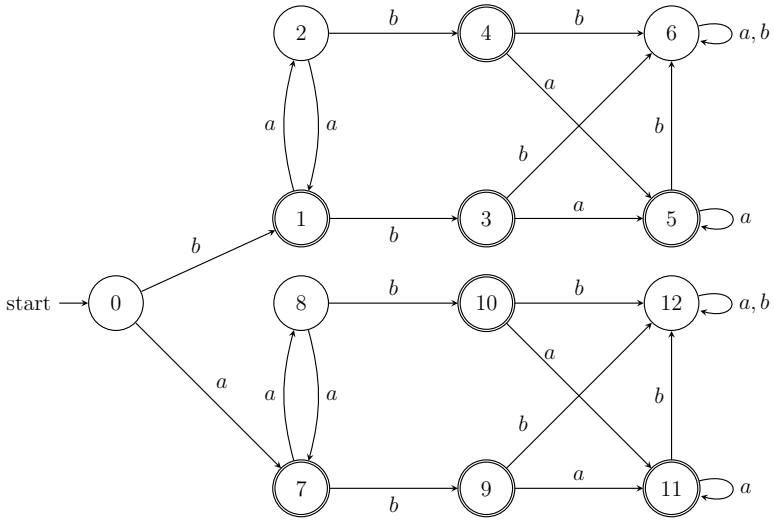


**Figure 14: A sample replicated-random DFA**

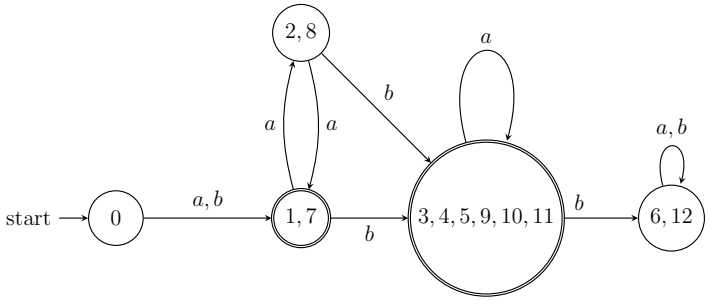


**Figure 15: Minimal DFA equivalent to the DFA in Figure 14**

The last data set is generated from the slow DFA introduced in Section 2.1.2.2. In this automata, $p_0$ is the start state, $F = \{p_n\}$ is the final state where $|Q| = n$, and for $0 \leq i < n$

$$\forall a \in \Sigma, \delta(p_i, a) = p_{i+1}$$

The transitions of the final state are all self-loops. This DFA is "already minimal".

## 4.3 Experiments

First we ran Moore-MR and Moore-MR-PPHF on slow automata with alphabet size 2 and doubling size of states starting with 2. Their communication cost is compared in Figure 16 which supports Table 5. It exhibits how the communication cost for Moore-MR without shrinking generated classes, is growing exponentially. In addition, from some point ($|Q| = 16$), the amount of data will go beyond of processing capacity of the cluster. This matter shows the important of investigating *horizontal growth* of data in a complexity model among others. However, by applying PPHF method, the communication cost has linear relation with number of states. Total execution time is plotted in Figure 17 and it also follows the same fact as communication cost.

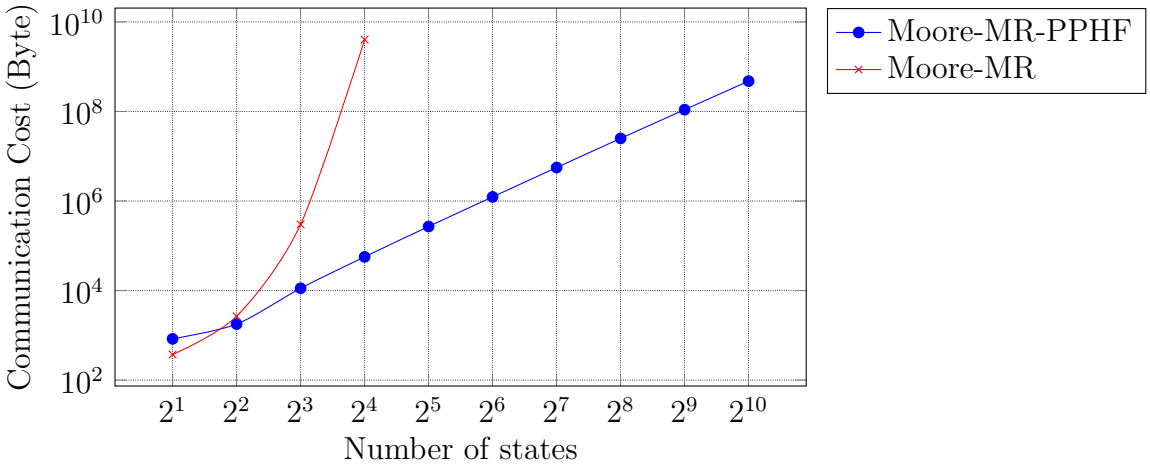


**Figure 16: Communication Cost of Moore's method with and without PPHF on slow DFA for the alphabet size $k = 2$**

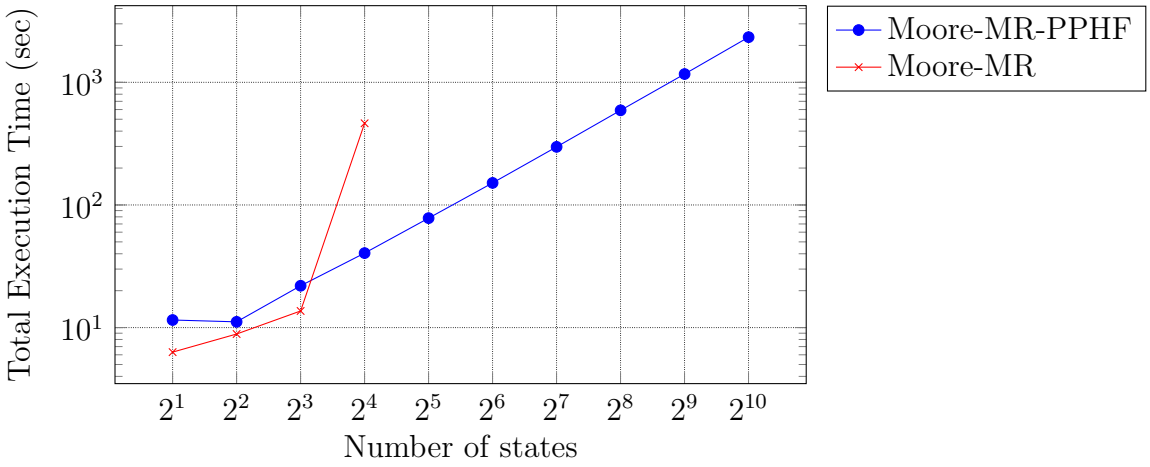


**Figure 17: Processing time of Moore's method with and without PPHF on slow DFA for the alphabet size $k = 2$**

The next experiment is about comparing behaviour of the algorithms on DFA with different topologies. First considering the slow automata, as it can be seen in Figure 18 and Figure 19, all the algorithms have the same communication cost and execution time. Figure 18 shows that communication cost and execution time in all three algorithms have polynomial relation with number of states as mentioned in Table 5. Hopcroft-MR will require more rounds to compute the minimal DFA as listed in Table 7 and consequently it takes more time to finish the process. However, the communication cost of Hopcroft-MR stays less than Hopcroft-MR-PAR and Moore-MR-PPHF.

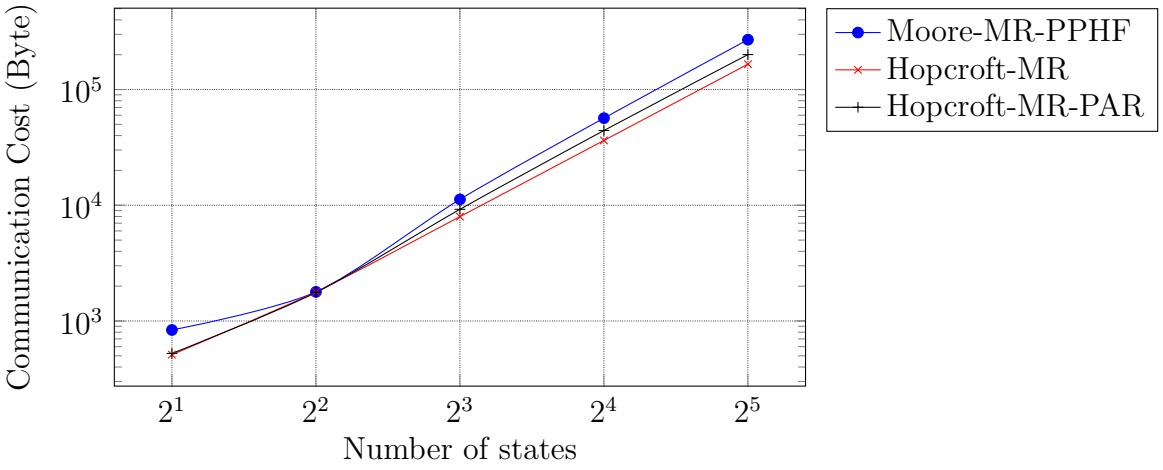


**Figure 18: Communication Cost of slow DFA for the alphabet size $k = 2$**

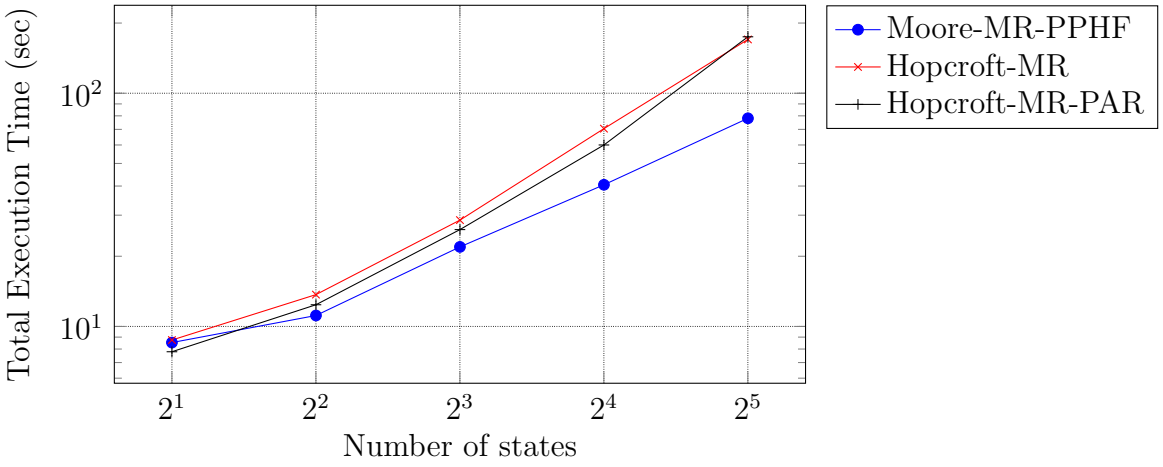


**Figure 19: Processing time of slow DFA for the alphabet size $k = 2$**

The second data set is *Circular* DFA. The main feature of this automata is that each state has $k$ incoming transitions. It makes uniform distribution of transitions in both mapping schema based on source state and target state. As a result, Figures

20 and 21 are exhibiting better performance of Hopcroft's algorithms in comparison with Moore-MR-PPHF.

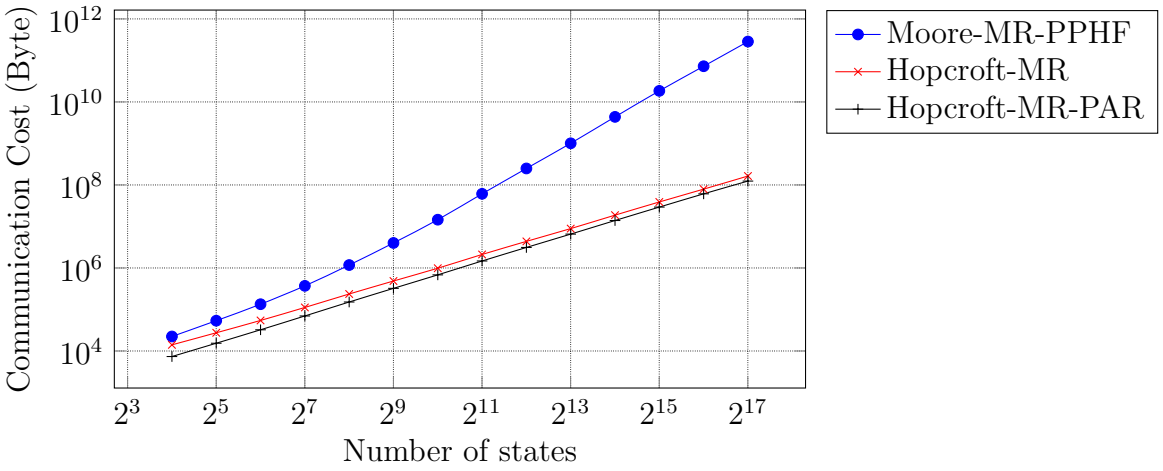


**Figure 20: Communication Cost of circular DFA for the alphabet size $k = 4$**

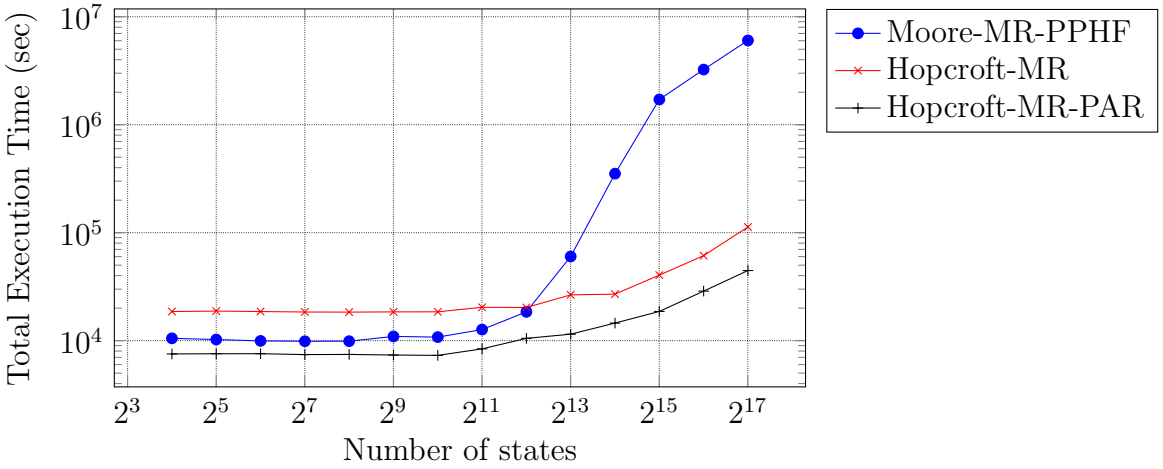


**Figure 21: Processing time of circular DFA for the alphabet size $k = 4$**

The last dataset is *Replicated-Random*. The number of alphabet is fixed and equal

to 4. The replicated-random DFA have $16 + 1$ states to $131,072 + 1$. Each DFA is replicated four times and parts are connected using an auxiliary state. The randomness prevents having uniform distribution of incoming transitions among the states. This fact injects skewness in the mapping schema for Hopcroft's algorithms. Figure 22 reflects the fact that Hopcroft-MR requires more communication amount than the others. Although Hopcroft-MR-PAR communicates almost same data as Moore-MR-PPHF, Figure 23 reveals that it requires more processing time than Moore-MR-PPHF to calculate the minimized version of DFA. Finally, for DFA with $32+1 = 2^5+1$ states, Hopcroft's algorithms had much better performance than Moore-MR-PPHF. This is due the fact that this DFA does not have any final records. Hopcroft's algorithms detect the matter in first round while Moore-MR-PPHF has to run 2 rounds.

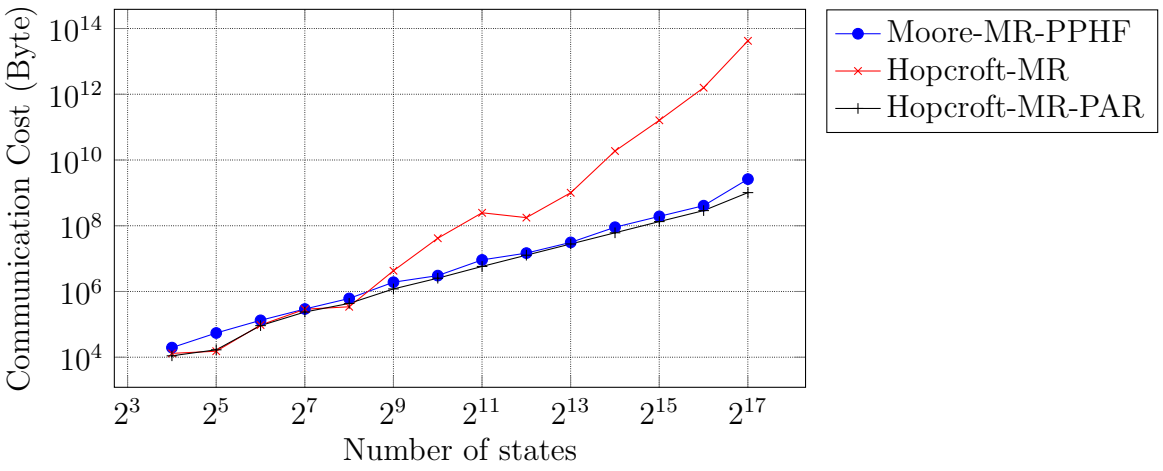


**Figure 22: Communication Cost of replicated-random DFA for alphabet size $k = 4$**

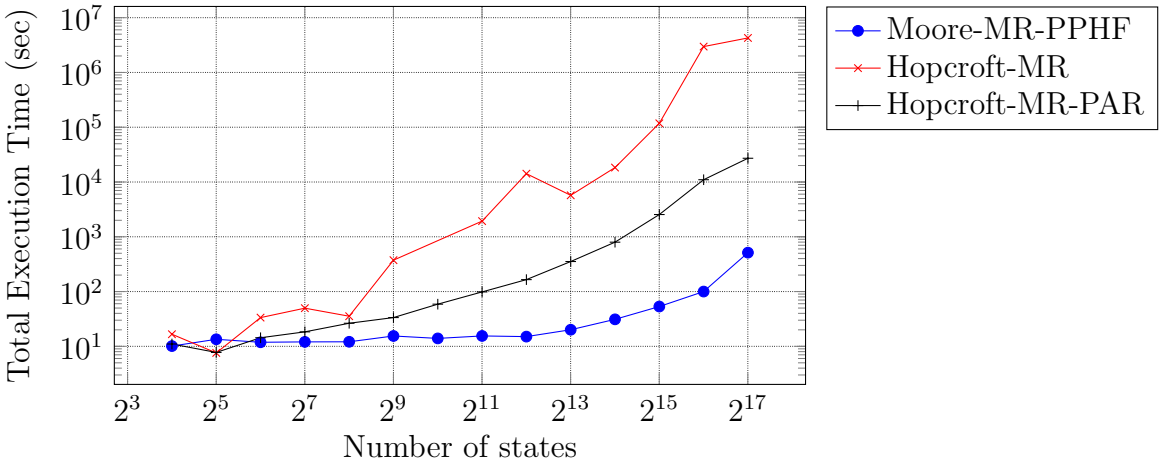


**Figure 23: Processing time of replicated-random DFA for alphabet size** $k = 4$

The number of rounds required for each algorithm is categorized by the data sets and listed in Table 7. For the *Linear* dataset, Moore-MR-PPHF requires exactly $n$ rounds where $n$ is the number of states in DFA. Hopcroft-MR takes $n + n/2 - 2 \approx 2n$ because of alphabet size $k = 2$ and Hopcroft-MR-PAR will do the minimization in $n - 2$ rounds. If $n = 1$, it does the procedure in 1 round. On the other hand, for *Circular* DFA, all the three algorithms accomplished the task in a fixed number of rounds. Lastly, running the algorithms on replicated-random DFA shows that Moore-MR-PPHF has finished the task in less than 5 rounds. Although Hopcroft-MR took much more rounds to do the minimization, by applying parallel processing of splitters in Hopcroft-MR-PAR, the number of rounds decreased dramatically.

**Table 7: Number of rounds for minimizing DFA using Moore-MR-PPHF, Hopcroft-MR, and Hopcroft-MR-PAR with different datasets**

| Data Set | Number of Alphabet | Number of States ($n$) | Moore-MR-PPHF | Hopcroft-MR | Hopcroft-MR-PAR |
|---|---|---|---|---|---|
| Linear | 2 | 2 to 1024 | $n$ | $n + n/2 - 2$ | $\max\{n-2, 1\}$ |
| Circular | 4 | 16 to 131072 | 3 | 6 | 1 |
| Random | 4 | $16 + 1$ | 3 | 4 | 4 |
| | 4 | $32 + 1$ | 2 | 1 | 1 |
| | 4 | $64 + 1$ | 4 | 13 | 4 |
| | 4 | $128 + 1$ | 4 | 19 | 6 |
| | 4 | $256 + 1$ | 4 | 22 | 9 |
| | 4 | $512 + 1$ | 5 | 75 | 13 |
| | 4 | $1024 + 1$ | 4 | 197 | 11 |
| | 4 | $2048 + 1$ | 5 | 487 | 16 |
| | 4 | $4096 + 1$ | 4 | 262 | 14 |
| | 4 | $8192 + 1$ | 4 | 797 | 10 |
| | 4 | $16384 + 1$ | 5 | 1003 | 19 |
| | 4 | $32768 + 1$ | 5 | 1810 | 25 |
| | 4 | $65536 + 1$ | 5 | 3709 | 21 |
| | 4 | $131072 + 1$ | 5 | 11486 | 25 |

# Chapter 5

# Conclusion and Future Work

## 5.1 Conclusion

In this research we studied methods for computing minimal equivalent automaton for large scale DFA using equivalence classes in Map-Reduce. Our focus was mainly on Moore's and Hopcroft's methods.

We proposed an algorithm for minimizing DFA in Map-Reduce based on Hopcroft's method, along with an enhancement to a previously work employed Moore's algorithm. Moreover, we showed that adding information to input records will cause *horizontal growth of data* [4] and it has to be controlled and measured in complexity models. For this matter, we proposed an efficient new parallel perfect hashing function in Map-Reduce model for numerical values. Furthermore, we extended the

Map-Reduce model in order to categorizing data on distributed file system for achieving more efficiency.

Additionally, we studied different measures of algorithms in Map-Reduce model with a focus on DFA minimization. We found that DFA minimization using equivalence classes are suitable for Map-Reduce model as we proved that the lower bound on replication of data is one and our algorithms are coincident with. We also found a lower and an upper bound for DFA minimization problem in parallel environments.

In addition, our analysis on topology of DFA supported by experimental results, shows that *Sensitivity on Skewness* is a critical measure for Map-Reduce algorithms.

The experiments done on different DFA showed that both methods have the same complexity in terms of amount of data has to be communicated in minimization process. However, it shows as well that Hopcroft's method is very sensitive to skewness.

## 5.2 Future Work

For the future works, the following question are interesting:

1. **Reducer capacity vs. Number of reducers trade-off**

   The capacity of reducer workers will directly affect the number of rounds for iterative problems such as DFA minimization. The question is that what is the relation between reducer capacity and number of rounds? How the problem

itself affects the trade-off?

2. **Investigating other methods of minimization**

   How the other methods of minimizing a DFA will behave in Map-Reduce model. We know that methods relying on equivalence classes are iterative and these problems show very bad results in Map-Reduce model. Now the question is a method like Brzozowski which computes the minimal DFA in two rounds, how can be compared with proposed algorithms in this study?

3. **Extending complexity model and class**

   Currently available complexity models and classes for Map-Reduce do not take into account data growth during iterations as well as sensitivity to skewness. We showed that how these to can decrease efficiency of Map-Reduce algorithms.

4. **Is it possible to compare Map-Reduce algorithms with others in different models?**

   The focus of Map-Reduce model is on solving data intensive problems in Big-Data concept where the others fail to run. To do so, scientists has to concern more on possibility and least in efficiency. The question is if it is wise to compare these two or not?

# References


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

[2] 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.

[3] Frédérique Bassino, Julien David, and Cyril Nicaud. On the average complexity of Moore's state minimization algorithm. *CoRR*, abs/0902.1048, 2009.

[4] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32Nd Symposium on Principles of Database Systems*, PODS '13, pages 273–284, New York, NY, USA, 2013. ACM.

[5] Jacek Becla and Daniel L. Wang. Lessons learned from managing a petabyte. In *Proceedings, 2nd Biennial Conference on Innovative Data Systems Research*, Stanford, CA, Jan 2005.

[6] Jean Berstel, Luc Boasson, Olivier Carton, and Isabelle Fagnot. Minimization of automata. *CoRR*, abs/1010.5318, 2010.

[7] Ronald V. Book, Christopher B. Wilson, and Xu Mei-Rui. Relativizing time, space, and time-space. *SIAM Journal on Computing*, 11(3):571–581, 1982.

[8] Kirk D. Borne. Scientific data mining in astronomy. *CoRR*, abs/0911.0505, 2009.

[9] Wilfried Brauer. On minimizing finite automata. *Bulletin of the EATCS*, 35:113–116, 1988.

[10] Parris M. Caulk. The design of a petabyte archive and distribution system for the NASA ECS project. In *Fourth NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 7–17. NASA, 1995.

[11] Sang Cho and Dung T. Huynh. The parallel complexity of coarsest set partition problems. *Inf. Process. Lett.*, 42(2):89–94, 1992.

[12] Sang Cho and Dung T. Huynh. The parallel complexity of finite-state automata problems. *Information and Computation*, 97(1):1–22, 1992.

[13] Julien David. The average complexity of Moore's state minimization algorithm is $O(n \log \log n)$. In *MFCS*, pages 318–329. Springer, 2010.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[16] Gösta Grahne, Shahab Harrafi, Ali Moallemi, and Adrian Onet. Computing NFA intersections in map-reduce. In *Proceedings of the Workshops of the EDBT/ICDT Joint Conference (EDBT/ICDT)*, pages 42–45, 2015.

[17] David Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2(2):97–109, 1973.

[18] Shahab Harrafi Saveh. Finite automata algorithms in map-reduce. Master's thesis, Concordia University, April 2015.

[19] 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.

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

[21] Joseph F. JáJá and Kwan Woo Ryu. An efficient parallel algorithm for the single function coarsest partition problem. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 230–239. ACM, 1993.

[22] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

[23] Eyal Kushilevitz. Communication complexity. *Advances in Computers*, 44:331–360, 1997.

[24] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

[25] Richard J. Lipton and Robert Sedgewick. Lower bounds for VLSI. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 300–307. ACM, 1981.

[26] Yossi Matias and Uzi Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, 1991.

[27] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[28] Bala Ravikumar and Xuanxing Xiong. A parallel algorithm for minimization of finite automata. In *The 10th International Parallel Processing Symposium*, pages 187–191, April 1996.

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

[30] 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*, pages 607–614. ACM, 2011.

[31] 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.

[32] H. M. M. ten Eikelder. Some algorithms to decide the equivalence of recursive types. Technical Report 31, Faculty of Computing Science, Eindhoven University of Technology, 1991.

[33] Ambuj Tewari, Utkarsh Srivastava, and Phalguni Gupta. A parallel DFA minimization algorithm. In *High Performance ComputingHiPC*, pages 34–40. Springer, 2002.

[34] György Turán. On the computational complexity of mapreduce. In *Distributed Computing: 29th International Symposium*, volume 9363 of *DISC 2015*, page 1, Tokyo, Japan, October 2015. Springer.

[35] Friedrich J. Urbanek. On minimizing finite automata. *Bulletin of the EATCS*, 39:205–206, 1989.

[36] Moshe Y. Vardi. Nontraditional applications of automata theory. In *Theoretical Aspects of Computer Software*, pages 575–597. Springer, 1994.

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

[38] Bruce W. Watson and Jan Daciuk. An efficient incremental DFA minimization algorithm. *Natural Language Engineering*, 9(01):49–64, 2003.

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

[40] Derick Wood. *Theory of Computation*. John Wiley, 1987.

[41] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 209–213, New York, NY, USA, 1979. ACM.

[42] Michael Yoeli. Canonical representations of chain events. *Information and Control*, 8(2):180 – 189, 1965.

[43] Paul C. Zikopoulos, Chris Eaton, Drik deRoos, Thomas Deutsch, and George Lapis. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011.