Modèle de calcul massivement parallèle en MapReduce pour quelques problèmes en théorie des automates

Bilal El Ghadyry

Thèse de Doctorat

Mention : Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique et Applications

présentée à *l'Ecole Doctorale en Sciences Technologie et Santé (ED 585)* de
l'Université du Littoral Côte d'Opale
et
au Centre D'Etudes Doctorales Sciences et Technologies de
l'Université Mohammed V de Rabat
par

Bilal El Ghadyry

pour obtenir le grade de Docteur de l'Université du Littoral Côte d'Opale

Modèle de calcul massivement parallèle en MapReduce pour quelques problèmes en théorie des automates

Soutenue le 17 décembre 2022, après avis des rapporteurs, devant le jury d'examen :

M. Mohamed Bouhdadi, Professeur, Université Mohammed V — Président
M^me^ Jihane Alami Chentoufi, Professeur, Université Ibn Tofail — Rapporteur
M. Mignot Ludovic, Maître de Conférences HDR, Université de Rouen Normandie — Rapporteur
M^me^ Yasmine Lamari, Maître de Conférences, Université Ibn Zohr — Examinateur

M. Sébastien Verel, Professeur, Université du Littoral Côte d'Opale — Directeur de thèse
M. Faissal Ouardi, Professeur, Université Mohammed V — Co-directeur

Numéro d'ordre : inconnu

Université du Littoral Côte d'Opale
Université Mohammed V de Rabat

École doctorale **Sciences, Technologie et Santé (ED 585)**
Unité de recherche **LISIC* et Équipe ANISSE****

Thèse présentée par **Bilal El ghadyry**

Soutenue le **17 décembre 2022**

En vue de l'obtention du grade de docteur de l'Université du Littoral Côte d'Opale et de l'Université Mohammed V de Rabat

Discipline **Informatique**
Spécialité **Informatique et Applications**

Modèle de calcul massivement parallèle en MapReduce pour quelques problèmes en théorie des automates

Thèse dirigée par Sébastien Verel directeur
Faissal Ouardi co-directeur

Composition du jury

Rapporteurs	Jihane Alami Chentoufi	Université Ibn Tofail de Kénitra	
	Ludovic Mignot	Université de Rouen Normandie	
Examinateurs	Mohamed Bouhdadi	Université Mohammed V de Rabat	président du jury
	Yasmine Lamari	Université Ibn Zohr d'Agadir	
Directeurs de thèse	Sébastien Verel	Université du Littoral Côte d'Opale	
	Faissal Ouardi	Université Mohammed V de Rabat	

Colophon

Mémoire de thèse intitulé « Modèle de calcul massivement parallèle en MapReduce pour quelques problèmes en théorie des automates », écrit par Bilal El ghadyry, achevé le 16 mars 2023, composé au moyen du système de préparation de document LaTeX et de la classe yathesis dédiée aux thèses préparées en France.

L'Université du Littoral Côte d'Opale et l'Université Mohammed V de Rabat n'entendent donner aucune approbation ni improbation aux opinions émises dans les thèses : ces opinions devront être considérées comme propres à leurs auteurs.

Mots clés : calcul parallèle, automate fini, automate pondéré, big data, mapreduce

Keywords: parallel computing, finite automata, weighted automata, big data, mapreduce

Cette thèse a été préparée au

LISIC⋆ et Équipe ANISSE⋆⋆
⋆Maison de la Recherche Blaise Pascal
50, rue Ferdinand Buisson
BP 719
62228 Calais Cedex
France

⋆⋆Équipe ANISSE,
Département d'informatique,
Faculté des sciences,
Université Mohammed V de Rabat,
Maroc

☎ (33)(0)3 21 46 36 53
(33)(0)3 21 46 55 75
secretariat@lisic.univ-littoral.fr
Site https://www-lisic.univ-littoral.fr/

⟨*épigraphe*⟩

Je dédie cette thèse à mes parents,
ma soeur et mes frères

Modèle de calcul massivement parallèle en MapReduce pour quelques problèmes en théorie des automates

Résumé

Dans cette thèse, nous étudions l'algorithmique parallèle à grande échelle de quelques problèmes en théorie des automates, à savoir la composition des machines à états finis pondérées, ainsi que la dérivation des séquences séparantes à partir des machines à états finis non déterministes. Nous adoptons dans notre étude, un modèle théorique de traitement parallèle récemment introduit appelé modèle de calcul massivement parallèle en MapReduce (CMP-MR) où le seul coût est donné par la quantité de communication entre les nœuds et le nombre d'itérations de l'algorithme. Nous avons proposé des algorithmes parallèles efficaces à grande échelle en MapReduce basés sur le modèle CMP-MR pour chacun des problèmes traités. Les résultats obtenus montrent clairement que le modèle CMP-MR offre un cadre théorique garanti pour le développement de nouveaux algorithmes parallèles efficaces à grande échelle en théorie des automates.

Mots clés : calcul parallèle, automate fini, automate pondéré, big data, mapreduce

Abstract

In this thesis, we tackle some known problems in automata theory using parallel approaches in large scale context, namely the composition of weighted finite state machines and the derivation of distinguishing sequences for nondeterministic finite state machines. We adopt in our study a theoretical model of parallel processing called the massive parallel computing model in MapReduce (MPC-MR) where the only cost is given by the amount of communication between the nodes of a cluster and the number of iterations of the algorithm. We have proposed efficient large-scale parallel MapReduce algorithms based on MPC-MR model for each of the addressed problems. The obtained results show clearly that the MPC-MR model offers a guaranteed theoretical framework for the development of new efficient parallel algorithms in automata theory.

Keywords: parallel computing, finite automata, weighted automata, big data, mapreduce

LISIC⋆ et Équipe ANISSE⋆⋆

⋆Maison de la Recherche Blaise Pascal – 50, rue Ferdinand Buisson – BP 719 – 62228 Calais Cedex – France – – ⋆⋆Équipe ANISSE, – Département d'informatique, – Faculté des sciences, – Université Mohammed V de Rabat, – Maroc

Remerciements

Je tiens à remercier particulièrement mes directeurs de thèse le professeur *Faissal Ouardi* et le professeur *Sébastien Verel* pour leurs encadrements tout au long de ce travail. Merci pour m'avoir donné l'opportunité de vivre cette aventure, de m'avoir fait confiance et pour avoir su être exigeant tout en me laissant libre d'explorer toutes les pistes que je souhaitais entreprendre. Je tiens à leur exprimer ma plus profonde gratitude pour le soutien moral, l'encouragement et l'extrême patience dont ils ont fait part.

Je suis très reconnaissant au professeur *Mohamed Bouhdadi* d'avoir accepté de présider le Jury de ma soutenance de thèse. Je remercie également les professeurs *Jihane Alami Chentoufi* et *Mignot Ludovic* qui ont accepté la tâche fastidieuse de rapporter cette thèse et le professeur *Yasmine Lamari* de bien vouloir participer au jury de cette thèse et d'examiner ce travail. Leur présence à la soutenance de cette thèse m'honore énormément.

Je tiens à remercier tous les membres de l'Équipe de Systèmes intelligents, Réseaux, Génie Logiciel et Algorithmes (ANISSE , UM5) et du Laboratoire d'Informatique Signal et Image de la Côte d'Opale (LISIC, ULCO). Je remercie le Centre National pour la Recherche Scientifique et Technique (CNRST, Maroc) et l'Université du Littoral Côte d'Opale (ULCO) pour m'avoir attribué le financement qui m'a permis de travailler dans de bonnes conditions.

Je remercie également les professeurs Mohammed Benkhalifa, professeur à Université Mohammed V de Rabat, Virginie Marion-Poty, professeur à Université du Littoral Côte d'Opale, Djelloul Ziadi, professeur à Université de Rouen Normandie, pour leurs encouragements, conseils et aide aussi bien sur le plan personnel que professionnel.

Je remercie par la même occasion Gaëlle Compiègne, Nihal Ferhane et Philippe Marion leurs aides tout au long du processus de recherche de cette thèse. J'adresse toute ma gratitude à tous mes ami(e)s et à toutes les personnes qui m'ont aidé dans la réalisation de ce travail. Enfin, les mots les plus simples étant les plus forts, j'adresse toute mon affection à ma famille.

Sommaire

Liste des tableaux

Table des figures

Introduction

Contexte général du travail

Au cours de la dernière décennie, le traitement de grandes masses de données sur de grands clusters distribués a suscité un grand intérêt des scientifiques dans différents domaines de recherche. Cette tendance a commencé avec le framework MapReduce [52], qui a été largement adopté par plusieurs autres systèmes, comme PigLatin [125], Hive [145], Scope [40], Dremmel [114], Spark [158] et Myria [153], pour n'en citer que quelques-uns. Tandis que les applications de ces systèmes sont diverses, la plupart impliquent des tâches de traitement de données relativement standards, telles que identifier les données pertinentes, nettoyer, filtrer, joindre, regrouper, transformer, extraire des caractéristiques et évaluer les résultats [43].

D'un point de vue algorithmique, Afrati and Ullman [3] ont introduit le modèle de Calcul Massivement Parallèle en MapReduce (CMP-MR) proposant ainsi un cadre théorique pour le développement des algorithmes parallèles optimaux sur de grands clusters distribués. Ce modèle est une simplification du modèle introduit par Valiant appelé Bulk Synchronous Parallel (BSP) [150], qui permet de séparer le coût de calcul du coût de communication, et d'étudier uniquement ce dernier.

De nombreux problèmes ont été étudiés en utilisant le modèle CMP-MR, notamment ceux portant sur les graphes [4, 16, 17, 15, 67, 22, 20, 21, 24, 28, 42, 50, 66, 64, 106, 132], sur classification [25, 27, 58, 65, 157, 73] et sur l'optimisation de fonction sous-modulaire [26, 59, 100, 117].

Le modèle CMP-MR est également utilisé pour savoir dans quelle mesure un problème donné accepte une solution MapReduce. Le coût d'un algorithme distribué selon le modèle CMP-MR est mesuré en fonction du coût de communication des machines du cluster et le nombre d'itérations de l'algorithme. L'entrée est répartie selon un schéma bien défini sur toutes les machines et le calcul se déroule en itérations synchronisées. À chaque itération, une machine exécute une tâche sur sa propre mémoire. Une fois la tâche terminée, chaque machine calcule un ensemble de messages. Ensuite, le système sous-jacent envoie les messages

vers la machine correspondante. Le coût global de l'algorithme distribué provient principalement des coûts de communication par machine. Par conséquent, la conception d'un algorithme efficace selon le modèle CMP-MR nécessite l'étude du compromis entre le degré de parallélisme et coût de communication dans une architecture distribuée. Ce modèle propose un schéma générique permettant de calculer des bornes inférieures sur le coût de communication en fonction de la taille maximale d'entrées pouvant être affectées à une machine et le taux de réplication d'une donnée dans les différentes machines.

Problématique

Les automates finis sont l'un des dispositifs les plus utiles et les plus pratiques pour modéliser une pléthore de phénomènes liés au calcul. Ils sont largement utilisés pour modéliser une variété de matériels et de logiciels tels que les logiciels de conception du comportement des circuits numériques, les composants d'analyse lexicale des compilateurs, les logiciels de comparaison de motifs, etc. [86].
Le fait de n'avoir qu'un nombre fini d'états fait des automates finis des dispositifs appropriés pour être implémentés dans un matériel ou un programme tel que ceux mentionnés ci-dessus.
Les automates finis sont utilisés pour décrire une classe majeure de langages formels appelés langages réguliers. Plus précisément, un langage L est dit régulier si et seulement s'il existe un automate fini A tel que le langage reconnu par A est égal à L. Un automate fini possède un ensemble d'états et de transitions qui permettent de consommer l'entrée en passant d'un état à un autre. Il existe essentiellement deux types d'automates finis :

— *Automates finis déterministes* (DFA). Un automate fini est déterministe s'il ne possède pas le choix dans ses déplacements. En d'autres termes, il n'y a qu'un seul état vers lequel une machine peut se déplacer sur un symbole d'entrée.
— *Automates finis non déterministes* (NFA). Par opposition aux DFA, un NFA est une machine dont les états peuvent avoir un ensemble de transitions basées sur un symbole d'entrée particulier. De plus, il a la possibilité de quitter son état actuel sans lire aucune entrée en autorisant une transition sur ϵ, le mot vide. Les NFAs avec ϵ-transitions sont appelés ϵ-NFAs.

En raison de leur simplicité et de leur capacité à modéliser des systèmes complexes, les machines à états finis (FSMs), qui représentent une classe d'automates finis, sont largement utilisées dans plusieurs domaines tels que les protocoles de communication [6], la recherche de motifs [98], la reconstruction d'événements

numériques [68], les contrats intelligents [112], les tests de charge distribués [87, 80], la génomique [144] et les systèmes réactifs [14]. Une FSM est un modèle qui possède un nombre fini d'états, d'entrées, de sorties, et un nombre fini de transitions, chacune étiquetée par une paire de symboles entrée/sortie. En outre, les FSMs sont les modèles sous-jacents des techniques de description formelle, telles que les diagrammes d'état, le langage de spécification et de description (SDL) [135], le langage de modélisation unifié (UML) [113], les dispositifs logiques programmables [49] et les contrats intelligents Ethereum [112].

À travers cette thèse, nous étudions quelques problèmes algorithmiques en théorie des automates en utilisant le modèle CMP-MR, montrant ainsi l'adaptabilité de ce modèle pour le calcul parallèle à grande échelle en théorie des automates.
Le présent travail traite deux problématiques, à savoir :

— La composition des machines à états finis pondérées [56] [1],
— La génération de séquences séparantes des états dans les machine à états finis non déterministes [57] [2].

La NP-dureté du problème de composition des machines à états finis pondérées [154, 9, 119] présente un défi qui nécessite la conception de méthodes efficaces en considérant plus que deux transducteurs à grande échelle. Nous introduisons pour la première fois un algorithme parallèle pour la composition de machines à états finis pondérées en utilisant le modèle CMP-MR.

Quant au second problème, il consiste en la construction parallèle en utilisant le modèle de calcul MapReduce des séquences séparantes minimales des différents états deux à deux d'une machine à états finis non déterministe observable complète. Il est à noter que ce dernier problème est classique dans le domaine de test et vérification des logiciels [45, 55, 76, 44, 96].

Organisation du manuscrit

Ce rapport est structuré en cinq chapitres.

Le premier chapitre constitue des préliminaires sur les langages formels, automates finis, paradigme du Big Data et le modèle de calcul MapReduce. Il introduit les définitions et les notations élémentaires qui seront utilisées dans la suite de ce rapport.

Le deuxième chapitre présente de façon détaillée le modèle de calcul massivement parallèle en MapReduce. Des exemples explicatifs sont introduits tout au long du deuxième chapitre afin de clarifier les différentes notions liées à ce

1. https://gitlab.com/bilalgh/compositionmr
2. https://gitlab.com/bilalgh/pds-mr

modèle.

Le troisième et le quatrième chapitres représentent le cœur de cette thèse. Ils contiennent une description de la méthodologie suivie pour la conception d'algorithmes optimaux en MapReduce pour deux problèmes connus en théorie des automates ainsi que les résultats expérimentaux obtenus sur une large variante de données à grande échelle. Le troisième chapitre présente une nouvelle approche parallèle pour le calcul massivement parallèle de la composition des machines à états finis pondérées à grande échelle. Quant au quatrième chapitre, il présente également une nouvelle approche parallèle pour la construction des séquences séparantes à partir des machines à états finis non déterministes en utilisant le modèle de calcul massivement parallèle en MapReduce.

Chapitre 1

Préliminaires

La théorie des automates est née de la convergence de plusieurs courants scientifiques, et parmi ces courants : le premier est issu des tentatives de logiciens tels que Church, Gödel ou Turing pour formaliser la notion de calcul et de machine [39, 5]. Cet effort a occupé toute la première moitié du vingtième siècle et pourtant, les automates finis, qui constituent le modèle le plus simple de machine, ne seront définis formellement que bien après les machines de Turing. Le deuxième est de linguistes tels que Chomsky qui, en cherchant à formaliser les langues naturelles, ont introduit les concepts de mots, langages, grammaires, que nous utilisons aujourd'hui. Ce chapitre introduit les définitions et les notations élémentaires de la théorie des automates qui seront utilisées dans la suite de ce rapport. Pour plus de détails sur les aspects formels de la théorie des automates finis, nous recommandons particulièrement la lecture de [136].

1.1 Théorie des langages

La théorie des langages est la base de la construction des automates, structures fondamentales en informatique théorique. En théorie des langages, l'ensemble des entités élémentaires est appelé l' *alphabet*. Une combinaison d'entités élémentaires est appelée un *mot*. Un ensemble de mots est appelé un *langage* [39].

1.1.1 Mots et langages

En informatique, mais aussi en mathématiques, en linguistique ou en biologie, les informations sont souvent représentées par des chaînes de caractères. Par exemple, pour les données informatiques, on utilise des suites de 0 et de 1, pour

l'information génétique, des suites formées des quatre caractères A (Adénine), C (Cytosine), G (Guanine) et T (Thymine) et pour les langues naturelles, les mots figurant dans un dictionnaire.

La formalisation commune à ces exemples est la suivante. Un alphabet est un ensemble fini dont les éléments sont appelés des lettres. Ainsi, on parle de l'alphabet binaire $\{0,1\}$, de l'alphabet du génome $\{A,C,G,T\}$, de l'alphabet latin usuel $\{a,...,z,A,...,Z\}$. Dans les exemples qui vont suivre, on utilisera le plus souvent des alphabets assez petits tels que $\{a,b,c\}$ ou $\{0,1,2\}$ et on notera Σ l'alphabet tout entier.

Un *mot* sur l'alphabet Σ est une suite finie de lettres de Σ. On note ces lettres par simple juxtaposition : ainsi le mot *abaabb* est un mot sur l'alphabet $\{a,b\}$. La longueur d'un mot u, notée $|u|$, est égale au nombre de lettres dans u, chaque lettre étant comptée autant de fois qu'elle apparaît. Ainsi, $|abaabb| = 6$. Il existe aussi un mot de longueur 0 appelé *mot vide*, que l'on note ϵ.

On appelle *langage* tout ensemble de mots sur un alphabet donné. Par exemple, les ensembles $\{aba, babaa, bb\}$ et $\{a^n b^n \mid n \geqslant 0\}$ sont des langages sur l'alphabet $\{a,b\}$. D'après le théorème de Kleene [97], un langage sur un alphabet A est *régulier*, appelé aussi *rationnel* ou reconnaissable si et seulement s'il est reconnu par un automate fini, et on peut le décrire à partir des lettres de l'alphabet A en utilisant trois opérations : l'union, la concaténation et l'étoile.

1.1.2 Représentation des langages

La plupart du temps, la définition d'une *classe de langages* $\mathbb{L}$ est liée à celle d'une classe de machines abstraites $\mathbb{R}$, appelées ici *représentations*, caractérisant tous les langages de $\mathbb{L}$ et seulement ces langages. La relation entre ces deux classes est formalisée par la *fonction de nommage* $\mathcal{L}_{\mathbb{R},\mathbb{L}}$ ou plus simplement $\mathcal{L} : \mathbb{R} \longrightarrow \mathbb{L}$ telle que :

1. $\forall R \in \mathbb{R}, \mathcal{L}(R) \in \mathbb{L}$,
2. $\forall L \in \mathbb{L}, \exists R \in \mathbb{R}$ tel que $\mathcal{L}(R) = L$.

Deux représentations R_1 et R_2 sont équivalentes si et seulement si $\mathcal{L}(R_1) = \mathcal{L}(R_2)$. Dans ce document, nous nous préoccuperons principalement des langages réguliers caractérisés par les automates à états finis.

1.2 Automates finis

Les automates finis sont des *machines abstraites* qui savent reconnaître l'appartenance ou la non-appartenance d'un mot à un langage régulier donné [39, 5]. Ces machines abstraites constituent un modèle théorique de référence. Dans

la pratique, nombreuses sont les applications qui implémentent la notion d'automates finis ou ses variantes. Un automate lit le mot écrit sur son ruban d'entrée et à partir d'un (ou plusieurs) état initial et à chaque lettre lue, il change d'état. Si, à la fin du mot, il est dans un état final, on dit qu'il reconnaît le mot lu.

1.2.1 Automates finis

Un automate fini A est la donnée d'un quintuplet $(\Sigma, Q, \delta, q_0, F)$ tel que :

- Σ est un ensemble fini de symboles,
- Q est un ensemble fini d'états,
- δ est un ensemble de règles de transition $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$,
- q_0 est l'état initial,
- F est un sous-ensemble de Q appelé l'ensemble des états finaux.

Un automate fini est déterministe (DFA) si et seulement si δ est une fonction de transition telle que $\delta : Q \times \Sigma \longrightarrow Q$, c'est-à-dire que les transitions à partir de chaque état sont déterminées de façon unique par la lettre d'entrée et on ne peut plus effectuer de transition par ϵ. Un tel automate se distingue ainsi d'un automate fini non déterministe (NFA), où au contraire plusieurs possibilités de transitions peuvent exister simultanément à partir d'un état et une lettre donnée. Dans un automate fini non déterministe, il peut y avoir le choix entre plusieurs chemins lors de la lecture d'un mot, et pour qu'un mot soit accepté, il suffit que ses lettres étiquettent un chemin d'un état initial à un état final. Cependant, il peut y avoir d'autres chemins ne menant pas à un état final et étiqueté par le mot accepté par ailleurs, ou même des lectures du mot s'arrêtant en cours de route.

Les deux classes d'automates finis, les automates finis déterministes et non déterministes, ont la même puissance d'expression : elles reconnaissent la même famille de langages, à savoir les langages rationnels, appelés aussi langages réguliers ou langages reconnaissables.

Exemple 1. *La Figure 1.1 schématise un exemple d'automate fini sur l'alphabet $\{a, b\}$. Cet automate possède trois états q_0, q_1 et q_2, entourés par des cercles sur la figure. L'état q_0 est l'état initial, ce qu'on indique par une flèche entrante. L'état q_2 est l'état final, ce qui est indiqué par double cercle. Cet automate possède aussi des transitions, représentées par des flèches étiquetées par des lettres de l'alphabet $\{a, b\}$. Le langage reconnu par cet automate est l'ensemble de tous les mots commençant par a et se terminant par b.*

1.2.2 Machines à états finis

Les machines à états finis (FSM, Finite State Machines) constituent une classe des automates finis avec sortie pour lesquels les sorties dépendent à la fois de

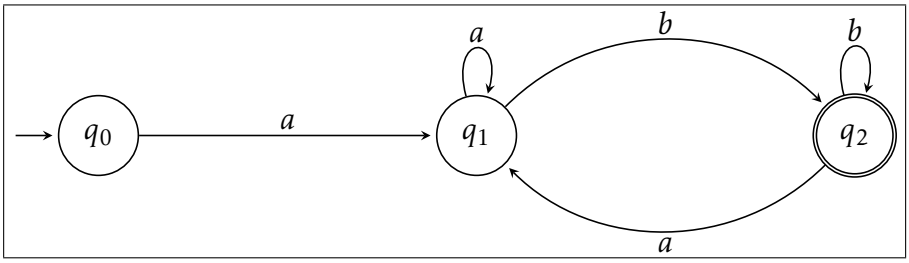

FIGURE 1.1 – Automate fini sur l'alphabet $\{a, b\}$

l'état courant et des symboles d'entrée. Cela signifie que l'étiquette de chaque transition est un couple formé d'une lettre d'entrée et d'une lettre de sortie. En particulier, la longueur du mot de sortie est égale à la longueur du mot d'entrée.

Formellement, une FSM M est définie par quadruplet $M = (S, I, O, E)$, tel que :

- S est un ensemble fini d'états,
- I est un ensemble fini de symboles d'entrée,
- O est un ensemble fini de symboles de sortie,
- $E \subseteq S \times I \times O \times S$ est un ensemble de transitions.

Pour une transition $t = (s, i, o, s') \in E$, on note $s[t]$ son origine ou état source s, $d[t]$ son état de destination ou état suivant de s', $I[t]$ son symbole d'entrée i, et $O[t]$ son symbole de sortie o. Soit s un état dans S, nous désignons par E_s le sous-ensemble de transitions de E ayant s comme état de départ, *i.e.* pour tout $t \in E_s$, on a $s[t] = s$. Une FSM est non déterministe si pour un état $s \in S$, il existe deux transitions $t, t' \in E$ avec $t \neq t'$ tel que $s[t] = s[t']$ et $I[t] = I[t']$.
Une FSM est dite *complète* si pour chaque paire $(s, i) \in S \times I$, il existe $(o, s') \in O \times S$ telle que $(s, i, o, s') \in E$. Dans le cas contraire, la machine est dite *partielle*.
Une FSM non déterministe complète est *observable* si pour chaque état s de la machine, il existe au plus une transition t sortante de s étiquetée par une paire entrée/sortie i/o donnée *i.e.* pour tout $t, t' \in E$ avec $t \neq t'$ tel que $s[t] = s[t']$ et $I[t] = I[t']$ alors $O[t] \neq O[t']$. Sinon, la machine est dite *non observable*.
Une FSM est dite *fortement connexe* si pour chaque état $s \in S$ il existe une séquence d'entrée qui permet à la FSM de passer d'un état arbitraire à l'état s.
Pour une FSM $M = (S, I, O, E)$, un état s et une entrée i, l'ensemble *i-successeur* de l'état s contient chaque état s' pour lequel il existe un symbole de sortie $o \in O$ tel que $(s, i, o, s') \in E$. Étant donné un sous-ensemble d'états $S' \subseteq S$ et un symbole d'entrée i, l'ensemble des états S' est le *i-successeur* de S si S' est l'union des ensembles *i-successeurs* sur tous les états de l'ensemble S.

Exemple 2. *La Figure 1.2 représente une machine à états finis M. Cette machine définie sur les ensembles d'entrée $I = \{a, b\}$, de sortie $O = \{0, 1\}$, et d'états $S =$*

$\{s_0, s_1, s_2, s_3, s_4\}$.

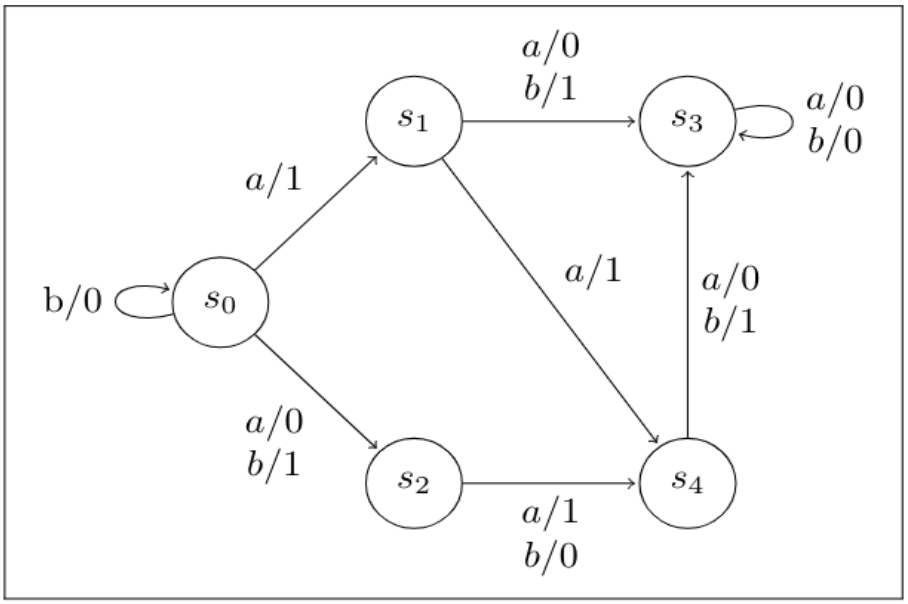

FIGURE 1.2 – Machine à états finis M

Cette FSM est non déterministe complète non observable.
L'ensemble a-successeur de l'état s_0 est l'ensemble des états $\{s_1, s_2\}$, le a-successeur de l'état s_1 est l'ensemble des états $\{s_3, s_4\}$. Donc, le a-successeur de l'ensemble des états $\{s_0, s_1\}$ est l'ensemble $\{s_1, s_2\} \cup \{s_3, s_4\} = \{s_1, s_2, s_3, s_4\}$.

1.2.3 Machines à états finis pondérées

Les machines à états finis pondérées (WFSM, Weighted Finite State Machines) sont une extension de la classe des machines à états finis dans lesquelles chaque transition, en plus de ses symboles d'entrée et de sortie, comporte un élément de pondération appartenant à un semi-anneau. Elles ont été utilisées dans un large éventail d'applications telles que le traitement d'images [94, 47], la reconnaissance vocale [121, 120] et plus récemment, en bioinformatique [10, 134].

Semi-anneaux. Un système $(\mathbb{K}, \oplus, \otimes, \underline{0}, \underline{1})$ est un *semi-anneau* si $(\mathbb{K}, \oplus, \underline{0})$ est un monoïde commutatif avec l'élément neutre $\underline{0}$; $(\mathbb{K}, \otimes, \underline{1})$ est un monoïde avec l'élément neutre $\underline{1}$; $\otimes$ est distributive par rapport au $\oplus$; et $\underline{0}$ est un annihilateur (annihilator) pour l'opération $\otimes$: pour tout $a \in \mathbb{K}$, $a \otimes \underline{0} = \underline{0} \otimes a = \underline{0}$. Certaines familles de semi-anneaux incluent le semi-anneau booléen $(\mathbb{B}, \vee, \wedge, 0, 1)$, le semi-anneau tropical $(\mathbb{R}_+ \cup \{\infty\}, min, +, \infty, 0)$, et le semi-anneau des réels $(\mathbb{R}, +, \times, 0, 1)$

[118].

Transducteurs à états finis pondérés. Soient A et B deux alphabets. Les transducteurs à états finis pondérés (WFST, Weighted Finite States Transducers), parfois appelés transducteurs avec multiplicités ou $\mathbb{K}$-transducteurs sur $A^* \times B^*$ constituent une classe plus générale que les FSMs pondérées aux quelles on rajoute des états initiaux avec des poids d'entrée et des états finaux avec des poids de sortie. Les étiquettes de transition d'un WFST sont dans $A \times B$.
Formellement, un transducteur pondéré T sur un semi-anneau $(\mathbb{K}, \oplus, \otimes, \underline{0}, \underline{1})$ est un 8-tuple $(A, B, Q, I, F, E, \lambda, \rho)$ tel que :

- A est un ensemble fini, appelé alphabet d'entrée ;
- B est un ensemble fini, appelé alphabet de sortie ;
- Q est un ensemble fini, l'ensemble des états ;
- I est un sous ensemble de Q, l'ensemble des états initiaux ;
- F est un sous ensemble de Q, l'ensemble des états finaux ;
- $E \subseteq Q \times (A \times B) \times \mathbb{K} \times Q$ est l'ensemble fini des transitions ;
- $\lambda : I \longrightarrow \mathbb{K}$ fonction de poids sur les états initiaux ;
- $\rho : F \longrightarrow \mathbb{K}$ fonction de poids sur les états finaux.

La taille d'un WFST $\mathcal{T}$, dénotée par $|\mathcal{T}|$, est le nombre de ses transitions. Étant donnée une transition $t \in E$, on note $s[t]$ son origine ou état source, $d[t]$ son état de destination ou état suivant, et $w[t]$ son poids. Un chemin $\pi = t_1 \cdots t_k$ est un élément de E^* composé de transitions consécutives telles que : $d[t_{i-1}] = s[t_i]$, $i = 2, \cdots, k$. Nous étendons s et d aux chemins : $s[\pi] = s[t_1]$ et $d[\pi] = d[t_k]$. La fonction de poids w peut également être étendue aux chemins en définissant le poids d'un chemin par le $\otimes$-produit du semi-anneau $\mathbb{K}$ des poids associés à ses transitions : $w[\pi] = w[t_1] \otimes \cdots \otimes w[t_k]$. Soit $q, q' \in Q$, nous dénotons par $P(q, q')$ l'ensemble des chemins allant de q vers q' et par $P(q, x, y, q')$ l'ensemble des chemins allant de q vers q' étiquetés en entrée par $x \in A^*$ et en sortie par $y \in B^*$. Ces définitions peuvent être étendues aux sous-ensembles $R, R' \subseteq Q$, par : $P(R, x, y, R') = \bigcup_{q \in R, q' \in R'} P(q, x, y, q')$. Un WFST $\mathcal{T}$ est régulé si le poids associé par $\mathcal{T}$ à chaque paire de mots d'entrée/sortie (x, y) :

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

est bien défini dans le semi-anneau $\mathbb{K}$. Nous avons $[\mathcal{T}](x, y) = 0$ quand $P(I, x, y, F) = \emptyset$.

Exemple 3. *La Figure 1.3 schématise un WFST sur $\{a, b, c\} \times \{a, b, c\}$ avec des multiplicités dans le semi-anneau probabiliste P. L'état q_a est l'état initial ayant comme*

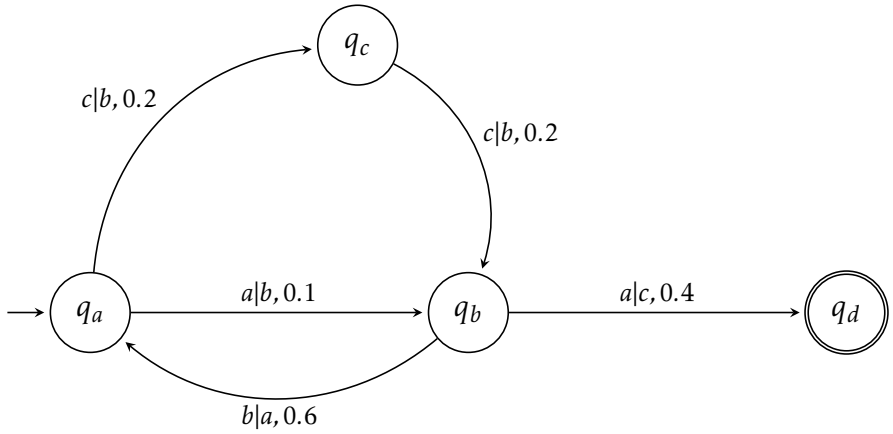

FIGURE 1.3 – Un WFST T sur le semi-anneau probabiliste P.

poids d'entrée 1, *i.e.* $\lambda(q_a) = 1$. *L'état* q_d *est un état final, ce qui est indiqué par double cercle, ayant comme poids de sortie* 1, *i.e.* $\rho(q_a) = 1$. *Par conséquent, le poids associé par ce WFST au paire de mots* (cca, bbc) *est calculé comme suit :*

$$\begin{aligned}
[\mathcal{T}](cca, bbc) &= \bigoplus_{\pi \in P(I, cca, bbc, F)} \lambda[s[\pi]] \otimes w[\pi] \otimes \rho[d[\pi]] \\
&= \lambda(q_a) \times w(q_a, c, b, 0.2, q_c) \times w(q_c, c, b, 0.2, q_b) \times w(q_b, a, c, 0.4, q_d) \times \rho(q_d) \\
&= 0.016
\end{aligned}$$

1.3 Paradigme du Big Data

Le Big Data est un concept relativement récent qui fait référence à un volume considérable et complexe de données. Ces données sont si importantes qu'elles sont difficiles à traiter et à analyser à l'aide des systèmes de gestion de base de données traditionnels. Le Big Data est apparu entre les années 1980 et 2000 et s'est rapidement développé avec le développement des technologies. John Mashey a fait une déclaration à USENIX en 1999 pour définir le Big Data comme un domaine informatique et cette idée a été reprise par la littérature scientifique pour analyser les données statistiques [61, 124].

Dans la suite de cette section, nous allons explorer diverses définitions du terme *Big Data* et discuter des plateformes disponibles pour le traitement de données à grande échelle. Puis, le framework MapReduce sera présenté en détails dans la section 1.4.

1.3.1 Définitions du Big Data

Le phénomène du "Big Data" est décrit par de nombreuses définitions disponibles dans la littérature scientifique, mais il n'est pas aussi simple qu'il y paraît de trouver une définition précise du Big Data. En effet, la signification du terme "Big Data" est en constante évolution. L'ensemble de ces définitions proviennent des entreprises de premier plan mondial et de professionnels de la recherche et du développement dans le domaine de technologie et de l'informatique tels que SAS, IBM, McKinsey et Gartner. Tous définissent le phénomène Big Data selon certaines dimensions ou caractéristiques [89, 23]. Ces caractéristiques doivent s'appliquer toutes ensemble pour indiquer qu'un ensemble de données est considéré comme du Big Data. Certaines de ces définitions sont données ci-dessous.

- **Définition 3Vs :** Le Big Data est défini selon les 3Vs [105], qui font référence au grand *Volume* de données, à leur *Vélocité* de production et de mise à jour, ainsi qu'à la *Variété* des données. Cette définition nécessite un traitement innovant et rentable pour permettre une meilleure compréhension et prise de décision. Le *Volume* se réfère à la quantité massive de données générées par les entreprises et les individus à partir de différentes sources. La *Vélocité* correspond à la rapidité à laquelle les données sont produites, capturées, partagées et mises à jour, ce qui s'accélère avec l'avancée des technologies. Enfin, la *Variété* se rapporte aux types et sources de données de plus en plus complexes et diversifiés par rapport aux données classiques, collectées à partir de multiples sources et formats différents. Les nouveaux types de données sont donc de plus en plus nombreux [41, 104].
- **Définition 5Vs :** Zhai *et al.* [159] ont proposé une définition élargie du framework 3V, en ajoutant deux caractéristiques supplémentaires : la *Véracité* et la *Valeur*. Selon cette définition, le Volume représente à la fois la taille de l'instance et la dimensionnalité des données, puisqu'il est possible de rencontrer des données de grande dimension dans des problèmes de petite taille d'échantillon. La *Véracité* ou fiabilité des données renvoie aux différents niveaux de qualité et de sécurité des données, tandis que la *Valeur* se rapporte aux avantages et aux informations pouvant être tirés de l'analyse du Big Data. Pour cela, il est généralement nécessaire d'adopter des approches de sélection d'entités permettant d'identifier un sous-ensemble pertinent et de grande valeur à partir d'un grand jeu de données original contenant des entités potentiellement non-pertinentes, redondantes, bruyantes ou manquantes [23, 104].

- **Définition SAS :** La définition proposée par SAS [1] élargit le framework 3V en incluant deux caractéristiques supplémentaires : la *Variabilité* et la *Véracité*. La *Variabilité* renvoie aux flux de données très incohérents qui peuvent survenir à certains moments, souvent associés à une augmentation de la vitesse et de la variété des données. Quant à la *Véracité*, elle souligne l'importance de la difficulté de relier, faire correspondre, nettoyer et transformer les données provenant de multiples sources différentes afin de garantir un contrôle de la qualité des données.

1.3.2 Plateformes du Big Data

Dans cette section, nous examinons les différentes plateformes d'analyse des Big Data sur la base d'une étude structurée proposée dans [139]. Selon cette étude, deux catégories de plateformes Big Data peuvent être distinguées : les plateformes de scalabilité horizontale et les plateformes de scalabilité verticale. La figure 1.4 montre les différentes plateformes de Big Data. Le traitement des Big Data repose principalement sur des modèles de programmation parallèles, tels que MapReduce, et des services cloud pour les Big Data.

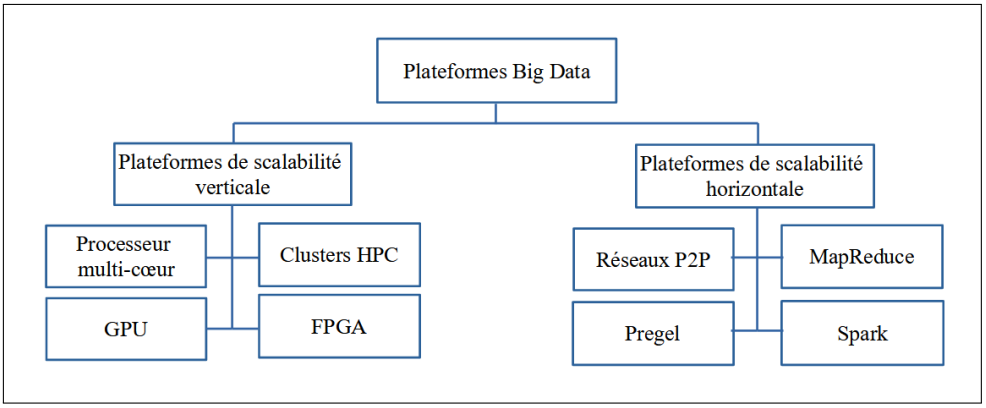

FIGURE 1.4 – Classification des différentes plateformes Big Data.

La scalabilité verticale

La scalabilité verticale rassemble des systèmes qui fonctionnent sur un seul serveur et permettent d'ajouter des ressources supplémentaires, telles que des processeurs, de la mémoire et du matériel rapide [139]. Cette catégorie comprend les clusters de calcul haute performance (HPC), les processeurs multi-cœurs, les processeurs graphiques (GPU) et les réseaux de portes programmables sur site (FPGA). Vous trouverez ci-dessous un bref aperçu de chacune de ces plateformes.

1. https://www.sas.com/fr_fr/insights/big-data/what-is-big-data.html

- ***Processeur multi-cœur*** est un processeur qui intègre plusieurs cœurs dans une seule puce. Les progrès significatifs réalisés au fil des ans dans le domaine des puces d'unité centrale ont contribué à l'émergence des architectures multicœurs et constituent un élément clé de la puissance de traitement requise pour les données volumineuses. Généralement, on distingue trois architectures ([7]). La première architecture partage de cache sur la puce entre les unités d'exécution, tandis que la seconde architecture fournit de cache dédiée pour chaque cœur. La troisième architecture adopte une approche hybride qui a subdivisé le cache en deux types de couches : des couches dédiées à un cœur particulier et d'autres partagées entre tous les cœurs [51].
- ***Clusters HPC (High-Performance Computing)*** est composé d'un grand nombre de cœurs. Les clusters sont le cadre de système HPC le plus couramment utilisé. Le calcul parallèle est plus efficace lorsqu'il est effectué sur plusieurs serveurs que sur des systèmes spécialisés. Les grappes HPC, également appelées superordinateurs, sont des machines construites avec plusieurs cœurs. Leur cache, leur mécanisme de communication et l'organisation de leur disque sont différents. Ils utilisent un matériel puissant et solide, optimisé pour le débit et la vitesse. Les principales architectures d'un HPC incluent l'architecture SISD (Single Instruction, Single Data), l'architecture SIMD (Single Instruction, Multiple Data) et l'architecture MIMD (Multiple Instructionrchitecture, Multiple Data) à mémoire partagée. En plus nous avons aussi les architectures SIMD et MIMD à mémoire distribuée ([63]).
- ***GPU (Graphical Processing Unit)*** attire l'attention en raison de son architecture de traitement massivement parallèle qui accélère les performances des applications nécessitant de grandes quantités de calculs en virgule flottante. Au départ, les GPU étaient dédiés au traitement des calculs en deux et trois dimensions et à la transformation des primitives géométriques de l'unité centrale en pixels à ombrer et à cartographier sur l'écran [126, 8]. Avec l'émergence de nouveaux langages de programmation parallèle, tels que CUDA et OpenCL, les GPU sont désormais utilisés pour d'autres applications que les applications graphiques, ce qui permet d'utiliser des unités de traitement graphique à usage général (GPGPU).
- ***FPGA (Field-Programmable Gate Array)*** est un dispositif de circuit intégré. Il est composé d'un tableau de blocs logiques programmables. Ces blocs logiques et commutateurs peuvent être configurés de différentes manières pour former des circuits complexes à sorties multiples ([37]). De tels circuits contiennent plusieurs autres sous-circuits et ont plus d'une sortie. Ces blocs sont interconnectés via des interconnexions reconfigu-

rables, constituées de segments de fil et de commutateurs programmables. Comme les blocs logiques, la structure des commutateurs programmables peut être conçue de différentes manières. Les configurations de tous les composants du FPGA sont décrites à l'aide d'un langage de description matérielle (HDL).

La scalabilité horizontale

La scalabilité horizontale revient au regroupement des systèmes qui répartissent la charge de travail sur de nombreux serveurs ou machines [139]. Et donc, elle inclut les réseaux peer-to-peer (P2P), framework MapReduce et Spark. Ci-dessous, une brève description est donnée pour chacune de ces plateformes.

- ***Le réseau P2P*** est une architecture distribuée qui ne nécessite pas de serveur centralisé pour contrôler le transfert d'informations entre pairs. Ce type de réseau se caractérise par ses externalités, son faible coût de possession et de partage, et son anonymat [116]. Le schéma le plus couramment utilisé dans la plateforme P2P est l'interface de transmission de messages (MPI). MPI fournit des abstractions pour garantir la communication entre les pairs et maintient les processus en vie pendant le fonctionnement du système, éliminant ainsi la nécessité de lire à plusieurs reprises les données sur le disque. Cependant, son utilisation a diminué avec le développement de nouveaux frameworks, tels que Hadoop MapReduce.
- ***MapReduce*** est modèle de programmation parallèle, introduit pour la première fois par Google [52] en 2004, est conçu pour lire, traiter et écrire de grandes quantités de données. Ce modèle se compose de deux fonctions principales, les fonctions Map et Reduce. La fonction Map prend un enregistrement (ou morceau de données) en entrée et produit un ensemble de paires clé-valeur intermédiaires. La phase de Reduce traite ensuite ces paires clé-valeur intermédiaires, en fusionnant toutes les valeurs attribuées à la même clé pour produire l'ensemble des valeurs correspondant à cette clé. Il existe également deux fonctions optionnelles utilisées pour affiner l'exécution des tâches MapReduce : le Partitionneur et le Combinateur. Le Partitionneur divise les clés intermédiaires en fonction du nombre de tâches Reduce ou de reducer spécifiés par le développeur. Le Combinateur résume les résultats intermédiaires produits par chaque tâche Map, optimisant ainsi le transfert des données sur le réseau vers la tâche Reduce. Toutes ces fonctions peuvent être programmées par le développeur.
- ***Pregel*** s'agit d'un modèle de calcul de graphes bien connu pour traiter les grands graphes comportant jusqu'à des milliards de sommets et des trillions d'arêtes [146]. Les structures basées sur Pregel sont employées

pour gérer de grands graphes distribués grâce à l'utilisation des modèles Bulk Synchronous Parallel (BSP). Cette plateforme analytique de graphes nécessite que les développeurs créent des programmes qui nécessitent une optimisation soignée et sont difficiles à maintenir. Le processus commence par un graphique d'entrée généré par HDFS chargé en mémoire de la machine [48]. L'utilisateur doit ensuite définir une fonction définie par l'utilisateur (UDF, User-Defined Function) compute (msgs) pour chaque sommet v, avec msgs étant l'ensemble des messages entrants envoyés dans les super-étapes précédentes [8].

- ***Apache Spark*** est la nouvelle génération de traitement des Big Data. Destiné aux applications à forte intensité de données exécutées sur des clusters de [158], Spark prend en charge les tâches itératives et charge les ensembles de données dans la mémoire à la demande. Il présente trois aspects fondamentaux : *Resilient Distributed Dataset* (RDD), les opérations parallèles et les variables partagées. Les RDD sont une collection partagée d'objets répartis sur un ensemble de machines, qui peuvent être stockés dans la mémoire principale pour plusieurs travaux MapReduce parallèles et peuvent être récupérés en cas de perte. Le deuxième aspect concerne les opérations parallèles qui peuvent être appliquées aux RDD, telles que Collect, Reduce et Foreach. Le dernier aspect concerne les variables de diffusion et les accumulateurs. Les variables de diffusion sont des variables en lecture seule mises en cache sur différents nœuds, tandis que les accumulateurs sont des variables ajoutées à certaines opérations associées.

1.4 Framework Hadoop MapReduce

Le framework Hadoop MapReduce est l'implémentation open-source la plus populaire du framework MapReduce et est devenu un des outils les plus utilisé pour le traitement par lot car il répond au attente en terme efficacité à stocker et à analyser des données structurées ou non structurées. Nous incluons dans cette section une discussion sur les généralités de Apache Hadoop, paradigme MapReduce et système de fichiers distribué.

1.4.1 Apache Hadoop

Apache Hadoop est l'un des frameworks open source les plus populaires pour l'environnement en cluster qui permet un stockage fiable, évolutif et distribué. Il aide également au traitement de grands ensembles de données grâce aux modèles de programmation simples. Il gère des clusters d'ordinateurs constitués d'une à

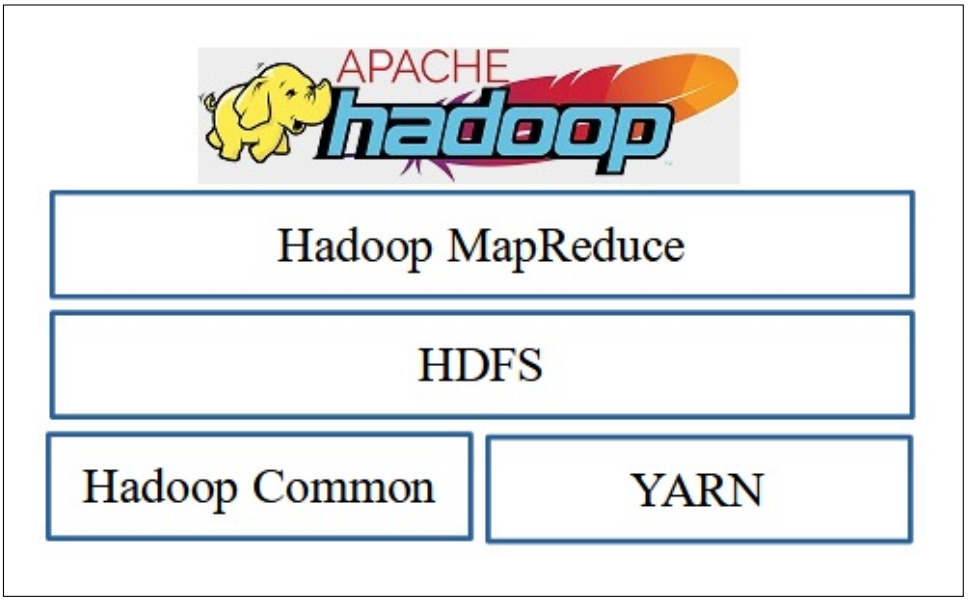

FIGURE 1.5 – Aperçu d'Apache Hadoop.

des milliers de machines, chacune offrant un calcul et un stockage locaux. La défaillance d'un nœud du cluster est automatiquement gérée en réattribuant sa tâche à un autre nœud [72].
Le projet Apache Hadoop comprend quatre modules fondamentaux : *Hadoop Common* ou *Hadoop Core* , *HDFS*, *YARN* et *Hadoop MapReduce*. La figure suivante schématise ces composants.

— *Hadoop Common* ou *Hadoop Core* fournit des services essentiels et des processus de base tels que l'abstraction du système d'exploitation et son système de fichiers. Il contient également les bibliothèques et scripts Java nécessaires pour démarrer Hadoop. Le package Hadoop Common fournit également le code source et la documentation, ainsi qu'une section de contribution qui comprend différents projets de la communauté Hadoop [72].
— *Hadoop Distributed File System (HDFS)* est un système de fichiers distribué développé par Apache Hadoop. Il garantit un stockage à haut débit et un accès aux données d'application sur les machines de la communauté, fournissant ainsi une bande passante globale très élevée à travers le cluster [72], haute tolérance aux pannes et prise en charge native de grands ensembles de données [35].
— *Hadoop Yet Another Resource Negotiator (YARN)* est une plateforme pour gérer les ressources du cluster et planifier des tâches. Il a été ajouté dans la version Hadoop 2.0 pour augmenter les capacités en résolvant la limite de

4000 nœuds et l'incapacité d'Hadoop à effectuer un partage de ressources à granularité fine entre plusieurs frameworks de calcul [152].
— *Hadoop MapReduce* est une implémentation du modèle de programmation MapReduce basé sur le système YARN pour le traitement parallèle à grand échelle [18].

1.4.2 Paradigme MapReduce

MapReduce est un paradigme de calcul proposé par Google en 2004 [52] qui permet le traitement parallèle de données à grande échelle. Il est facile à utiliser et exprime une grande variété de problèmes sous forme de calcul MapReduce de manière flexible, ce qui simplifie le traitement des données à grande échelle [52]. Le modèle de programmation MapReduce est un système permettant de traiter l'unité d'information de base dans une paire clé-valeur, où clé et valeur sont deux objets.

Ce modèle de calcul comporte trois étapes principales : Map, Shuffle et Reduce, comme schématisé dans la Figure 1.6.

Au cours de l'étape de mappage, le modèle lit chaque paire clé-valeur d'un fichier d'entrée donné. Ensuite, le Mapper traite une paire à la fois en appelant la fonction map définie par l'utilisateur, produit en sortie un multi-ensemble fini de nouvelles paires clé-valeur et détermine les ensembles de nouvelles paires par une fonction de hachage. Cela permet à différentes machines de traiter les entrées d'une map différente d'une manière simple et parallèle. L'étape de shuffle se produit automatiquement, elle est effectuée par Hadoop pour gérer l'échange des données intermédiaires entre la tâche map et la tâche reduce. On peut diviser cette étape en trois phases. La phase de tri produit l'ensemble des clés intermédiaires reçues du mapper en mémoire tampon dans un ordre particulier. Elle aide le reducer à savoir qu'une nouvelle tâche de reduce doit commencer lorsque la clé suivante dans les données d'entrée triées est différente de la précédente. La phase de fusion regroupe toutes les valeurs d'entrée intermédiaires ayant la même clé dans une liste et crée la paire (clé, liste de valeur) . La phase de partitionnement détermine dans quel reducer une paire (clé, liste de valeur) sera envoyée. Elle est basée sur une fonction de hachage qui associe et envoie une paire à un reducer. Dans la dernière étape, les reducers qui reçoivent les paires triées (clé, liste de valeurs) peuvent être exécutés simultanément tout en opérant sur différentes clés, ils réduisent un ensemble de valeurs intermédiaires qui partagent une clé à un nouvel ensemble plus petit de valeurs.

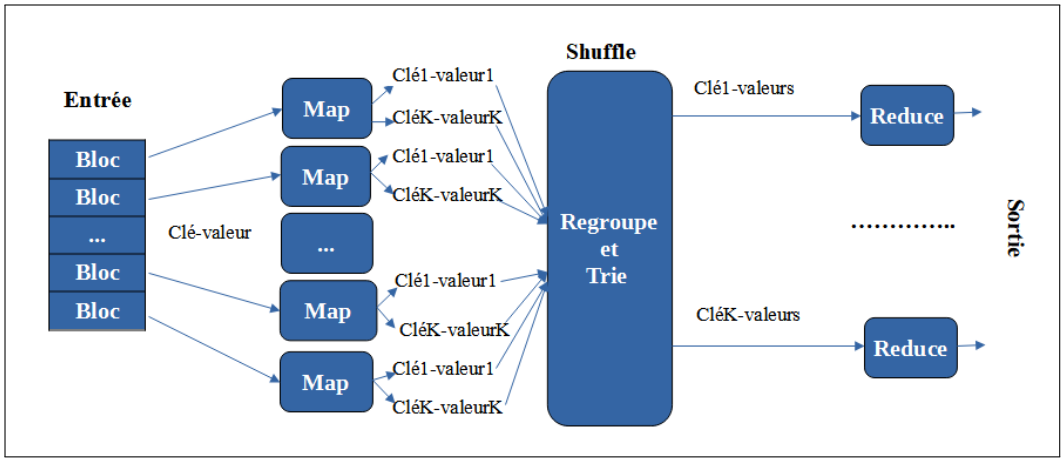

FIGURE 1.6 – Principales étapes de MapReduce.

1.4.3 Système de fichier distribué de Hadoop Apache

Système de fichier distribué de Hadoop Apache (HDFS, Hadoop Distributed File System) a été développé en utilisant la conception de systèmes de fichiers distribués. Il s'agit de l'un des composants basiques du framework Hadoop Apache, et plus précisément de son système de stockage. Il est exécuté sur du matériel de base (nœud de calcul). Contrairement à d'autres systèmes distribués, HDFS est hautement tolérant aux pannes et conçu à l'aide de matériel à faible coût.

HDFS contient de très grandes quantités de données et permet un accès plus facile. Pour stocker des données aussi volumineuses, les fichiers sont stockés sur plusieurs machines. Ces fichiers sont stockés de manière redondante afin de sauver le système d'éventuelles pertes de données en cas de panne. HDFS rend également les applications accessibles au traitement parallèle. La Figure 1.7 présente l'architecture du système de fichier distribué de Hadoop Apache.

2. https ://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

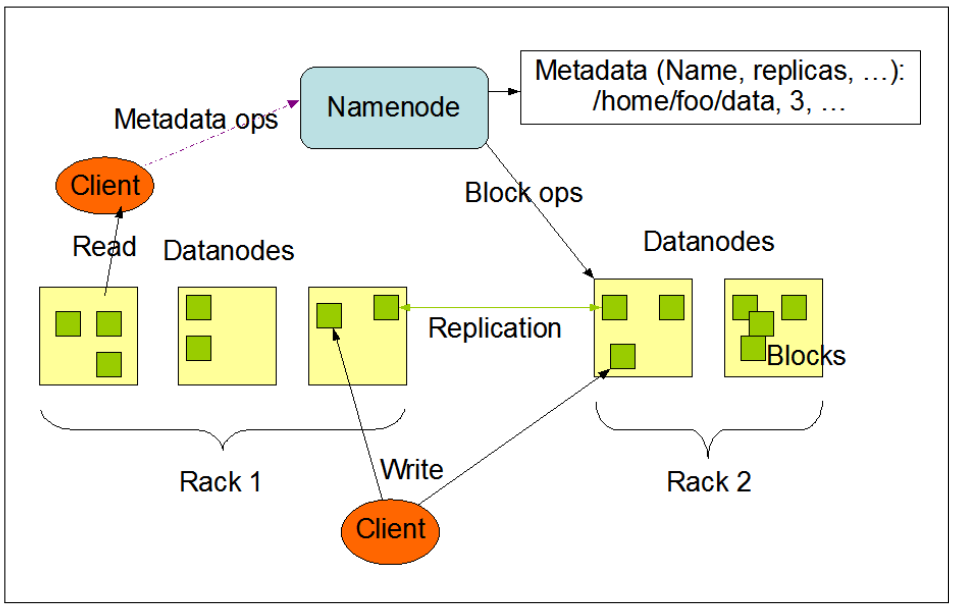

FIGURE 1.7 – Architecture du système de fichier distribué.[2]

Chapitre 2

Modèle de calcul massivement parallèle en MapReduce

Ce chapitre présente un modèle simple de traitement parallèle conçu pour analyser le coût de communication dans les algorithmes de traitement parallèle des données, connu sous le nom de modèle de Calcul Massivement Parallèle en MapReduce (CMP-MR) [95, 3]. Ce modèle se concentre sur le coût de communication entrant, par machine, et est une simplification du modèle BSP (Bulk Synchronous Processing), introduit par Valiant [150], dans lequel seul le coût de communication est modélisé et le coût de calcul est ignoré. Il est considéré comme une abstraction de l'architecture sans partage (Shared-Nothing architecture) [53], où le coût de communication lors de l'échange de données peut dominer le temps d'exécution. Cette architecture est adoptée aujourd'hui par les grands systèmes de traitement de données tels que le système COSMOS de Microsoft [99], où les tâches réparties sur le système de fichiers distribué s'exécutent souvent sur plus de 10 000 nœuds [131]. Afin de présenter le modèle CMP-MR de manière simple et détaillée, ce chapitre est basé sur les références suivantes Koutris *et al.* [99], Leskovec *et al.* [110], et Afrati *et al.* [1, 2, 3].

2.1 Aperçu Modèle de Calcul Massivement Parallèle

Dans cette section, on présente le modèle de Calcul Massivement Parallèle (CMP) avec ses trois paramètres, la manière dont les données sont initialement distribuées et discute la performance du modèle CMP.

2.1.1 Formalisme de Calcul Massivement Parallèle

Le Calcul Massivement Parallèle (CMP) est un modèle de traitement qui permet à des centaines ou des milliers de nœuds de calcul de fonctionner en parallèle sur des parties distinctes d'une tâche de calcul [99].
Un cluster contenant p machines, chacune avec son propre processeur et sa mémoire locale. Tout au long de ce chapitre, on utilisera le terme *processeurs* pour désigner les machines ou les serveurs d'un cluster. Les p processeurs sont connectés par des canaux privés via un réseau complet, ce qui signifie que chaque processeur peut envoyer ou recevoir des données de n'importe quel autre processeur du cluster. Le calcul se déroule par itérations, ou étapes, où chaque itération se compose de deux phases : le calcul et la communication. Dans la *phase de calcul*, les processeurs effectuent des calculs locaux en opérant uniquement sur les données disponibles localement. Dans la *phase de communication*, les processeurs échangent les données en communiquant avec tous les autres processeurs du cluster. La Figure 2.1 illustre le modèle de Calcul Massivement Parallèle.

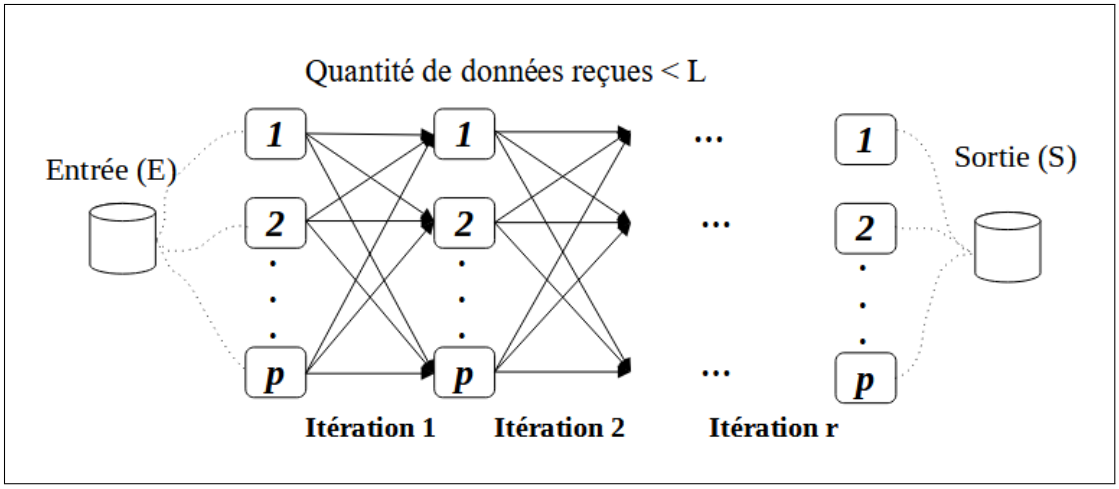

FIGURE 2.1 – Modèle de Calcul Massivement Parallèle [99].

les notations utilisées sont les suivantes :

- E : la taille de l'entrée.
- L : un ensemble fini d'états.
- p : le nombre de processeurs ou de machines dans le cluster.
- r : le nombre d'itérations ou étapes de l'algorithme.
- S : la taille des données de sortie.

2.1.2 Répartition initiale des données

Les *données d'entrée* sont initialement réparties arbitrairement entre les p processeurs. On peut utiliser une fonction de hachage pour "équilibrer la charge" des données initiales sur tous les processeurs. E est mesurée en nombre d'éléments ou de tuples de données ; dans certains cas, l'entrée est mesurée en nombre de bits plutôt qu'en nombre de tuples. Un algorithme doit pouvoir calculer correctement la réponse, quelle que soit la façon dont les données d'entrée sont initialement partitionnées entre les p processeurs. Les données de sortie doivent être présentes dans l'union des mémoires des p processeurs à la fin du calcul.

Dans le traitement parallèle des données, il est nécessaire de mettre une hypothèse sur la manière dont les données sont initialement partitionnées, car la taille de sortie du traitement des données est souvent plus grande que la taille de l'entrée. Pour s'adapter à cela, les données de sortie sont stockées de manière distributive, plutôt qu'un seul processeur ne détenant pas le résultat entier.

Aucune restriction ne sera imposée sur les capacités de calcul des processeurs ou le schéma d'échange de données pendant la phase de communication, et les paramètres du modèle consistent en le nombre de processeurs, le nombre d'itérations et le coût de communication par processeur.

2.1.3 Paramètres du modèle CMP

Les paramètres considérés dans le modèle CMP par Koutris *et al.* [99] sont les suivants :

- Le *nombre de processeurs* p, qui est une entrée fixe pour l'algorithme. Les p processeurs ne communiquent que par réseau. En pratique, certains systèmes distribués permettent que le nombre de processeurs de changer dynamiquement. Dans les modèles abstraits de MapReduce [95], p correspond au nombre maximal de mappers pendant la phase *map*, ou au nombre maximal de reducers pendant la phase *reduce*.
- Le *nombre d'itérations* r, qui est un paramètre important dans la conception d'algorithmes distribués, car il mesure le nombre de synchronisations entre les processeurs nécessaires à la terminaison de l'algorithme. Chaque étape de synchronisation supplémentaire a un coût, car elle nécessite que tous les processeurs attendent le processeur le plus lent, implique un coût physique associé au suivi de la progression de chaque processeur et nécessite une attention particulière pour s'assurer que les données sont réparties uniformément entre les processeurs. Par conséquent, minimiser le nombre d'itérations r est un objectif important lors de la conception d'algorithmes.

- Le *Coût de communication/Charge L* défini comme étant la quantité maximale de données reçues par un processeur pendant une itération. Une métrique associée est le coût total de communication, qui est la quantité totale de données échangées par tous les processeurs durant toutes les itérations. Formellement, on désigne par $L_u^{(k)}$ la quantité de données que le processeur $u = 1, \ldots, p$ reçoit durant l'itération $k = 0, 1, \ldots, r$. La charge maximale est définie comme suit :

$$L \stackrel{\text{def}}{=} \max_{u=1}^{p} \max_{k=1}^{r} L_u^{(k)}$$

La *communication totale* dans un algorithme parallèle est la somme de toutes les données échangées entre les processus durant son exécution :

$$C \stackrel{\text{def}}{=} \sum_{u=1}^{p} \sum_{k=1}^{r} L_u^{(k)}$$

Noté que $E \leq C \leq r \cdot p \cdot L$.

En principe, les algorithmes CMP doivent être conçus de manière à minimiser le temps d'exécution total. Pour ce faire, il faut optimiser à la fois le temps de calcul local et le nombre d'itérations. Le temps de calcul local dépend du coût de communication L et des performances physiques du processeur. Le nombre d'itérations r est plus difficile à optimiser, car il n'existe pas de formule permettant de déterminer le temps d'exécution total en fonction de ces deux paramètres. Par conséquent, le modèle CMP doit être considéré comme un problème d'optimisation multiobjectifs afin de minimiser correctement le temps d'exécution total.

2.1.4 Performance du modèle CMP

Pour évaluer la performance du modèle CMP, on se base sur la mesure du parallélisme à l'aide de deux paramètres simples, l'*accélération* et la *scalabilité* [53]. Pour une entrée donnée, l'accélération est le taux d'augmentation des performances de l'algorithme lorsque le nombre de processeurs est augmenté. Une accélération idéale est linéaire. Cependant, en pratique, il est plus commun de constater une accélération sous-linéaire. La scalabilité est la vitesse de l'algorithme lorsque les données d'entrée et le nombre de processeurs augmentent proportionnellement. Une scalabilité idéale est constante, ce qui signifie que si

la taille des données double, la même performance peut être obtenue en doublant le nombre de processeurs. Dans la réalité, la scalabilité est généralement sous-constante.

L'accélération et la scalabilité, d'un modèle CMP, sont déterminés par la charge de l'algorithme lors de l'utilisation de différents nombres de processeurs et le temps nécessaire pour le traitement local à chaque processeur. La charge de l'algorithme, *L(E, p)*, est calculée en utilisant un nombre fixe d'itérations et une taille d'entrée E lors de l'utilisation de p processeurs. En supposant que l'algorithme local prend du temps linéaire par rapport à la quantité de données reçues au cours d'une itérations, c'est-à-dire que le temps est égal à *O(L)*. L'accélération est donnée par [99] :

$$\begin{aligned} f(p) &= \frac{L(E,1)}{L(E,p)} \\ &= \frac{c}{L(E,p)} \end{aligned}$$

pour une certaine constante c indépendante de p, tandis que la scalabilité est donnée par [99] :

$$\begin{aligned} g(t) &= \frac{L(E,p)}{L(t\times E, t\times p)} \\ &= \frac{c'}{L(t\times E, t\times p)} \end{aligned}$$

2.2 Modèle de calcul MapReduce comme modèle CMP : CMP-MR

Cette section présente un modèle permettant de mesurer la qualité des algorithmes MapReduce, notamment le compromis entre le parallélisme et le coût de la communication dans un calcul MapReduce. Il est à noter que la communication entre les tâches Map et Reduce est souvent la partie la plus longue de ces tâches. Le modèle fournit une méthode générique pour déterminer les bornes inférieures du coût de communication en fonction du nombre maximal d'entrées pouvant être affectées à un reducer.

2.2.1 Analyse de coût de communication

Le coût de communication d'un algorithme est la taille totale des entrées de toutes les tâches qui composent l'algorithme. Ce coût est utilisé pour mesu-

rer l'efficacité de l'algorithme et ne tient généralement pas compte du temps nécessaire à l'exécution de chaque tâche. Bien qu'il existe des cas où le temps d'exécution des tâches soit plus significatif, ceux-ci sont rares dans la pratique [3].

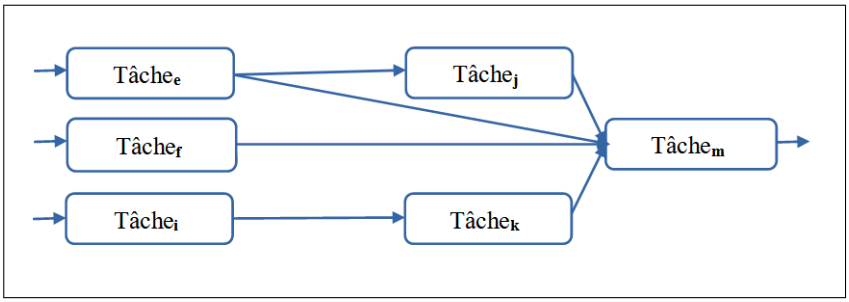

FIGURE 2.2 – Exemple d'un flux de travail entre plusieurs tâches MapReduce.

L'importance du coût de la communication est liée à la vitesse typique de l'interconnexion pour un cluster informatique, qui est relativement lente par rapport à la vitesse à laquelle un processeur exécute des instructions. De plus, lorsque plusieurs nœuds de calcul communiquent en même temps, il peut y avoir une concurrence pour l'interconnexion, ce qui signifie que le nœud de calcul peut effectuer beaucoup de travail sur un élément d'entrée reçu pendant qu'il est en cours de livraison. Même si la tâche est exécutée sur un nœud de calcul qui possède une copie du ou des blocs sur lesquels la tâche fonctionne, le bloc est généralement stocké sur disque, ce qui peut augmenter le temps nécessaire pour déplacer les données en mémoire principale.
Le coût de communication est souvent le coût dominant dans les calculs distribués. Par conséquent, lors de la prise en compte du coût d'un algorithme, la taille d'entrée est prise en compte, par opposition à la taille de sortie. C'est parce que la taille de sortie d'une tâche peut être incluse dans la taille d'entrée de la tâche réceptrice, et que la taille de sortie de l'algorithme global est généralement beaucoup plus petite que la taille d'entrée ou les données intermédiaires produites [110].

2.2.2 Communication et parallélisme pour MapReduce

Le modèle proposé par Afrati *et al.* [3] offre un aspect formel du compromis inhérent entre le coût de communication et le degré de parallélisme, et aide à analyser les problèmes qui se prêtent à une solution MapReduce. Ce modèle se base sur deux paramètres qui caractérisent les algorithmes MapReduce. La *taille du reducer* q et le *taux de réplication* r.

Un reducer est un élément qui combine une clé-de-reduce générée par les mappers avec la liste de valeurs qui lui est associée, et qui est transmise à un reduce travailleur (ou reduce-worker, qui fait le traitement). La *taille d'un reducer* est la borne supérieure du nombre de valeurs qui sont autorisées à apparaître dans la liste associée à une seule clé. Afin d'augmenter le parallélisme, les tailles de reducer doivent être réduites. Cela peut nécessiter une refonte de la définition d'une "clé" dans le but de créer un plus grand nombre de reducers plus petits, et donc un plus grand parallélisme si suffisamment de nœuds de calcul sont disponibles.
Le choix d'une taille de reducer suffisamment petite pour être sûr que le calcul associé à un seul reducer puisse être exécuté entièrement dans la mémoire principale du nœud de calcul où se trouve sa tâche Reduce permet de réduire considérablement le temps d'exécution. Cela évite d'avoir à déplacer les données à plusieurs reprises entre la mémoire principale et le disque, quel que soit le calcul effectué par les reducers. Le coût de calcul total des reducers est la somme des coûts de calcul pour le traitement de toutes les valeurs associées à chaque clé.

Le *taux de réplication* d'un algorithme MapReduce est le nombre moyen de paires clé-valeur créées à partir de chaque entrée afin de mesurer la quantité de communication entre la phase map et la phase reduce. Souvent, mais pas toujours, le coût de la communication est le coût dominant d'un algorithme MapReduce.

Formellement, supposons que nous ayons p reducers et que les entrées $q_i \leq q$ soient affectées au $i^{\text{ème}}$ reducer. Soit $|I|$ le nombre total de différentes entrées, alors le taux de réplication est donné par l'expression $r = \sum_{i=1}^{p} q_i/|I|$ [3, 110].

2.2.3 Schémas de Mappage d'un algorithme MapReduce

Pour chaque problème soluble par un algorithme MapReduce, il y a un ensemble d'entrées et de sorties qui sont liées entre elles par une relation plusieurs-plusieurs décrivant quelles entrées sont nécessaires pour produire quelles sorties. Chacun de ces algorithmes doit avoir un schéma de mappage, qui exprime comment les sorties sont produites par les différents reducers utilisés par l'algorithme. En d'autres termes, Le schéma de mappage pour un problème donné avec une taille de reducer q consiste à affecter les entrées à un ou plusieurs reducers, tels qu'aucun reducer ne soit attribué plus que q entrées et chaque sortie du problème soit couverte par un reducer qui aura toutes les entrées associées à cette sortie.

La possibilité de créer un schéma de mappage pour n'importe quelle taille de reducer est ce qui différencie les problèmes qui peuvent être résolus par un

seul travail MapReduce de ceux qui ne peuvent pas l'être.

Le fait qu'une sortie soit liée à une certaine entrée signifie que chaque entrée traitée par la tâche Map générera au moins une paire clé-valeur pour être utilisée lors du calcul de cette sortie. La valeur peut ne pas correspondre exactement à l'entrée, mais elle sera dérivée de celle-ci. Il est important de noter que chaque entrée et sortie liée aura une paire clé-valeur unique qui doit être communiquée. Il n'est pas nécessaire d'avoir plus d'une paire clé-valeur pour une entrée et une sortie données, car l'entrée peut être transmise au reducer sans transformation et toutes les transformations qui ont été appliquées à l'entrée par la fonction Map peuvent être appliquées par la fonction Reduce au reducer pour cette sortie [3, 110].

2.2.4 Absence de certaines entrées

Un algorithme pour un problème qui peut ne pas fournir toutes ses sorties nécessite un schéma de mappage afin d'assurer que toutes les sorties peuvent être produites. En effet, toutes les entrées, ou tout sous-ensemble d'entre elles, peuvent être présentes, et sans schéma de mappage, toute sortie qui est liée à ces entrées, mais qui n'est pas directement couverte par le reducer de l'algorithme ne sera pas produite.

La seule conséquence de l'absence de certaines entrées est que nous devons peut-être modifier la taille du reducer q lors du choix d'un algorithme parmi la famille des algorithmes possibles. Plus précisément, lorsque la taille du reducer est choisie de manière que l'entrée tienne dans la mémoire principale, il peut être nécessaire de l'augmenter pour tenir compte de l'absence de certaines entrées [110].

2.2.5 Borne inférieure du taux de réplication

Dans cette section, on présente la méthode qui nous sert à dériver toutes les bornes inférieures utilisées dans nos travaux. Alors que les bornes supérieures de r pour tous les problèmes sont dérivés à l'aide d'algorithmes constructifs, cette technique générique, introduit par Afrati, Ullman *et al.* [3, 110], permet de dériver les bornes inférieures en quatre étapes.

1. **Calcul de $g(q)$:** Prouver une borne supérieure, $g(q)$, sur le nombre de sorties qu'un reducer peut produire, si q est le nombre d'entrées reçues.
2. **Calcul de $|I|$ et $|O|$:** Déterminer le nombre total d'entrées $|I|$ et de sorties $|O|$ produites par le problème.
3. **L'inégalité :** Supposer qu'un cluster de p reducers, et que le $i^{\text{ème}}$ reducer reçu $q_i < q$ entrées. Observer que $\sum_{i=1}^{p} g(q_i)$ ne doit pas être inférieur au

nombre de sorties $|O|$ calculées à l'étape (2).

$$\sum_{i=1}^{p} g(q_i) \geq |O| \tag{2.1}$$

4. **Taux de réplication :** Manipuler l'inégalité de l'équation (2.1) pour obtenir une borne inférieure sur le taux de réplication, qui est $\sum_{i=1}^{p} q_i = |I|$. Noter que la dernière étape ci-dessus peut nécessiter une manipulation astucieuse pour factoriser le taux de réplication. Tout d'abord, faites en sorte d'isoler un seul facteur q_i de $g(q_i)$:

$$\sum_{i=1}^{p} g(q_i) \geq |O| \Rightarrow \sum_{i=1}^{p} q_i \frac{g(q_i)}{q_i} \geq |O| \tag{2.2}$$

En supposant que $\frac{g(q_i)}{q_i}$ est monotone et croissante en q_i, on peut utiliser le fait que $\forall q_i : q_i \leq q$ pour obtenir de l'équation (2.2) :

$$\sum_{i=1}^{p} q_i \frac{g(q)}{q} \geq |O| \tag{2.3}$$

Maintenant, on divise les deux côtés de l'équation (2.3) par la taille de l'entrée, pour obtenir une formule avec le taux de réplication à gauche :

$$r = \frac{\sum_{i=1}^{p} q_i}{|I|} \geq \frac{q \times |O|}{g(q) \times |I|} \tag{2.4}$$

L'équation 2.4 fournit une borne inférieure pour r. Dans l'exemple 4, on va analyser le coût de communication du problème de la multiplication de deux matrices dans une seule itération MapReduce.

Exemple 4. *Pour expliquer le compromis entre r et q, on va examiner un problème connu sous le nom de multiplication matricielle. Dans ce problème, on se donne deux matrices carrées A et B de la taille $n \times n$ pour calculer leur produit C, la Figure 2.3 illustre l'opération de produit de deux matrices.*

Dans ce qui suit, on présente un schéma de mappage pour calculer la multiplication de deux matrices en une seule itération MapReduce, comme l'illustre la Figure 2.4. Tout d'abord, chaque sortie dépend de plusieurs entrées, et non seulement de deux ou trois. En particulier, la sortie c_{ij} dépend d'une ligne entière de A et d'une colonne

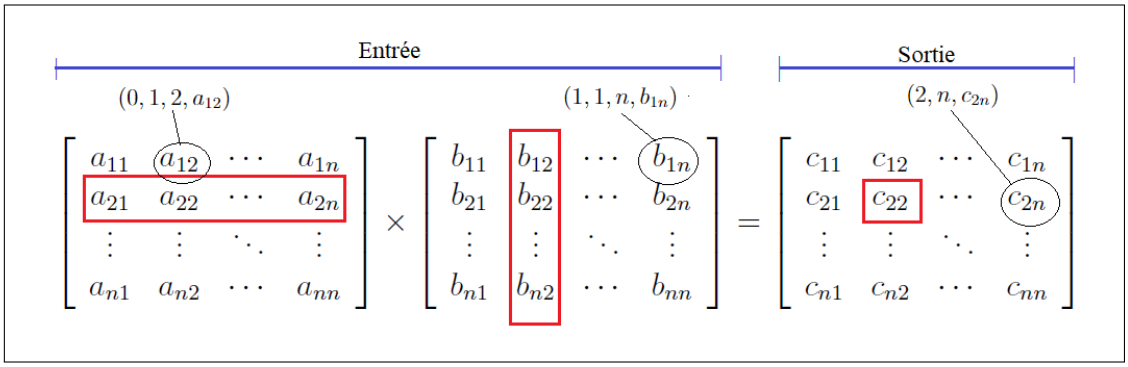

FIGURE 2.3 – Multiplication de deux matrices.

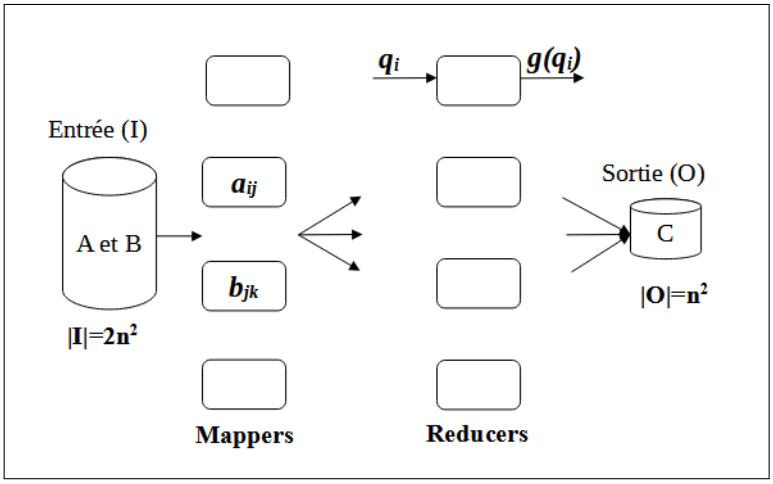

FIGURE 2.4 – Multiplication de deux matrices dans une seule itération MapReduce.

entière de B, c'est-à-dire, 2n entrées, comme suit :

$$\begin{aligned} c_{ij} &= a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} \\ &= \sum_{k=1}^{n} a_{ik}b_{kj} \end{aligned}$$

Par ailleurs, nous détaillons les fonctions Map et Reduce. On suppose que la phase de pré-traitement stocke tous les éléments de deux matrices A et B dans un fichier texte. Un élément e_{ij} d'une matrice d'entrée sera représenté par un 4-tuple comme suit : $(indice(e_{ij}), i, j, e_{ij})$, tel que $indice(e_{ij})$ représente l'indice d'ordre de la matrice contenant l'élément e_{ij} (par exemple un bit, 0 pour A et 1 pour B), i et j respectivement la ligne et la colonne de e_{ij} pour définir sa position dans la matrice.

La fonction Map produit un ensemble de paires (clé, valeur) à partir de chaque enregistrement d'entrée. En d'autres termes, si l'élément d'entrée e_{ij} de la matrice A alors clé $= (i,k)$ et $valeur = (0, j, e_{ij})$, sinon clé $= (k, j)$ et $valeur = (1, i, e_{ij})$ pour $k = 1, 2, \cdots, n$.

La fonction Reduce reçoit une liste des valeurs de $2n$ éléments associée à la clé (i, j), tel que n éléments de A et n de B. Elle divise les éléments par rapport à l'indice des matrices auxquels ils appartiennent en deux ensembles triés. La foncrion Reduce calcule le produit des éléments de deux ensembles qui ont le même ordre, et après la somme pour calculer la valeur $c_{i,j}$ de la matrice C, et sera stockée sous la forme $(i, j, c_{i,j})$.

La borne inférieure de taux de réplication du produit matriciel en MapReduce dans une seule itération en découlant la procédure générique précédemment présentée, on a :

1. $g(q) = \frac{q}{2n}$ *et* $q = 2n$
2. $|I| = 2n^2$ *et* $|O| = n^2$
3. *La borne inférieure pour r :*

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

Alors

$$r \geq \frac{q \times n^2}{\frac{q}{2n} \times 2n^2}$$

Donc

$$r \geq n$$

Pour conclure, tous les algorithmes MapReduce doivent avoir un taux de réplication minimal, égal à la borne inférieure r, pour garantir une réponse complète. Tout algorithme MapReduce qui présente un taux de réplication inférieur à cette borne ne fournira pas une réponse complète.

Résumé

Ce chapitre a été principalement consacré à la présentation du modèle de calcul massivement parallèle en MapReduce (CMP-MR) qui est une partie très importante de toutes nos contributions. Ce modèle est essentiellement développé par d'Afrati, Ullman *et al.*, qui base sur deux paramètres, le *coût de la communication* et la *taille du reducer*. Ce modèle fournit une méthode générique pour déterminer les bornes inférieures du coût de communication en fonction du

nombre maximal d'entrées pouvant être affectées à un reducer, taille du reducer, pour mesurer la qualité des algorithmes MapReduce, notamment le compromis entre le parallélisme et le coût de la communication dans un calcul MapReduce.

Chapitre 3

Composition massivement parallèle des transducteurs à états finis pondérés

Les transducteurs à états finis pondérés (WFST, Weighted finite states transducers) ont été largement utilisées dans différentes applications telles que le traitement d'images numériques [48], la reconnaissance vocale [85], la traduction automatique à grande échelle [32], la cryptographie [143] et plus récemment, en bio-informatique [133] où les noyaux rationnels par paires sont calculés pour la prédiction du réseau métabolique et de nombreuses autres applications [29, 88, 75]. Les WFSTs sont des machines à états finis dans lesquelles chaque transition, en plus de son symbole d'entrée, est augmentée d'un symbole de sortie d'un nouvel alphabet possible et comporte un élément de pondération d'un semi-anneau. Les WFSTs peuvent être utilisés pour définir une correspondance entre deux langues. Dans certains cas, les pondérations représentent l'incertitude ou la variabilité de l'information. Par exemple, les WFSTs introduites dans la reconnaissance vocale [121] sont utilisées pour attribuer différentes prononciations au même mot avec des rangs ou des probabilités différentes. En classification et en apprentissage, les méthodes du noyau, comme les machines à vecteurs de support, sont largement utilisées [115]. Dans [119], Mohri *et al.* ont introduit la théorie du noyau rationnel. Le calcul d'un noyau rationnel peut être effectué efficacement en utilisant un algorithme rapide pour la composition des WFSTs. Le calcul de la composition des WFSTs est principalement basé sur la composition standard des machines à états finis non pondérés. Il prend en entrée deux ou plusieurs WFSTs $(\mathcal{T}_i)_{1\leq i\leq n}$, et il produit en sortie un WFST $\mathcal{T}$, le résultat de la composition de tous les WFST d'entrée, de telle sorte que l'alphabet d'entrée de $\mathcal{T}_{i+1}$ coïncide avec l'alphabet de sortie de $\mathcal{T}_i$. La complexité temporelle pour

calculer cette opération est en $O(\prod_{i=1}^{n} |\mathcal{T}_i|)$ telle que $|\mathcal{T}_i|$ représente le nombre d'états dans $\mathcal{T}_i$ [119, 154] (c'est-à-dire, si on considère n WFSTs en l'entrée, chacun ayant m états, le WFST résultant peut atteindre la limite exponentielle de m^n états).

3.1 Composition des transducteurs à états finis pondérés

La composition est une opération fondamentale utilisée pour générer des transducteurs à états finis pondérés complexes à partir de machines plus petites. Soit $\mathbb{K}$ un semi-anneau commutatif et soient $\mathcal{T}_1$ et $\mathcal{T}_2$ deux WFSTs tels que l'alphabet d'entrée de $\mathcal{T}_2$ coïncide avec l'alphabet de sortie de $\mathcal{T}_1$. Supposons que la somme infinie $\bigoplus_z \mathcal{T}_1(x,z) \times \mathcal{T}_2(z,y)$ est bien définie dans $\mathbb{K}$ pour tout $(x,y) \in A^* \times B^*$.
La composition de $\mathcal{T}_1$ et $\mathcal{T}_2$ produit un WFST, noté $\mathcal{T}_1 \circ \mathcal{T}_2$, et est définie pour tout x, y par [119] :

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

Allauzen et Mohri ont présenté un algorithme efficace pour la composition des WFSTs [9]. Les états de la composition $\mathcal{T}_1 \circ \mathcal{T}_2$ de deux WFSTs $\mathcal{T}_1$ et $\mathcal{T}_2$ sont identifiés par des paires d'un état de $\mathcal{T}_1$ et un état de $\mathcal{T}_2$. Son état initial est la paire des états initiaux de $\mathcal{T}_1$ et $\mathcal{T}_2$ ayant comme poids d'entrée la somme des poids d'entrée dans le semi-anneau des deux états initiaux. Ses états finaux sont des paires d'un état final de $\mathcal{T}_1$ et d'un état final de $\mathcal{T}_2$ ayant comme poids de sortie la somme des poids de sortie dans le semi-anneau des deux états finaux. Soit $E_{\mathcal{T}}$ l'ensemble des transitions d'un WFST $\mathcal{T}$. La règle suivante spécifie comment dériver une transition de $\mathcal{T}_1 \circ \mathcal{T}_2$ à partir des transitions appropriées de $\mathcal{T}_1$ et $\mathcal{T}_2$:

$$(q_1, a, b, w_1, q_2) \in E_{\mathcal{T}_1} \text{ et } (q_1', b, c, w_2, q_2') \in E_{\mathcal{T}_2} \Longrightarrow ((q_1, q_1'), a, c, w_1 \otimes w_2, (q_2, q_2')) \in E_{\mathcal{T}_1 \circ \mathcal{T}_2}$$

Dans le pire des cas, toutes les transitions sortantes d'un état q de $\mathcal{T}_1$ correspondent à toutes celles de $\mathcal{T}_2$ sortant de l'état q', ainsi la complexité temporelle et spatiale de la composition est en $O(|\mathcal{T}_1||\mathcal{T}_2|)$ [121].

Exemple 5. *La Figure 3.1 illustre l'opération de composition entre deux WFSTs $\mathcal{T}_1$ et $\mathcal{T}_2$.*

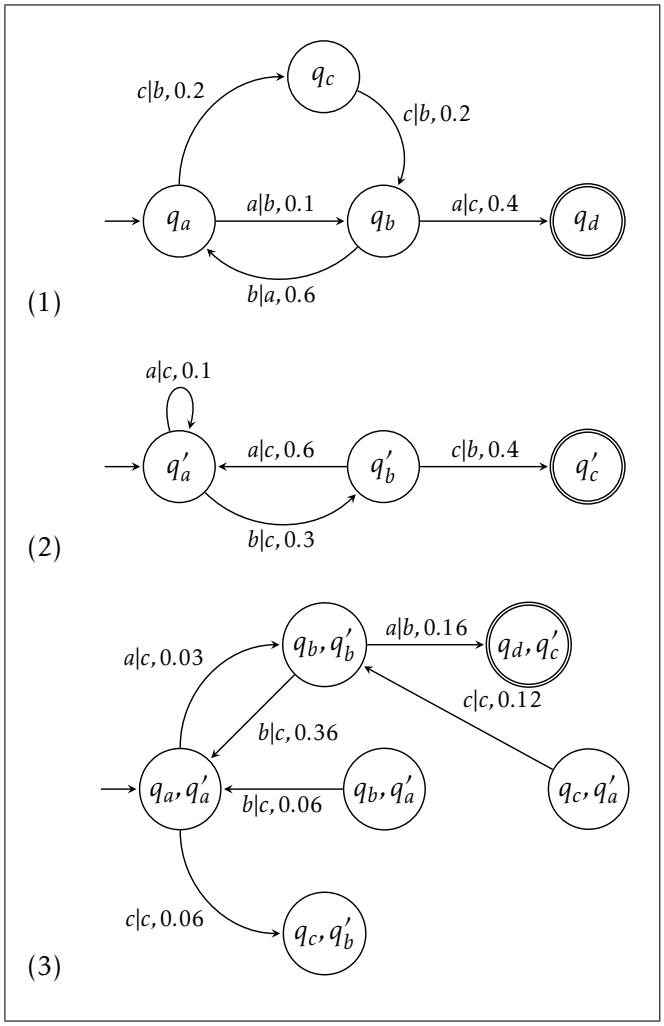

FIGURE 3.1 – (1) WFST $\mathcal{T}_1$ et (2) WFST $\mathcal{T}_2$ sur le semi-anneau tropical. (3) résultat de la composition de $\mathcal{T}_1$ et $\mathcal{T}_2$.

3.2 Composition en MapReduce des transducteurs à états finis pondérés

Dans cette section, nous introduisons le premier algorithme massivement parallèle en MapReduce pour le calcul de la composition de plusieurs WFSTs. Nous présentons trois méthodes pour effectuer la composition des WFSTs en MapReduce. Nous étudions également le coût de communication optimal selon le modèle introduit par Afrati *et al.* [3] et analysons le taux de réplication des méthodes proposées afin de montrer leurs efficacités.

3.2.1 Algorithme générique en MapReduce pour la composition des WFSTs

Dans ce qui suit, nous présentons une approche générique pour calculer la composition des WFSTs en une seule itération MapReduce. Par ailleurs, nous définissons et détaillons respectivement les fonctions Map et Reduce. La phase de pré-traitement de notre algorithme stocke dans un fichier texte toutes les transitions des WFSTs $(\mathcal{T}_i)_{1\leq i\leq n}$, chacun ayant m états. Une transition t d'un WFST $\mathcal{T}_i$ sera représentée par un 4-tuple comme suit : (t,type(s[t]),type(d[t]),index(t)), tel que type() représente une fonction qui associe un état à un élément de l'ensemble {i (initial), f (final), if (initial et final), s (simple)} et $index(t) = i$ représente l'indice d'ordre du WFST $\mathcal{T}_i$ contenant la transition t.

Algorithme 1 : Map

Entrée : paire $\langle clé, valeur\rangle$, où *clé* représente un identifiant d'instance arbitraire et $valeur$ est la forme de 4-tuple associée à la transition t.

Sortie : collection de paires $\langle k, t\rangle$, où k est une clé associée à la transition t

```
// créer la transition t à partir de la valeur d'entrée
1   t ← getTransitionFrom(valeur);
// générer l'ensemble des clés associées à la transition t
2   setOfKeys ← getTransitionSetKeys(t);
// répliquer la transition t en l'envoyant aux reducers
par la fonction Emit()
3 foreach k in setOfKeys do
4     Emit(k,t) ;
```

La fonction Map produit un ensemble de paires (*clé*, *valeur*) à partir de chaque enregistrement d'entrée. En d'autres termes, la transition d'entrée est répliquée et associée à toutes les clés générées à partir de celle-ci en fonction de la méthode de mappage (ligne 2 de l'algorithme 1). Les sorties intermédiaires ajoutent un facteur de taux de réplication dans le coût de l'algorithme MapReduce. Les sorties de la fonction Map sont introduites dans l'étape Shuffle. Dans notre implémentation, nous produisons l'étape Shuffle automatiquement.

Algorithme 2 : Reduce

Entrée : paire $\langle key, values \rangle$, où $values$ est un ensemble de transitions ayant la même clé clé.

Sortie : Transitions du résultat de la composition.

```
// regrouper les transitions en fonction de l'indice de
leur WFST sous forme de liste d'ensembles TransList
```

1 **foreach** *transition t in values* **do**

2 *add t to TransList[index(t)]*;

```
// joindre les ensembles de transitions TransList[i] de
TransList en appliquant le produit cartésien
```

3 JoinedTransList $= \bigotimes_{i=1}^{n} TransList[i]$;

```
// calculer la composition à partir de la liste
JoinedTransList
```

4 **foreach** *element* $(t_1, \cdots, t_n)$ *in JoinedTransList* **do**

5 $t = t_1 \circ \cdots \circ t_n$;

6 Emit(*null*,*t*) ;

La fonction Reduce effectue la composition des différentes listes de transitions produites à partir de l'étape Shuffle comme suit : une transition est sélectionnée à partir d'un WFST de telle sorte que le symbole d'entrée de WFST $\mathcal{T}_{i+1}$ coïncide avec le symbole de sortie du WFST $\mathcal{T}_i$ précédent. De plus, la fonction Reduce regroupe les transitions par rapport à l'indice des WFSTs auxquels elles appartiennent dans une liste d'ensembles (ligne 2 de l'algorithme 2), et calcule le produit cartésien de ces ensembles (ligne 3 de l'algorithme 2).
Dans ce qui suit, nous discuterons le coût de communication de l'algorithme proposé selon le modèle d'Afrati *et al.* [3].

3.2.2 Analyse du coût de communication

Le modèle de coût de communication introduit par Afrati *et al.* [3] offre un cadre complet pour analyser les problèmes et optimiser les performances de calcul sur toute architecture distribuée en étudiant explicitement le compromis entre le coût de communication et le degré de parallélisme. En appliquant ce modèle sur le framework MapReduce, nous pouvons déterminer l'algorithme pertinent pour un problème donné en analysant le compromis entre la taille du reducer et le coût de communication en une seule itération MapReduce. Il y a deux paramètres qui représentent le compromis impliqué dans la conception d'un algorithme optimal en MapReduce : le premier est la taille du reducer,

notée q, qui représente la taille de la plus grande liste des valeurs $values$ associée à une clé key qu'un reducer peut recevoir. Le coût global est la somme des coûts de calcul sur chaque reducer traitant toutes ses valeurs associées. Le deuxième paramètre est la quantité de communication entre l'étape Map et l'étape Reduce. Le coût de communication, noté r, est défini comme le nombre moyen de paires clé-valeur que les Mappers créent à partir de chaque entrée. Formellement, supposons que nous ayons p reducers et que les entrées $q_i \leq q$ soient affectées au $i^{\text{ème}}$ reducer. Soit $|I|$ le nombre total de différentes entrées, alors le taux de réplication [3] est donné par l'expression $r = \sum_{i=1}^{p} q_i/|I|$.

Notons que la limitation de la taille du reducer permet plus de parallélisme. Une taille petite des reducers nous oblige à redéfinir la forme de clé afin de permettre plus de reducers et ainsi permettre plus de parallélisme sur les nœuds disponibles.

Borne inférieure du taux de réplication

Le taux de réplication est destiné à modéliser le coût de communication, c'est-à-dire la quantité totale de données envoyées des Maps aux reducers. Le compromis entre la taille du reducer q et le taux de réplication r, est exprimé par une fonction f, telle que $r = f(q)$. La première tâche dans la conception d'un bon algorithme en MapReduce pour un problème donné est de déterminer la fonction f, qui représente la borne inférieure du taux de réplication r [3].

Calculons maintenant la borne supérieure $g(q)$, sur le nombre de sorties qui peuvent être produites par un reducer de taille q pour la composition des WFSTs. On considère n WFSTs déterministes $(\mathcal{T}_i)_{1\leq i\leq n}$, chacun ayant $\frac{|\mathcal{T}_i|}{|A_i|}$ transitions pour chaque alphabet d'entrée. Soit $\mathcal{T}$ le résultat de la composition $\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$. Pour calculer une transition de $\mathcal{T}$, un reducer doit recevoir n transitions, une de chaque WFST $\mathcal{T}_i$. Par conséquent, le WFST résultant $\mathcal{T}$ peut avoir $\prod_{i=1}^{n} |\mathcal{T}_i|$ transitions. Supposons qu'un reducer de taille q reçoit $\frac{q}{n}$ transitions de chaque WFST $\mathcal{T}_i$ uniformément distribuées de telle sorte que l'alphabet d'entrée de $\mathcal{T}_{i+1}$ coïncide avec l'alphabet de sortie de $\mathcal{T}_i$. Le lemme suivant nous donne une borne supérieure sur la sortie d'un reducer.

Lemme 1. *Lors du calcul de la composition $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$ un reducer de taille q ne peut produire plus que $g(q) = \frac{q}{n}$ sorties.*

À partir de [3], on peut calculer une borne inférieure du taux de réplication pour la composition des WFSTs en fonction de q en utilisant l'expression

suivante :

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

où $|I|$ désigne la taille d'entrée et $|O|$ la taille de sortie. La taille d'entrée pour notre problème est la somme des transitions de tous les WFSTs d'entrée $\mathcal{T}_i$, $|I| = \sum_{i=1}^{n} |\mathcal{T}_i|$, et la taille de la sortie est la taille de $\mathcal{T}$, $|O| = |A_1| \times \frac{\prod_{i=1}^{n} |\mathcal{T}_i|}{\prod_{i=1}^{n} |A_i|}$.

Par conséquent, la borne inférieure du taux de réplication pour la composition des WFSTs est donnée dans la proposition suivante.

Proposition 1. *La borne inférieure du taux de réplication r nécessaire pour la composition $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$ est donnée par :*

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

3.2.3 Méthodes de mappage pour l'algorithme MapReduce

Cette section comprend la description de trois méthodes de mappage permettant de produire des schémas de clés qui mappe un ensemble de transitions vers le même reducer. Explicitement, nous définissons la fonction `getTransitionFrom(t)` de l'algorithme 1, la ligne 2. Nous présentons également, dans cette section, une analyse formelle du coût de communication en calculant une borne supérieure du taux de réplication pour chaque méthode de mappage. Rappelons que nous considérons la composition de n WFSTs, ayant chacun m états.

Méthode de mappage basée sur les états

La première méthode de mappage, basée sur les états, génère un ensemble de clés de la forme $K_{état} = (i_1, i_2, \ldots, h_{état}(s[t]), \ldots, i_n)$, à partir d'une transition $t \in E_i$ d'un WFST $\mathcal{T}_i$, où $h_{état}$ est une fonction de hachage de Q_i vers l'ensemble $\{1, \cdots, m\}$ et $i_j \in \{1, \cdots, m\}$.

Exemple 6. *Considérons les WFSTs $\mathcal{T}_1$ et $\mathcal{T}_2$ de l'exemple 5. Calculons l'ensemble des clès $K_{état}$ pour la composition $\mathcal{T}_1 \circ \mathcal{T}_2 \circ \mathcal{T}_1$ selon la méthode de mappage, basée sur les états. Nous avons $n = 5$, $|Q_1| = 4$ et $|Q_2| = 3$. Soit $h_{état}$ une fonction de hachage définit comme suit : $h_{état}(q_a) = h_{état}(q'_a) = 1$, $h_{état}(q_b) = h_{état}(q'_b) = 2$, $h_{état}(q_c) = h_{état}(q'_c) = 3$ et $h_{état}(q_d) = 4$. l'ensemble des clés seront alors générées à partir des transitions comme*

suit :

	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_1$
Transitions	$t_1 = (q_a, c, b, 0.2, q_c)$ $t_2 = (q_a, a, b, 0.1, q_b)$ $t_3 = (q_b, b, a, 0.6, q_a)$ $t_4 = (q_b, a, c, 0.4, q_d)$ $t_5 = (q_c, c, b, 0.2, q_b)$	$t'_1 = (q'_a, a, c, 0.1, q'_a)$ $t'_2 = (q'_a, b, c, 0.3, q'_b)$ $t'_3 = (q'_b, a, c, 0.6, q'_a)$ $t'_4 = (q'_b, c, b, 0.4, q'_c)$	$t_1 = (q_a, c, b, 0.2, q_c)$ $t_2 = (q_a, a, b, 0.1, q_b)$ $t_3 = (q_b, b, a, 0.6, q_a)$ $t_4 = (q_b, a, c, 0.4, q_d)$ $t_5 = (q_c, c, b, 0.2, q_b)$
	$\mathcal{T}_1$ i_1	$\mathcal{T}_2$ i_2	$\mathcal{T}_1$ i_3
	$h_{état}(s[t_1]) = h(q_a) = 1$ $h_{état}(s[t_2]) = 1$ $h_{état}(s[t_3]) = 2$ $h_{état}(s[t_4]) = 2$ $h_{état}(s[t_5]) = 3$	$h_{état}(s[t'_1]) = h(q'_a) = 1$ $h_{état}(s[t'_2]) = 1$ $h_{état}(s[t'_3]) = 2$ $h_{état}(s[t'_4]) = 2$	$h_{état}(s[t_1]) = h(q_a) = 1$ $h_{état}(s[t_2]) = 1$ $h_{état}(s[t_3]) = 2$ $h_{état}(s[t_4]) = 2$ $h_{état}(s[t_5]) = 3$

L'ensemble des clés généré par la fonction `getTransitionFrom(`t_1`)` *à partir de la transition t*1 *du premier* $\mathcal{T}_1$ *est égale alors à* {(1,1,1), (1,1,2), (1,1,3), (1,1,4), (1,2,1), (1,2,2), (1,2,3), (1,2,4), (1,3,1), (1,3,2), (1,3,3), (1,3,4)}. *Donc la transition* t_1 *sera répliquée et envoyée aux reducers* $1 \times |Q_2| \times |Q_1| = 12$ *fois.*

Les mappeurs produisent l'ensemble des paires (clé,valeur) de la forme $\langle (i_1, i_2, \ldots, h(s[t]), \ldots, i_n), t \rangle$. Par conséquent, si nous considérons m^n reducers, chaque transition est envoyée à m^{n-1} reducers.
La fonction $g(q)$ sera affectée par la présence à l'intérieur du même reducer de certaines transitions qui ne peuvent pas être composées. Cela nous donne une nouvelle borne supérieure sur le nombre de sorties que chaque reducer peut produire formellement

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

La proposition suivante donne une borne supérieure du taux de réplication.

Proposition 2. *Le taux de réplication r dans le schéma de mappage basé sur les états est*

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

Notons que à partir des propositions 1 et 2, la borne supérieure du taux de réplication dépasse la borne inférieure d'un facteur $\frac{q}{n \times |A_1|}$. Par conséquent, ce schéma de mappage est approprié lorsque l'on considère un alphabets d'entrée

de taille petite.

Méthode de mappage basée sur l'alphabet d'entrée

Dans la deuxième méthode, les transitions seront mappées par leur alphabet d'entrée. Définissons la clé $K_{alphabet} = (j_1, j_2, \cdots, j_i, \cdots, j_n)$ associée à un symbole d'entrée a et soit h_a une fonction de hachage sur l'alphabet d'entrée vers l'ensemble $\{1, 2, \cdots, k\}$. On peut associer une transition $t = (s_i, a_i, b_i, w_i, d_i) \in E_i$ d'un WFST $\mathcal{T}_i$ à une clé $K_{alphabet}$, s'il existe j_i tel que $j_i = h_{alphabet}(a_i)$.

Exemple 7. *Considérons les FSMs de l'Exemple 6 et calculons l'ensemble des clès $K_{alphabet}$ pour la composition $\mathcal{T}_1 \circ \mathcal{T}_2 \circ \mathcal{T}_1$ selon la méthode de mappage, basée sur l'alphabet d'entrée. Soit $h_{alphabet}$ une fonction de hachage définit comme suit : $h_{alphabet}(a) = 1$, $h_{alphabet}(b) = 2$, et $h_{alphabet}(c) = 3$. l'ensemble des clés seront alors générées à partir des transitions comme suit :*

$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_1$
j_1	j_2	j_3
$h_{alphabet}(I[t_1]) = h_a(c) = 3$	$h_{alphabet}(I[t'_1]) = h_a(a) = 1$	$h_{alphabet}(I[t_1]) = h_a(c) = 3$
$h_{alphabet}(I[t_2]) = 1$	$h_{alphabet}(I[t'_2]) = 2$	$h_{alphabet}(I[t_2]) = 1$
$h_{alphabet}(I[t_3]) = 2$	$h_{alphabet}(I[t'_3]) = 1$	$h_{alphabet}(I[t_3]) = 2$
$h_{alphabet}(I[t_4]) = 1$	$h_{alphabet}(I[t'_4]) = 3$	$h_{alphabet}(I[t_4]) = 1$
$h_{alphabet}(I[t_5]) = 3$		$h_{alphabet}(I[t_5]) = 3$

L'ensemble des clés généré par la fonction `getTransitionFrom(`t_1`)` *à partir de la transition t_1 du premier $\mathcal{T}_1$ est égale alors à {(3,1,1), (3,1,2), (3,1,3), (3,2,1),(3,2,2), (3,2,3), (3,3,1), (3,3,2), (3,3,3)}. Donc la transition t_1 sera répliquée et envoyée aux reducers $1 \times |A_2| \times |A_1| = 9$ fois.*

Nous avons $\prod_{j=1}^{n} |A_j|$ reducers disponibles et chaque transition de $\mathcal{T}_i$ est envoyée à $\frac{\prod_{j=1}^{n} |A_j|}{|A_i|}$ reducers. Puisque la tâche Map traite $\sum_{i=1}^{n} |\mathcal{T}_i|$ transitions et chaque WFST $\mathcal{T}_i$ a $\frac{|\mathcal{T}_i|}{|A_i|}$ transitions associées à un symbole d'entrée $c \in |A_i|$, le nombre total de transitions envoyées à chaque reducer est $n \times \frac{|\mathcal{T}_i|}{|A_i|}$. Ce nombre peut être estimé à $n \times m$. Cependant, la fonction $g(q)$ est influencée par la présence d'une combinaison incompatible de symboles de sortie et d'entrée à l'intérieur du même reducer. Cela nous donne une nouvelle borne supérieure sur le nombre de sorties que chaque reducer peut produire, formellement nous avons

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

Proposition 3. *Le taux de réplication dans le schéma de mappage basé sur les alphabets d'entrée est*

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

À partir des propositions 1 et 3, la borne supérieure du taux de réplication dépasse la borne inférieure théorique d'un facteur $\frac{q}{n \times |Q_1|}$. Par conséquent, le mappage basé sur l'alphabet d'entrée est le mieux adapté aux situations où les WFSTs sont de tailles petites.

Méthode de mappage hybride basé sur les alphabets d'entrée et de sortie

Dans la troisième méthode, nous proposons un mappage hybride basée à la fois sur les alphabets d'entrée et de sortie. En effet, les clés seront associées à une paire de symboles d'entrée et de sortie. Formellement, une transition $t \in E_i$ d'un WFST $\mathcal{T}_i$ sera associée à un ensemble de clés de la forme $K_{hybride} = (j_1, j_2, \cdots, h_a^i(t), h_b^o(t), \cdots, j_{n-1})$, où $h_a^i()$ est une fonction de hachage sur l'alphabet d'entrée $\bigcup_{i=1}^{n} A_i$ vers $\{1, 2, \cdots, k\}$ et h_b^o est une fonction de hachage sur l'alphabet de sortie $\bigcup_{i=1}^{n} B_i$ vers $\{1, 2, \cdots, k\}$.

Exemple 8. *Considérons les FSMs de l'Exemple 6 et calculons l'ensemble des clès $K_{hybride}$ pour la composition $\mathcal{T}_1 \circ \mathcal{T}_2 \circ \mathcal{T}_1$ selon la méthode de mappage hybride basé sur les alphabets d'entrée et de sortie. Soit $h_{hybride}$ une fonction de hachage définit comme suit : $h_{hybride}(t) = (h_a^i(I[t]), h_b^o(O[t]))$ tel que $h_a^i(a) = h_b^o(a) = 1$, $h_a^i(b) = h_b^o(b) = 2$, et $h_a^i(c) = h_b^o(c) = 3$,. l'ensemble des clés seront alors générées à partir des transitions comme suit :*

$\mathcal{T}_1$ j_1	$\mathcal{T}_2$ j_2	$\mathcal{T}_1$ j_3
$h_{hybride}(t_1) = (3,2)$	$h_{hybride}(t'_1) = (1,3)$	$h_{hybride}(t_1) = (3,2)$
$h_{hybride}(t_2) = (1,2)$	$h_{hybride}(t'_2) = (2,3)$	$h_{hybride}(t_2) = (1,2)$
$h_{hybride}(t_3) = (2,1)$	$h_{hybride}(t'_3) = (1,3)$	$h_{hybride}(t_3) = (2,1)$
$h_{hybride}(t_4) = (1,3)$	$h_{hybride}(t'_4) = (3,2)$	$h_{hybride}(t_4) = (1,3)$
$h_{hybride}(t_5) = (3,2)$		$h_{hybride}(t_5) = (3,2)$

L'ensemble des clés généré par la fonction `getTransitionFrom(`t_1`)` *à partir de la transition t_1 du premier $\mathcal{T}_1$ est égale alors à {(2,1), (2,2), (2,3)}. Donc la transition t_1*

sera répliquée et envoyée aux reducers $k = 3$ *fois, et la transition* t'_1 *sera envoyée à un seul reducer* $(1, 3)$.

Les transitions de $\mathcal{T}_1$ seront mappées selon leurs symboles de sortie et ceux de $\mathcal{T}_n$ selon leurs symboles d'entrée. Par conséquent, le nombre de reducers devient de l'ordre k^{n-1} et la fonction $g(q) = |Q_1|$.
Puisque les transitions de $\mathcal{T}_2$ à $\mathcal{T}_{n-1}$ sont envoyées aux k^{n-3} reducers, nous avons :

Proposition 4. *Le taux de réplication dans la méthode de mappage hybride est strictement inférieur au taux de réplication dans la méthode de mappage basée sur l'alphabet d'entrée.*

En comparant les propositions 2, 3, et 4, nous en déduisons que la borne supérieure du taux de réplication dans la méthode de mappage hybride est la plus proche de la borne inférieure théorique. Ainsi, cette méthode est la mieux adaptée aux situations où la taille de l'alphabet est inférieure ou égale au nombre d'états.

3.3 Implémentation et évaluation expérimentale

Cette section comprend différentes expériences nécessaires pour évaluer l'efficacité des méthodes proposées en termes de coût de communication et temps d'exécution pour la composition de cinq WFSTs $\mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_5$.

Des expériences sont menées sur une grande variété de WFSTs générés aléatoirement à l'aide de la bibliothèque FAdo [11] avec diverses combinaisons de paramètres comprenant le nombre d'états $|Q|$, la taille de l'alphabet d'entrée $|A|$ et la taille de l'alphabet de sortie $|B|$.

3.3.1 Configuration de cluster de calcul

Nos expériences ont été réalisées avec la plateforme Hadoop 2.7. sur le cluster de calcul scientifique français Grid'5000 [33]. Nous avons utilisé pour nos expériences un cluster composé de 15 nœuds, 30 CPUs, 300 cœurs. Chaque nœud est une machine équipée de deux processeurs Intel Xeon E5-2630 v4 à 10 cœurs par processeur, 256 Go de mémoire principale, et deux disques durs (HDD) de 300 Go. Les machines sont connectées par un réseau Ethernet 10 Gbps et fonctionnent sous Debian 9 64 bits.

3.3.2 Génération de données expérimentales

Nous avons généré aléatoirement une grande variété de jeu des données en deux phases. Nous avons tout d'abord généré un ensemble d'automates

finis déterministes en utilisant la bibliothèque FAdo [11], qui est un projet open source fournissant un ensemble d'outils pour la manipulation symbolique des automates. FAdo est basé sur l'énumération et la génération initialement d'automates finis déterministes connectés [142]. Ensuite, sur les transitions des automates produits de l'étape précédente, nous ajoutons aléatoirement selon une distribution uniforme des poids et un certain degré de non déterminisme sur les symboles de sortie. Notre technique de génération produit des WFSTs basés sur deux paramètres : le nombre d'états m et la taille de l'alphabet d'entrée k. Ainsi, on peut définir la densité de transition du WFST généré comme le rapport $\frac{|E|}{k \times m}$, la densité d'états finaux comme rapport $\frac{|F|}{m}$ et considérer un état initial unique comme dans l'approche de génération introduite dans [111].

3.3.3 Analyse expérimentale du coût de communication

Nous évaluons le coût de communication des méthodes proposées sur des jeux de données à grande échelle. Le coût de communication est défini comme étant le nombre total de paires clé-valeur transférées depuis la phase Map vers la phase Reduce. Il peut être optimisé en minimisant le paramètre de taux de réplication, c'est-à-dire le nombre de copies d'entrée envoyées aux reducers.

Le tableau suivant présente la relation entre les tailles des données en entrée et le coût de communication :

Les résultats obtenus montrent que le coût de communication de la méthode de mappage hybride est minimal dans toutes les combinaisons considérées de nombres d'états et de tailles d'alphabet. Ceci est dû au fait que le nombre de copies de transition envoyées aux reducers avec cette méthode est inférieur à celui des autres méthodes comme prouvé formellement dans la proposition 4. Dans certains cas particuliers de WFSTs, lorsque le nombre d'état est inférieure à la taille de l'alphabet, le mappage basé sur les états a un coût de communication plus petit. Ce résultat coïncide avec les propositions 2 et 3.

3.3.4 Analyse expérimentale du coût de calcul

Le coût de calcul est le temps requis pour exécuter une tâche MapReduce. Les figures ci-dessous, 3.2,3.3,3.4 et 3.5, présentent les résultats comparatifs en termes de temps d'exécution entre les trois méthodes : le mappage basé sur les états (State), le mappage basé sur l'alphabet d'entrée (Input) et le mappage hybride (InOut).

Taille d'entrée					Coût de communication (GB)		Taille de sortie	
Taille de fichier (KB)	$\sum_1^5 \lvert E_i \rvert$ ($\times 10^3$)	$\lvert Q_i \rvert$	$\lvert A_i \rvert$	$\lvert B_i \rvert$	Mappage hybride	Mappage basée sur l'alphabet	Mappage basée sur les états	$\lvert E \rvert$ ($\times 10^9$)	Taille (GB)
132	6	75	16	16	10	22	10699	38	1765
227	10	63	32	32	252	601	9032	31.8	1486
342	15	47	64	64	5896	14402	4189	14.7	684
411	18	60	60	60	5465	13327	13327	46.7	2194

Tableau 3.1 – La relation entre les tailles des données en entrée et le coût de communication

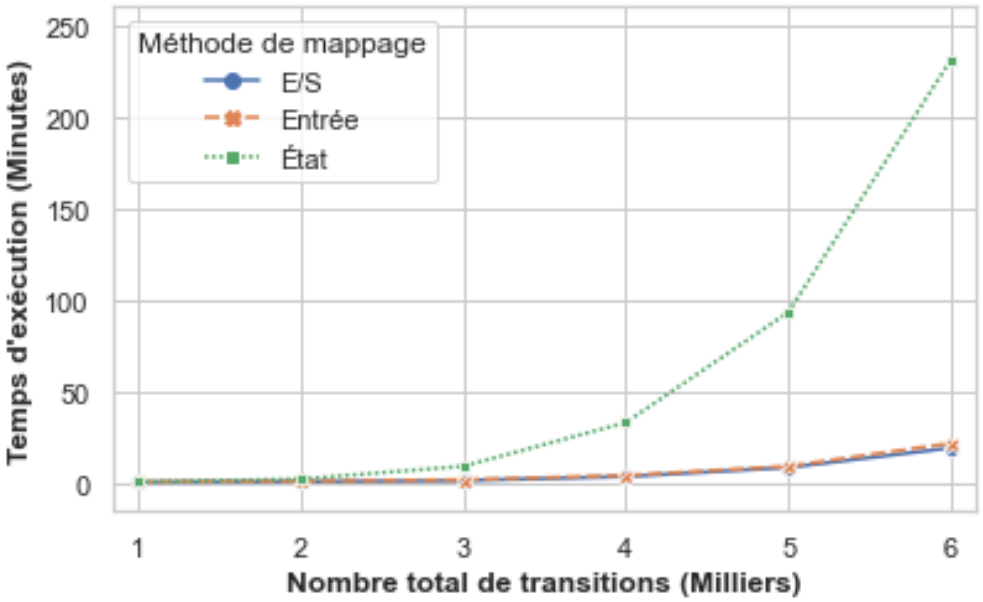

FIGURE 3.2 – Temps d'exécution des trois méthodes pour la taille de l'alphabet K = 16.

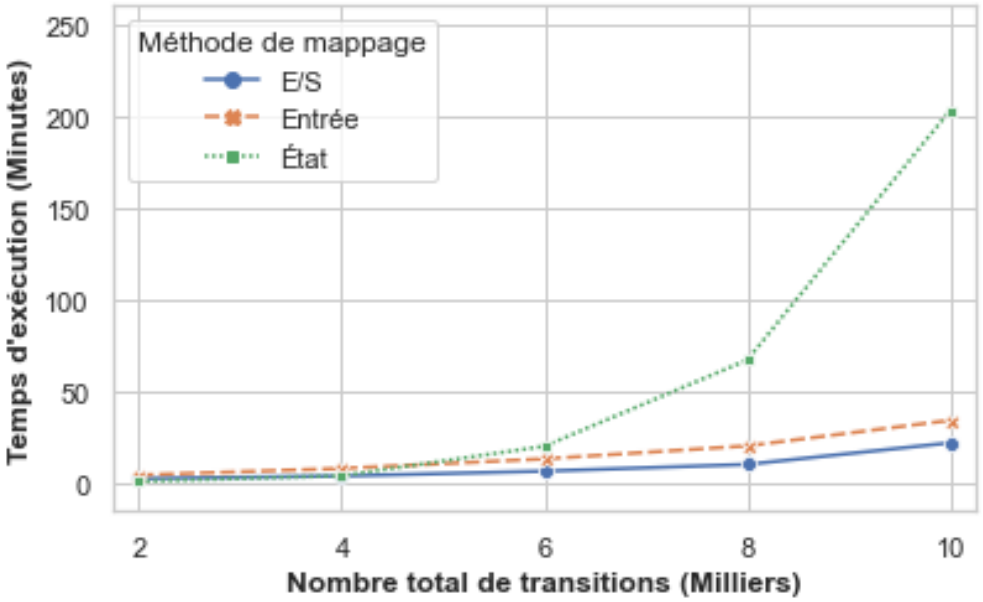

FIGURE 3.3 – Temps d'exécution des trois méthodes pour la taille de l'alphabet K=32.

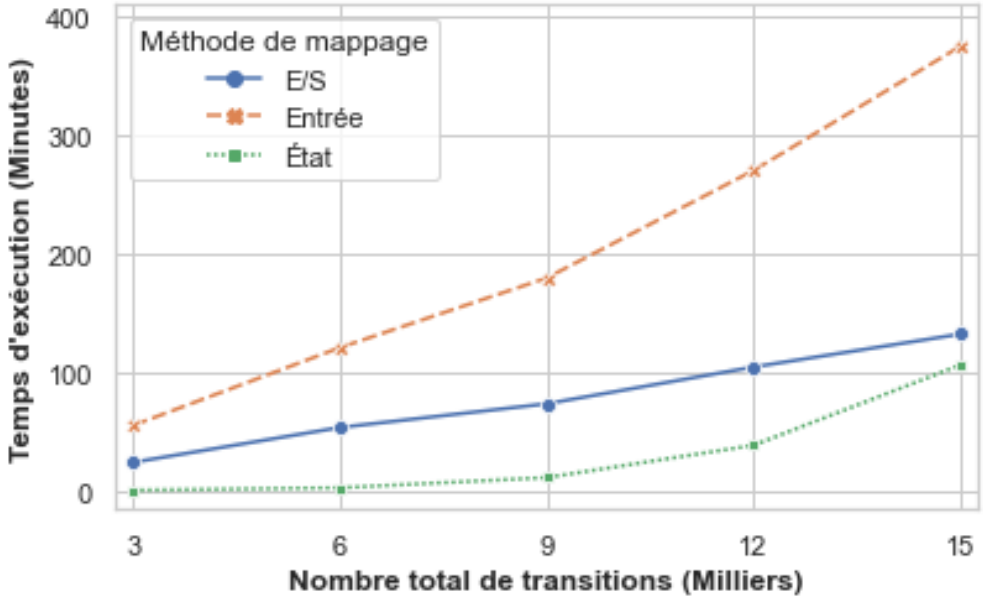

FIGURE 3.4 – Temps d'exécution des trois méthodes pour la taille de l'alphabet K=64.

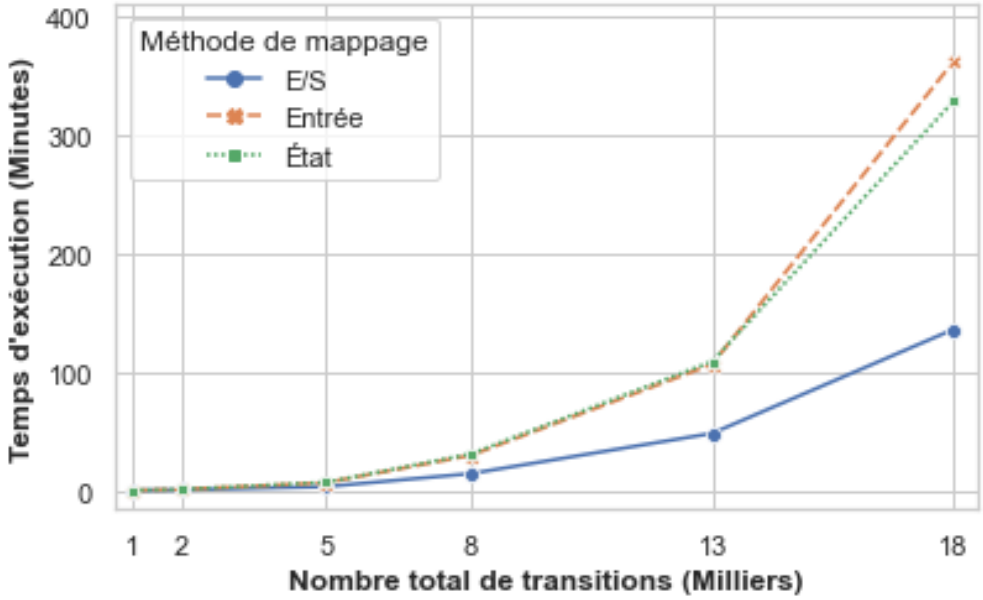

FIGURE 3.5 – Temps d'exécution des trois méthodes lorsque le nombre d'états est égal à la taille de l'alphabet.

Les figures 3.2, 3.3 et 3.4 présentent le temps d'exécution des trois méthodes

proposées pour différentes tailles d'ensembles de données en variant la taille de l'alphabet à 16, 32 ou 64 . La figure 3.5 montre une comparaison des trois méthodes lorsque la taille de l'alphabet est égale au nombre d'états. Comme prévu, la méthode hybride est nettement plus efficace lorsque la taille de l'alphabet est inférieure ou égale au nombre d'états. La réduction du taux de réplication diminue le temps utilisé par les mappeurs pour répliquer chaque transition et évite l'existence de transitions qui ne peuvent pas être composées au sein du même reducer. En même temps, il réduit le nombre de transitions affectées à un reducer. D'un autre côté, l'utilisation d'un nombre adéquat de reducers diminue le temps d'attente qu'un reducer passe pour utiliser un processeur.

Résumé

Dans ce chapitre, nous avons présenté une nouvelle approche parallèle pour la calcul de la composition des WFSTs à grande échelle en utilisant le framework MapReduce. Nous avons proposé trois méthodes MapReduce. De plus, nous avons analysé le coût de communication et de calcul pour chacune des méthodes proposées.
Enfin, nous avons évalué la performance des trois méthodes sur différents ensembles de jeu de données. Les résultats obtenus montrent que la meilleure méthode en termes de temps d'exécution est celle qui minimise le nombre de reducers et optimise le taux de réplication des entrées.

Chapitre 4

Dérivation massivement parallèle des séquences séparantes

Les séquences séparantes sont largement utilisées dans les tests de conformité sur les machines à états finis pour résoudre le problème d'identification des états. Dans ce chapitre, nous abordons le problème d'évolutivité rencontré lors de la dérivation de séquences séparantes à partir de machines à états finis non déterministes observables complètes en utilisant une version MapReduce massivement parallèle de l'Algorithme Exact [140]. Tout d'abord, nous donnons un aperçu concis de l'Algorithme Exact pour dériver des séquences séparantes à partir de machines à états finis non déterministes. Ensuite, nous proposons un algorithme massivement parallèle pour ce problème en utilisant l'approche MapReduce et nous analysons son coût de communication en utilisant le modèle d'Afrati *et al.* [3]. Enfin, nous menons diverses expériences comparatives sur une variété de jeu de données des machines à états finis non déterministes afin de montrer l'efficacité à grande échelle de notre approche.

4.1 Machines à états finis et tests de conformité

Les systèmes logiciels récents à grande échelle, composés de milliers ou de millions de composants matériels et logiciels en interaction, sont devenus des composants critiques de nombreuses infrastructures importantes dans notre société, notamment ceux utilisés pour les services bancaires, les soins de santé, le contrôle du trafic aérien, la téléphonie et de nombreux autres secteurs. Les systèmes logiciels se composent de centaines ou milliers d'ordinateurs et de millions de lignes de code et ils exécutent des tâches complexes presque en permanence. Ils soulèvent un large éventail de problèmes techniques et non

techniques qui peuvent devenir critiques. D'autres sont apparus récemment en raison de l'augmentation de l'échelle et du degré d'interconnexion des systèmes logiciels. Les tests de logiciels étant une étape essentielle du processus de développement de logiciels, nous devons utiliser des approches efficaces pour réduire à la fois le coût et le temps des tests.

La phase de test (Testing) est une partie indispensable du développement des systèmes pour garantir le bon fonctionnement et trouver des aspects du comportement des systèmes modélisés [108]. Cette phase est bien étudiée dans le domaine de la recherche sur les tests des systèmes basés sur les FSM. Cependant, afin d'extraire certaines informations sur une implémentation sous test (IUT) d'une FSM inconnu, des expériences seront menées sur cette IUT [54]. Le but de ces expériences est de vérifier si l'implémentation d'un modèle est conforme à sa spécification, en appliquant des séquences de vérification (Checking Sequences, CSs) à l'IUT, en observant les réponses de sortie correspondantes et en formulant une conclusion sur l'IUT [103]. En d'autres termes, il faut reconnaître l'état de l'IUT. La reconnaissance de l'état peut être accomplie en utilisant des CSs comme des séquences séparantes [148], des séquences uniques entrée-sortie [6, 79], des ensembles de caractérisation (W-Set) [78], ou des séquences de synchronisation (également connues sous le nom de séquences de réinitialisation) [90], quand ces séquences existent. La motivation pour étudier de telles séquences provient de différents domaines tels que la robotique, la bio-informatique, le calcul propositionnel, les tests basés sur des modèles, les tests distribués, et bien d'autres domaines [31, 36, 45, 60, 82, 90, 123, 77, 148, 149, 151]. La littérature contient de nombreuses techniques permettant de générer automatiquement des CSs [45, 122, 36, 83, 91, 107, 137, 138]. La plupart des approches se composent, en principe, de trois parties : l'initialisation, l'identification des états et la vérification des transitions.

Dans ce travail, nous avons étudié le problème de génération des séquences séparantes (Distinguishing Sequences, DSs) à partir des Machines à Etats Finis (FSM) à grande échelle en utilisant le modèle de calcul massivement parallèle en MapReduce. Les DSs ont été introduites afin de répondre au problème d'identification d'état, qui consiste à trouver une séquence d'entrée qui produit des sorties différentes pour chaque état initial d'une FSM. Ce problème a été initialement décrit dans l'article de Moore [122] en 1956, et en 1964, Hennie [76] a fourni le premier algorithme automatisé de génération des séquences de test à partir des FSMs. Les DSs, lorsqu'elles existent, conduisent à des tests plus courts [54]. Il existe deux types de DSs, adaptatives (Adaptive Distinguishing Sequence, ADS) et prédéfinies (Preset Distinguishing Sequence, PDS)[102] pour différentes classes de FSMs (déterministes [54], non déterministes [84], complètes [141], partielles [127], observables [13]). La PDS est une séquence d'entrée unique fixe

qui peut être utilisée pour distinguer chaque état de la machine [70]. Pour une ADS, l'entrée suivante dépend de la sortie de l'entrée courante. Elle consiste en un arbre où chaque chemin de la racine vers les feuilles représente une séquence d'entrée spécifique à l'état représenté par la feuille. Pour obtenir la plus courte DS d'une paire d'états, Spitsyna *et al.* ont conçu l'Algorithme Exact (EA) [140]. Il est principalement basé sur deux étapes : À partir d'une FSM M, on construit un arbre des successeurs tronqué à partir de l'intersection de deux FSMs initialisés M/s_1 (c'est-à-dire la FSM M avec comme état initial s_1) M/s_2 (c'est-à-dire la FSM M avec comme état initial s_2), ensuite on dérive la plus courte DS. Ils ont proposé une méthode d'analyse de la relation de séparabilité entre les FSMs qui peut être utilisée pour dériver une plus courte DS (si elle existe) de deux FSMs donnés (ou pour deux états d'une FSM donnée) et ils ont démontré que pour deux états d'une FSM, sa borne supérieure devient exponentielle.
Lorsque qu'on considère des FSMs non déterministes, Alur *et al.* [12] ont montré que la longueur d'une DS pour tous les états peut atteindre une limite exponentielle. De plus, la complexité pour décider de l'existence d'une PDS est PSPACE-complète, et elle est EXPTIME-complète pour décider l'existence d'une ADS.
Dans ce chapitre, nous proposons une version optimale massivement parallèle en MapReduce de l'Algorithme Exact. Les différentes expériences menées ont montré que notre approche est efficace, scalable et meilleure que les différentes approches parallèles de l'état de l'art pour ce problème.

Travaux connexes

Approches parallèles pour dériver des séquences séparantes

Diverses études ont récemment porté sur l'utilisation de techniques de traitement parallèle afin de dériver des CSs dans un contexte à grande échelle : des séquences UIO [79], des identifiants d'état harmonisés et ensembles de caractérisation [78], des séquences de synchronisation [147, 93]. Pour la génération de DSs, Hierons et Türker [81], et El-Fakih *et al.* [62] ont introduit des implémentations parallèles de l'EA sur les architectures Central Processing Unit (CPU) et Graphics Processing Unit (GPU) [128]. Ils ont mené des expériences approfondies sur une grande variété de classes de FSM, avec différentes architectures CPU-GPU et charges de travail. Les résultats obtenus montrent que leurs approches sont suffisamment efficaces pour les données à grande échelle et que le temps d'exécution pour dériver des DS à partir de FSM non déterministes augmente exponentiellement par rapport à différents paramètres tels que le degré de non déterminisme, le nombre de transitions et le rapport entre la taille de l'alphabet d'entrée et la taille de l'alphabet de sortie.

Afin de réduire le temps d'exécution de la construction de la table des successeurs de toutes les paires d'états pour un FSM non déterministe donné, El-fakih *et al.* [62] ont proposé différentes approches parallèles multithreads basées sur une architecture CPU et GPU multi-cœurs. Ils ont utilisé deux options à savoir : les plateformes logicielles robustes et les implémentations GPU utilisant la plateforme CUDA. Ils ont également proposé et évalué une solution de réseau de stations de travail basée sur la théorie de charge divisible (Divisible Load Theory, ou DLT). Ils ont mené leurs expériences sur une variété de FSMs non déterministes avec un grand nombre de symboles d'entrée et de sortie. leurs expériences mettent en évidence la différence entre les algorithmes proposés en termes d'accélération et temps d'exécution.

Dans [81], Hierons et Türker ont considéré les FSMs non déterministes partiellement observables et ont étudié le problème d'évolutivité pour la construction de séquences PDSs et ADSs pour tous les états. Ils ont proposé un algorithme de génération des ADSs qui peut traiter des entrées jusqu'à 2048 fois plus que les algorithmes existants de construction des ADSs et un algorithme de génération des PDSs qui peut traiter des entrées jusqu'à 8 fois plus que les algorithmes existants de génération des PDS. Leur approche est basée sur le parallélisme disponible sur des machines à GPU, appelé *the thin thread strategy* en utilisant la mémoire globale du dispositif et maximisant ainsi le nombre de threads afin de maximiser le degré de parallélisme. Les résultats de leurs expériences sont satisfaisantes et montrent que l'algorithme proposé peut dériver des DSs à partir de FSMs non déterministes partielles observables ayant 32000 états en un temps acceptable.

4.2 Aperçu de l'algorithme exact

Dans cette section, nous présentons un aperçu concis de l'Algorithme Exact [140] qui construit les plus courtes DSs, si elles existent, à partir d'une FSM non déterministe observable complète.
À partir d'une FSM M, nous considérons deux FSMs avec un état initial différent, notées M/s_1 et M/s_2. L'Algorithme Exact (EA) sera appliqué alors sur une seule FSM M pour dériver une DS entre deux états s_1 et s_2 de M. Notons que, dans la suite, l'ordre des paires d'états ne fait pas de différence entre (s_0, s_1) et (s_1, s_0). Afin de dériver une DS, l'EA est implémenté à l'aide de la méthode BFS (Breadth-First Search) qui parcourt l'arbre des successeurs en largeur c'est à dire niveau par niveau.

Algorithme 3 : Algorithme exact (EA)

Entrée : Deux états différents s_k et s_l d'une $M = (S, I, O, E)$ complète observable et non déterministe.

Sortie : Une DS courte des états s_k et s_l (si elle existe) ou le message que les états s_k et s_l ne sont pas séparables.

Étape de l'intersection

// *Dériver l'arbre des successeurs* Arbre *à partir de* $M/s_k \cap M/s_l$

foreach *transition* $(t, t') \in E \times E$ **do**
  if $I[t] = I[t']\&\&O[t] = O[t']$ **then**
    insert(Arbre,$((s[t],s[t']),I[t],(d[t],d[t']))$)

if Arbre *est une FSM complète* **then**
  Les états s_k et s_l sont non-séparables.
else
  dériver l'arbre des successeurs de $M/s_k \cap M/s_l$.

Étape de dérivation

// *Un nœud intermédiaire du* $j^{ème}$ *niveau étiqueté avec un sous-ensemble P d'états de l'intersection*
// *Un noeud courant Courant, au* $p^{ème}$ *niveau, étiqueté avec le sous-ensemble P de paires d'états*

if *il existe une entrée i telle que chaque paire de l'ensemble P n'a pas de i-successeurs*
or *il existe un nœud à un* j^{ime} *niveau,* $j < p$*, étiqueté avec un sous-ensemble R d'états tel que* $P \supseteq R$*,*
or *pour une paire* (s, t) *de l'ensemble P et une sortie o, la séquence d'entrée/sortie io prend les états s et t au même état.* **then**
  Le nœud courant est considéré comme nœud feuille
if *aucun des chemins de l'arbre dérivé n'est terminé* **then**
  return s_k et s_l sont non-séparables.
else
  une plus courte séquence αi où α étiquette le chemin de la racine de l'arbre à une feuille, est une plus courte séquence séparante de s_k et s_l, **return** αi.

L'EA est divisé en deux étapes principales : l'étape de l'intersection et l'étape de dérivation. Dans la première étape, nous calculons l'intersection $M/s_1 \cap M/s_2$. Si le résultat de l'intersection est une FSM partielle (non complète) alors nous

dérivons un arbre des successeurs *Arbre*, sinon nous renvoyons le message que les deux états ne sont pas séparables. Dans la deuxième étape, nous dérivons à partir de l'arbre des successeurs *Arbre* précédemment construite une DS courte pour la paire d'états du nœud racine, s'il existe, par la méthode classique de parcours en largeur d'un arbre, comme présenté dans l'algorithme 3. La racine de cet arbre, qui se trouve au 0ème niveau, est l'état initial (s_k, s_l) de l'intersection ; les nœuds de l'arbre sont étiquetés par des sous-ensembles d'états de l'intersection. Étant donné j niveaux de l'arbre des successeurs *Arbre*, tel que $j \geq 0$, un nœud interne du $j$ème niveau étiqueté avec un sous-ensemble P d'états de l'intersection, et une entrée i, il existe une arête sortante de ce nœud interne étiqueté par i vers le nœud étiqueté avec le sous-ensemble des i-successeurs des paires d'états du sous-ensemble P. Un nœud courant *Courant*, au $p$ème niveau, $p \geq 0$, étiqueté avec le sous-ensemble P de paires d'états, est déclaré comme étant un *noeudfeuille* si l'une des conditions de la ligne 10, 11 ou 12 de l'Algorithme 3 est vérifiée. Ensuite, s'il n'existe pas de nœud feuille suivant la condition de la ligne 10, alors la paire d'états du nœud racine est non-séparable. Sinon, s'il existe un nœud feuille étiqueté par le sous-ensemble P d'états tel que pour une certaine entrée i, chaque état de l'ensemble P n'a pas de i-successeurs, alors nous dérivons une DS αi où α est l'étiquette du chemin du nœud racine de l'arbre vers le nœud feuille.

Le nombre de sous-ensembles de paires d'états possibles pour une FSM ayant n états est de $2^{\frac{n^2}{2}}$ (le nombre de sous-ensembles possibles P dans l'Algorithme 3). Par conséquent, la complexité temporelle dans le pire des cas de la première étape d'EA est en $O(2^{\frac{n^2}{2}})$. Dans la deuxième étape de l'EA, la dérivation d'une DS, si elle existe, à partir de l'arbre des successeurs précédemment construit, est effectuée à l'aide de la méthode classique de recherche en largeur (Breadth-First Search), comme présenté dans l'Algorithme 3. Ensuite, la complexité temporelle de l'EA dans le pire des cas est de $O(2^{\frac{n^2}{2}} \times \frac{n^2}{2} \times |I| \times |O|)$. En considérant deux FSMs ayant respectivement n et m états, il est montré dans [140] que la longueur de la plus courte DS est au plus 2^{mn-1} et que cette borne supérieure peut être atteinte. par conséquent, la borne supérieure pour une seule FSM devienne 2^{n^2-1}.

Cependant, les expérimentations de [140] ont montré l'existence d'une grande classe de paires de FSMs ayant n et m états tels que la longueur de la plus courte DS d'une paire d'états est inférieure à mn et moins que n^2 lorsque l'on considère une seule FSM. Les expériences montrent également que l'existence d'une séquence DS dépend de manière significative du degré de non déterminisme des FSMs considérées [140].

Exemple 9. *Considérons la FSM M de l'Exemple 2 et appliquons l'EA pour dériver*

une DS entre l'état s_0 et l'état s_1. La Figure 4.1 montre l'arbre des successeurs dérivé de l'étape 1 d'EA. Les nœuds de cet arbre des successeurs sont étiquetés de n_1 à n_7.

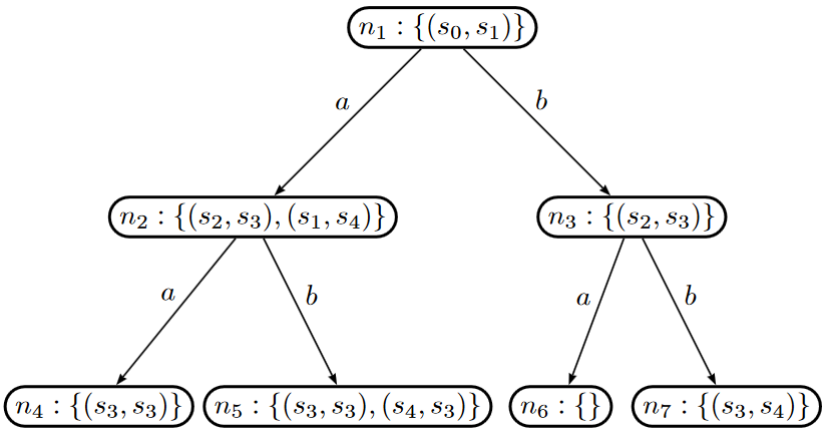

FIGURE 4.1 – L'arbre des successeurs de $M/0 \cap M/1$

Comme l'intersection de M/s_0 et M/s_1 n'est pas une FSM complète, le nœud racine n_1 associé à la paire d'états (s_0, s_1) est au niveau $j = 0$. Les successeurs de n_1 sont les nœuds n_2 et n_3. Ensuite, pour le niveau $j = 1$, on obtient les paires d'états aux nœuds n_4 et n_5 comme successeurs de n_2, et les successeurs de n_3 sont les paires d'états aux nœuds n_6 et n_7. Le nœud n_4 a une paire avec l'état répété (s_3, s_3), donc, nous ne considérons pas n_4 pour une exploration plus approfondie. Le nœud n_6 est étiqueté avec l'ensemble vide, c'est-à-dire, il n'y a aucun successeur pour n_3 sous aucun symbole d'entrée. Par conséquent, la séquence d'entrée "ba" qui commence à partir de la racine et mène au nœud étiqueté avec l'ensemble vide est une courte DS pour les FSMs M/s_0 et M/s_1.

4.3 Algorithme MapReduce pour la dérivation des séquences séparantes

Dans cette section, nous présentons et analysons notre version parallèle de l'EA en utilisant une approche parallèle basée sur le framework MapReduce pour extraire un ensemble de courtes DSs à partir d'une FSM non déterministe observable complète. Notre méthode améliore les approches parallèles précédentes dans le sens où elle fournit efficacement une courte DSs pour chaque paire d'états d'une FSM à grande échelle.

4.3.1 Solution MapReduce

La solution proposée se compose en deux étapes MapReduce : l'étape d'intersection et l'étape de dérivation de DSs courtes ou l'étape de dérivation (par abréviation).

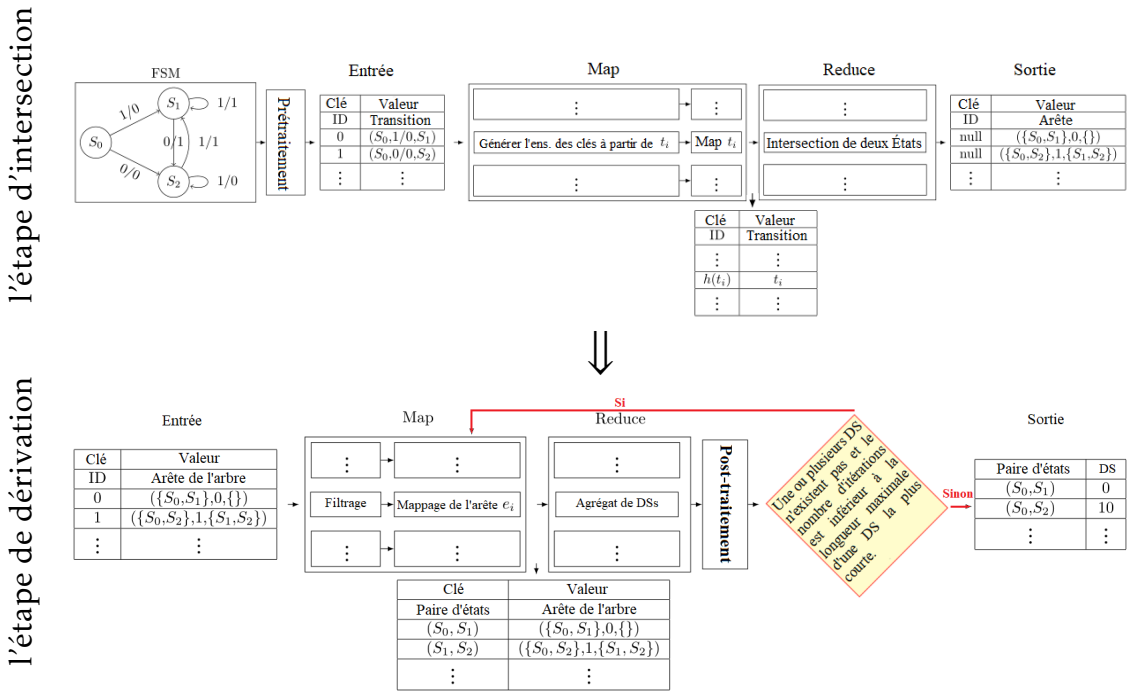

FIGURE 4.2 – Un aperçu de l'algorithme exact parallèle utilisant MapReduce.

La figure 4.2 illustre le workflow de notre solution, qui prend en entrée une FSM de grande taille et produit un ensemble de courtes DSs pour toutes les paires d'états après un calcul parallèle en MapReduce en deux phases. Initialement, une FSM d'entrée $M = (S, I, O, E)$ est prétraitée pour produire un fichier texte où chaque ligne ($valeur$) représente une transition $t = (s[t], I[t], O[t], d[t])$. Ce fichier est l'entrée de l'étape d'intersection. La fonction map produit un ensemble de paires $\langle clé, valeur \rangle$. Dans notre cas, pour une transition t, elle génère un ensemble de clés associées. Ensuite, l'ensemble des paires produites par la fonction map sont regroupées et envoyées à la fonction reduce. Cette dernière reçoit les transitions ayant la même clé et calcule leur intersection, c'est-à-dire que pour deux transitions t_i et t_j ayant la même *entre/sortie*, le résultat sera une arête dans l'arbre des successeurs $(\{(s[t_i], s[t_j])\}, I[t_i], \{(d[t_i], d[t_j])\})$, sinon $(\{(s[t_i], s[t_j])\}, I[t_i], \{\})$. Dans la deuxième étape, nous proposons un algorithme

MapReduce itératif pour dériver une courte DS si elle existe. L'entrée de cette étape est la sortie de l'étape d'intersection précédente. La fonction map de cette étape prend une arête de l'arbre des successeurs (n_s, l, n_d) où n_s désigne le noeud source, l est l'étiquette et n_d est le noeud de destination et produit un ensemble de clés associées. Pour une arête donnée, si son noeud de destination est vide, la clé associée sera son noeud source, sinon chaque paire d'états dans son noeud de destination devient la nouvelle clé associée. Ensuite, la fonction de reduce reçoit l'ensemble des arêtes ayant la même clé et divise l'ensemble des arêtes associées en deux sous-ensembles, le premier contient toutes les arêtes ayant un nœud de destination vide, l'autre contient le reste des arêtes. Puis, le produit cartésien de ces deux sous-ensembles est calculé, pour deux arêtes e et e', si le noeud source de e' est un sous-ensemble de l'ensemble des paires du noeud destination de e, alors l'arête résultante sera $(n_s(e), l(e)l(e'), n_d(e) \setminus n_s(e'))$. Ce processus est équivalent à un BFS dans l'arbre des successeurs de l'EA, lorsque nous concaténons les étiquettes des arêtes pour dériver les DSs. La Figure 4.3 montre l'étape de dérivation de l'arbre successeur présenté dans la Figure 4.1. La première itération de l'étape de dérivation traite l'arbre des successeurs reçu de l'étape d'intersection et fait correspondre ses arêtes à différents reducers en suivant le schéma de mappage décrit précédemment. Par exemple, dans la Figure 4.3, la paire (s_2, s_3) dans le noeud n_3 fait correspondre toutes les arêtes ayant les formes suivantes $((s_2, s_3), *, \perp)$, $((s_2, s_3), *, \{\})$ ou $(*, *, (s_2, s_3))$ à un reducer donné. Alors, le produit cartésien entre les arêtes ayant la forme $(*, *, (s_2, s_3))$ et les arêtes ayant la forme $((s_2, s_3), *, \{\})$ ou $((s_2, s_3), *, \perp)$ est calculé dans le même reducer pour produire des arêtes compactes ayant la forme $(*, **, \{\})$ ou $(*, **, \perp)$. Ainsi, le reducer représenté par (s_2, s_3) produit l'arête $((s_0, s_1), ba, \{\})$ à partir de deux arêtes $((s_0, s_1), b, (s_2, s_3))$ and $((s_2, s_3), a, \{\})$. Ensuite, cette arête est affectée, lors de la deuxième itération, au reducer associé à la clé (s_0, s_1). Le même processus sera répété de manière itérative dans les prochains itérations MapReduce de l'étape de dérivation jusqu'à ce que la condition d'arrêt soit vraie. Enfin, si une DS n'existe pas et que le nombre d'itérations est inférieur à la limite maximale, nous répétons l'étape de dérivation en considérant la sortie comme une entrée de l'itération suivante, sinon la sortie finale est l'ensemble des paires ainsi que leurs courtes DSs si elles existent.

Dans la section suivante, nous présentons une analyse de coût de communication dans le framework MapReduce pour ce problème en utilisant le modèle introduit par d'Afrati *et al.* [3].

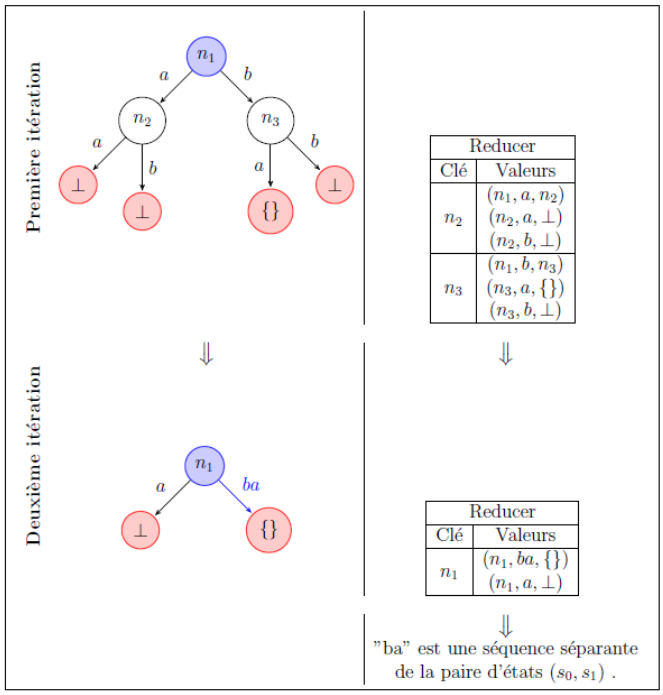

FIGURE 4.3 – L'étape de dérivation de $M/s_0 \cap M/s_1$.

4.3.2 Analyse de coût de communication

Le modèle de coût de communication introduit par Afrati *et al.* [3] offre une bonne méthode pour analyser les problèmes et optimiser les performances de tout environnement de calcul distribué par une étude de la relation de compromis entre le coût de communication et le degré de parallélisme. En appliquant ce modèle à un framework MapReduce, nous pouvons déterminer le meilleur algorithme pour un problème en analysant le compromis entre la taille du reducer et le coût de communication pour un algorithme en MapReduce. Deux paramètres représentent ce compromis pour concevoir un algorithme optimal en MapReduce : Le premier est la taille du reducer, désignée par q, qui représente la taille de la plus grande liste de valeurs associées à une clé qu'un reducer peut recevoir. Le second paramètre est le coût de communication entre l'étape map et l'étape reduce. Le coût de communication, désigné par r, est défini comme le nombre moyen de paires clé-valeur que les mappers créent à partir de chaque entrée.
Formellement, supposons que nous avons p reducers et $q_i \leq q$ entrées sont affectées au i^{th} reducer. Soit $|In|$ le nombre total d'entrées différentes, alors le taux de réplication est donné par la formule $r = \sum_{i=1}^{p} q_i/|In|$ [3].

À partir de [3], nous calculons une limite inférieure du taux de réplication pour l'intersection de FSMs en fonction de q par la formule suivante :

$$r \geq \frac{q \times |Out|}{g(q) \times |In|}$$

où $|In|$ désigne la taille de l'entrée, $|Out|$ la taille de la sortie, et $g(q)$ le nombre de sorties qui peuvent être produites par un reducer de taille q. Puisque nous considérons une FSM non déterministe observable complète, nous avons $|In| = |E|$ and $|Out| = \frac{n(n-1)}{2} \times |I|$. Donc

Proposition 5. *La borne inférieure du taux de réplication est comme suit :*

$$r \geq \frac{n(n-1)}{2} \times \frac{q \times |I|}{g(q) \times |E|}$$

Notons que la limitation de la taille des reducers permet plus de parallélisme. Une petite taille des reducers nous oblige à redéfinir le schéma de mappage afin de permettre un plus grand nombre de reducers de petites tailles, et ainsi permettre plus de parallélisme en utilisant les nœuds disponibles.

4.3.3 Algorithme MapReduce pour l'étape d'intersection

Dans cette section, nous présentons l'implémentation MapReduce de l'étape d'intersection d'une version modifiée des algorithmes proposés dans [69]. Notons que notre approche produit un arbre des successeurs, également appelé table des successeurs, pour toutes les paires d'états d'une FSM non déterministe observable complète. Les expériences conduites dans [62] montrent que lors de la dérivation des séquences séparantes, le temps de construction de l'arbre des successeurs prend 96% du temps global de l'EA. C'est pourquoi trois méthodes seront présentées dans cette section pour la construction de l'arbre des successeurs.
L'Algorithme 4 ci-dessous contient la définition des fonctions map et reduce de l'étape d'intersection. La fonction map produit un ensemble de clés basé sur un schéma défini à partir des transitions de la FSM d'entrée. La fonction de reduce effectue l'intersection des transitions reçues depuis la phase map.

Les méthodes de mappage proposées sont basées respectivement : sur les états, sur les symboles de l'alphabet d'entrée, et à la fois sur les états et les symboles de l'alphabet d'entrée.
Formellement, soit $M = (S, I, O, E)$ une FSM non déterministe observable complète ayant n états et $t = (s, i, o, d)$ une transition dans E. Un mapper produit un ensemble de clés à partir de la transition t en utilisant une fonction de hachage

Algorithme 4 : Étape d'intersection avec MapReduce

Entrée : $M = (S, I, O, E)$, une FSM non déterministe observable complète.
Sortie : Arbre, un arbre des successeurs de toutes les paires d'états.

1 **Fonction** Map(*clé*, *valeur*) **:**
  Data : Paire $\langle clé, valeur \rangle$, où *clé* représente un identifiant d'instance arbitraire et *valeur* est une transition $t \in E$.
  Result : Collection de paires $\langle k, t \rangle$, où k est une clé associée à la transition t.
  // créer la transition t à partir de la valeur d'entrée
2 t = getTransitionFrom(*valeur*)
  // générer l'ensemble des clés associées à la transition t
3 *clés* = getKeysFromTransition(t)
  // répliquer la transition t
4 **foreach** *k in clés* **do**
5 Emit (k,t)

6 **Function** Reduce(*key*, *valeurs*) **:**
  Data : Paire $\langle clé, valeurs \rangle$, où *valeurs* est un ensemble de transitions ayant la même clé *clé*.
  Result : Partie dérivée de l'arbre des successeurs.
  // Calculer l'intersection de l'ensemble des transitions de valeurs et les ajouter à l'arbre des successeurs Arbre
7 **foreach** $(t, t') \in valeurs \times valeurs$ **do**
8 **if** $I[t] = I[t']$ *and* $O[t] = O[t']$ **then**
9 ajouter l'arête $((s[t], s[t']), I[t], (d[t], d[t']))$ à Arbre

h. Cette fonction de hachage est intégrée dans la fonction `getKeysFromTransition()` de la Ligne 3 de l'Algorithme 4. Dans la suite, nous détaillons les trois méthodes de mappage utilisées dans la fonction `getKeysFromTransition()`.

Méthode de mappage basée sur les états

Dans la première méthode de mappage, à partir d'une transition $t \in E$, les mappers produisent un ensemble de paires clé-valeur de la forme $\langle clé, t \rangle$, où $clé = \langle h_S(s[t]), s \rangle$, pour tout $s \in S$ tel que $s[t] \neq s$ et h_S une fonction de hachage définie de S vers l'ensemble $\{1, \cdots, n\}$. Dans ce cas, nous avons $\frac{n(n-1)}{2}$ reducers. Dans cette méthode, la fonction $g(q)$, qui est le nombre de sorties pouvant être produites par un reducer de taille q, peut être affectée par la présence de transitions avec des étiquettes différentes à l'intérieur du même reducer. Formellement, puisque nous considérons une FSM non déterministe observable complète, on a $q \leq 2 \times (|I| + |I| \times |O|)$ et $g(q) = |I|$. Ainsi, la proposition suivante donne la borne supérieure du taux de réplication pour cette méthode.

Proposition 6. *Le taux de réplication r dans le schéma de mappage basé sur les états est*

$$r \leq (n-1)$$

Méthode de mappage basée sur les alphabets d'entrée

Dans la deuxième méthode, nous avons un reducer pour chaque alphabet d'entrée. Ainsi, le nombre de reducers est égal à la taille de l'alphabet d'entrée $|I|$. Les mappers enverront chaque transition t au reducer correspondant à son symbole d'entrée $I[t]$.

Plus précisément, à partir d'une transition $t \in E$, les mappers produisent un ensemble de paires clé-valeur ayant la forme $\langle clé, t \rangle$, et $clé = h_{In}(I[t])$ tel que h_{In} soit une fonction de hachage définie de I à $\{1, \cdots, |I|\}$. Nous aurons maintenant $g(q) = \frac{n(n-1)}{2}$, et $q \leq n + n \times |O|$.
En supposant que les symboles de l'alphabet sont uniformément distribués, donc nous avons

Proposition 7. *Le taux de réplication dans le schéma de mappage basé sur les alphabets d'entrée est optimal et égal à* 1.

Méthode de mappage hybride, basée sur les états et les alphabets d'entrée

Dans la dernière méthode, nous proposons un mappage hybride entre la première et la deuxième méthode. En d'autres termes, les clés seront basées sur les états et les alphabets d'entrée en même temps. Alors, on considère la forme

de la clé *clé*= $(s[t], s, I[t])$ tel que $s \in S$ et $s \neq s[t]$. Le nombre de reducers dans ce cas est égal à $\frac{n(n-1)}{2} \times |I|$, la taille du reducer $q \leq 2 \times |O|$ et chaque reducer produira au plus un arc de l'arbre des successeurs. Par conséquent, nous déduisons la borne supérieure du taux de réplication dans la proposition suivante.

Proposition 8. *Le taux de réplication dans la méthode de mappage hybride est* $r \leq (n-1)$.

Théorème 1. *L'algorithme 4 calcule l'arbre des successeurs pour toutes les paires d'états d'une FSM* $M = (S, I, O, E)$ *en une seule itération MapReduce et une borne inférieure de coût de communication égale à* $\frac{n(n-1)}{2} \times \frac{q \times |I|}{g(q) \times |E|}$, *où* $n = |S|$.

Démonstration. Dans la fonction map de l'algorithme 4, `getTransitionFrom(t)` renvoie l'ensemble des clés associées à la transition t par rapport à une méthode de mappage donnée. Ensuite, il envoie la transition t à tous les reducers indexés par ces clés. Afin de s'assurer que l'algorithme construit correctement l'arbre des successeurs, il est nécessaire d'avoir la propriété (*) : toutes les transitions avec le même symbole d'entrée et de sortie dans le même reducer. Ensuite, la fonction de reducer calcule leur intersection par paire pour étendre l'arbre des successeurs. En utilisant les méthodes de mappage proposées, nous avons :

— pour le mappage basé sur les états, un reducer $\langle s_i, s_j \rangle$ reçoit des mappers toutes les transitions partant de l'état s_i ou s_j ; en conséquence, toutes les transitions sortantes de l'état s_i ou de l'état s_j sont à l'intérieur de ce reducer. Ensuite, la propriété (*) est vérifiée en utilisant cette méthode de mappage.
— pour le mappage basé sur les alphabets d'entrée, un reducer reçoit des mappers les transitions ayant le même symbole d'entrée c ; Puis à l'intérieur d'un reducer, nous avons aussi toutes les transitions ayant le même symbole d'entrée et de sortie.
— pour le mappage hybride, un reducer $\langle s_i, s_j, c \rangle$ reçoit des mappers des transitions ayant le même symbole d'entrée c et partant de l'état s_i ou s_j ; alors la propriété (*) est évidemment vérifiée.

La borne inférieure de coût de communication est une conséquence de la Proposition 5. □

4.3.4 Algorithme MapReduce pour l'étape de dérivation

Dans cette étape, nous utilisons un algorithme MapReduce itératif pour dériver un ensemble de courtes DSs pour toutes les paires d'états d'une FSM non déterministe observable complète. À chaque itération, plusieurs mappers s'exécutent en parallèle et produisent une collection de paires $\langle clé, edge \rangle$. Ensuite,

Algorithme 5 : Étape de dérivation avec MapReduce

Entrée : L'arbre des successeurs de la FSM M.
Sortie : Une DS la plus courte pour chaque paire d'états.

```
1  Function Map(clé, valeurs) :
       Data : Paire ⟨clé, valeur⟩, où clé représente un identifiant d'instance
              arbitraire et valeur est une arête de l'arbre des successeurs.
       Result : Collection de paires ⟨k, arête⟩, où k est une clé associée à
                l'arête arête.
       /* créer l'arête arête à partir de la valeur d'entrée */
2      arête = getEdgeFrom(valeur)
3      if n_d (arête) est vide then
4          k = n_s (arête)
5          Emit (k, arête)
6      else
7          listeClés = n_d (arête)
           /* listeClés est un ensemble de paires d'états      */
8          foreach k in listeClés do
9              Emit (k, arête)

10 Function Reduce(clé, valeurs) :
       Data : Paire ⟨clé, valeurs⟩, où valeurs est un ensemble d'arêtes
              ayant la même clé clé.
       Result : Arbre des successeurs moins les feuilles.
       /* Diviser valeurs en deux sous-ensembles disjoints
       d'arêtes valeurs_1 et valeurs_2.                          */
11     valeurs_1 = {e ∈ valeurs | n_d(e) n'est pas vide}
12     valeurs_2 = valeurs \ valeurs_1
13     foreach (e_1, e_2) in valeurs_1 × valeurs_2 do
           /* construire un suffixe commun d'un ensemble de
           séquences séparantes                                  */
14         l(e_1) = l(e_1)l(e_2)
15         n_d (e_1) = n_d (e_1) \ n_s (e_2)
16     Emit (null, e_1)
```

plusieurs reducers s'exécutent en parallèle et traite l'entrée reçue de la phase de mapper pour trouver une DS, si elle existe.

Dans cette étape, nous dérivons une courte DS pour chaque paire d'états. Cette étape reçoie de l'étape d'intersection $\frac{n(n-1)}{2} \times |I|$ arêtes d'arbre des successeurs, ensuite, produit $\frac{n(n-1)}{2}$ paires d'états ainsi que des courtes DSs associées s'elles existent. Pour cela, nous utilisons une seule méthode de mappage basée sur les états. Chaque fonction map prend en entrée une arête e de l'arbre des successeurs et produit un ensemble de paires $\langle clé, e \rangle$. où la clé *clé* est la paire d'états du nœud source $n_s(e)$ si le nœud de destination $n_d(e)$ est vide, sinon l'ensemble de clés est l'ensemble des paires d'états du nœud de destination de l'arête e.

Dans ce cas, nous avons $\frac{n(n-1)}{2}$ reducers où chaque reducer pourrait contenir $q \leq (\frac{n(n-1)}{2}) \times |I|$ arêtes. Le nombre de sorties pouvant être produites est alors $g(q) = 1$ dans la dernière itération.

Proposition 9. *Le taux de réplication r dans chaque itération MapReduce de l'étape de dérivation est :* $r \leq \frac{n(n-1)}{2}$

Théorème 2. *L'algorithme 5 dérive une courte DS, si elle existe, pour toutes les paires d'états d'une FSM* $M = (S, I, O, E)$ *en mn MapReduce itérations au maximum, où* $m = |E|$ *et* $n = |S|$.

Démonstration. Par construction, il est claire qu'une DS est l'étiquette d'un chemin allant du nœud racine à un nœud feuille indexé par l'ensemble vide {} dans l'arbre des successeurs. Pendant chaque itération MapReduce de l'Algorithme 5, l'arbre des successeurs *Arbre* est compacté par niveau jusqu'à atteindre la condition d'arrêt. Sans perte de généralité, considérons un nœud feuille ne_k indexé par l'ensemble vide qui est situé au $k$ième niveau de l'arbre des successeurs, et soit $prec(n)$ l'ensemble des nœuds prédécesseurs du nœud n. Dans chaque itération MapReduce, le nœud ne_k remplace tous les nœuds situés au niveau $k-1$, appartenant à l'ensemble $prec(ne_k) = \{ne^1_{k-1}, \cdots, ne^l_{k-1}\}$. En conséquence, l'arbre des successeurs est compacté de la manière suivante : l'étiquette x d'une arête (ne^i_{k-1}, x, ne_k) est concaténée avec toutes les étiquettes de l'ensemble des arêtes $\{(n, y, ne^i_{k-1} \mid n \in prec(ne^i_{k-1})\}$, pour tout $1 \leq i \leq l$, pour produire l'ensemble des arêtes $\{(n, yx, ne_k \mid n \in prec(ne^1_{k-1})\}$. Le nombre d'itérations MapReduce dans l'Algorithme 5 est lié à la condition d'arrêt qui est vraie lorsque le nœud racine de l'arbre des successeurs est atteint et qu'un ensemble des DSs est dérivé, si elles existent, en moins de mn itérations. □

4.4 Implémentation et résultats expérimentaux

Cette section comprend des expériences approfondies afin d'évaluer l'efficacité des méthodes proposées en termes de coût de communication et temps d'exécution.
Les expériences sont conduites sur des FSMs non déterministes observables complètes générées de manière aléatoire par rapport aux nombre d'états, la taille des alphabets d'entrée et de sortie, le degré de non déterminisme et le *rang*. Nous menons cinq expériences différentes puis nous calculons et représentons la moyenne des résultats obtenus dans les figures correspondantes. Enfin, nous comparons les méthodes proposées en termes de coût de communication et temps d'exécution requis dans le framework MapReduce pour dériver l'arbre des successeurs, et pour extraire une courte DS pour chaque paire d'états si elle existe.

4.4.1 Configuration de cluster de calcul

Nos expériences ont été réalisées avec la plateforme Hadoop 2.7. sur le cluster de calcul scientifique français Grid'5000 [33]. Nous avons utilisé pour nos expériences un cluster composé de 15 nœuds, 30 CPUs, 300 cœurs. Chaque nœud est une machine équipée de deux processeurs Intel Xeon E5-2630 v4 à 10 cœurs par processeur, 256 Go de mémoire principale, et deux disques durs (HDD) de 300 Go. Les machines sont connectées par un réseau Ethernet 10 Gbps et fonctionnent sous Debian 9 64-bits.

4.4.2 Méthode de génération de données

Nous avons généré de manière aléatoire une grande variété d'ensembles de données en deux phases en fonction de différentes combinaisons de paramètres tels que la taille de l'alphabet d'entrée $|I|$, la taille de l'alphabet de sortie $|O|$, le nombre d'états $|S|$, le degré de non déterminisme D et le rang R. Premièrement, nous avons généré aléatoirement des automates finis déterministes complets de manière en utilisant la méthode décrite dans la competition *Abbadingo One* (*voir* `http://abbadingo.cs.nuim.ie`). Et après, nous avons sélectionné $\{(D \times |S| \times |I|)\}$ transitions pour un degré de non determinsme D égal à 20, 40, 60 ou 80. Enfin, nous ajoutons à la FSM non déterministe obtenue la propriété d'observabilité, en dupliquant aléatoirement $(\frac{R \times |O|}{100})$ chaque transition où le rang d'observabilité R est égale respectivement à 20-30, 40-50, 60-70 ou 80-90.
Le Tableau 4.1 présente tous les jeux de données utilisés dans nos expériences ainsi que leurs propriétés respectives. Afin d'obtenir des résultats précis, nous

avons généré cinq échantillons $S_i^1, \cdots, S_i^5$ pour chaque jeu de données S_i du Tableau 4.1. Les résultats des différentes expériences sont la moyenne des résultats obtenus à partir de ces échantillons.

4.4.3 Analyse du coût de communication

Le coût de communication est égal au nombre total de paires clé-valeur transférées de la phase map à la phase reduce. Il peut être optimisé en minimisant le paramètre de taux de réplication, c'est-à-dire, le nombre de copies d'entrée envoyées aux reducers.

Dans le tableau suivant, nous avons sélectionné quelques jeux de données pour étudier la relation entre les tailles des jeux de données considérés et le coût de communication :

Comme nous avons introduit trois méthodes de mappage dans l'étape d'intersection de notre approche, nous présentons le coût de communication pour les jeux de données considérés dans la figure 4.4.

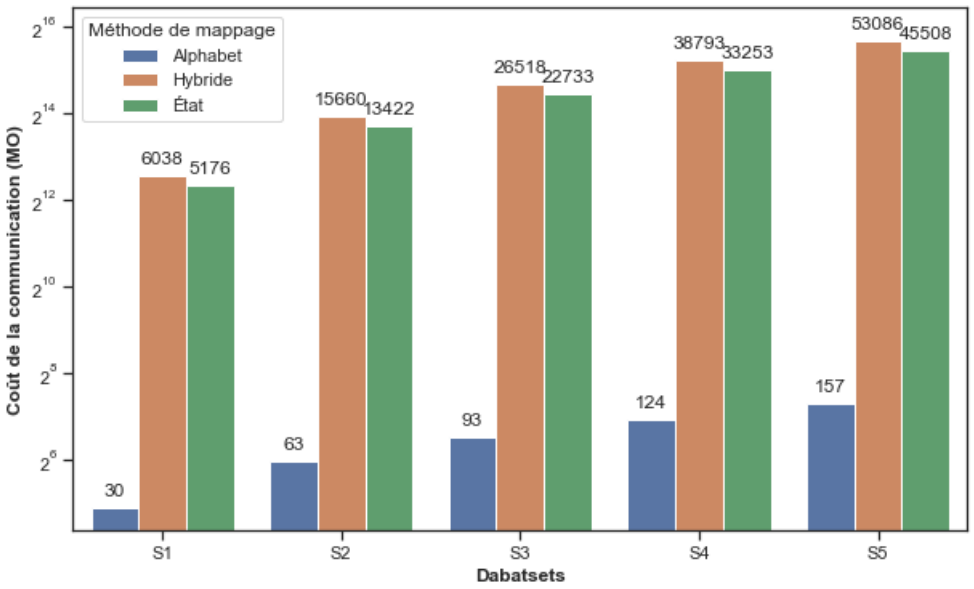

FIGURE 4.4 – Coût de communication des trois méthodes proposées pour l'étape d'intersection

Dans la figure 4.4, les résultats obtenus montrent clairement que la méthode de mappage basée sur l'alphabet d'entrée est plus performante que les autres méthodes de mappage en termes de coût de communication. Cela est dû au fait que le nombre de copies de transition envoyées aux reducers en utilisant

Paramètre	Base de données	Taille de fichier (MB)	$\lvert E\rvert$ ($\times 10^6$)	$\lvert S\rvert$	$\lvert I\rvert$	$\lvert O\rvert$	D (%)	R (%)
Nombre de transitions	S_1	14.24	1	143	143	143	70	50
	S_2	29.86	2	180	180	180	70	50
	S_3	44.74	3	205	205	205	70	50
	S_4	59.94	4	225	225	225	70	50
	S_5	76.22	5	243	243	243	70	50
Degré de non-déterminisme	S_6	13.26	0.896	200	200	200	20	50
	S_7	26.88	1.816	200	200	200	40	50
	S_8	38.61	2.608	200	200	200	60	50
	S_9	51.28	3.464	200	200	200	80	50
Taille de l'alphabet d'entrée	S_{10}	17.41	1.203	250	50	250	70	50
	S_{11}	35.82	2.458	250	100	250	70	50
	S_{12}	49.99	3.345	250	150	250	70	50
	S_{13}	78.06	5.160	250	200	250	70	50
	S_{14}	93.64	6.144	250	250	250	70	50
Rang	S_{15}	14.51	0.980	200	200	200	50	20
	S_{16}	27.51	1.860	200	200	200	50	40
	S_{17}	38.19	2.580	200	200	200	50	60
	S_{18}	50.01	3.380	200	200	200	50	80
Nombre d'états	S_{19}	4.62	0.367	100	100	100	70	50
	S_{20}	10.82	0.790	200	100	100	70	50
	S_{21}	17.53	1.248	300	100	100	70	50
	S_{22}	21.68	1.524	400	100	100	70	50
	S_{23}	29.83	2.080	500	100	100	70	50
	S_{24}	31.73	2.202	600	100	100	70	50
	S_{25}	40.69	2.813	700	100	100	70	50
	S_{26}	44.19	3.048	800	100	100	70	50
	S_{27}	49.82	3.429	900	100	100	70	50
	S_{28}	58.51	4.020	1000	100	100	70	50

Tableau 4.1 – Jeux de données utilisés dans les différentes expériences

Base de données	Taille de sortie de l'intersection = Taille de l'entrée de la dérivation (MO)	Étape de dérivation			
		Phase de dérivation		Phase de post-traitement	
		Communication Coût (GO)	Taille de la sortie (GO)	Communication Coût (GO)	Taille de la sortie (GO)
S_1	138	4.8	188	211.4	1
S_2	357	15.7	793.8	869	1.5
S_3	603	29.8	1738.7	1882.5	2
S_4	882	47.9	3078.5	3308.6	2.4
S_5	1205	70.7	4930.4	5270	3

TABLEAU 4.2 – Coût de communication des trois méthodes proposées pour l'étape d'intersection

cette méthode est inférieur aux autres selon la proposition 7. Dans certains cas particuliers de FSMs, lorsque le nombre d'états est inférieur à la taille de l'alphabet d'entrée, le mappage basé sur les états a un coût de communication plus réduit. Cela correspond aux résultats des propositions 6 et 8.

4.4.4 Analyse du coût de calcul

Le coût de calcul est le temps nécessaire pour exécuter une tâche MapReduce. Les graphes ci-dessous présentent les résultats comparatifs en termes de temps d'exécution des méthodes proposées en variant différents paramètres tels que le nombre d'états $|S|$, la taille des alphabets d'entrée $|I|$, la taille des alphabets de sortie $|O|$, le degré de non déterminisme D, le rang R, et le nombre total de transitions. Notons que nous fournissons le temps d'exécution réel sur le cluster de calcul considéré.

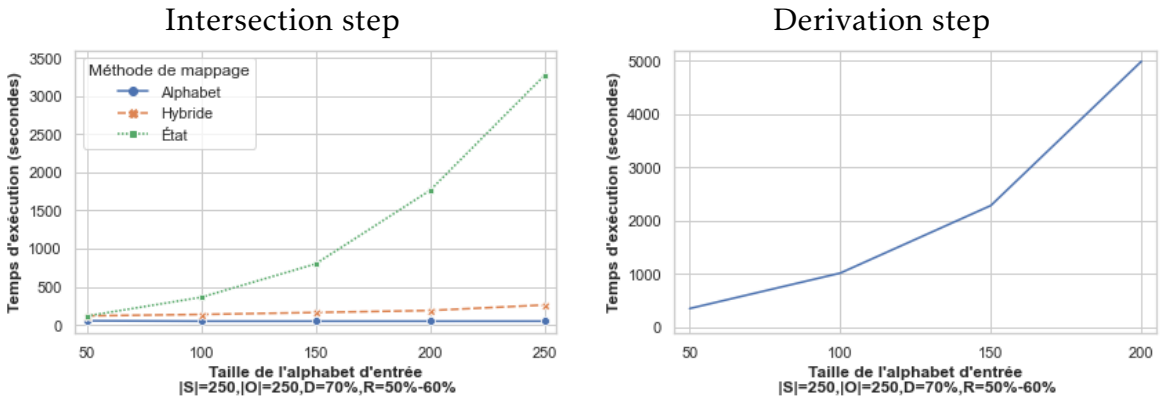

FIGURE 4.5 – Temps d'exécution versus la taille de l'alphabet d'entrée

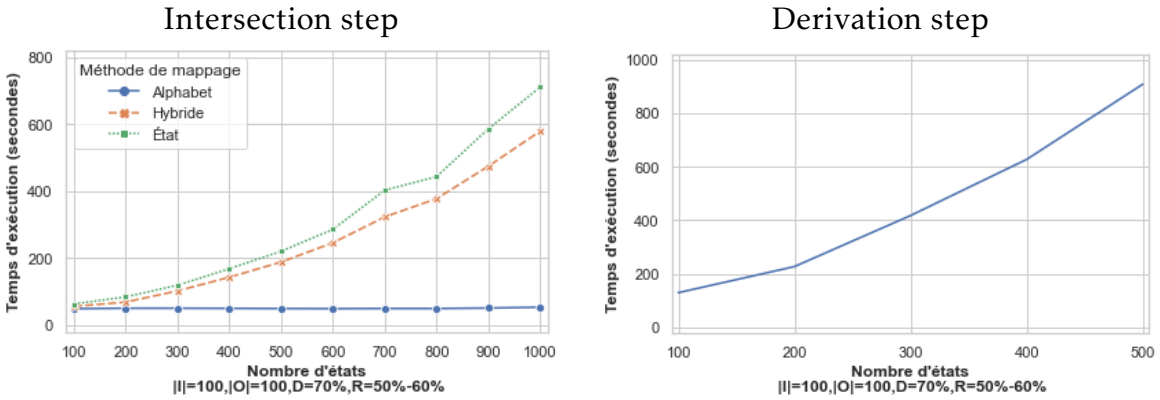

FIGURE 4.6 – Temps d'exécution versus le nombre d'états

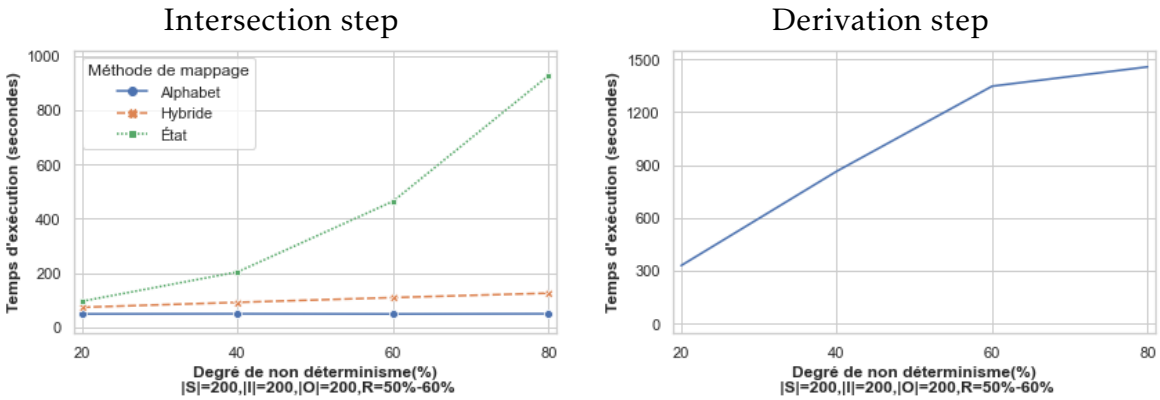

FIGURE 4.7 – Temps d'exécution versus le degré de non déterminisme

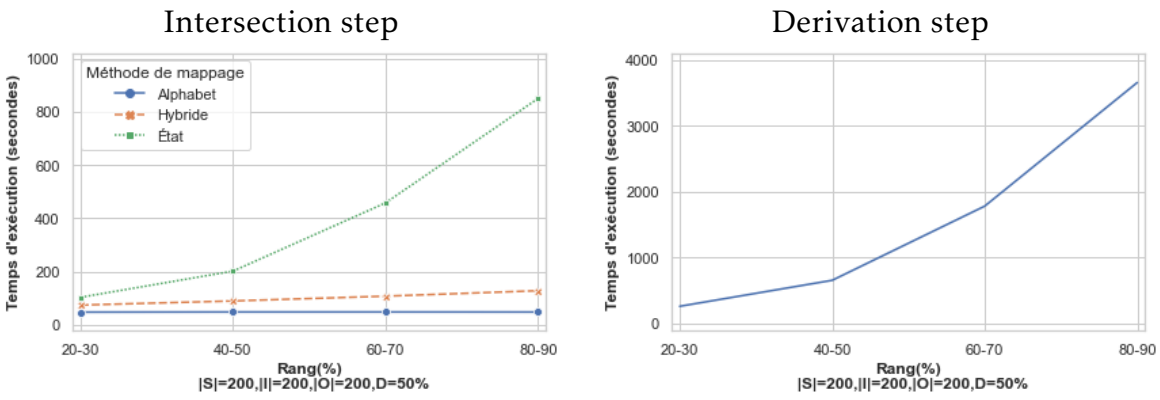

FIGURE 4.8 – Temps d'exécution versus le rang

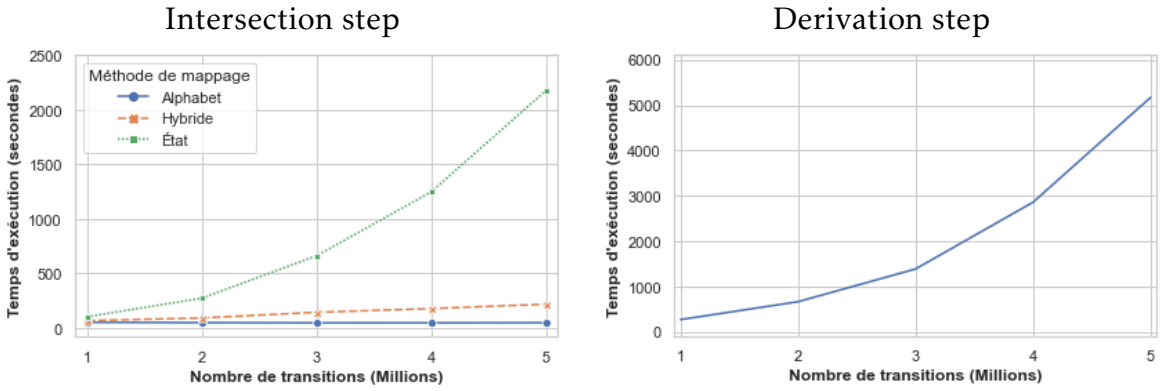

FIGURE 4.9 – Temps d'exécution versus le nombre de transitions

Les Figures 4.5, 4.6, 4.7, 4.8 et 4.9 nous montrent des courbes croissantes qui présentent le temps d'exécution des méthodes proposées pour l'étape de l'intersection et dérivation pour différents jeux de données en variant la taille de l'alphabet d'entrée dans la Figure 4.5, le nombre d'états dans la Figure 4.6, le degré de non déterminisme dans la Figure 4.7, le rang dans la Figure 4.8 et le nombre de transitions dans la Figure 4.9. La méthode basée sur l'alphabet d'entrée est nettement plus efficace lorsque la taille de l'alphabet est inférieure ou égale au nombre de reducers disponibles car chaque alphabet définit un reducer. Cependant, si nous avons un grand nombre de ressources, cela implique un gaspillage de celles-ci et chaque reducer peut contenir beaucoup de transitions qui viennent avec le même alphabet d'entrée à partir d'états différents. La méthode de Mappage hybride est offre plus de parallélisme par rapport aux autres méthodes où chaque reducer reçoit les transitions nécessaire pour produire le résultat conduisant ainsi à une durée de fonctionnement réduite. Par ailleurs, cela peut prendre un temps considérable dans la phase de Mappage afin de dupliquer chaque transition et demandera par conséquent un nombre également considérable de reducers par rapport aux ressources disponibles. Ce qui induit un temps d'attente pour un ensemble de reducers, ainsi une augmentation du temps d'exécution globale. La méthode de mappage basée sur les états, est la moins performante, car elle produit un taux de réplication et un nombre de reducers importants.
Dans l'étape de dérivation, nous avons proposé une seule méthode de mise en correspondance, qui est basée sur les états. Les résultats montrent que le temps d'exécution est presque linéaire dans tous les cas. Selon les résultats des expériences, minimiser le taux de réplication diminue le temps utilisé par les mappers pour répliquer chaque transition, et évite une lecture/écriture des résultats intermédiaires. En même temps, cela réduit le nombre de transitions

qui sont envoyées à un reducer.

D'autre part, l'utilisation d'un nombre suffisant de reducers diminue le temps d'attente d'un reducer pour utiliser un CPU. Par conséquent, nous obtenons un schéma MapReduce parallèle optimal pour produire un ensemble de courtes DSs pour une FSM non déterministe observable complète.

4.4.5 Étude comparative

Nous avons réalisé également un ensemble d'expériences comparatives pour évaluer l'efficacité et l'évolutivité des méthodes proposées par rapport aux approches de l'état de l'art. Pour cela, nous avons opté pour la métrique speedup, qui est définie comme la rapidité de la méthode parallèle par rapport à la méthode séquentielle, pour évaluer la performance de nos méthodes MapReduce avec une approche basée sur un calcul multi-threads [92]. Nous avons conduit des expériences sur un seul nœud du cluster décrit précédemment. Pour chaque expérience, nous exécutons différentes méthodes cinq fois sur 40 threads afin de déterminer celle qui donne les meilleures performances et analyser l'effet du nombre de transitions et du rang de non déterminisme de la FSM selon la métrique speedup. Nous avons alors comparé les trois méthodes MapReduce proposées précédemment dans l'étape d'intersection de notre approche avec une implémentation parallèle multi-threads basée sur OpenMP de l'étape séquentielle Algorithme Exact (EA) sur un CPU multicore [71] en utilisant OMP4J [30]. OMP4J est une implémentation open-source d'OpenMP. Les expériences confirment les résultats précédents. La figure 4.10 illustre l'accélération pour différents jeux de données du tableau 4.2. On peut voir que la méthode de mappage basée sur les alphabets d'entrée est plus efficace pour les différents jeux de données. La figure 4.11 montre l'accélération en fonction des rangs de non déterminisme R 50–60%, 60–70%, 70–80%, et 80–90%. Ces expériences montrent des gains de performance constants pour la méthode basée sur les alphabets dans différentes situations. Cette performance est due au faible taux de réplication entre map et reduce pour cette méthode, et aussi au fait que chaque reducer ne reçoit que les transitions utiles pour le calcul du résultat final. De plus, les performances de la méthode basée sur les alphabets d'entrée sont plus rapides que celles de la méthode basée sur OpenMP, et l'accélération de cette méthode croît de manière exponentielle lorsque le nombre de transitions et le rang de non déterminisme de la FSM augmentent.

Nous avons obtenu des résultats satisfaisants prouvant que l'approche proposée pour dériver de courtes DSs à partir de FSMs non déterministes. Cependant, MapReduce n'est pas conçu pour le traitement itératif, et les données doivent être écrites sur le disque après chaque itération, ce qui augmente considérablement le nombre d'E/S sur le disque. Combiner les résultats des différents nœuds après

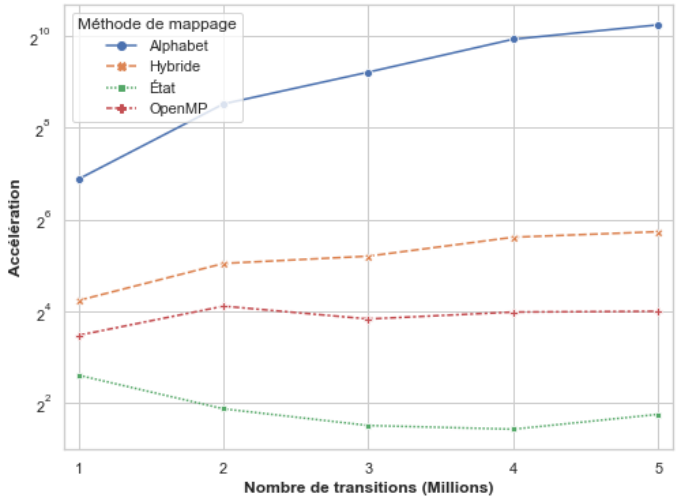

FIGURE 4.10 – Accélération versus le nombre de transitions

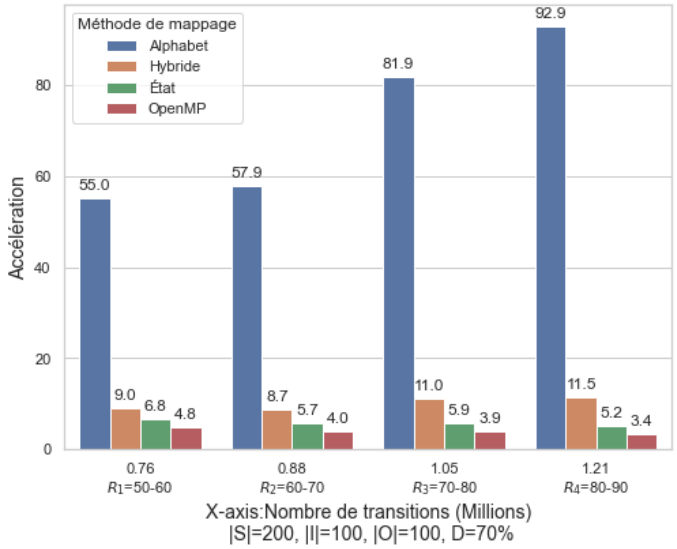

FIGURE 4.11 – Accélération versus le rang du non déterminisme.

chaque itération représente un défi important en raison de la structure complexe du réseau. Pour surmonter cette limitation dans l'étape de dérivation, nous pouvons utiliser le framework I2 MapReduce [160], qui introduit un modèle de graphe bipartite (MapReduce Bipartite Graph) pour représenter les calculs itératifs et incrémentaux, et qui contient une boucle entre les mappers et les reducers.

Résumé

Les FSMs sont largement utilisées dans divers domaines d'application, tels que les protocoles de communication, la recherche de motifs, les contrats intelligents, la génomique et les systèmes réactifs. Les tests de conformité sur des FSMs consistent à dériver des séquences séparantes, puis à les appliquer à une machine ou à une implémentation sous test IUT afin d'identifier l'état du système. De nombreux travaux ont considérés la classe des FSMs déterministes. Cependant, les systèmes actuels sont souvent modélisés par des FSMs non déterministes, dans lequelles une DS peut atteindre une limite exponentielle ce qui pose un chanllenge de sa construction à grande échelle.

Dans ce travail, nous avons abordé le problème de scalabilité rencontré lors de la dérivation des DSs à partir des machines à états finis (FSMs) complètes, observables et non déterministes, en introduisant une version MapReduce massivement parallèle de l'Elgorithme Exact. Notre approche est basée sur deux étapes MapReduce : l'étape d'intersection et l'étape de dérivation de courtes DSs. Dans la première étape, nous avons proposé trois méthodes MapReduce basées respectivement sur un mappage basé sur les états, un mappage basé sur les alphabets d'entrée et un mappage hybride basé à la fois sur les états et les alphabets d'entrée. L'introduction de trois méthodes est justifiée par le fait que le temps nécessaire pour cette étape représente environ 96% du temps total dans l'EA. Dans la deuxième étape, un algorithme MapReduce itératif est introduit pour dériver un ensemble de courtes DSs. Pour les deux étapes, nous avons analysé le coût de communication en utilisant le modèle d'Afrati *et al.* qui offre un aspect formel du compromis inhérent entre le coût de communication et le degré de parallélisme dans un environnement de calcul distribué. Nous avons mené des expériences approfondies sur une large classe des FSMs générés aléatoirement. Ces expériences sont évaluées en termes de coût de communication, temps d'exécution et l'accélération. Nous avons également comparé les résultats des algorithmes proposés avec une approche basé sur un calcul parallèle en multi-threads en termes d'accélération. Les résultats obtenus montrent que l'algorithme proposé est efficace et beaucoup plus scalable : Notre algorithme MapReduce a été capable de traiter des FSMs ayant jusqu'à 5 millions de transitions,

ce qui représente 150 fois plus grand que les algorithmes existants de génération des DSs et avec une accélération 2^6 plus que l'algorithme d'intersection basé sur OpenMP.

Conclusions et perspectives

Dans ce travail de thèse, nous nous sommes intéressé au domaine de traitement parallèle de données massives, et plus précisément le domaine de calcul massivement parallèle appliqué à quelques problèmes en théorie des automates finis. C'est dans cette optique que nous avons mené à bien notre travail, dans le but de développer de nouveaux algorithmes parallèles pour les problèmes considérés en utilisant le modèle de calcul massivement parallèle en MapReduce. L'étude faite a montrée l'adaptabilité de ce modèle pour le calcul parallèle à grand échelle en théorie des automates. Les sections suivantes présentent les principales conclusions, contributions et perspectives obtenues tout au long de cette thèse. Le présent travail traite deux problématiques à savoir :

La composition des machines à états finis pondérées

la NP-dureté du problème de composition des machines à états finis pondérées [154, 9, 119] présente un défi qui nécessite la conception de méthodes efficaces en considérant plus que deux transducteurs à grande échelle. Nous introduisons pour la première fois un algorithme parallèle pour la composition de machines à états finis pondérées en utilisant le modèle CMP-MR.

La génération de séquences séparantes des états dans les machine à états finis non déterministes

Quant au deuxième problème, il consiste en la construction parallèle en MapReduce des séquences séparantes minimales des différents états deux à deux d'une machine à états finis non déterministe observable complet en utilisant également le modèle CMP-MR. Il est à noter que ce dernier problème est classique dans le domaine de test et vérification des logiciels [45, 55, 76, 44, 96]. Les résultats obtenus sont très satisfaisants, ce qui ouvre la voie pour l'étude des autres types de séquences dans le même contexte.

Le travail de recherche de cette thèse a permis d'atteindre des résultats très satisfaisantes pour le traitement parallèle des machines à états finis à grande échelle en utilisant le modèle de calcul massivement parallèle en MapReduce.

Cependant, il reste encore quelques problèmes qui méritent d'être améliorés et de faire l'objet de recherches plus poussées afin de proposer des modèles de calcul optimales et plus efficaces pour un traitement à grande échelle. Par ailleurs, notre étude ouvre une voie prometteuse pour l'utilisation de l'approche proposée pour certaines applications industrielles.

- Décodeur de reconnaissance automatique de la parole : Un décodeur joue un rôle important dans un système de reconnaissance automatique de la parole où il intègre des informations acoustiques et linguistiques pour générer la séquence de mots la plus probable pour un signal vocal d'entrée. Les décodeurs basés sur les machines à états finis pondérées peuvent composer efficacement diverses informations de source, notamment l'acoustique, l'arbre de décision du contexte phonétique, le lexique et le modèle de langue. Grâce à des opérations telles que la détermination et la minimisation, les machines à états finis pondérées rendent les décodeurs très compacts. Elle fonctionnent généralement plus efficacement, de manière concise et élégante par rapport aux autres modèles classiques. Cependant, lorsque les sources de connaissances deviennent énormes, comme un modèle de langage d'ordre élevé ou un lexique de plusieurs millions de mots, les machines à états finis pondérées composées peuvent être inefficace en termes de mémoire ou même infaisable à construire. Nous envisageons alors d'appliquer notre approche de composition massivement parallèle des machines à états finis pondérées pour relever ce défis pour cette application.
- Spark pour les algorithmes MapReduce iteratifs :
  L'implémentation de l'approche proposée pour le génération des séquences séparantes à partir des machines à états finis a été réaliser sur la plateforme Hadoop. Sachant que la plateforme Spark permet d'effectuer un traitement de données massives de manière distribuée en utilisant un moteur d'execution DAG avancé qui supporte un flux de données acyclique et un calcul in-memory. Ses principaux avantages sont sa vitesse, sa simplicité d'usage, et sa polyvalence. Nous envisageons adapter notre solution de génération des séquences séparantes à partir des machines à états finis sur la plateforme Spark afin d'améliorer le temps globale de calcul.
- Génération des séquences de test de conformité à partir des machines à états finis :
  Nous prévoyons également d'étudier les algorithmes massivement parallèles pour la génération des séquences de types UIO, W-method, l'ensemble caractéristique et les séquences de vérification pour les machines à états finis complets/partiels, observable/non obsevabe et déterministes/

non déterministes. Enfin, il est possible d'adapter l'approche proposée aux problèmes d'inférence des machines à états finis.

- Les items fréquents en data mining :
  L'exploration des ensembles d'items fréquents (FIM) est une tâche essentielle dans l'analyse des données, car elle est responsable de l'extraction des événements, des modèles ou des éléments qui se produisent fréquemment dans les données. Cependant, les solutions algorithmiques pour l'extraction de ce type de motifs ne sont pas simples car la complexité de calcul augmente de façon exponentielle avec le nombre d'éléments dans les données. Récemment, des approches de génération des FIM basées sur les machines à états finis ont été introduites en utilisant principalement les opérations de déterminisation et d'intersection. Dans un contexte à grande échelle, notre approche basée sur le modèle CMP-MR peut être une solution efficace pour cette application.

Bibliographie

[1] Foto N Afrati et Jeffrey D Ullman. « Optimizing joins in a map-reduce environment ». In : *Proceedings of the 13th International Conference on Extending Database Technology*. 2010, p. 99-110.

[2] Foto N Afrati et al. « Cluster computing, recursion and datalog ». In : *International Datalog 2.0 Workshop*. Springer. 2010, p. 120-144.

[3] Foto N Afrati et al. « Upper and Lower Bounds on the Cost of a Map-Reduce Computation ». In : *Proceedings of the VLDB Endowment* 6.4 (2013).

[4] Kook Jin Ahn et Sudipto Guha. « Access to data and number of iterations : Dual primal algorithms for maximum matching under resource constraints ». In : *ACM Transactions on Parallel Computing (TOPC)* 4.4 (2018), p. 1-40.

[5] Alfred V Aho et Jeffrey D Ullman. *The theory of parsing, translation, and compiling*. T. 1. Prentice-Hall Englewood Cliffs, NJ, 1972.

[6] Alfred V Aho et al. « An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours ». In : *IEEE transactions on communications* 39.11 (1991), p. 1604-1615.

[7] Shameem Akhter et Jason Roberts. *Multi-core programming*. T. 33. Intel press Hillsboro, 2006.

[8] Ahmed Hussein Ali. « A survey on vertical and horizontal scaling platforms for big data analytics ». In : *International Journal of Integrated Engineering* 11.6 (2019), p. 138-150.

[9] Cyril Allauzen et Mehryar Mohri. « N-way composition of weighted finite-state transducers ». In : *International Journal of Foundations of Computer Science* 20.04 (2009), p. 613-627.

[10] Cyril Allauzen, Mehryar Mohri et Ameet Talwalkar. « Sequence kernels for predicting protein essentiality ». In : *Proceedings of the 25th international conference on Machine learning*. 2008, p. 9-16.

[11] André Almeida et al. « FAdo and GUItar : tools for automata manipulation and visualization ». In : *International Conference on Implementation and Application of Automata*. Springer. 2009, p. 65-74.

[12] Rajeev Alur, Costas Courcoubetis et Mihalis Yannakakis. « Distinguishing tests for nondeterministic and probabilistic machines ». In : *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. 1995, p. 363-372.

[13] Rajeev Alur et David L Dill. « A theory of timed automata ». In : *Theoretical computer science* 126.2 (1994), p. 183-235.

[14] Rajeev Alur et Thomas A Henzinger. « Computer-aided verification : An introduction to model building and model checking for concurrent systems ». In : *Draft, www-cad. eecs. berkeley. edu/~ tah/CavBook* (1998).

[15] Alexandr Andoni, Clifford Stein et Peilin Zhong. « Log Diameter Rounds Algorithms for 2-Vertex and 2-Edge Connectivity ». In : *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[16] Alexandr Andoni et al. « Parallel algorithms for geometric graph problems ». In : *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. 2014, p. 574-583.

[17] Alexandr Andoni et al. « Parallel graph connectivity in log diameter rounds ». In : *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2018, p. 674-685.

[18] R Archana, Ravindra S Hegadi et T Manjunath. « A big data security using data masking methods ». In : *Indonesian Journal of Electrical Engineering and Computer Science* 7.2 (2017), p. 449-456.

[19] HamidReza Asaadi, Dounia Khaldi et Barbara Chapman. « A comparative survey of the HPC and big data paradigms : Analysis and experiments ». In : *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2016, p. 423-432.

[20] Sepehr Assadi et Sanjeev Khanna. « Randomized composable coresets for matching and vertex cover ». In : *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 2017, p. 3-12.

[21] Sepehr Assadi, Xiaorui Sun et Omri Weinstein. « Massively parallel algorithms for finding well-connected components in sparse graphs ». In : *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, p. 461-470.

[22] Sepehr Assadi et al. « Coresets meet EDCS : algorithms for matching and vertex cover on massive graphs ». In : *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2019, p. 1616-1635.

[23] Omer Aziz, Muhammad Shoaib Farooq et Adel Khelifi. « Domain and Challenges of Big Data and Archaeological Photogrammetry With Blockchain ». In : *IEEE Access* 10 (2022), p. 101495-101514.

[24] Bahman Bahmani, Ravi Kumar et Sergei Vassilvitskii. « Densest subgraph in streaming and MapReduce ». In : *Proceedings of the VLDB Endowment* 5.5 (2012), p. 454-465.

[25] Bahman Bahmani et al. « Scalable k-means++ ». In : *Proceedings of the VLDB Endowment* 5.7 (2012), p. 622-633.

[26] Rafael da Ponte Barbosa et al. « A new framework for distributed submodular maximization ». In : *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. Ieee. 2016, p. 645-654.

[27] MohammadHossein Bateni et al. « Distributed Balanced Clustering via Mapping Coresets. » In : *NIPS*. 2014, p. 2591-2599.

[28] Soheil Behnezhad, Mohammad Taghi Hajiaghayi et David G Harris. « Exponentially faster massively parallel maximal matching ». In : *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2019, p. 1637-1649.

[29] Slimane Bellaouar et al. « Weighted Automata Sequence Kernel ». In : *Proceedings of the 9th International Conference on Machine Learning and Computing*. 2017, p. 48-55.

[30] Petr Bělohlávek et Antonín Steinhauser. « OMP4J - OpenMP for Java ». Thèse de doct. Univerzita Karlova, Matematicko-fyzikální fakulta, 2015. url : http://www.omp4j.org/.

[31] Yaakov Benenson et al. « Programmable and autonomous computing machine made of biomolecules ». In : *Nature* 414.6862 (2001), p. 430-434.

[32] Graeme Blackwood et al. « Large-scale statistical machine translation with weighted finite state transducers ». In : *Proceeding of the 2009 conference on Finite-State Methods and Natural Language Processing*. 2009, p. 39-49.

[33] Raphaël Bolze et al. « Grid'5000 : a large scale and highly reconfigurable experimental grid testbed ». In : *The International Journal of High Performance Computing Applications* 20.4 (2006), p. 481-494.

[34] Josh Bongard et Hod Lipson. « Active coevolutionary learning of deterministic finite automata ». In : *Journal of Machine Learning Research* 6.Oct (2005), p. 1651-1678.

[35] Dhruba Borthakur. « The hadoop distributed file system : Architecture and design ». In : *Hadoop Project Website* 11.2007 (2007), p. 21.

[36] Raymond T Boute. « Distinguishing sets for optimal state identification in checking experiments ». In : *IEEE Transactions on Computers* 100.8 (1974), p. 874-877.

[37] Stephen D Brown et al. « Introduction to FPGAs ». In : *Field-Programmable Gate Arrays*. Springer, 1992, p. 1-11.

[38] Rajkumar Buyya. « High performance cluster computing : Architectures and systems (volume 1) ». In : *Prentice Hall, Upper SaddleRiver, NJ, USA* 1.999 (1999), p. 29.

[39] Olivier Carton. *Langages formels, calculabilité et complexité*. T. 28. Vuibert, 2008.

[40] R Chaiken et al. « SCOPE : Easy and E cient Parallel Processing of Massive Datasets ». In : VLDB.

[41] Aftab Ahmed Chandio, Nikos Tziritas et Cheng-Zhong Xu. « Big-data processing techniques and their challenges in transport domain ». In : *ZTE Communications* 1.010 (2015), p. 1-21.

[42] Yi-Jun Chang et al. « The complexity of (Δ+ 1) coloring in congested clique, massively parallel computation, and centralized local computation ». In : *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, p. 471-480.

[43] Surajit Chaudhuri. « What next ? A half-dozen data management research goals for big data and the cloud ». In : *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. 2012, p. 1-4.

[44] Kwang-Ting Cheng et Avinash S Krishnakumar. « Automatic functional test generation using the extended finite state machine model ». In : *30th ACM/IEEE Design Automation Conference*. IEEE. 1993, p. 86-91.

[45] Tsun S. Chow. « Testing software design modeled by finite-state machines ». In : *IEEE transactions on software engineering* 3 (1978), p. 178-187.

[46] Nicolas Colin et Henri Verdier. *L'âge de la multitude-2e éd. : Entreprendre et gouverner après la révolution numérique*. Armand Colin, 2015.

[47] Karel Culik et Jarkko Kari. « Digital images and formal languages ». In : *Handbook of formal languages*. Springer, 1997, p. 599-616.

[48] Karel Culik II et Ivan Friš. « Weighted finite transducers in image processing ». In : *Discrete Applied Mathematics* 58.3 (1995), p. 223-237.

[49] Robert Czerwinski et Dariusz Kania. *Finite state machine logic synthesis for complex programmable logic devices*. 2013.

[50] Artur Czumaj et al. « Round compression for parallel matching algorithms ». In : *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. 2018, p. 471-484.

[51] Zineb Dafir, Yasmine Lamari et Said Chah Slaoui. « A survey on parallel clustering algorithms for big data ». In : *Artificial Intelligence Review* 54.4 (2021), p. 2411-2443.

[52] Jeffrey Dean et Sanjay Ghemawat. « MapReduce : simplified data processing on large clusters ». In : *Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation-Volume 6*. 2004, p. 10-10.

[53] David DeWitt et Jim Gray. « Parallel database systems : The future of high performance database systems ». In : *Communications of the ACM* 35.6 (1992), p. 85-98.

[54] Rita Dorofeeva et al. « FSM-based conformance testing methods : A survey annotated with experimental evaluation ». In : *Information and Software Technology* 52.12 (2010), p. 1286-1297.

[55] Andre Drumea et Camelia Popescu. « Finite state machines and their applications in software for industrial control ». In : *27th International Spring Seminar on Electronics Technology : Meeting the Challenges of Electronics Technology Progress, 2004.* T. 1. IEEE. 2004, p. 25-29.

[56] Bilal Elghadyry, Faissal Ouardi et Sébastien Verel. « Composition of weighted finite transducers in MapReduce ». In : *Journal of Big Data* 8.1 (2021), p. 1-15.

[57] Bilal Elghadyry et al. « Efficient parallel derivation of short distinguishing sequences for nondeterministic finite state machines using MapReduce ». In : *Journal of Big Data* 8.1 (2021), p. 1-27.

[58] Alina Ene, Sungjin Im et Benjamin Moseley. « Fast clustering using MapReduce ». In : *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2011, p. 681-689.

[59] Alina Ene et Huy Nguyen. « Random coordinate descent methods for minimizing decomposable submodular functions ». In : *International Conference on Machine Learning*. PMLR. 2015, p. 787-795.

[60] David Eppstein. « Reset sequences for monotonic automata ». In : *SIAM Journal on Computing* 19.3 (1990), p. 500-510.

[61] Olivier Ezratty. *Les usages de l'intelligence artificielle*. T. Cinqième édition. 2021.

[62] Khaled El-Fakih et al. « Parallel algorithms for reducing derivation time of distinguishing experiments for nondeterministic finite state machines ». In : *International Journal of Parallel, Emergent and Distributed Systems* 33 (2018), p. 197-210.

[63] Michael J Flynn et Kevin W Rudd. « Parallel architectures ». In : *ACM Computing Surveys (CSUR)* 28.1 (1996), p. 67-70.

[64] Buddhima Gamlath et al. « Weighted matchings via unweighted augmentations ». In : *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, p. 491-500.

[65] Mohsen Ghaffari, Silvio Lattanzi et Slobodan Mitrović. « Improved parallel algorithms for density-based network clustering ». In : *International Conference on Machine Learning*. PMLR. 2019, p. 2201-2210.

[66] Mohsen Ghaffari et Jara Uitto. « Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation ». In : *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2019, p. 1636-1653.

[67] Mohsen Ghaffari et al. « Improved massively parallel computation algorithms for mis, matching, and vertex cover ». In : *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. 2018, p. 129-138.

[68] Pavel Gladyshev et Ahmed Patel. « Finite state machine approach to digital event reconstruction ». In : *Digital Investigation* 1.2 (2004), p. 130-149.

[69] G. Grahne et al. « Computing NFA Intersections in Map-Reduce ». In : *EDBT/ICDT Workshops*. 2015.

[70] Canan Güniçen et al. « The relation between preset distinguishing sequences and synchronizing sequences ». In : *Formal Aspects of Computing* 26.6 (2014), p. 1153-1167.

[71] Abdul Rahim Haddad, Khaled El-Fakih et Gerassimos Barlas. « Parallel implementation for deriving preset distinguishing experiments of nondeterministic finite state machines ». In : *2017 7th International Conference on Modeling, Simulation, and Applied Optimization (ICMSAO)*. IEEE. 2017, p. 1-6.

[72] apache HADOOP. *Welcome to apache hadoop*. 2020. URL : http://hadoop.apache.org.

[73] Joonas HÄMÄLÄINEN, Tommi KÄRKKÄINEN et Tuomo ROSSI. « Improving scalable K-means++ ». In : *Algorithms* 14.1 (2021), p. 6.

[74] Maher HELAOUI. « Cours : Logique Informatique ». In : ().

[75] Lars HELLSTEN et al. « Transliterated mobile keyboard input via weighted finite-state transducers ». In : *Proceedings of the 13th International Conference on Finite State Methods and Natural Language Processing (FSMNLP 2017)*. 2017, p. 10-19.

[76] FC HENNINE. « Fault detecting experiments for sequential circuits ». In : *1964 Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*. IEEE. 1964, p. 95-110.

[77] RM HIERONS. « Using a minimal number of resets when testing from a finite state machine ». In : *Information Processing Letters* 6.90 (2004), p. 287-292.

[78] Robert M HIERONS et Uraz Cengiz TÜRKER. « Parallel algorithms for generating harmonised state identifiers and characterising sets ». In : *IEEE Transactions on Computers* 65.11 (2016), p. 3370-3383.

[79] Robert M HIERONS et Uraz Cengiz TÜRKER. « Parallel algorithms for testing finite state machines : Generating UIO sequences ». In : *IEEE Transactions on Software Engineering* 42.11 (2016), p. 1077-1091.

[80] Robert M HIERONS et Uraz Cengiz TÜRKER. « Distinguishing Sequences for Distributed Testing : Preset Distinguishing Sequences ». In : *The Computer Journal* 60.1 (2017), p. 110-125.

[81] Robert M HIERONS et Uraz Cengiz TÜRKER. « Parallel algorithms for generating distinguishing sequences for observable non-deterministic fsms ». In : *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26.1 (2017), p. 1-34.

[82] Robert M HIERONS et Hasan URAL. « UIO sequence based checking sequences for distributed test architectures ». In : *Information and Software Technology* 45.12 (2003), p. 793-803.

[83] Robert M HIERONS et Hasan URAL. « Optimizing the length of checking sequences ». In : *IEEE Transactions on Computers* 55.5 (2006), p. 618-629.

[84] Robert M. HIERONS. « Adaptive testing of a deterministic implementation against a nondeterministic finite state machine ». In : *The Computer Journal* 41.5 (1998), p. 349-355.

[85] Joachim Hofer et Georg Stemmer. *Optimizations to decoding of WFST models for automatic speech recognition*. US Patent 10,127,902. 2018.

[86] John E Hopcroft, Rajeev Motwani et Jeffrey D Ullman. « Introduction to automata theory, languages, and computation ». In : *Acm Sigact News* 32.1 (2001), p. 60-65.

[87] Sara Hsaini, Salma Azzouzi et My El Hassan Charaf. « A temporal based approach for MapReduce distributed testing ». In : *International Journal of Parallel, Emergent and Distributed Systems* 36.4 (2021), p. 293-311.

[88] Rongqing Huang et Ilya Oparin. *Applying neural network language models to weighted finite state transducers for automatic speech recognition*. US Patent 10,354,652. 2019.

[89] Abou_el_ela Abdou Hussein. « Fifty-six big data V's characteristics and proposed strategies to overcome security and privacy challenges (BD2) ». In : *Journal of Information Security* 11.4 (2020), p. 304-328.

[90] Guy-Vincent Jourdan, Hasan Ural et Hüsnü Yenigün. « Reduced checking sequences using unreliable reset ». In : *Information Processing Letters* 115.5 (2015), p. 532-535.

[91] Guy-Vincent Jourdan et al. « Lower bounds on lengths of checking sequences ». In : *Formal aspects of computing* 22.6 (2010), p. 667-679.

[92] Sol Ji Kang, Sang Yeon Lee et Keon Myung Lee. « Performance comparison of OpenMP, MPI, and MapReduce in practical problems ». In : *Advances in Multimedia* 2015 (2015).

[93] Sertaç Karahoda et al. « Multicore and manycore parallelization of cheap synchronizing sequence heuristics ». In : *Journal of Parallel and Distributed Computing* 140 (2020), p. 13-24.

[94] Jarkko Kari. « Image Processing Using Finite Automata. » In : *Recent Advances in Formal Languages and Applications* 25 (2006), p. 171-208.

[95] Howard Karloff, Siddharth Suri et Sergei Vassilvitskii. « A model of computation for mapreduce ». In : *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM. 2010, p. 938-948.

[96] Onur Kilincceker et al. « Model-Based Ideal Testing of GUI Programs–Approach and Case Studies ». In : *Ieee Access* 9 (2021), p. 68966-68984.

[97] Stephen C. Kleene. « Representation of events in nerve nets and finite automata ». In : *Automata studies* 34 (1956), p. 3-42.

[98] Donald E Knuth, James H Morris Jr et Vaughan R Pratt. « Fast pattern matching in strings ». In : *SIAM journal on computing* 6.2 (1977), p. 323-350.

[99] Paraschos Koutris, Semih Salihoglu, Dan Suciu et al. « Algorithmic aspects of parallel data processing ». In : *Foundations and Trends® in Databases* 8.4 (2018), p. 239-370.

[100] Ravi Kumar et al. « Fast greedy algorithms in mapreduce and streaming ». In : *ACM Transactions on Parallel Computing (TOPC)* 2.3 (2015), p. 1-22.

[101] N. Kushik et al. « On adaptive experiments for nondeterministic finite state machines ». In : *International Journal on Software Tools for Technology Transfer* 18 (2014), p. 251-264.

[102] Natalia Kushik, Khaled El-Fakih et Nina Yevtushenko. « Preset and adaptive homing experiments for nondeterministic finite state machines ». In : *International Conference on Implementation and Application of Automata*. Springer. 2011, p. 215-224.

[103] Richard Lai. « A survey of communication protocol testing ». In : *Journal of Systems and Software* 62.1 (2002), p. 21-46.

[104] Yasmine Lamari. « New Heuristics for Clustering Big Data ». In : (2019).

[105] Douglas Laney. *3D Data Management : Controlling Data Volume, Velocity, and Variety*. Rapp. tech. META Group, 2001.

[106] Silvio Lattanzi et al. « Filtering : a method for solving graph problems in mapreduce ». In : *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. 2011, p. 85-94.

[107] David Lee et Mihalis Yannakakis. « Testing finite-state machines : State identification and verification ». In : *IEEE Transactions on computers* 43.3 (1994), p. 306-320.

[108] David Lee et Mihalis Yannakakis. « Principles and methods of testing finite state machines-a survey ». In : *Proceedings of the IEEE* 84.8 (1996), p. 1090-1123.

[109] Sungchul Lee, Ju-Yeon Jo et Yoohwan Kim. « Hadoop Performance Analysis Model with Deep Data Locality ». In : *Information* 10.7 (2019), p. 222.

[110] Jure Leskovec, Anand Rajaraman et Jeffrey David Ullman. *Mining of Massive Datasets*. 2014.

[111] Ted Leslie. « Efficient approaches to subset construction ». Thèse de doct. University of Waterloo (Canada), 1995.

[112] Anastasia Mavridou et Aron Laszka. « Designing secure ethereum smart contracts : A finite state machine based approach ». In : *International Conference on Financial Cryptography and Data Security*. Springer. 2018, p. 523-540.

[113] William E McUmber et Betty HC Cheng. « A general framework for formalizing UML with formal languages ». In : *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE. 2001, p. 433-442.

[114] Sergey Melnik et al. « Dremel : interactive analysis of web-scale datasets ». In : *Proceedings of the VLDB Endowment* 3.1-2 (2010), p. 330-339.

[115] Zhong Meng et Biing-Hwang Juang. « Minimum Semantic Error Cost Training of Deep Long Short-Term Memory Networks for Topic Spotting on Conversational Speech. » In : *INTERSPEECH*. 2017, p. 2496-2500.

[116] Dejan S Milojicic et al. *Peer-to-peer computing*. 2002.

[117] Baharan Mirzasoleiman et al. « Distributed Submodular Maximization : Identifying Representative Elements in Massive Data. » In : *In Advances in Neural Information Processing Systems*. 2013, p. 2049-2057.

[118] Mehryar Mohri. « Weighted finite-state transducer algorithms. An overview ». In : *Formal Languages and Applications* (2004), p. 551-563.

[119] Mehryar Mohri. « Weighted automata algorithms ». In : *Handbook of weighted automata*. Springer, 2009, p. 213-254.

[120] Mehryar Mohri, Fernando Pereira et Michael Riley. « Weighted finite-state transducers in speech recognition ». In : *Computer Speech & Language* 16.1 (2002), p. 69-88.

[121] Mehryar Mohri, Fernando Pereira et Michael Riley. « Speech recognition with weighted finite-state transducers ». In : *Springer Handbook of Speech Processing*. Springer, 2008, p. 559-584.

[122] E Moore. « Gedanken-Experiments. In Automata Studies ». In : *C. Shannon and J. McCarthy, editors, Automata Studies. Princeton University Press* (1956).

[123] Balas K Natarajan. « An algorithmic approach to the automated design of parts orienters ». In : *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE. 1986, p. 132-142.

[124] Anna Nesvijevskaia. « Phénomène Big Data en entreprise : processus projet, génération de valeur et Médiation Homme-Données ». Thèse de doct. Conservatoire national des arts et metiers-CNAM, 2019.

[125] Christopher Olston et al. « Pig latin : a not-so-foreign language for data processing ». In : *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, p. 1099-1110.

[126] John D Owens et al. « GPU computing ». In : *Proceedings of the IEEE* 96.5 (2008), p. 879-899.

[127] Alexandre Petrenko et Nina Yevtushenko. « Testing from partial deterministic FSM specifications ». In : *IEEE Transactions on Computers* 54.9 (2005), p. 1154-1165.

[128] Petr Pospichal, Jiri Jaros et Josef Schwarz. « Parallel genetic algorithm on the cuda architecture ». In : *European conference on the applications of evolutionary computation*. Springer. 2010, p. 442-451.

[129] Tao R. *Finite Automata and Application to Cryptography*. 2009.

[130] Anand Rajaraman et Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[131] Raghu Ramakrishnan et al. « Azure data lake store : a hyperscale distributed file service for big data analytics ». In : *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, p. 51-63.

[132] Vibhor Rastogi et al. « Finding connected components in map-reduce in logarithmic rounds ». In : *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, p. 50-61.

[133] Abiel Roche-Lima, Michael Domaratzki et Brian Fristensky. « Pairwise Rational Kernels Obtained by Automaton Operations ». In : *International Conference on Implementation and Application of Automata*. Springer. 2014, p. 332-345.

[134] Abiel Roche-Lima et Ruppa K Thulasiram. « Bioinformatics algorithm based on a parallel implementation of a machine learning approach using transducers ». In : *Journal of Physics : Conference Series*. T. 341. 1. IOP Publishing. 2012, p. 012034.

[135] Anders Rockstrom et Roberto Saracco. « SDL-CCITT specification and description language ». In : *IEEE Transactions on Communications* 30.6 (1982), p. 1310-1318.

[136] Jacques Sakarovitch. *Eléments de théorie des automates*. Vuibert, Paris, 2003.

[137] Adenilso Simao et Alexandre Petrenko. « Checking completeness of tests for finite state machines ». In : *IEEE Transactions on Computers* 59.8 (2010), p. 1023-1032.

[138] Adenilso Simao, Alexandre Petrenko et Nina Yevtushenko. « On reducing test length for FSMs with extra states ». In : *Software testing, verification and reliability* 22.6 (2012), p. 435-454.

[139] Dilpreet Singh et Chandan K Reddy. « A survey on platforms for big data analytics ». In : *Journal of big data* 2.1 (2015), p. 8.

[140] Natalia Spitsyna, Khaled El-Fakih et Nina Yevtushenko. « Studying the separability relation between finite state machines ». In : *Software Testing, Verification and Reliability* 17.4 (2007), p. 227-241.

[141] Peter H Starke. *Abstract automata*. North-Holland ; New York, American Elsevier, 1972.

[142] Deian Tabakov et Moshe Y Vardi. « Experimental evaluation of classical automata constructions ». In : *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2005, p. 396-411.

[143] Renji Tao. *Finite automata and application to cryptography*. Springer, 2008.

[144] Niclas Thomas et al. « Decombinator : a tool for fast, efficient gene assignment in T-cell receptor sequences using a finite state machine ». In : *Bioinformatics* 29.5 (2013), p. 542-550.

[145] Ashish Thusoo et al. « Hive : a warehousing solution over a map-reduce framework ». In : *Proceedings of the VLDB Endowment* 2.2 (2009), p. 1626-1629.

[146] Le-Duc Tung. « Pregel meets UnCAL : A systematic framework for transforming big graphs ». In : *2015 31st IEEE International Conference on Data Engineering Workshops*. IEEE. 2015, p. 250-254.

[147] Uraz Cengiz Türker. « Parallel brute-force algorithm for deriving reset sequences from deterministic incomplete finite automata ». In : *Turkish Journal of Electrical Engineering and Computer Sciences* 27.5 (2019), p. 3544-3556.

[148] Uraz Cengiz Türker, Tonguç Ünlüyurt et Hüsnü Yenigün. « Effective algorithms for constructing minimum cost adaptive distinguishing sequences ». In : *Information and Software Technology* 74 (2016), p. 69-85.

[149] Uraz Cengiz Türker et Hüsnü Yenigün. « Complexities of some problems related to synchronizing, non-synchronizing and monotonic automata ». In : *International Journal of Foundations of Computer Science* 26.01 (2015), p. 99-121.

[150] Leslie G Valiant. « A bridging model for parallel computation ». In : *Communications of the ACM* 33.8 (1990), p. 103-111.

[151] MP Vasilevskii. « Failure diagnosis of automata ». In : *Cybernetics* 9.4 (1973), p. 653-665.

[152] Vinod Kumar Vavilapalli et al. « Apache hadoop yarn : Yet another resource negotiator ». In : *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[153] Jingjing Wang et al. « The Myria Big Data Management and Analytics System and Cloud Services. » In : *CIDR*. Citeseer. 2017.

[154] H Todd Wareham. « The parameterized complexity of intersection and composition operations on sets of finite-state automata ». In : *International Conference on Implementation and Application of Automata*. Springer. 2000, p. 302-310.

[155] Tom White. *Hadoop : The definitive guide*. " O'Reilly Media, Inc.", 2012.

[156] Xindong Wu et al. « Data mining with big data ». In : *IEEE transactions on knowledge and data engineering* 26.1 (2013), p. 97-107.

[157] Grigory Yaroslavtsev et Adithya Vadapalli. « Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p-distances ». In : *35th International Conference on Machine Learning (ICML'18)*. 2018.

[158] Matei Zaharia et al. « Spark : Cluster computing with working sets. » In : *HotCloud* 10.10-10 (2010), p. 95.

[159] Yiteng Zhai, Yew-Soon Ong et Ivor W Tsang. « The emerging" big dimensionality" ». In : *IEEE Computational Intelligence Magazine* 9.3 (2014), p. 14-26.

[160] Yanfeng Zhang et Shimin Chen. « i2MapReduce : incremental iterative MapReduce ». In : *Proceedings of the 2nd International Workshop on Cloud Intelligence*. 2013, p. 1-4.

[161] Paul Zikopoulos et Chris Eaton. *Understanding big data : Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.