MINISTÈRE ALGERIÈNNE DE L'ENSEIGNEMENT SUPÉRIEUR, DE LA RECHERCHE SCIENTIFIQUE ET DE LA TECHNOLOGIE

Université Saad Dahleb Blida
Faculté des Sciences,Département d'Informatique

MEḾOIRE DE MASTER

Présenté en vue d'obtenir un
diplôme de Master en Informatique
Option : Systèmes Informatiques et Réseaux

Algorithmes Parallèles efficaces pour la Construction de la Transformées de Burrows-Wheeler

Réalisé Par:
Tidafi Omar Abdelaziz
Meziani Walid

Nom du promoteur: Mr.Kameche Abdallah Hichem
Grâde : MAB
Nom de l'encadreur: Mr.Belazzougui Djamal
Grâde : Maître de recherche
Affiliation: CERIST
Jury :
Présidente: Mme.Nasri Ahlem
Examinatrice: Mme.Lahiani Nesrine

Année universitaire 2018/ 2019

Table des Matières

I Généralités Sur l'Algorithmique Du Texte **13**

I.1 Introduction 14
I.2 Définitions 14
I.3 Compression 15
I.4 Recherche de motif 18
I.5 Quelques structures de données pour l'indexation 20
I.5.1 Arbre des suffixes 20
I.5.2 Vecteur de bits et l'Arbre Ondulé (Wavelet Tree) 23
I.6 La transformée de Burrows-Wheeler 25
I.6.1 Construction 25
I.6.2 Indexation utilisant la BWT 28
I.6.3 Compression utilisant la BWT 31

II Paradigme de Parallélisme **32**

II.1 Introduction 33
II.2 Multithread 33
II.2.1 Fonctionnement 33
II.2.2 Performances 34
II.3 OpenMP 34
II.4 SIMD 35
II.5 calcul parallèles 37

III Algorithmes de Construction de Tableau de Suffixe **40**

III.1 Introduction 41
III.2 Algorithmes de Doublage de Préfixes 41
III.2.1 Algorithme de Manber et Myers 42
III.2.2 Algorithme de Larsson et Sadakane 49
III.3 Algorithmes Récursifs 55
III.4 Algorithmes de tri induit 58

IV Implémentations **62**

IV.1 Introduction 63
IV.2 Construction de la table de suffixes 63
IV.3 Construction de la BWT 65
IV.3.1 Méthode 66

IV.3.2 Parallélisme 69
IV.3.3 Coût mémoire 70
IV.4 Conclusion 70

V Tests Et Résultats **71**
V.1 Introduction 72
V.2 Environnent d'implémentation 72
V.2.1 Équipements Matériels 72
V.2.2 Équipements Logiciels 72
V.2.3 Compilation 73
V.3 Métriques de Performance 73
V.3.1 Accélération d'un traitement (Speedup) 74
V.4 Étude du Comportement de l'Algorithme de Construction de la BWT . . . 77
V.4.1 Étude du temps et de l'espace de Construction de l'Algorithme . . . 77
V.4.2 Calcul de l'Accélération (Speedup) 80
V.4.3 Conclusion 82

Résumé

Dans le cadre de notre travail, on s'est intéressé au fameux problème de **compression de données**. La **Transformée de Burrows-Wheeler** [1] (**BWT**) est l'un des outils les plus reconnu dans la résolution de ce problème. Mais sa construction est vraiment coûteuse que sa soit en terme de mémoire ou en terme de temps de calcul. Pour cela on a développé un **algorithme parallèle** qui vise principalement la **diminution de l'espace de travail** ainsi que le temps d'exécution construisant un **tableau de suffixe** qui sert à calculer la Transformée d'une manière **efficace**.
Les tests et comparaisons effectué sur des implémentations existantes déjà et sur notre implémentation montre bien l'efficacité de cette dernière surtout en terme d'espace de travail et aussi en temps d'exécution qui est acceptable.

Mots clés: compression de données, Transformée de Burrows-Wheeler, BWT, algorithme parallèle, diminution de l'espace de travail, tableau de suffixe, efficace.

Abstract

As part of our work, we have focused on the famous problem of **data compression**. The **Burrows-Wheeler Transform** [1] (**BWT**) is one of the most recognized tools in solving this problem. But its construction is really expensive, whether in terms of memory or computing time. For this purpose, we have developed a **parallel algorithm** that mainly aims to **reduce the workspace** and the execution time by building a **suffix array** which serves to calculate the BWT **efficiently**.
The tests performed on our implementation show its efficiency in terms of workspace and also execution time that is acceptable.

Keywords: data compression, Burrows-Wheeler Transform, BWT, parallel algorithm,reduce the workspace, suffix array, efficiently.

Remerciements

On tient à remercier toutes les personnes qui ont contribué de près ou de loin au succès de notre projet, et qui nous ont aidé lors de sa réalisation et lors de la rédaction de ce travail. Tout d'abord, On adresse nos remerciements à nos chers parents qui sans eux rien de cela ne serait possible. Ils nous ont soutenu tout au long de notre parcours et nous ont offert un environnement de travail idéal. En suite, on tient à remercier vivement notre encadreur Mr. BELAZZOUGUI DJAMAL, qui nous a donné énormément d'aides dans nos recherches, pour le temps passé ensemble et pour le partage de son expertise. Mr. BELAZZOUGUI, qui nous a orienté, conseillé et suivi tout au long de notre projet, ce qui nous a permis d'acquérir beaucoup de connaissances indispensables dans la réalisation des différentes étapes de ce travail. On tient aussi à remercier nos collègues pour leur esprit d'équipe. Et à remercier cordialement Mr. KAMECHE, pour son accueil, son attention, et ses précieux conseils grâce à quoi on a pu réaliser un bon boulot. Enfin, On tient à remercier toutes les personnes qui nous ont soutenu, aidé et ont contribué, même de loin, à la création de notre projet.

Liste des Tableaux

I.1 Le tableau de suffixe de la chaîne $x = abracadabra\$$. 22
I.2 Le tableau de suffixe et le tableau de suffixe inverse de la chaîne $x = abracadabra\$$. 23
I.3 Application du BWT avec la chaîne de saisie $x = abracadabra\$$. 27
I.4 La relation entre le BWT et le SA avec la chaîne de saisie $x = abracadabra\$$. 28
I.5 Taille et temps de requête du FM-index et de ses variations d'après [2]. . . 30

III.1 $SA\ [i]$ partitionnée en 6 compartiments en fonction du premier symbole de chaque suffixe: *\$, a, b, c, d, r*. 44
III.2 Balayage h-casier. 46
III.3 Début du balayage h-casier. 47
III.4 L'étape 2 du balayage h-casier. 47
III.5 L'étape 3 du balayage h-casier. 48
III.6 L'étape 4 du balayage h-casier. 48
III.7 exécution de l'algorithme LS(version de base) avec la chaîne d'entrée $x = abracadabra\ \$$ et un symbole sentinelle \$ ajouté à la fin de la chaîne. 53
III.8 tableau, qui incluons tous les n-uplets et triplets possibles utilisés pour la comparaison dans une routine de fusion. 57

IV.1 tri alphabétique des suffixes de la chaine *abracadabra*. 65

V.1 Le temps et l'espace nécessaires pour la construction de l'algorithme par rapport au nombre de partition et le nombre de threads pour le fichier de **KING JAMES**. 78
V.2 Le temps et l'espace nécessaires pour la construction de l'algorithme par rapport au nombre de partition et le nombre de threads pour le fichier d'*ADN*. 78
V.3 Le temps et l'espace nécessaires pour la construction de l'algorithme par rapport au nombre de partition et le nombre de threads pour le fichier du langage anglais. 79
V.4 Temps d'exécution et espace utilisé par $DBWT$ pour les trois fichiers (**KING JAMES, ADN, ANGLAIS**). 79

Liste des Figures

I.1 Les différentes opérations de modifications d'une séquence. Le caractère représente une absence de lettre, il n'est utilisé que pour la représentation d'exemples. Une substitution est représentée en (1), un B devient A, une délétion est en (2), on supprime la lettre B, et une insertion est en (3), on ajoute une lettre A. Ici la distance d'édition entre chaque couple est de 1. (4) La distance d'édition entre les mots ABBA et AAB est 2, en substituant la deuxième lettre et supprimant la dernière 15

I.2 Arbre de Huffman pour définir le code du texte de 7 lettres BANANA$. Le code est ensuite utilisé pour écrire le texte : 0011011011111, utilisant 13 bits. Le caractère A est très fréquent (présent 3 fois), il est donc représenté avec un seul bit : 1. À l'inverse, le caractère B n'est que peu présent (1 fois) et est représenté en 3 bits : 001. 16

I.3 Découpage du mot BANANA$ avec l'algorithme LZ76. Les trois premières lettres sont des premières occurrences, elles provoquent la formation de trois facteurs de taille 1. Les quatrième et cinquième facteurs AN et A sont déjà présents en position 1 et 3. Le dernier facteur est la première occurrence de $ et est de taille 1. Le codage de BANANA$ est B, A, N, (1,2), (3,1), $. 17

I.4 Programmation dynamique pour aligner le texte $ATGGA$ et le texte $ACGG$ avec un alignement global. Ici $C_{ins} = C_{del} = C_{sub} = 1$. La case $M[5][4]$ (entourée) nous indique que l'alignement se fait avec 2 erreurs. Le chemin fléché nous indique les modifications apportées au premier texte: délétion de la lettre A en dernière position et substitution du T en C en deuxième position. 20

I.5 Programmation dynamique pour trouver les occurrences du motif ACGG dans un texte par alignement semi-global. Nous avons : $C_{ins} = C_{del} = C_{sub} = 1$. Nous autorisons au maximum une erreur. Les positions entourées sont les minimums locaux. Nous trouvons un alignement exact ainsi que deux alignements provoquant une erreur, mais dont les débuts sont à la même position. Il y a donc deux occurrences du motif à une erreur près. . 20

I.6 Arbre des suffixes compacté du mot $BANANA\$$. Le chemin allant de la racine à la feuille étiquetée 3, représente le suffixe $ANA\$$, se trouvant à la position 3 du mot. La recherche du motif ANA, nous conduit à un nœud interne (nœud plein). Deux feuilles en sont les descendantes, le motif est présent deux fois dans le mot, aux positions 3 et 1. 21

I.7 Découpage d'un vecteur B pour permettre le fonctionnement de la fonction *rank* en $O(1)$. Les indices proches des blocs sont les valeurs des tables superblocks et blocks. La fonction *rank*(1,10,B) est résolue en faisant la somme : *superblock*[0] + *block*[2] + *smallrank*[00][1] + *smallrank*[10][0] = $0 + 5 + 0 + 1 = 6$ . . . 24
I.8 représentation Wavelet Tree du mot T = BANANA$. (gauche) Nous voulons connaître *T[3]*. Le quatrième bit du nœud q_0 est 0 donc la lettre est dans le sous -arbre gauche de q_0. C'est le deuxième 0 de q_0, le deuxième bit du nœud q_1 est 1. La lettre est donc dans le sous-arbre droit, il s'agit d'un A. (droite) Nous cherchons toutes les occurrences de la lettre N dans le texte. La feuille dont le label est N est dans le sous-arbre doit de son parent, donc nous regardons tous les bits 1 du nœud q_2. Celui-ci nous indique qu'il y a 2 occurrences de la lettre N. Ces bits sont les 2e et 3e bits de q_2 et ce nœud est dans le sous-arbre droit de q_0. Les occurrences de N sont les 2e et 3e bits 1 de q_0, soient les lettres aux positions 2 et 4. . . . 25
I.9 Construction de la transformée de Burrows-Wheeler pour le texte BANANA$. La première matrice présente toutes les rotations du mot. La seconde, M a les rotations triées dans l'ordre lexicographique. . . . 26
I.10 Formule de la relation entre le BWT et le SA. . . . 28

II.1 Graphe nominal simplifié état transition d'un thread (avec préemption). . . 34
II.2 Modèle d'exécution de OpenMP, Parallélisme *Fork-Join*. . . . 35
II.3 Le triplement ordinaire de quatre nombres de 8 bits. La CPU charge un nombre de 8 bits dans R1, le multiplie par R2, puis enregistre la réponse de R3 dans la RAM. Ce processus est répété pour chaque numéro. . . . 36
II.4 Charge de travail de l'algorithme séquentiel et de l'algorithme parallèle. . . 38
II.5 Somme des coûts de tous les processeurs (Coût total). . . . 39

III.1 Algorithme de MM. . . . 45
III.2 Algorithme de *SAIS*. . . . 59
III.3 Exemple d'exécution de l'algorithme *SAIS*, en utilisant la chaîne d'entrée X = *abracadabra*$, avec les trois étapes. . . . 61

IV.1 Fonction *ConstTabSuff*(*char* , *int*) illustrant la méthode naïve pour la construction de la table suffixes. . . . 64
IV.2 Affichage après l'exécution de l'algorithme sur la chaine *abracadabra*. . . . 65
IV.3 Déroulement des opérations de l'algorithme. **(1)** plusieurs processeurs chargent différentes parties des sous-chaînes contenues dans un fichier initial; **(2)** Les suffixes sont partitionnés dans des parties qui leurs convient; **(3)** plusieurs processeurs chargent et trient les suffixes qui appartiennent aux mêmes partitions; **(4)** les codes BWT de chaque partition sont sortis dans des fichiers temporaires; **(5)** concaténation des fichiers temporaires pour obtenir le BWT final. . . . 68
IV.4 Principe de l'algorithme *quicksort* . . . 68

V.1 Les trois types de l'accélération possibles. 74
V.2 l'allure classique d'une accélération expérimental. 75
V.3 Le temps de construction de la *BWT* pour le fichier de **KING JAMES**. . 80
V.4 Histogramme de la consommation mémoire des trois fichiers par notre algorithme et dbwt 81
V.5 Représentation de la courbe de l'accélération par rapport au nombre de threads lancés. 82

Introduction Générale

Une transformée de Burrows-Wheeler (BWT) [1] est une transformation d'un texte en texte, ce qui est utile pour de nombreuses applications, y compris la compression de données, l'indexation de texte intégral compressé, l'exploration de modèles pour n'en nommer que quelques-uns. Pour avoir BWT, on construit en général un tableau de suffixe (SA) d'abord, puis on obtient BWT à partir de SA. Bien que les deux étapes puissent être faites en temps linéaire dans la longueur du texte [3,4], elle nécessite un grand espace de travail. Bien que l'espace pour le résultat de BWT est n $\log \sigma$ bits où n est la longueur du texte et σ est la taille de l'alphabet, l'espace pour SA est de n log n bits ce qui est très considérable en terme d'espace de travaille. Par exemple, dans le cas des génomes humains, $n = 3,0 * 10^9$ et $\sigma = 4$, la taille de BWT est d'environ 750 Mo, et celle de SA est de 12 Go. Par conséquent, l'espace de travail est environ 16 fois plus grand que celui de BWT.

Le problème du tri des suffixes se résume à trouver un ordre lexicographique de tous les suffixes d'une chaîne. Le tableau de suffixes (SA) est une donnée élégante et compacte. qui stocke les positions de départ des suffixes triés. Le concept de l'AS a été proposé en 1990 par Manber et Myers [5] comme substitut peu encombrant de la structure de données du suffixe [6, 7]. Le SA trouve une application importante dans différents domaines tels que la stringologie [8], l'analyse génomique [9], la bioinformatique [10], et bien d'autres. Par exemple, on peut utiliser l'AS pour déterminer rapidement si un texte donné contient un modèle requis. Au lieu de construire une structure d'index pour le texte par des requêtes répétitives et un prétraitement du texte, le SA fournit une solution de structure d'index immédiate. Ainsi, pour trouver toutes les correspondances d'un motif dans un texte, il faut identifier chaque suffixe qui commence par un motif. Une solution simple nécessite une double passe d'une recherche binaire [11] pour localiser le premier et le dernier index de l'intervalle contenant le motif. Ce type d'index est appelé index plein texte [12].

Une autre application de SA est la compression de données. La table de suffixes joue un rôle important dans la transformée de Burrows-Wheeler (BWT) introduite par Burrows et Wheeler en 1994. La BWT d'une chaîne de caractères est obtenu par une génération des rotations cycliques de cette chaîne, en les triant lexicographiquement et en extrayant le dernier caractère de ces chaînes triées. BWT est réversible avec un minimum de surcharge de données. La BWT peut être calculée efficacement avec une simple modification de la la table de suffixe. C'est un outil puissant pour transformer des données en une forme adaptée aux algorithmes de compression sans perte intégrés dans des compresseurs populaires tels que bzip2 [13].

Une façon triviale de construire le tableau de suffixes du texte d'entrée est d'utiliser un algorithme de tri (p.ex. *quick sort* ou *merge sort* [14, 15]) pour trouver l'ordre lexicographique des suffixes. Cette approche se traduira en $O(n^2 log\ n)$ complexité temporelle parce que les comparaisons $O(n\ log\ n)$ sont utilisées et que chaque comparaison de chaînes prend $O(n)$ temps. Toutefois, nous pouvons améliorer considérablement cette complexité

en sachant qu'il existe une certaine dépendance entre les suffixes. En fait, les suffixes sont des chaînes de caractères qui se chevauchent.

Actuellement, les algorithmes de construction de tableaux de suffixes (SACA) qui tirent parti de ces connaissances peuvent être classés en trois grandes catégories : doublement de préfixe , tri récursif et tri induit [16]. Les SACA à doublement de préfixe utilisent le rang des préfixes pour déterminer l'ordre des suffixes, où la longueur de chaque préfixe est doublée à chaque itération. Le SACA récursif approche les suffixes de groupe en deux sous-ensembles selon certains critères. Un sous-ensemble avec 2/3 ou moins de suffixes est trié récursivement, et son ordre est utilisé pour induire l'ordre du second sous-ensemble. Une fois les deux sous-ensembles triés, le tri par fusion est utilisé pour combiner les résultats. Les algorithmes de tri induit utilisent l'information sur le sous-ensemble trié de suffixes pour induire l'ordre des autres suffixes.

En outre, ces dernières années, la construction de tableaux de suffixe en temps linéaire [17] a été parallélisée avec succès. Cependant, jusqu'à présent, la plupart des implémentations parallèles ne sont pas efficaces en mémoire.

L'objectif de notre travail est d'étudier et de développer des algorithmes parallèles pour la construction de ces structures de données qui sont efficaces en mémoire.

Ce mémoire est organisé comme suit : nous commençons au chapitre I par une brève introduction aux définitions et notations de base en stringologie. Puis nous étudions les propriétés de base de la SA. Finalement, nous décrivons le BWT et montrons comment il peut être construit efficacement par la SA. En chapitre II, on donnera une vue générale sur le paradigme de Parallélisme ainsi que les API et les directives qu'ils l'utilise. Dans le chapitre III on passe en revue sur les travaux antérieurs concernant la construction de la Table de Suffixes. Nous discutons des trois méthodes fondamentales pour la construction de tableaux de suffixes. Nous étudions les principes de fonctionnement des SACAs en fonction de chaque méthode. En chapitre IV, on présente nos contributions concernant la construction de SA en illustrant l'algorithme naïf, puis on passe à notre algorithme principal de construction de la BWT. Dans le dernier chapitre (V), on mettra en œuvre notre implémentation grâce à des tests de hautes performances effectués sur le Cluster IBNBADIS spécifique à l'organisme d'accueil CERIST. On fini ce mémoire par une conclusion générale résumant le travail réalisé et donnant nos futures perspectives.

Chapitre I

Généralités Sur l'Algorithmique Du Texte

I.1 Introduction

L'algorithmique du texte, aussi nommée Stringologie, est le domaine de recherche sur les algorithmes agissants sur un texte. Dans ce chapitre on définit les notions qui seront utilisées tout au long de ce mémoire. On introduit à la section I.2 quelques notations de base sur la Stringologie. Puis par la suite, on présente deux algorithmes de compression dans la section I.3. Parmi les problèmes importants du domaine, on trouve la localisation de motif dans un texte, présentée en section I.4. Les algorithmes font souvent appel à des structures de données, certaines d'entre elles sont détaillées dans la section I.5. Dans la section I.6 on présente la transformée de Burrows-Wheeler, un algorithme réorganisant les lettres d'un texte. Nous étudions le principe de fonctionnement de ce dernier, qui sera largement examiné dans les prochains chapitres. Et on conclut par l'algorithme de compression bzip2 qui est basé sur cette transformée et qui offre actuellement l'un des meilleurs taux de compression.

I.2 Définitions

Un **alphabet** est un ensemble non vide d'éléments appelés les **lettres** Une lettre est un symbole quelconque. Une **séquence** de taille n est une suite finie de lettres $\omega = \omega_0\omega_1\ldots\omega_{n-1}$ appartenant à un alphabet $\sum$ - {$} de taille σ, sauf le caractère final qui est le symbole $. Un **mot**, ou **texte**, est composé d'une ou de plusieurs séquences. La longueur du mot ω, notée $|\omega|$, est le nombre de lettres qui le compose.
Le **mot vide** ϵ est le mot de longueur nulle. Chaque lettre d'un mot ω est placée à une **position** de ω. La lettre à la position i du mot ω, ω_i, est la i+1-ème lettre de ω.

Le mot y est un **facteur** ou **occurrence** du mot xyz, avec x ; y ; $z \in \sum^*$. Si x est vide, alors y est un **préfixe** de xyz. Si z est vide, alors y est un **suffixe** de xyz. On dit que y **apparaît** à la position i de ω si la première lettre de y est placée à la position i de ω. On écrira $T_{i,j} = t_i t_{i+1} \cdots t_j$ un facteur du texte T de longueur *j-i+1* apparaissant à la position i.

Le mot yx est une **rotation**, ou **permutation circulaire**, du mot xy. Il existe n rotations d'un mot de taille n. La *i+1-ème* rotation du mot $\omega = \omega_0\,\omega_1 \cdots \omega_{n-1}$ est le mot $\omega_i \cdots \omega_{n-1}\omega_0 \cdots \omega_{i-1}$.

L'ordre lexicographique est un ordre sur est mots induit par un ordre sur les lettres. Soient v et w, deux mots sur l'alphabet $\sum$, $v < w$ si soit v est un préfixe de w, soit $v = xay$ et w $= xbz$ et $a < b$, avec $a,b \in \sum$ et $x,y,z \in \sum^*$.

Soient v et w, deux mots sur l'alphabet $\sum$, avec $v \neq w$. La **distance** d'édition, ou distance de Levenshtein, entre ces deux mots est le plus petit nombre d'opérations nécessaires pour transformer v en w. Les opérations possibles sont (voir Figure I.1) :

- **la substitution** d'une lettre par une autre
- **la délétion** d'une lettre
- **l'insertion** d'une lettre

AABAAA AABAAA AAB-AA ABBA
AAAAAA AA-AAA AABAAA AAB-
(1) (2) (3) (4)

Figure I.1: Les différentes opérations de modifications d'une séquence. Le caractère représente une absence de lettre, il n'est utilisé que pour la représentation d'exemples. Une substitution est représentée en (1), un B devient A, une délétion est en (2), on supprime la lettre B, et une insertion est en (3), on ajoute une lettre A. Ici la distance d'édition entre chaque couple est de 1. (4) La distance d'édition entre les mots ABBA et AAB est 2, en substituant la deuxième lettre et supprimant la dernière

I.3 Compression

Nous pouvons représenter un texte en utilisant moins d'espace mémoire que sous sa forme originale en utilisant une **compression**. Une compression sans perte signifie que le fichier compressé après sa décompression on obtient un résultat strictement identique à l'original, contrairement à la compression avec perte. Le rapport entre le volume avant et après compression est le **quotient de compression**. L'entropie est une mesure de la surprise d'un texte, elle indique si un texte peut être bien compressé ou non. Si après la lecture d'une partie du texte, nous pouvons déterminer la lettre suivante, alors l'entropie est faible. L'**entropie** d'un texte T est notée $H_x(T)$ où x est l'**ordre** de l'entropie. $H0(T)$ est l'entropie d'ordre zéro de T, où nous considérons la possibilité d'apparition de chaque lettre indépendamment et $Hk(T)$, l'entropie d'ordre k de T qui est la probabilité d'apparition d'une lettre en fonction des k précédentes. Il est possible de transformer le texte avant de le compresser pour avoir un meilleur taux de compression, par exemple avec une transformée de Burrows-Wheeler, voir la section 1.5.

Nous présentons deux algorithmes de compression : le codage de Huffman et l'algorithme LZ76.

Le **codage de Huffman** [18] est un algorithme de compression de données sans perte. Il consiste à coder chaque élément avec un code à longueur variable déterminé grâce à un arbre binaire (voir Figure I.2). Les doublons élément/poids constituent les feuilles de notre arbre, le poids étant le nombre d'occurrences de l'élément. Puis nous construisons l'arbre de la manière suivante : les deux nœuds de plus petit poids sont descendants d'un nouveau nœud de poids égal à leur somme, le tout jusqu'à obtenir un seul nœud. Pour

obtenir le code d'un élément nous suivons le chemin de la racine à la feuille correspondant à cet élément et ajoutons un bit 0 (resp.1) à chaque fois que nous allons dans le sous-arbre gauche (resp.droit).
A partir d'une distribution de probabilité connue, un code de Huffman propose la plus courte longueur de texte pour un codage par symbole. La longueur du code d'un élément est déterminée par la fréquence d'apparition de cet élément dans le texte. La taille d'un texte T codé avec le codage de Huffman est fonction de l'entropie d'ordre *0 (H0(T))*.

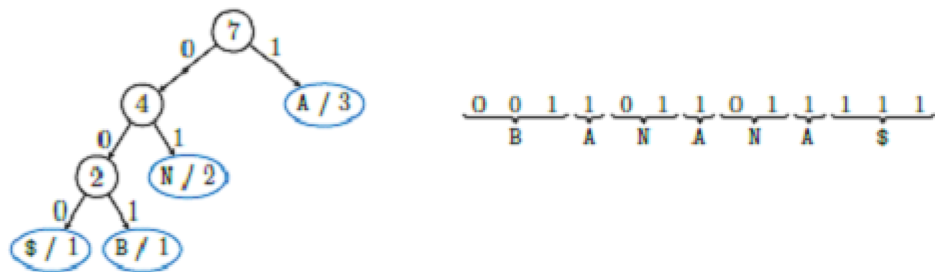

Figure I.2: Arbre de Huffman pour définir le code du texte de 7 lettres BANANA$. Le code est ensuite utilisé pour écrire le texte : 0011011011111, utilisant 13 bits. Le caractère A est très fréquent (présent 3 fois), il est donc représenté avec un seul bit : 1. À l'inverse, le caractère B n'est que peu présent (1 fois) et est représenté en 3 bits : 001.

LZ76 [19] et ses variantes (LZ77, LZ78...) sont des algorithmes de compression de données sans perte, proposés par Lempel et Ziv. Ils utilisent des références vers des facteurs connus, c'est une compression par dictionnaire. L'algorithme LZ76 découpe un texte T de taille n en $n' \leq$ n facteurs z_i appelés phrases, tels que $z_0 z_1 \cdots . \ z_{n'-1} =$ T, voir Figure I.3. Ce découpage du texte T sur un alphabet de taille σ produit $n' = O(n/\log_\sigma n)$ phrases [2]. Supposons que nous avons découpé $T_{0,i-1}$ en facteurs $z_0 z_1 \cdots z_{i'-1}$, le facteur $z_{i'}$ est :

- soit t_i si cette lettre n'est jamais apparue dans $T_{0,i-1}$,
- soit le plus long préfixe $z_{i'}$ de $T_{i,n-1}$, tel que $z_{i'}$ est présent dans $T_{0,i-1}$. La position de $z_{i'}$ dans $T_{0,i-1}$, est appelée la source de $z_{i'}$

Le codage utilisant cette découpe est composé de doublons (source, taille) et de lettres uniques (exceptionnel car utilisé seulement lors de leur première occurrence).

Figure I.3: Découpage du mot BANANA$ avec l'algorithme LZ76. Les trois premières lettres sont des premières occurrences, elles provoquent la formation de trois facteurs de taille 1. Les quatrième et cinquième facteurs AN et A sont déjà présents en position 1 et 3. Le dernier facteur est la première occurrence de $ et est de taille 1. Le codage de BANANA$ est B, A, N, (1,2), (3,1), $.

I.4 Recherche de motif

Un problème courant dans l'algorithmique du texte est la **recherche de motif exact ou approché**. Un motif est un mot dont nous cherchons toutes les occurrences dans un texte. Ce problème est très répandu en bio-informatique pour reconstituer un génome à partir de segments issus de séquençage ou pour rechercher un gène particulier dans une séquence.

L'algorithme naïf de recherche de motif exact consiste à tester si le motif est présent à chaque position du texte. Il a une complexité quadratique. Beaucoup d'autres sont plus rapides : *KMP* [20], *Boyer-Moore* [21], le parallélisme de bits [22]. Les mots à jokers (ou graines) utilisés sur les mêmes algorithmes que précédemment [23] et l'alignement par programmation dynamique (voir section I.4) permettent de rechercher des motifs approchés. Les meilleurs algorithmes ont une vitesse d'exécution proportionnelle à la taille du motif, ils sont exécutés sur des structures nécessitant un prétraitement dont le temps est lié à la taille du texte.

Les structure d'indexation organisent un texte pour le traiter plus facilement et plus rapidement. Ces structures nécessitent un pré-traitement du texte et permettent au mieux une recherche de motif en temps proportionnel à la longueur du motif et non du texte. Pour arriver à de tels résultats, certaines structures d'indexation nécessitent beaucoup de mémoire. De plus, certaines structures permettent la recherche de motifs exacts comme approchés.

Parmi eux se trouve l'arbre des suffixes qui reconnaît tous les suffixes d'un texte et permet le dénombrement des occurrences d'un motif P en temps $O(\mid P \mid)$. Cet arbre est plus amplement défini dans la section I.5.1.

Programmation dynamique pour la comparaison de mots:

La programmation dynamique est une classe d'algorithmes qui consistent à découper un problème en sous-problèmes, les résoudre du plus petit au plus grand en stockant les résultats intermédiaires jusqu'à arriver au résultat du problème original. L'une de ses applications, l'alignement de textes, calcule la plus petite distance d'édition entre deux textes T de taille t et S de taille s.

Soient C_{sub}, C_{del} et C_{ins} des coûts positifs associés aux trois opérations d'édition. La programmation dynamique consiste à remplir une matrice M de taille $(t+1)\times(s+1)$, dont toute case $M[i][j]$ indique le meilleur coût de l'alignement entre$T_{0,i-1}$ et $S_{0,j-1}$.

La matrice est résolue de la manière suivante :

$$M[i][j] = \min \begin{cases} M[i][j-1] + c_{ins} \\ M[i-1][j] + c_{del} \\ M[i-1][j-1] \text{ si } t_i = s_j \\ M[i-1][j-1] + c_{sub} \text{ sinon} \end{cases}$$

L'initialisation de la matrice est fonction du problème à résoudre. Si nous cherchons la meilleure manière d'aligner T et S, nous effectuons un alignement **global**, et utilisons l'algorithme de Needleman-Wunsch [24]. L'initialisation de la matrice est :
$M[i][0] = i \times C_{del}$ et $M[0][j] = j \times C_{ins}$, $\in [0, t]$, $j \in [0, s]$. Le coût de l'alignement entre T et S est la valeur se trouvant dans la case $M[t][s]$.

La recherche de motifs P, proches à k erreurs près, dans le texte T revient à effectuer un alignement **semi-global** entre T et P puis à chercher les cases dont les résultats sont inférieurs à k sur la dernière ligne. L'initialisation de la matrice est :
$M[i][0] = 0$ et $M[0][j] = j \times C_{ins}$, $\in [0, t]$, $j \in [0, s]$. Voir la **Figure** I.5.

Un alignement **local** permet de trouver des facteurs communs à deux textes, il s'agit de l'algorithme de Smith-Waterman [25], la matrice est initialisée comme suit :
$M[i][0] = M[0][j] = 0$, $\in [0, t]$, $j \in [0, s]$. Toute les cases de la matrice ayant une valeur inférieure à k reflètent un alignement local avec moins de k erreurs. Nous ne nous intéressons ici qu'aux deux premières méthodes d'alignement.

Afin de trouver l'alignement global, nous commençons à la case $M[t][s]$ et traversons la matrice jusqu'à arriver à la case $M[0][0]$ avec la règle suivante : depuis la case $M[i][j]$, la case suivante est la case, parmi $M[i\text{-}1][j]$, $M[i][j\text{-}1]$ et $M[i\text{-}1][j\text{-}1]$, dont la valeur est minimale. En cas d'égalité l'une ou l'autre des cases est choisie. Notons qu'à partir d'une case, plusieurs alignements sont possibles.

Dans un alignement semi-global à k erreurs près, nous commençons aux cases $M[i][s]$, telles que $M[i][t] \leq k$, puis nous traversons la matrice de la même manière que précédemment jusqu'à la première case $M[j][0]$, $\in [1, s]$ trouvée. Voir schéma I.5.

		A	T	G	G	A
	0	1	2	3	4	5
A	1	0	1	2	3	4
C	2	1	1	2	3	4
G	3	2	2	1	2	3
G	4	3	3	2	1	2

Figure I.4: Programmation dynamique pour aligner le texte $ATGGA$ et le texte $ACGG$ avec un alignement global. Ici $C_{ins} = C_{del} = C_{sub} = 1$. La case $M[5][4]$ (entourée) nous indique que l'alignement se fait avec 2 erreurs. Le chemin fléché nous indique les modifications apportées au premier texte: délétion de la lettre A en dernière position et substitution du T en C en deuxième position.

I.5 Quelques structures de données pour l'indexation

Les structures de données permettent d'organiser les données afin de les traiter plus facilement. Nous pouvons rechercher un motif dans un texte très rapidement grâce à un arbre des suffixes (voir **section I.5.1**). Les vecteurs de bits et l'Arbre Ondulé (**Wavelet Tree**), définis dans la **section I.5.2**, accèdent et comptent les éléments d'un texte (bit ou lettre) rapidement.

		A	T	A	C	G	G	A	C	T	G	G	C	T
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	0	1	1	1	0	1	1	1	1	1	1
C	2	1	1	1	0	1	2	1	0	1	2	1	1	2
G	3	2	2	2	1	0	1	2	1	1	1	2	2	2
G	4	3	3	3	2	1	0	1	2	2	1	1	2	3

Figure I.5: Programmation dynamique pour trouver les occurrences du motif ACGG dans un texte par alignement semi-global. Nous avons : $C_{ins} = C_{del} = C_{sub} = 1$. Nous autorisons au maximum une erreur. Les positions entourées sont les minimums locaux. Nous trouvons un alignement exact ainsi que deux alignements provoquant une erreur, mais dont les débuts sont à la même position. Il y a donc deux occurrences du motif à une erreur près.

I.5.1 Arbre des suffixes

L'**arbre des suffixes** d'un mot est un automate déterministe acyclique reconnaissant l'ensemble des suffixes du mot, voir Figure I.6. Un chemin est l'ensemble des transitions menant de la racine à une feuille. Un chemin représente le mot issu de la concaténation

des étiquettes de ses transitions. Chaque feuille indique la position de début du suffixe représenté par le chemin. L'arbre des suffixes peut être compacté en transformant chaque chemin unaire en une transition ; l'étiquette de la transition est alors un mot, représenté par sa longueur et un pointeur sur le texte. Il peut être construit en temps linéaire [26–28] tout en utilisant un espace mémoire de $O(n \log n)$ bits. Il permet de compter les occurrences d'un motif P de taille m en temps optimal $O(m)$. En partant de la racine de l'arbre, il faut lire le motif jusqu'à :

- Soit ne plus pouvoir avancer dans l'arbre car la transition par la lettre suivante du motif n'existe pas, et alors le motif n'est pas présent dans l'arbre ;
- Soit avoir fini de lire le motif et :
  - Soit on se trouve sur une feuille, et le motif se trouve une seule fois dans le texte ;
  - Soit on se trouve sur un nœud interne, et le motif est présent autant de fois qu'il y a de feuilles dans le sous-arbre descendant de ce nœud (donnée qui peut être stockée dans le nœud).

La localisation des motifs se fait en temps optimal $O(m + occ)$, avec occ le nombre d'occurrences du motif.

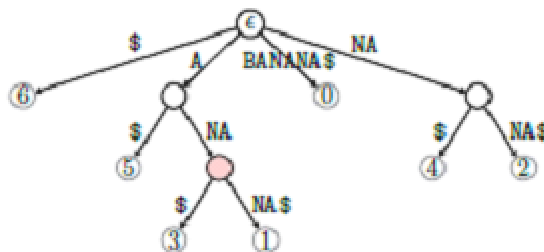

Figure I.6: Arbre des suffixes compacté du mot *BANANA*$. Le chemin allant de la racine à la feuille étiquetée 3, représente le suffixe *ANA*$, se trouvant à la position 3 du mot. La recherche du motif *ANA*, nous conduit à un nœud interne (nœud plein). Deux feuilles en sont les descendantes, le motif est présent deux fois dans le mot, aux positions 3 et 1.

Soit $x\ [0 \cdots n]$ est une chaîne de longueur $n + 1$ sur un alphabet indexé (c'est-à-dire un alphabet où chaque symbole est associé à un entier appartenant à une plage limitée). Nous supposons que $x\ [n] = \$$ est un symbole spécial appelé la sentinelle, généralement identifié par un signe dollar $ tel que $ est lexicographiquement plus petit que tout autre symbole de $\sum$. On note le i-ème suffixe de x comme $x_i = x\ [\ i \cdots n]$ ou suffixe i. Le tableau de suffixe de la chaîne x, écrit sous la forme SA_x ou simplement SA, est un tableau d'entiers $SA\ [0 \cdots n]$, qui stocke une permutation des entiers $0, \cdots, n$, tel que:

$x_{SA[0]} < x_{SA[1]} < \cdots < x_{SA[n]}$.
En d'autres termes, x_i est le j-ème suffixe le plus petit de la chaîne x dans l'ordre lexicographique croissant si et seulement si $SA\ [j] = i$.

Exemple 1.

Suffixe	*i*	Suffixe trié	SA[*i*]
abracadabra$	0	$	11
bracadabra$	1	a$	10
racadabra$	2	abra$	7
acadabra$	3	abracadabra$	0
cadabra$	4	acadabra$	3
adabra$	5	adabra$	5
dabra$	6	bra$	8
abra$	7	bracadabra$	1
bra$	8	cadabra$	4
ra$	9	dabra$	6
a$	10	ra$	9
$	11	racadabra$	2

Table I.1: Le tableau de suffixe de la chaîne x = *abracadabra$*.

L'exemple 1. nous indique que $x_1 1$ est le 0-ème suffixe le plus petit, $x_1 0$ le 1e plus petit suffixe, etc. (en supposant que nous utilisions une chaîne indexée à zéro et l'ordre lexicographique). Notez que $|x\ |=\ |SA\ |$et que SA est la permutation de l'ensemble 0,$\cdots$, 11. Le tableau de suffixe de la chaîne inversé x, écrit sous la forme ISA_x ou simplement ISA, est un tableau d'entiers $ISA\ [0,\ \cdots,\ n]$, de sorte que pour tout i avec $0 \leq i \leq n$, l'égalité $ISA\ [SA\ [i]] = i$. Plus précisément, la relation entre SA et ISA peut être exprimée comme suit: $ISA\ [i] = j \Leftrightarrow SA\ [j] = i$. Le tableau de suffixes inverses est également appelé rangs de suffixes lexicographiques, car $ISA\ [i]$ spécifie le rang du i-ème suffixe parmi les suffixes ordonnés lexicographiquement. L'expression $ISA\ [i] = j$ implique que le suffixe x_i est le j-ème suffixe le plus petit de la chaîne x dans l'ordre lexicographique croissant [51, 64]. Nous pouvons calculer SA et ISA, l'un à partir de l'autre, en temps linéaire comme suit: $SA\ [ISA\ [i]] = i$, $0 \leq i \leq n$ et $ISA\ [SA\ [j]] = j$, $0 \leq j \leq n$.

Exemple 2.

Suffixe	i	Suffixe trié	SA[i]	ISA[i]
abracadabra$	0	$	11	3
bracadabra$	1	a$	10	7
racadabra$	2	abra$	7	11
acadabra$	3	abracadabra$	0	4
cadabra$	4	acadabra$	3	8
adabra$	5	adabra$	5	5
dabra$	6	bra$	8	9
abra$	7	bracadabra$	1	2
bra$	8	cadabra$	4	6
ra$	9	dabra$	6	10
a$	10	ra$	9	1
$	11	racadabra$	2	0

Table I.2: Le tableau de suffixe et le tableau de suffixe inverse de la chaîne x = *abracadabra$*.

L'exemple 2. nous indique que x_0 est le 3-ème suffixe le plus petit, x_1 le 7-ème suffixe et ainsi de suite. Pour calculer la *SA* à partir d'*ISA*, nous prenons les mesures suivantes: *SA* [*ISA* [0]] = *SA* [3] = 0, *SA* [*ISA* [1]] = *SA* [7] = 1, etc. Pour calculer *ISA* à partir de *SA*, procédez comme suit: *ISA* [*SA* [0]] = *ISA* [11] = 0, *ISA* [*SA* [1]] = *ISA* [10] = 1, etc. Évidemment, nous pouvons calculer la *SA* à partir d'*ISA* et vice versa en temps linéaire.

I.5.2 Vecteur de bits et l'Arbre Ondulé (Wavelet Tree)

Dans un **vecteur de bits** B, $B[i]$ est le i+1-ème élément de B et B [i,j] est le vecteur $b_i, b_{i+1}, \cdots, b_j$. La fonction $rank(b,i,B)$ renvoie le nombre de fois que le bit de valeur b apparaît dans le préfixe $B[0,i]$. La fonction $select(b,j,B)$ renvoie la position i du j-ème bit de valeur b dans B. Soit le vecteur de bits B= 0101111000100011, représenté dans la Figure I.7. Il y a $rank(1,10,B)$=6 bits 0 dans le facteur $B[0,10]$ et le quatrième bit 1 de B se trouve à la position $select(1,4,B)$=5. Raman et al. [29] et Pagh [30] ont prouvé qu'un vecteur de bits B peut être stocké en utilisant $nH_0(B)+o(n)$ bits tout en supportant les fonctions *rank* et *select* en $O(1)$, où $H_0(B)$ est l'entropie d'ordre 0 de B. La méthode est appelée **RRR vecteurs**.

De manière simplifiée, le fonctionnement de $rank$(1,i,B) est le suivant : soit un vecteur de bits B de longueur n (en supposant n puissance de 4 pour l'explication). Nous découpons B en superblocks de $\log_2 n$ bits. Chacun de ces superblocs est découpé en blocs de longueur log n. Une table superblock indique le nombre de bits 1 se trouvant avant le superblock courant. Une table block indique le nombre de bits 1 se trouvant avant

le bloc courant, à l'intérieur du superblock courant. Nous stockons une table *smallrank* de taille [0,-1] [0, log $\frac{n}{2}$]. Cette table indexe chaque vecteur de longueur log $\frac{n}{2}$ et indique le nombre de bits 1 se trouvant dans chaque préfixe : $smallrank[V][j]$ indique le nombre de bits 1 se trouvant dans $V_{0,j}$. $rank(1,i,B)$ est la somme des valeurs des cases du superblock, du bloc ainsi que de *smallrank* courants (voir Figure I.7).

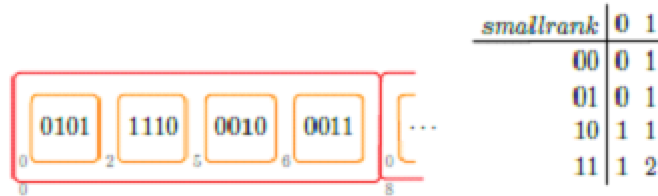

Figure I.7: Découpage d'un vecteur B pour permettre le fonctionnement de la fonction *rank* en $O(1)$. Les indices proches des blocs sont les valeurs des tables superblocks et blocks. La fonction $rank(1,10,B)$ est résolue en faisant la somme : $superblock[0] + block[2] + smallrank[00][1] + smallrank[10][0] = 0 + 5 + 0 + 1 = 6$

Afin de permettre la résolution efficace de la fonction select, le vecteur de bits n'est plus découpé en blocs de même taille mais en blocs ayant x bits 1, avec x valant log_2 n, ou log n. Pour les blocs trop longs (ayant beaucoup de bits 0 pour peu de bits 1), les solutions sont écrites explicitement.

La solution stockant le vecteur de bit en $nH_0(B)+o(n)$ bits est une optimisation de la solution présentée ci-dessus. Les blocs sont triés par classe, avec la classe 1 étant tous les blocs ayant un seul bit 1, la classe 2, tous les blocs ayant deux bits 1, $\cdots$ Il y a $\log(n+1)$ classes possibles et dans chaque classe k, il y a $(\frac{\log(n+1)}{k})$ vecteurs. Ainsi chaque bloc peut être représenté par son numéro de classe en $\log\log\ n$ bits et sa position dans la classe k en log $(\frac{\log(n+1)}{k})$ bits.

L'**Arbre Ondulé** (WT) permet d'étendre les fonctions *rank* et *select* à un alphabet quelconque. Nous pouvons par exemple compter les occurrences d'une lettre dans un facteur ou accéder à une occurrence d'une lettre en temps $O(\log\sigma)$. Défini par Grossi et al. [31], le WT est un arbre binaire où chaque symbole de l'alphabet correspond à une feuille et chaque nœud interne est un vecteur de bits. La racine est un vecteur de bits où chaque position correspond à l'élément qu'il indexe. Les positions marquées 0 correspondent aux éléments dont les feuilles se situent dans le sous-arbre gauche. Les feuilles correspondantes à une position marquée 1 se trouvent dans les sous-arbre droit. Ce processus est répété récursivement jusqu'aux feuilles.

Pour un texte de longueur n construit à partir d'un alphabet de taille σ, la construction d'un arbre ondulé équilibré nécessite un temps de $O(n\lceil\log\sigma\rceil)$ [32] et une taille de $n \log\sigma\ (1+O(1))$ bits [31]. Les vecteurs de bits d'un WT répondent aux fonctions *rank* et *select*. Cela permet à un WT équilibré d'accéder à la lettre $W[i]$ en temps $O(\log\sigma)$ et d'accéder à toutes les positions d'une lettre c en temps $O(occ * \log\sigma)$, avec occ le nombre d'occurrences de c, voir schéma I.8. Un WT ayant la forme d'arbre de Huffman, construit en temps $O(nH_0)$ [33], permet d'accéder à une lettre d'un texte T avec un temps moyen de $O(H_0(T))$. Un WT permet d'obtenir une réorganisation d'une séquence de lettres (en considérant les lettres dans l'ordre des feuilles) [34].

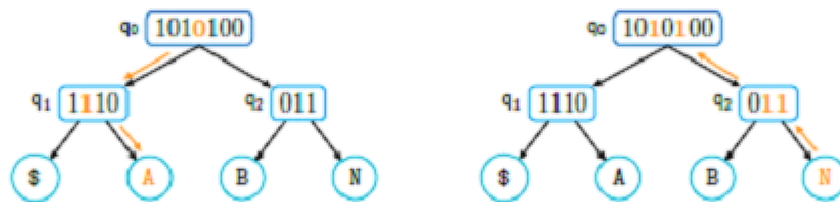

Figure I.8: représentation Wavelet Tree du mot T = BANANA\$. (gauche) Nous voulons connaître $T[3]$. Le quatrième bit du nœud q_0 est 0 donc la lettre est dans le sous -arbre gauche de q_0. C'est le deuxième 0 de q_0, le deuxième bit du nœud q_1 est 1. La lettre est donc dans le sous-arbre droit, il s'agit d'un A. (droite) Nous cherchons toutes les occurrences de la lettre N dans le texte. La feuille dont le label est N est dans le sous-arbre doit de son parent, donc nous regardons tous les bits 1 du nœud q_2. Celui-ci nous indique qu'il y a 2 occurrences de la lettre N. Ces bits sont les 2e et 3e bits de q_2 et ce nœud est dans le sous-arbre droit de q_0. Les occurrences de N sont les 2e et 3e bits 1 de q_0, soient les lettres aux positions 2 et 4.

I.6 La transformée de Burrows-Wheeler

La transformation de Burrows-Wheeler (BWT) est un processus de transformation d'une chaîne d'entrée en une autre chaîne dotée d'une structure appropriée pour une compression efficace. Une propriété remarquable de cette structure est qu'une chaîne générée par le BWT a tendance à contenir un grand nombre de sous-chaînes composées de caractères identiques consécutifs. Pour tirer parti de cette propriété et coder efficacement une chaîne générée par le BWT, un codeur Move-to-Front (MTF) [9] et un codeur de Huffman [31] ou arithmétique sont utilisés. Le BTW est réversible, ce qui permet une récupération de la chaîne d'origine avec une surcharge de données minimale.

I.6.1 Construction

La **transformée de Burrows-Wheeler** (BWT) [1] est un algorithme réversible qui réorganise les lettres d'un texte, permettant une meilleure compression de celui-ci. Elle

est formée de la manière suivante : soit un texte T, dont la lettre finale est \$. Créons toutes les rotations possibles du texte, puis trions-les dans l'ordre lexicographique, ceci forme la matrice M. Nous nommons la première colonne de M, F et la dernière L. L est le texte transformé par la BWT. Les opérations précédentes sont représentées sur la Figure I.9, où le texte BANANA\$ est transformé en ANNB\$AA. Sur ce court texte, l'utilité de la BWT n'est pas évidente à voir. Mais sur un texte plus long, la BWT construit beaucoup de zones de lettres consécutives identiques, permettant une meilleure compression.

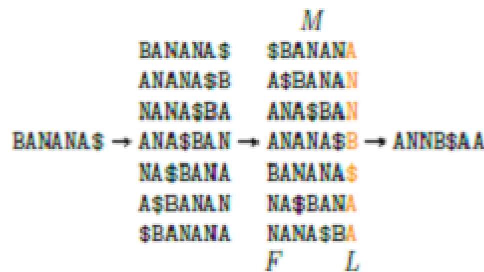

Figure I.9: Construction de la transformée de Burrows-Wheeler pour le texte BANANA\$. La première matrice présente toutes les rotations du mot. La seconde, M a les rotations triées dans l'ordre lexicographique.

Formellement, le texte transformé est la concaténation de la dernière lettre des rotations du texte, rangées dans l'ordre lexicographique. En pratique, les rotations ne sont pas créées, et nous utilisons des références vers les positions du texte. Nous nommons $T^{bwt} = L$, la transformée de Burrows-Wheeler d'un texte T. Soit t_i une lettre de T se trouvant à la position j dans T^{bwt}, nous disons que t_i correspond à la lettre T_j^{bwt}.

Le texte transformé par la BWT est plus facilement compressible que le texte original car des suites de lettres identiques se sont formées. Lorsque nous avons transformé le texte BANANA\$ dans nos exemples, les rotations ANA\$BAN et ANANA\$B sont placées successivement. Leurs rotations NA\$BANA et NANA\$BA commencent de la même manière et sont proches dans M créant une succession de lettres A dans L. Le même argument est valable pour tous les mots du texte : toute région sera susceptible de contenir beaucoup de fois peu de lettres différentes.

T^{bwt} peut être stocké en utilisant $nH_k(T) + o(n)$ bits [35], où H_k est l'entropie d'ordre $k \leq \log_\sigma n + \omega(1)$ du texte T de taille n.

Donc le BWT d'une chaîne x $[0 \cdots \mathrm{n}]$ est généré comme suit: **(1)** Ajoutez un symbole sentinelle \$ à la fin de la chaîne x tel que $x[n] = \$$ **(2)** Construisez$n + 1 \times n + 1$ matrices R contenant toutes les rotations (décalages cycliques). de x. **(3)** Trier les rotations dans

R lexicographiquement. **(4)** La dernière colonne de R est la sortie du BWT [12].

Exemple 1.

i	**Rotations (R)**	**Rotations triées**	**BWT[i]**
0	abracadabra$	$abracadabra	a
1	bracadabra$a	a$abracadabr	r
2	racadabra$ab	abra$abracad	d
3	acadabra$abr	abracadabra$	$
4	cadabra$abra	acadabra$abr	r
5	adabra$abrac	adabra$abrac	c
6	dabra$abraca	bra$abracada	a
7	abra$abracad	bracadabra$a	a
8	bra$abracada	cadabra$abra	a
9	ra$abracadab	dabra$abraca	a
10	a$abracadabr	ra$abracadab	b
11	$abracadabra	racadabra$ab	b

Table I.3: Application du BWT avec la chaîne de saisie $x = abracadabra\$$.

Notez que lorsque des décalages cycliques sont appliqués sur la chaîne abracadabra $, le symbole sentinelle $ se décale de droite à gauche jusqu'à ce qu'il atteigne la première position de la dernière rotation. Le résultat de la transformation est *ard $ rcaaaabb*. Bien que la longueur de la chaîne d'entrée soit petite, la structure de la chaîne de sortie contient des sous-chaînes de caractères identiques consécutifs, faciles à compresser à l'aide des codeurs mentionnés au début de ce chapitre. En pratique, les chaînes d'entrée de longueur suffisamment grande sont susceptibles de générer davantage de sous-chaînes de caractères identiques consécutifs, du fait de la transformation BTW, et donc l'efficacité de la compression a tendance à augmenter.
L'efficacité de l'approche initiale pour calculer le BTW d'une chaîne se résume à l'efficacité du tri lexicographique des rotations de la matrice Burrows-Wheeler (**BWW**). En d'autres termes, le tri des rotations est un goulot d'étranglement dans le BWT en ce qui concerne la complexité temporelle de cette opération. Une approche naïve consiste à trier les rotations à l'aide d'un algorithme de tri basé sur des comparaisons tel que **quicksort** ou **mergesort**. Cependant, une approche plus sophistiquée et efficace consiste à utiliser le tableau de suffixes pour résoudre le problème du tri des rotations [3, 46].
Observez que les rotations triées ressemblent aux suffixes triés obtenus à partir du tableau de suffixes de la chaîne $x = abracadabra\ \$$. Nous remarquons dans l'exemple 2. que chaque rotation triée peut être obtenue en extrayant un suffixe trié correspondant de x et en l'ajoutant au début de x.

Exemple 2.

i	**Rotations triées**	**Suffixes triés de x**	**SA $[i]$**	**BWT$[i]$**
0	$abracadabra	$	11	a
1	a$abracadabr	a$	10	r
2	abra$abracad	abra$	7	d
3	abracadabra$	abracadabra$	0	$
4	acadabra$abr	acadabra$	3	r
5	adabra$abrac	adabra$	5	c
6	bra$abracada	bra$	8	a
7	bracadabra$a	bracadabra$	1	a
8	cadabra$abra	cadabra$	4	a
9	dabra$abraca	dabra$	6	a
10	ra$abracadab	ra$	9	b
11	racadabra$ab	racadabra$	2	b

Table I.4: La relation entre le BWT et le SA avec la chaîne de saisie x = *abracadabra$*.

La relation entre le BWT et le SA peut être formulée comme suit:

$$\mathrm{BWT}[i] = \begin{cases} x[\mathrm{SA}[i]-1] & \text{if } \mathrm{SA}[i] > 0 \\ \$ & \text{if } \mathrm{SA}[i] = 0 \end{cases}$$

Figure I.10: Formule de la relation entre le BWT et le SA.

Cette relation implique que BWT $[i]$ de la chaîne x peut être construit en prenant un caractère qui est positionné à gauche du suffixe dans le tableau de suffixes SA $[i]$.

Actuellement plusieurs logiciels de compression de documents utilisent la BWT, couplée à un codage de Huffman. Nommée méthode **BZIP2** [36], elle est intégrée dans **7zip**, Winrar, $\cdots$ La BWT est aussi utilisée en bio-informatique pour aligner des séquences sur une référence avec la méthode **BWA** [37].

I.6.2 Indexation utilisant la BWT

Grâce à la recherche inverse, une structure d'indexation, FM-index, est une compression sans perte basée sur la Transformée de Burrows-Wheeler, avec quelques similitudes avec le Tableau des suffixes. Cette méthode de compression a été créée par Paolo Ferragina et Giovanni Manzini [38].Cet algorithme peut, en plus de la compression, être utilisé pour trouver de façon efficace le nombre d'occurrences d'un motif dans le texte compressé, ainsi que pour localiser la position de chaque occurrence du motif dans le texte compressé.

Le FM-index et plusieurs de ses variantes ont vu le jour. Ils permettent de compter et localiser des motifs dans le texte indexé. Chaque index est évalué en fonction de sa

complexité en taille et en temps d'exécution des fonctions : $Count_Occ(P)$ qui compte le nombre d'occurrences d'un motif P et $Locate_Occ(P)$ qui localise les occurrences du motif P dans le texte initial.

Le **FM-index** [38] est la première structure d'indexation à atteindre une taille proche de l'entropie du texte indexé. Elle se construit à partir de la BWT du texte sur laquelle on applique plusieurs encodages : move-to-front, run-length encoding puis un code préfix. Ferragina et Manzini indiquent que sa taille est de l'ordre de $O(H_k(T)) + o(1)$ et a une borne maximale de $5nH_k(T) + g_k \log_n$.

Algorithm 1 *Count_Occ*(): Compte le nombre d'occurrences du motif P de longueur p

```
l = P[p], i = p
sp = C[l] + 1, ep = C[l + 1]
while ((sp ≤ ep)et(i ≥ 2)) do
  l = P[i − 1]
  sp = C[l] + Occ(l, sp − 1) + 1
  ep = C[l] + Occ(l, ep)
  i = i − 1
  if ep < sp then
    return 0
  else
    return ep - sp +1
  end if
end while
```

Plusieurs implémentations ont été proposées, réduisant la taille de l'index et augmentant la vitesse d'exécution des fonctions de requête.

Une implémentation du FMI a été proposée par Sadakane [39] puis par Ferragina et al. [40] et Mäkinen et Navarro [35] en utilisant un Wavelet Tree. Cette dernière, nommée **WT-FMI**, permet une exécution de la fonction $Occ(l, i)$ en $O(\log \sigma)$.

Mäkinen et Navarro [41] ont voulu exploiter le grand nombre de lettres identiques consécutives pour pouvoir représenter la BWT de T : Soit T^{bwt} un texte de taille n, il peut être décomposé en n' sous-mots de lettres identiques. T peut être représenté par le mot $S = l_1 l_2 \cdots l_{n'}$ et le vecteur de bits $B = 10^{l_1-1}10^{l_2-1} \cdots 10^{l_{n'}-1}$ Par exemple si T^{bwt} = aaacbbccc = $a^3c^1b^2c^2$, nous avons S = acbc et B = 1001101000. Les fonctions rank et select sur S et B nous permettent de retrouver T^{bwt}.

Le **RL-FMI** regroupe les idées précédentes : nous indexons seulement le mot S provenant de T^{bwt} avec un Wavelet Tree. Une optimisation peut être faite en utilisant un Wavelet Tree ayant la forme d'un arbre de Huffman. Le WT a alors une taille de $nH_0(S)$. Les requêtes s'exécutent en $O(H_0(S))$. La taille totale du RL-FMI est de $nH_k log\sigma(1 + o(1))$ bits.

Le **AF-FMI** [40], puis l'implémentation de Mäkinen et Navarro [42], permettent d'atteindre un index de taille $nH_k + o(n \log \sigma)$ bits. La première implémentation utilise une "boosting compression", la seconde utilise des RRR vecteurs [29]. Leurs fonctions utilisent une table $Occ[s_i,l]$ de taille σ^{k+1} $log\ n$ bits indiquant le nombre d'occurrences de la lettre l avant le contexte si, ainsi qu'un vecteur de bits de taille $O(\sigma^k \log n)$ bits indiquant le début de chaque contexte avec un bit 1.

La table I.5 résume la taille des index ainsi que les temps d'exécution des fonctions *Count_Occ*() et *Locate_Occ*() [2].

	FM-Index	WT-FMI	RI-FMI	AF-FMI
Taille totale	$5nH_k + o(n \log \sigma)$	$nH_0 + o(n \log \sigma)$	$nH_k \log \sigma + 2n$	$nH_k + o(n \log \sigma)$
Count_Occ	$O(p)$	$O(p \times (1 + \frac{\log \sigma}{\log \log n}))$		
Locate_Occ par occurrence	$O(\log^{1+\epsilon} n)$	$O(\log^{1+\epsilon} n \frac{\log \sigma}{\log \log n})$		
Condition	$\sigma = o(\frac{\log n}{\log \log n})$ $k \leq \log_\sigma(\frac{n}{\log n})$ $-\omega(1)$	$\sigma = o(n)$	$\sigma = o(n)$ $k \leq \log_\sigma n - \omega(1)$	$\sigma = o(n)$ $k \leq \alpha \log_\sigma n$ pour $0 < \alpha < 1$

Table I.5: Taille et temps de requête du FM-index et de ses variations d'après [2].

I.6.3 Compression utilisant la BWT

Algorithme de compression *bzip2*

bzip2 [36] est à la fois le nom d'un algorithme de compression de données et d'un logiciel libre développé par Julian Seward entre 1996 et 2000 qui l'implémente. L'algorithme bzip2 utilise la transformée de Burrows-Wheeler avec le codage de Huffman. Le taux de compression est la plupart du temps meilleur que celui de l'outil classique gzip.

bzip2 compresse la plupart des fichiers plus efficacement que les anciens algorithmes de compression *LZW* (*.Z*) et Deflate (*.zip* et *.gz*), mais il est considérablement plus lent. *LZMA* est généralement plus économe en espace que *bzip2* aux dépens d'une vitesse de compression encore plus lente, tout en ayant une décompression beaucoup plus rapide.

bzip2 compresse les données en blocs d'une taille comprise entre 100 et 900 *ko* et utilise la transformation **Burrows** – **Wheeler** pour convertir les séquences de caractères récurrentes fréquentes en chaînes de lettres identiques. Il applique ensuite la transformation et le codage de **Huffman** [18]. L'ancêtre de *bzip2*, *bzip*, utilisait un codage arithmétique à la place de **Huffman**. La modification a été apportée en raison d'une restriction de brevet logiciel [43].

Les performances de *bzip2* sont asymétriques, la décompression étant relativement rapide. Motivée par le temps *CPU* considérable nécessaire à la compression, une version modifiée appelée *pbzip2* a été créée en 2003 et prend en charge le multi-threading, offrant des améliorations de la vitesse quasi linéaire sur les ordinateurs multi-processeurs et multi-cœurs. Depuis mai 2010, cette fonctionnalité n'a pas été intégrée au projet principal.

Chapitre II

Paradigme de Parallélisme

II.1 Introduction

Dans ce chapitre on va parler du parallélisme et de son mécanisme de fonctionnement en présentant les différentes interfaces qui mène à son exploitation. dans la première section de ce chapitre on parlera du principe du MultiThread ainsi que ces utilisations pour l'amélioration des performances des processeurs. Par la suite, nous aborderons l'API OpenMP, cette interface multiperformes qui offre un contrôle important sur la mémoire partagée. Ensuite, on donne une vue générale sur le principe de SIMD qui décrit les ordinateurs avec plusieurs éléments de traitement qui effectuent la même opération sur plusieurs points de données simultanément. Et on finira par une explication détaillée sur les programme séquentiels et les programmes parallèles aussi que les métriques de performances pour chacun.

II.2 Multithread

II.2.1 Fonctionnement

Un processeur est dit multithread s'il est capable d'exécuter efficacement plusieurs threads simultanément. Contrairement aux systèmes multiprocesseurs (tels les systèmes **multi-cœur**), les threads doivent partager les ressources d'un unique cœur. Le multi-thread est supporté par un *OS* qui possède un ordonnanceur (**scheduler**) qui est responsable du séquencement temporel des processus et thread. Les algorithmes d'ordonnancement réalisent la sélection parmi les processus actifs de celui qui va obtenir l'utilisation d'une ressource, que ce soit l'unité centrale, ou bien un périphérique d'entrée-sortie. La préemption est la possibilité qu'a le système de suspendre un processus/thread(Exécution d'une suite d'instruction dans un processus). Tous les Threads partagent le même espace mémoire au sein de la machine virtuelle. Ils peuvent accéder/modifier tous les objets publics. Chaque thread possède une priorité, Un thread de plus haute priorité peut interrompre et suspendre un thread de plus basse priorité en cours d'exécution.Chaque thread fonctionne jusqu'à ce qu'il passe la main à un autre thread, ou atteint un appel système. des problèmes peuvent être créer si un thread attend qu'une ressource devienne disponible, car il peut alors se bloquer et entraîner le blocage des autres threads. alors chaque thread fonctionne pendant un quantum de temps fixé par avance avec une marge de sécurité (chemin le plus long à l'exécution).Et des procédé de synchronisation sont utilisés (par exemple les moniteurs).

L'*OS* possède les informations suivantes, qui lui permettent de changer de contexte d'exécution: Thread *ID*, Registres sauvegardés (pointeur de pile, pointeur d'instruction (dans la plupart des processeurs , le pointeur d'instruction (PC : Pointer Counter) est incrémenté après chaque exécution d'une instruction vers la prochaine instruction à exécuter.)), Pile ou pile (variable locale, variable temporaire, adresse de retour), Masque de signaux et enfin la priorité (scheduling information).

Chaque thread possède les informations suivantes, qui lui permettent de changer de

contexte d'exécution: Text segment (instructions), Data segment (static and global data), *BSS* segment (uninitialized data), Open file descriptors, Signals, Current working directory et enfin User and group *IDs*.

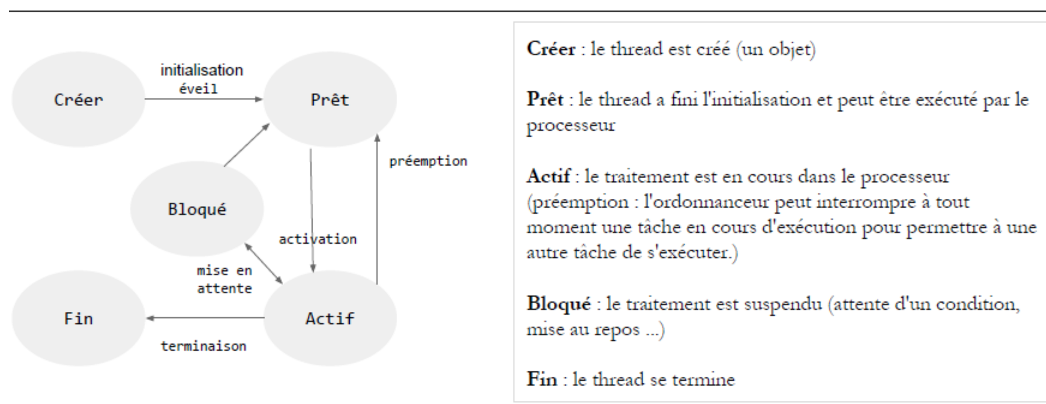

Figure II.1: Graphe nominal simplifié état transition d'un thread (avec préemption).

II.2.2 Performances

En utilisent le multithread avec plusieurs processeurs, il est possible de réaliser plus de calcul à chaque instant, de faire la parallélisation des algorithmes (si possible), l'acquisition des données, la simplification de la structure logiciel (chaque fonctionnalité est gérée par un thread), d'avoir une architecture plus robuste (une partie de l'application ne met plus en péril tout le reste) et une tâche de surveillance en parallèle d'une tâche de traitement.

L'*OS* n'a pas besoin de créer de nouveau emplacement dans la mémoire comme dans le cas des processus, L'*OS* n'a pas besoin également d'allouer de nouvelles structures pour assurer le suivi de l'état des fichiers ouverts et incrémenter le nombre de références sur les descripteurs de fichiers ouverts, Il est trivial de partager des données entre les threads, Les mêmes variables globales et statiques peuvent être lues et écrites entre tous les threads d'un processus.

Avec un processus **mono-thread** , le système d'exploitation ne peut rien faire pour laisser le processus tirer parti de plusieurs processeurs (pas de multithread).

II.3 OpenMP

OpenMP (Open Multi-Processing) [44] est une interface de programmation d'application (*API*) prenant en charge la programmation multitraitement en mémoire partagée multi-plateformes en *C*, *C* ++ et Fortran [45], sur la plupart des plateformes, architectures de

jeux d'instructions et systèmes d'exploitation, y compris Solaris, AIX. , HP-UX, Linux, macOS et Windows. Il se compose d'un ensemble de directives de compilation, de routines de bibliothèque et de variables d'environnement qui influent sur le comportement au moment de l'exécution [46]. OpenMP utilise un modèle portable et évolutif qui offre aux programmeurs une interface simple et flexible pour développer des applications parallèles pour des plates-formes allant de l'ordinateur de bureau standard au superordinateur.

Une application construite avec le modèle hybride de programmation parallèle peut être exécutée sur un cluster d'ordinateurs utilisant à la fois OpenMP et MPI (Message Passing Interface) [47], de telle sorte que OpenMP soit utilisé pour le parallélisme au sein d'un nœud (multicœur), tandis que MPI est utilisé pour le parallélisme entre les nœuds. Des efforts ont également été déployés pour exécuter OpenMP sur des systèmes de mémoire partagée distribuée par logiciel [48], afin de traduire OpenMP en MPI [49] et d'étendre OpenMP à des systèmes de mémoire non partagés.

Modèle d'exécution de OpenMP, Parallélisme *Fork-Join* [50]:

1. Le thread Maître lance une équipe de threads selon les besoins (région parallèle).
2. Le parallélisme est introduit de façon incrémentale ; le programme séquentiel évolue vers un prog. parallèle

la **Figure** II.2 illustre le modèle d'exécution précédent :

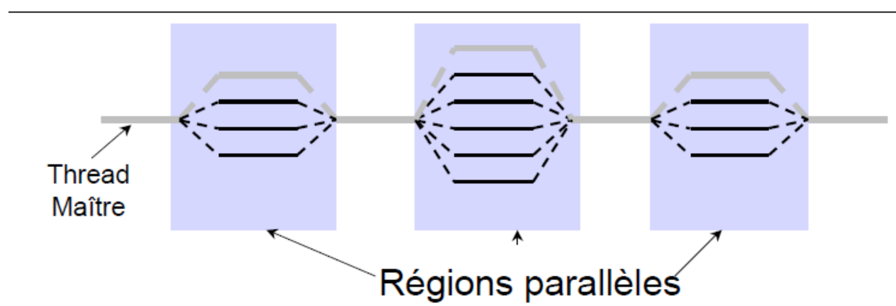

Figure II.2: Modèle d'exécution de OpenMP, Parallélisme *Fork-Join*.

II.4 SIMD

L'Instruction Simple, données multiples - Single Instruction on Multiple Data (*SIMD*) [51] - est une classe d'ordinateurs parallèles dans la taxonomie de **Flynn**. Il décrit les ordinateurs avec plusieurs éléments de traitement qui effectuent la même opération sur plusieurs points de données simultanément. De telles machines exploitent le parallélisme

au niveau des données, mais pas la simultanéité: il existe des calculs simultanés (parallèles), mais un seul processus (instruction) à un moment donné. La plupart des conceptions de *CPU* modernes incluent des instructions *SIMD* pour améliorer les performances de l'utilisation multimédia. *SIMD* ne doit pas être confondu avec *SIMT*, qui utilise des threads.

Le modèle *SIMD* convient particulièrement bien aux traitements dont la structure est très régulière, comme c'est le cas pour le calcul matriciel. Généralement, les applications qui profitent des architectures *SIMD* sont celles qui utilisent beaucoup de tableaux, de matrices, ou de structures de données du même genre (**voir Figure** II.3). On peut notamment citer les applications scientifiques, ou de traitement du signal.

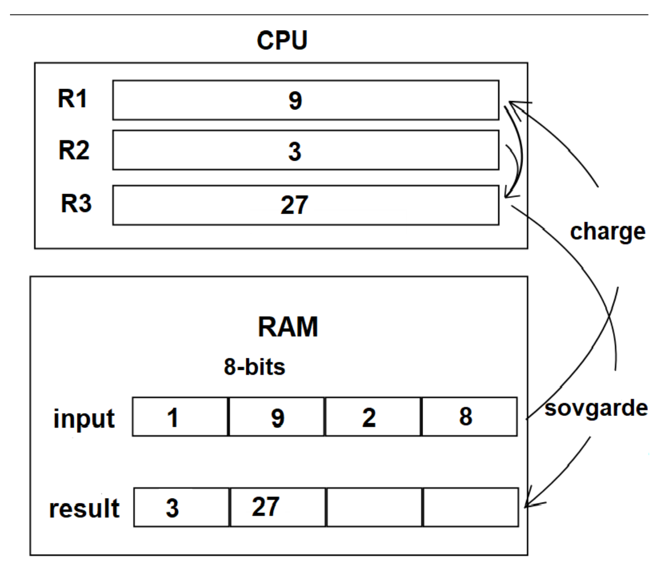

Figure II.3: Le triplement ordinaire de quatre nombres de 8 bits. La CPU charge un nombre de 8 bits dans R1, le multiplie par R2, puis enregistre la réponse de R3 dans la RAM. Ce processus est répété pour chaque numéro.

Les unités de traitement graphique (*GPU*) modernes sont souvent de larges implémentations *SIMD*, capables de créer des branches, des charges et des magasins sur 128 ou 256 bits à la fois. Les dernières instructions *AVD*-512 *SIMD* d'Intel traitent désormais 512 bits de données à la fois. les processeurs à base de *SIMD*: x86 (*Intel, AMD*) avec les technologies. *SSE,AVX, AVX*512. Power (*IBM*) avec Power *ISA* AArch64 (ARM, Cavium, Fujistu) avec *SVE* Nec Vector Engine Processor

SIMD pour Single Instruction on Multiple Data, Faire une même instruction sur plusieurs données en même temps (vecteurs), C'est un niveau de parallélisme disponible dans les coeurs, Seules quelques instructions ont une version vectorielle et Les instructions vectorielles vont aussi vite que les instructions scalaires.

Puissance théorique en DP La puissance crête d'un cœur se calcul comme le produit des composants suivant :

- La fréquence vectorielle du cœur (peut être différente de la fréquence du cœur).
- Le nombre d'instruction flottante vectorielle par cycle (ex 1 addition et 1 multiplication).
- Le nombre d'opération flottante par instruction vectorielle (dépend de la taille des registres).

On a pour exemple:

Nehalem /Westmere (SSE) : Ghz $\times$ 2 $\times$ 2

Sandy bridge/Ivy bridge (AVX) : Ghz $\times$ 2 $\times$ 4

Haswell/Broadwell (AVX2) : Ghz $\times$ 4 $\times$ 4

KNL/SkyLake/CascadeLake (AVX512) : Ghz $\times$ 4 $\times$ 8

AMD EPYC (AVX2) : Ghz $\times$ 2 $\times$ 4

II.5 calcul parallèles

Programme séquentiel : évaluation de la performance par la mesure du temps d'exécution.

Programme parallèle : évaluation plus complexe, Les performances d'un code parallèle vont dépendre :

- du nombre de processeurs

- de la machine sur laquelle il va s'exécuter
- et bien d'autres critères (cf plus loin)$\cdots$

Autrement dit, Un programme séquentiel rapide peut s'avérer, une fois parallélisé, très peu efficace.

Inversement, un programme séquentiel "moins rapide" peut s'avérer mieux parallélisable.

La charge de travail d'un algorithme parallèle/séquentiel:

La charge de travail d'un algorithme séquentiel $W_{sequentiel}$ se décompose en deux parties, travail non parallélisable (coût fixe) et travail pouvant être parallélisé (voir **(1). dans Fegure II.5**). **(2).** dans **Fegure II.5** explique La charge de travail d'un algorithme parallèle $W_{parallele}$ avec p processeurs. Dans un programme parallèle, il faut prendre en compte une surcharge de travail W_{com} due aux synchronisations des tâches (voir **(3)).** dans **Fegure II.5**.

(1). $$W_{séquentiel} = \underbrace{W_{fixe}}_{\text{Travail non parallélisable (coût fixe)}} + \underbrace{W_{parallélisable}}_{\text{Travail pouvant être parallélisé}}$$

(2). $$W_{parallèle}(p) = W_{fixe} + W_{parallélisable}/p$$

(3). $$W_{parallèle}(p) = W_{fixe} + W_{parallélisable}/p + W_{com}$$

Figure II.4: Charge de travail de l'algorithme séquentiel et de l'algorithme parallèle.

Pour calculer le temps d'exécution, il faut avoir nombre de processus ou de tâches, car c'est le temps écoulé entre le début du calcul parallèle et la terminaison de la dernière tâche. Pour calculer le coût total il suffit de faire la somme des coûts de tous les processeurs.

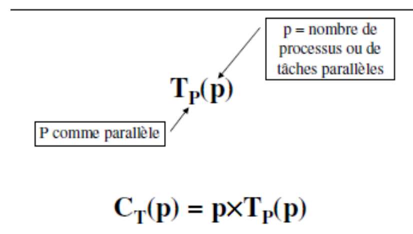

Figure II.5: Somme des coûts de tous les processeurs (Coût total).

En résumer, les performances d'un calcul parallèle s'évaluent à l'aide de plusieurs mesures comme l'accélération ou l'équilibrage.

Les performances parallèles sont essentiellement impactées par:

- les communications : il en faut le moins possible,et leurs messages doivent être les plus grands possibles
- le déséquilibrage de charge induit des attentes inutiles sur les processeurs les moins chargés : il faut une répartition équitable du travail

Chapitre III

Algorithmes de Construction de Tableau de Suffixe

III.1 Introduction

Dans ce chapitre, nous étudions trois classes principales d'algorithmes utilisés pour la construction de tableaux de suffixes. Chaque classe de tels algorithmes applique une technique spécifique pour réussir sa tâche. Nous analysons les idées principales qui sous-tendent chaque technique et examinons comment elles sont mises en œuvre par certaines des SACA les plus importantes de chaque classe.
La technique de préfixe-doublage a été introduite et appliquée pour la construction de suffixes par Manber et Myers [5] en 1990. Leur travail a été inspiré et motivé par Karp et al. [52] qui ont proposé l'idée de la technique en 1972. Plus tard, Larsson et Sadakane [53] ont optimisé l'algorithme MM en réalisant une vitesse environ 10 fois supérieure sans appliquer de parallélisme [16].
En 2003 et 2004, plusieurs nouveaux algorithmes de construction de matrices de suffixes récursifs à temps linéaire ont été développés. Il s'agit de l'algorithme de Karkarainen et Sanders [3], de Ko et Aluru [4], de Kim et al [54] et de Hon et al [55]. Ils sont basés sur un modèle de division et de conquête récursif similaire. Le premier algorithme basé sur la technique de tri induit a été proposé par Itoh et Tanaka [55] en 1999. De 2000 à 2006, d'autres SACA de tri induit ont été développés. Il s'agit de l'algorithme de Seward [56], de Burkhardt et Karkkainen [57], de Manzini et Ferragina [58], de l'algorithme de Baron et Bresler [59] et de Maniscalco et de Puglisi [60] algorithme. En 2009, Nong et al. [61] ont mis au point un SACA de tri induit rapide et léger, qui a ensuite été amélioré et optimisé sans aucun parallélisme par Mori [62].

III.2 Algorithmes de Doublage de Préfixes

L'idée de l'approche de doublage de préfixes est de déduire l'ordre des suffixes à partir des rangs lexicographiques de leurs préfixes. Dans un premier temps, nous trions les suffixes lexicographiquement par leurs préfixes de longueur un. Essentiellement, cela signifie que nous trions de manière lexicographique chaque suffixe en fonction du premier caractère. Ensuite, nous regroupons les suffixes qui commencent par un caractère identique dans des compartiments. Un compartiment qui ne contient qu'un seul suffixe est appelé singleton et est considéré comme trié. Un compartiment contenant plusieurs suffixes est appelé non-singleton et doit être trié.
À chaque tour, nous doublons la longueur du préfixe de chaque suffixe dans tous les compartiments non singleton, en ignorant les singles. Ensuite, nous utilisons les rangs de préfixes du tour précédent pour obtenir les rangs de préfixes du tour en cours et pour trier les suffixes dans chaque compartiment non singleton en fonction des nouveaux rangs de leurs préfixes. En d'autres termes, nous utilisons l'ordre relatif des suffixes calculés dans le tour t pour déduire leur ordre dans le tour $t + 1$. Une fois que le suffixe peut être distingué de manière unique par le rang du préfixe correspondant, sa position dans le tableau de suffixes est fixée, et il est marqué comme un singleton. Le processus de doublage se termine lorsque tous les compartiments sont singleton. Nous disons que les suffixes sont dans l'ordre h s'ils sont triés lexicographiquement en fonction de leurs pre-

miers symboles h. Le h-casier contient des suffixes dans leur ordre-h. Les suffixes dans l'ordre h peuvent avoir les mêmes rangs lorsque la valeur de h est relativement petite par rapport à la longueur de la chaîne. La longueur du préfixe est définie par l'ordre h des suffixes. Tous les singleton casier représentent des suffixes déjà triés en fonction de leurs préfixes h-length. L'avantage de cette méthode est qu'une fois que nous connaissons l'ordre h des suffixes, nous pouvons déterminer leur position dans des sillons de $2h$ dans le temps O (n).
Lorsque nous parcourons les suffixes dans leur ordre h, nous en déduisons l'ordre du suffixe i du rang lexicographique de son préfixe $i + h$ calculé au tour h-1 (où le tour 0 est le tri initial des caractères uniques). A chaque itération, nous doublons h, soit $h = 2_k$ pour $k \in$ 0, 1, 2,$\cdots$. Il faut au plus 0 tours (log n) jusqu'à ce que chaque casier de $2h$ devienne un singleton. Par conséquent, le temps total d'exécution de cette méthode est O (n log n).

III.2.1 Algorithme de Manber et Myers

L'algorithme de tri du suffixe **Manber** et **Myers** (**MM**) s'exécute en log_2 $(n + 1)$ étapes. À chaque étape, il effectue un tri implicite de $2h$ en balayant les compartiments de gauche à droite. Cette opération prend O (n) temps. Pour trier implicitement les suffixes, Manber et Myers s'appuient sur l'idée suivante. Considérons deux suffixes i et j qui résident dans le même compartiment. Nous voulons déterminer leur ordre h. Pour cela, nous devons comparer lexicographiquement leurs préfixes $i + h$ et $j + h$. Nous supposons que nous connaissons l'ordre relatif des suffixes $i + h$ et $j + h$. Par conséquent, nous pouvons utiliser leur ordre pour trier les suffixes i et j. Cette idée conduit à l'observation suivante:

Observation 1: Soit SA_h le tableau de suffixes au stade h. Soit $i \in [a, b]$ la position du suffixe appartenant au compartiment occupant l'intervalle $[a, b]$ dans SA_h. En traversant SA_h de gauche à droite, chaque suffixe d'ordre h:
SA_h $[i]$ - h> 0
définit l'ordre des suffixes sur $2h$ dans les h-casiers correspondants.

Cette observation signifie ce qui suit. Considérons le premier casier, en balayant SA_h de gauche à droite, et prenons le suffixe i comme premier suffixe de ce casier, tel que i - h $\geq$ 0. Puisque le suffixe i est classé premier dans son h-casier, puis le suffixe i - h doit être classé premier dans son panier de $2h$, car les h symboles suivants du suffixe i - h sont les premiers symboles h du suffixe i. Ensuite, nous déplaçons le suffixe i - h vers la première position dans son h-casier. Pour chaque suffixe i, l'algorithme déplace le suffixe i - h vers la position disponible suivante dans son h-casier. Initialement, nous créons deux tableaux booléens B_h et B_2h, ainsi que trois tableaux d'entiers *SA*, *Cnt* et *ISA*. La taille de tous les tableaux est n. *SA* est le tableau de suffixes qui stocke les suffixes dans l'ordre h. *ISA* est l'inverse de *SA*, défini comme *ISA* [*SA* [i]] = i pour tout $i \in [0, n)$. B_h marque les têtes de godet (c'est-à-dire aide à identifier les limites gauche et droite de h-casier). B_2h marque les préfixes qui ont été déplacés. *Cnt* stocke la prochaine position disponible

dans le compartiment en cours d'analyse. Le premier préfixe est déplacé vers le haut de son h-casier, Cnt $[i]$ est incrémenté et le préfixe suivant passe à la deuxième position, etc. Dans la ligne 4 de l'algorithme 1, nous effectuons l'initialisation de SA, B_h et B_2h. Nous trions les suffixes de la chaîne d'entrée $T = T$ [0] T $[1] \cdots$ T $[n$-1] en fonction du premier symbole et enregistrez le résultat dans SA. Cette procédure peut être effectuée en exécutant une sorte de **radix** clé-valeur, où les éléments de T sont des clés et les éléments de SA sont des valeurs. Initialement, nous remplissons SA avec des entiers consécutifs de $[0, n)$. Les valeurs initiales de B_h sont calculées en comparant lexicographiquement le premier symbole des suffixes d'ordre h adjacents, à savoir T $[SA$ $[i$ - 1]] et T $[SA$ $[i]]$. Les valeurs initiales de B_2h sont définies sur 0. Après l'initialisation, l'algorithme entre dans la boucle While principale (ligne 6). Il s'exécute jusqu'à ce que tous les compartiments soient entièrement triés, c'est-à-dire que chaque compartiment est un singleton. Cette condition est satisfaite lorsque toutes les valeurs de B_h sont définies sur 1 ou lorsque le nombre de compartiments est n. Au début de chaque itération, nous calculons les rangs des suffixes dans l'ordre h, à savoir, ISA $[SA$ $[i]] = i$ et définissons Cnt $[i]$ sur 0 pour tout $0 \leq i < n$.
Dans la ligne 15, nous commençons la traversée de SA de gauche à droite. Soit $[a,$ $b]$ l'intervalle occupé par le h-casier en cours d'analyse dans SA. On note s le rang de chaque suffixe obtenu à partir de SA $[i]$ - $h \geq 0$. Pour chaque i, $a \leq i \leq b$, on fixe ISA $[s] = ISA$ $[s] + Cnt$ $[s]$, incrémenter Cnt $[$ $s]$ et définissez B_2h sur 1. Tous les suffixes qui se distinguent uniquement par les rangs de leurs préfixes-h sont déplacés vers la position définie par Cnt dans le compartiment-h correspondant.
Avant de passer au compartiment suivant, nous analysons à nouveau le compartiment actuel. et mettez à jour les valeurs de B_2h comme indiqué à la ligne 25. Pour tous les suffixes déplacés, nous identifions le suffixe le plus à gauche et utilisons sa position pour marquer la tête de son compartiment de 2_h, tout en réinitialisant les valeurs B_2h des autres suffixes au sein du même h-casier. Plus précisément, fixons B_2h à 0 pour tout $v \in [ISA$ $[s] + 1$, u - 1$]$, de sorte que la position de chaque suffixe déplacé s soit marquée dans B_2h et

$$u = \min t : t > ISA[s] \text{ and } (B_h[t] \text{ or not } B_2h[t]).$$

Cette expression implique que nous identifions la limite gauche de chaque casier de 2_h, tandis que la limite droite est préservée par B_h. Ci-dessous, nous présentons un exemple d'exécution de l'implémentation MM disponible dans la **Figure III.1** à l'aide de la chaîne d'entrée T = *abracadabra $*, où $ est la sentinelle. Au début de l'étape $h = 1$, on a:

i	$B_h[i]$	$B_2h[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	1	1	11 = $
1	1	0	0	6	0 = abracadabra$
2	0	0	0	10	3 = acadabra$
3	0	0	0	1	5 = adabra$
4	0	0	0	8	7 = abra$
5	0	0	0	1	10 = a$
6	1	0	0	9	1 = bracadabra$
7	0	0	0	1	8 = bra$
8	1	0	0	6	4 = cadabra$
9	1	0	0	10	6 = dabra$
10	1	0	0	1	2 = racadabra$
11	0	0	0	0	9 = ra$

Table III.1: SA $[i]$ partitionnée en 6 compartiments en fonction du premier symbole de chaque suffixe: *$, a, b, c, d, r.*

À ce stade, SA $[i]$ est partitionnée en 6 compartiments en fonction du premier symbole de chaque suffixe: *$, a, b, c, d, r.* Les suffixes dans SA $[i]$ pour $i = 0, 8, 9$ sont déjà triés et leurs compartiments sont donc singleton. Nous analysons SA $[i]$, considérons un h-casier à la fois et effectuons les opérations suivantes:

```
1  procedure MM(T, n)
     Input: T — input string, n — input string length.
     Output: Sorted lexicographically suffixes of T stored in SA.
2    Define integer arrays: SA(n), ISA(n), Count(n)
3    Define boolean arrays: B_h(n), B_2h(n)
4    Initialize* SA, B_h and B_2h
5    h ← 1
6    while ∃ non-singleton h-bucket do
7      for each bucket[a, b]† do
8        Count[a] ← 0
9        for i ∈ [a, b] do
10         ISA[SA[i]] ← a
11       end
12     end
13     Cnt[ISA[n − h]] ← Cnt[ISA[n − h]] + 1
14     B_2h[ISA[n − h]] ← true
15     for each bucket[a, b] do
16       for i ∈ [a, b] do
17         s ← SA[i] − h
18         if s ≥ 0 then
19           head ← ISA[s]
20           ISA[s] ← head + Cnt[head]
21           Cnt[head] ← Cnt[head] + 1
22           B_2h[ISA[s]] ← true
23         end
24       end
25       for i ∈ [a, b] do
26         s ← SA[i] − h
27         if s ≥ 0 and B_2h[ISA[s]] then
28           u ← min{t : t > ISA[s] and (B_h[t] or not B_2h[t])}
29           for v ∈ [ISA[s] + 1, u − 1] do
30             B_2h[v] ← false
31           end
32         end
33       end
34     end
35     for i ∈ [0, n − 1] do
36       SA[ISA[i]] ← i
37       B_h[i] ← B_h[i] or B_2h[i]
38     end
39     h ← 2h
40   end
41   for i ∈ [0, n − 1] do
42     ISA[SA[i]] ← i
43   end
44   return SA
45 End
```

Figure III.1: Algorithme de MM.

balayage casier 1: $T_1 0$ est déplacé vers la position 1 de son casier, SA [0] - 1 = 10, ISA [10] = 1, définissez ISA [10] = 1 + Cnt [1], B_2h [1] = 1. et incrémenter Cnt [1].**balayage casier 2**: T_2 est déplacé vers la position 10 de son casier, SA [2] - 1 = 2, ISA [2] = 10, set ISA [2] = 10 + Cnt [10], B_2h [10] = 1. et incrémenter Cnt [10]. T_4 est déplacé vers la position 8 de son compartiment, SA [3] - 1 = 4, ISA [4] = 8, définit ISA [4] = 8 + Cnt [8], B_2h [8] = 1 et incrémente Cnt [8]. T_6 est déplacé vers la position 9 de son compartiment, SA [4] -1 = 6, ISA [6] = 9, définit ISA [6] = 9 + Cnt [9], B_2h [9] = 1 et incrémente Cnt [9]. T_9 est déplacé vers la position 11 de son compartiment, SA [5] -1 = 9, ISA [9] = 10, définit ISA [9] = 10 + Cnt [10], $B - 2h$ [11] = 1 et incrémente Cnt [10]. B_2h [11] est mis à 0 (en balayant à nouveau le compartiment). **balayage casier 3**: T_0 est déplacé vers la position 2 de son casier, SA [6] - 1 = 0, ISA [0] = 1, set ISA [0] = 1 + Cnt [1], B_2h [2] = 1 et incrémenter Cnt [1]. T_7 est déplacé vers la position 3 de son compartiment, SA [7] - 1 = 7, ISA [7] = 1, définissez ISA [7] = 1 + Cnt [1], B_2h [3] = 1 et incrémentez Cnt [1]. B_2h [3] est mis à 0 (en balayant à nouveau le compartiment). **balayage casier 4**: T_3

est déplacé vers la position 4 de son casier, *SA* [8] - 1 = 3, *ISA* [3] = 1, set *ISA* [3] = 1 + *Cnt* [1], B_2h [4] = 1 et incrémenter *Cnt* [1]. **balayage casier 5**: T_5 est déplacé vers la position 5 de son casier, *SA* [9] - 1 = 5, *ISA* [5] = 1, définissez *ISA* [5] = 1 + *Cnt* [1], B_2h [5] = 1. et incrémenter *Cnt* [1]. **balayage casier 6**: T_1 est déplacé vers la position 6 de son casier, *SA* [10] - 1 = 1, *ISA* [1] = 6, set *ISA* [1] = 6 + *Cnt* [6], B_2h [6] = 1 et incrémenter *Cnt* [6]. T_8 est déplacé vers la position 7 de son compartiment, *SA* [11] - 1 = 8, *ISA* [8] = 6, définissez *ISA* [8] = 6 + *Cnt* [6], B_2h [7] = 1 et incrémentez *Cnt* [6]. B_2h [7] est mis à 0 (en balayant à nouveau le compartiment). A la fin de l'étape h = 1, on a:

i	$B_h[i]$	$B_2h[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	1	2	11 = $
1	1	1	5	6	10 = a$
2	1	1	0	10	0 = abracadabra$
3	0	0	0	4	7 = abra$
4	1	1	0	8	3 = acadabra$
5	1	1	0	5	5 = adabra$
6	1	1	2	9	1 = bracadabra$
7	0	0	0	3	8 = bra$
8	1	1	1	7	4 = cadabra$
9	1	1	1	11	6 = dabra$
10	1	1	2	1	2 = racadabra$
11	0	0	0	0	9 = ra$

Table III.2: Balayage h-casier.

Notez que lors du balayage h-casier, nous avons dérivé de nouvelles positions pour les suffixes T_{i-h} et les avons stockées dans *ISA* [i]. À la fin de l'étape, nous avons effectué le déplacement réel, ce qui est démontré dans l'algorithme 1, lignes 35 à 38. De plus, nous avons fusionné les valeurs de $B_h$et B_2h en effectuant une opération OU logique entre elles et en stockant les résultats dans B_h. Au début de l'étape h = 2, on a:

i	$B_h[i]$	$B_2h[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	0	2	11 = $
1	1	1	1	6	10 = a$
2	1	1	0	10	0 = abracadabra$
3	0	0	0	4	7 = abra$
4	1	1	0	8	3 = acadabra$
5	1	1	0	5	5 = adabra$
6	1	1	0	9	1 = bracadabra$
7	0	0	0	2	8 = bra$
8	1	1	0	6	4 = cadabra$
9	1	1	0	10	6 = dabra$
10	1	1	0	1	2 = racadabra$
11	0	0	0	0	9 = ra$

Table III.3: Début du balayage h-casier.

Le nombre de compartiments est passé de 6 à 9. À l'étape précédente, l'algorithme a trié avec succès trois suffixes: T_3, T_5 et T_10. À ce stade, nous disposons de 6 casiers simples et de 3 casiers qui doivent encore être triés. Nous réinitialisons Cnt $[i]$ et B_2h $[i]$ en définissant Cnt $[i] = 0$ pour tout i, B_2h $[ISA$ $[n$ - $h]] = 1$ et en incrémentant Cnt $[ISA$ $[n$ - $h]]$. Ensuite, nous calculons de nouvelles valeurs pour ISA $[i]$ qui sont dérivées des nouveaux numéros de compartiment. Nous effectuons les mêmes opérations que dans l'étape précédente, mais avec les valeurs du tableau ci-dessus. En conséquence, à la fin de l'étape $h = 2$, on a:

i	$B_h[i]$	$B_2h[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	0	2	11 = $
1	1	1	1	7	10 = a$
2	1	1	2	11	0 = abracadabra$
3	0	0	0	4	7 = abra$
4	1	1	1	8	3 = acadabra$
5	1	1	1	5	5 = adabra$
6	1	1	2	9	8 = bra$
7	1	1	0	3	1 = bracadabra$
8	1	1	1	6	4 = cadabra$
9	1	1	1	10	6 = dabra$
10	1	1	2	1	9 = ra$
11	1	1	0	0	2 = racadabra$

Table III.4: L'étape 2 du balayage h-casier.

Nous avons déplacé T_9 à 10, T_8 à 6, T_5 à 5, T_1 à 7, T_3 à 4, T_6 à 9, T_2 à 11, T_4 à 8, T_0 à 2, T_7 à 3. Au début de l'étape $h = 4$, on a:

i	$B_h[i]$	$B_2h[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	0	2	11 = $
1	1	1	0	7	10 = a$
2	1	1	0	11	0 = abracadabra$
3	0	0	0	4	7 = abra$
4	1	1	0	8	3 = acadabra$
5	1	1	0	5	5 = adabra$
6	1	1	1	9	8 = bra$
7	1	1	0	2	1 = bracadabra$
8	1	1	0	6	4 = cadabra$
9	1	1	0	10	6 = dabra$
10	1	1	0	1	9 = ra$
11	1	1	0	0	2 = racadabra$

Table III.5: L'étape 3 du balayage h-casier.

À partir de l'étape précédente, nous avons trié de manière unique les suffixes T_1, T_2, T_8 et T_9. En conséquence, au début de cette étape, nous avons 10 casiers singleton sur 11. Seuls deux suffixes ont le même rang T_0 et T_7 et doivent donc être triés. Après exécution des lignes 7 à 39, nous obtenons les résultats suivants à la fin de l'étape $h = 4$:

i	$B_h[i]$	$B_2h[i]$	$Cnt[i]$	$ISA[i]$	$SA[i]$
0	1	1	0	3	11 = $
1	1	1	0	7	10 = a$
2	1	1	2	11	0 = abracadabra$
3	0	1	0	4	7 = abra$
4	1	1	1	8	3 = acadabra$
5	1	1	1	5	5 = adabra$
6	1	1	1	9	8 = bra$
7	1	1	1	2	1 = bracadabra$
8	1	1	1	6	4 = cadabra$
9	1	1	1	10	6 = dabra$
10	1	1	0	1	9 = ra$
11	1	1	1	0	2 = racadabra$

Table III.6: L'étape 4 du balayage h-casier.

Nous avons déplacé T_7 à 2, T_6 à 9, T_3 à 4, T_1 à 7, T_4 à 8, T_0 à 3, T_2 à 11 et T_5 à 5. L'algorithme met à jour *SA* et B_h. La boucle while principale se termine car tous les compartiments sont triés. L'algorithme reconstruit *ISA* [*i*] à partir de *SA* [*i*] pour tout *i* et sort le tableau de suffixes *SA*.

III.2.2 Algorithme de Larsson et Sadakane

Le tri des casiers est l'un des principaux facteurs affectant l'efficacité des *SACA* à préfixe doublé. **Manber** et **Myers** [5] (*MM*) ont été les premiers à décrire et à appliquer la technique de préfixe-doublage dans leur *SACA* à l'aide d'un tri en fonction du temps linéaire. Les principes de fonctionnement de l'algorithme *MM* sont basés sur le balayage des suffixes dans l'ordre *h*, en les triant implicitement en se déplaçant vers le haut de leur compartiment et en ré-étiquetant les compartiments dans un ordre $2h$. Tout ce processus prend du temps linéaire. Cependant, le *MM* est inefficace dans la pratique en raison du nombre important de travaux redondants [16].
Selon [16,57,63], l'algorithme de **Larsson** et **Sadakane** [53] (*LS*) est l'une des implémentations les plus efficaces basée sur la technique de préfixe-doublage dans un contexte non parallèle. Bien que sa complexité temporelle soit identique à celle de l'algorithme *MM*, $O(n \log n)$, en pratique, il est beaucoup plus rapide. L'algorithme *LS* est basé sur l'algorithme *MM* avec plusieurs ajustements importants. Premièrement, le *LS* élimine les numérisations inutiles. Deuxièmement, cela évite une grande quantité de travail redondant, tel que le regroupement inactif de parties déjà triées d'*ISA*. Troisièmement, il garde l'espace mémoire au même niveau que le *MM*. L'algorithme *LS* n'analyse pas l'ensemble du tableau; au lieu de cela, il garde la trace des casiers triés et les ignore lors des tours suivants. La *LS* trie explicitement chaque *h*-casier à l'aide du tri rapide de Bentley-McIlroy (**BMQS**) [64]. Le **BMQS** est une version optimisée du quicksort qui utilise un partitionnement à extrémité divisée et intègre une solution efficace pour l'échange d'éléments. Les systèmes **QMQS** sont égaux pour les éléments et les amènent au milieu du tableau, tandis que seuls les éléments distincts sont triés de manière récursive. Cette méthode est très efficace pour les entrées avec un grand nombre d'éléments égaux, car il faut moins de temps pour échanger des éléments égaux et les placer au milieu du tableau que de faire des appels récursifs.
Larsson et **Sadakane** ont formulé le problème comme suit. Soit $T[0 \cdots n] = T[0]\ T[1] \cdots T[n]$ est une chaîne composée de $n + 1$ caractères, de sorte que la chaîne d'entrée réelle occupe le sous-réseau $T[0 \cdots n)$ et $T[n] = \$$ est la sentinelle. À chaque étape *h*, un tableau entier *I* stocke les suffixes dans leur ordre *h*. Les valeurs de *I* sont des entiers appartenant à l'intervalle $[0, n]$. En d'autres termes, le tableau entier I est un tableau de suffixes qui ordonne les suffixes lexicographiquement en fonction des *h* premiers symboles. *I* est dans l'état final lorsque chaque suffixe peut être distingué de manière unique. L'algorithme *LS* utilise l'observation suivante de **Manber** et **Myers**:

Observation 2: Lorsque nous balayons les suffixes dans l'ordre *h*, nous pouvons utiliser le rang du suffixe T_{i+h} comme clé de tri pour le suffixe T_h et le placer dans l'ordre $2h$.

L'observation 2 trouve son application dans l'algorithme *LS* de la manière suivante. Initialement, nous plaçons tous les suffixes dans leur ordre 1 en les triant lexicographiquement en considérant uniquement le premier caractère de chaque suffixe. Ensuite, lorsque $h \geq 1$, $h = 2^i$, la position du suffixe $T_i + 2^{h-1}$ calculé à l'étape 2^{h-1} est une clé de tri pour le suffixe *Ti*. Dans chaque itération, nous doublons h et considérons deux fois plus de symboles par suffixe que dans l'itération précédente. Au total, ce processus prend O (log n) itérations jusqu'à ce que chaque compartiment devienne un singleton.
Larsson et Sadakane introduisent les concepts suivants dans la description de leur mise en œuvre. Un sous-réseau $I\ [i \cdots j]$ contenant un nombre maximal de suffixes adjacents égaux lexicographiquement selon les premiers symboles est un groupe. Le numéro de groupe d'un groupe est la position du dernier suffixe appartenant à ce groupe, c'est-à-dire pour tous les suffixes du groupe $I\ [i \cdots j]$ le numéro de groupe est j. Un groupe qui ne contient qu'un seul suffixe est un groupe trié; sinon, le groupe n'est pas trié. Tous les groupes triés voisins sont fusionnés en un groupe trié combiné.
L'algorithme *LS* conserve trois tableaux d'entiers *I*, *V* et *L*. Le rôle de *I* et *V* est similaire à celui du tableau de suffixes d'ordre *h* et de son inverse. **Larsson** et **Sadakane** utilisent *I* pour stocker des suffixes dans leur ordre *h*. Nous utilisons *V* pour *mapper* chaque suffixe d'ordre *h* à son rang calculé à partir du numéro de groupe auquel il appartient. Par rapport à l'approche *MM*, **Larsson** et **Sadakane** marquent les groupes en fonction de la position du suffixe le plus à droite dans chaque groupe. Nous utilisons L pour suivre les longueurs des groupes et facilite le processus de fusion des groupes triés. Si un groupe occupant un sous-réseau $I\ [i \cdots j]$ n'est pas trié, alors nous définissons $L\ [i] = j - i + 1$, et s'il s'agit d'un groupe trié combiné, nous annulons la valeur et définissons $L\ [i] = -(j - i + 1)$. Les valeurs négatives utilisées pour marquer la longueur du groupe trié combiné permettent à l'algorithme de sauter par-dessus les groupes triés.
Dans la première étape, nous affectons une position à chaque suffixe directement à partir de la chaîne d'entrée *T*. Ensuite, les suffixes sont triés par leur position initiale et stockés dans *I*. L'algorithme associe chaque suffixe T_i à sa position i. Chaque suffixe est trié par son premier caractère, c'est-à-dire pour chaque suffixe T_i, l'algorithme utilise $T\ [i]$ comme clé de tri et sa position i comme valeur. Ensuite, l'algorithme initialise V en attribuant le numéro de groupe à chaque suffixe, identifie les groupes triés et non triés et stocke les résultats dans L. Ceci met fin à la phase d'initialisation et donne l'ordre 1. L'algorithme maintient le tri *I* par étapes en doublant la valeur de *h* dans chaque étape. Notez que

Observation 3: Lorsque les suffixes sont dans l'ordre *h*, chaque suffixe T_i appartenant à un groupe trié possède un rang distinct, déduit de son préfixe $i + h$.

Cela signifie qu'une fois que certains groupes sont triés, la position des suffixes dans ces groupes est fixée dans *I*. Le problème est réduit à la réorganisation des groupes non triés. Nous trions chaque groupe non trié $I\ [j \cdots k]$ en utilisant $V\ [I\ [i] + h]$ comme clé de tri pour le suffixe i, pour tout $i \in I\ [j \cdots k]$. Les numéros de groupe uniques obtenus à partir de $V\ [I\ [i] + h]$ définissent la partition de $I\ [j \cdots k]$ dans de nouveaux groupes. Cela me met dans l'ordre $2h$. Nous calculons les numéros de groupe pour l'ordre de $2h$

et mettons à jour L en conséquence. Une description de haut niveau de l'algorithme *LS* (version de base) est donnée ci-dessous:

1. Remplissez I avec une suite de nombres de 0 à n représentant la position de départ de chaque suffixe dans la chaîne en entrée $x = x_0x_1 \cdots x_n$. Sort I en utilisant x_i comme clé pour i. Réglez h sur 1.
2. Pour chaque suffixe i, $0 \leq i \leq n$, définissez $V\ [i] = j$, où j est la position la plus à droite de chaque groupe dans I contenant le suffixe i.
3. Pour chaque groupe non trié occupant le sous-réseau $I\ [i\ \cdots\ j]$ définissez $L\ [i] = j$ - i + 1, et s'il s'agit d'un groupe trié combiné, définissez $L\ [i]$ = - (j - i + 1).
4. Traitez chaque groupe non trié occupant le sous-réseau $I\ [k \cdots$l] avec *BMQS*, en utilisant $V\ [I\ [i] + h]$ comme clé pour chaque suffixe i dans $I\ [k \cdots$ l].
5. Pour chaque paire de clés uniques $V\ [I\ [i] + h]$ et $V\ [I\ [j] + h]$, telles que i 6 = j, où le suffixe i et le suffixe j sont dans le groupe non trié $I\ [k \cdots$ l] marque les positions de division i et j.
6. Double h. Combinez la longueur des groupes triés, utilisez les positions de scission pour partitionner I en nouveaux groupes, mettez à jour V et L en conséquence.
7. 7. Si tous les groupes sont triés, c'est-à-dire $L\ [0]$ = -n, arrêtez-vous. Sinon, passez à 4.

Nous exécutons l'algorithme *LS* avec la chaîne d'entrée abracadabra \$. Lors de l'initialisation (étapes 1 à 3), nous trions les suffixes en utilisant x_i comme clé pour i et stockons le résultat dans I. Tout d'abord, nous remplissons I de nombres représentant les positions de départ des suffixes.

$$I = \overset{0}{\mathtt{a}}\overset{1}{\mathtt{b}}\overset{2}{\mathtt{r}}\overset{3}{\mathtt{a}}\overset{4}{\mathtt{c}}\overset{5}{\mathtt{a}}\overset{6}{\mathtt{d}}\overset{7}{\mathtt{a}}\overset{8}{\mathtt{b}}\overset{9}{\mathtt{r}}\overset{10}{\mathtt{a}}\overset{11}{\mathtt{\$}},$$

puis nous trions les suffixes en fonction du premier symbole et créons des groupes avec des symboles lexicographiquement égaux

$$I = \overset{11}{\mathtt{\$}}\,\Big|\,\overset{0}{\mathtt{a}}\overset{3}{\mathtt{a}}\overset{5}{\mathtt{a}}\overset{7}{\mathtt{a}}\overset{10}{\mathtt{a}}\,\Big|\,\overset{1}{\mathtt{b}}\overset{8}{\mathtt{b}}\,\Big|\,\overset{4}{\mathtt{c}}\,\Big|\,\overset{6}{\mathtt{d}}\,\Big|\,\overset{2}{\mathtt{r}}\overset{9}{\mathtt{r}},$$

où chaque groupe est séparé par la barre verticale et la position de départ de chaque suffixe d'ordre h est affichée au-dessus de celui-ci. Ensuite, nous calculons les numéros de groupe en utilisant la position la plus à droite du suffixe dans chaque groupe lorsque nous numérisons de gauche à droite. Nous créons de nouveaux groupes, attribuons des numéros de groupe à chaque suffixe de la chaîne d'entrée et stockons le résultat dans V:

$$\overset{0}{\$}\,|\,\mathtt{a\,a\,a\,a\,\overset{5}{a}}\,|\,\mathtt{b\,\overset{7}{b}}\,|\,\overset{8}{\mathtt{c}}\,|\,\overset{9}{\mathtt{d}}\,|\,\mathtt{r\,\overset{11}{r}} \quad \Rightarrow \quad V = \overset{5}{\mathtt{a}}\,\overset{7}{\mathtt{b}}\,\overset{11}{\mathtt{r}}\;\overset{5}{\mathtt{a}}\,\overset{8}{\mathtt{c}}\,\overset{5}{\mathtt{a}}\,\overset{9}{\mathtt{d}}\,\overset{5}{\mathtt{a}}\,\overset{7}{\mathtt{b}}\,\overset{11}{\mathtt{r}}\;\overset{5}{\mathtt{a}}\,\overset{0}{\$}.$$

Dans l'étape suivante, nous calculons la longueur de chaque groupe et les plaçons dans L:

$$L = \overset{-1}{\$}\,|\,\overset{5}{\mathtt{a}}\,\mathtt{a\,a\,a\,a}\,|\,\overset{2}{\mathtt{b}}\,\mathtt{b}\,|\,\overset{-1}{\mathtt{c}}\,|\,\overset{-1}{\mathtt{d}}\,|\,\overset{2}{\mathtt{r}}\,\mathtt{r}.$$

Ainsi, les groupes sont disposés comme suit: le groupe 5 a la longueur 5, les groupes 7 et 11, les deux ont la longueur 2. Les groupes 0,8 et 9 ont une longueur négative et sont considérés comme triés.

Exemple. Nous exécutons l'algorithme LS(version de base) avec la chaîne d'entrée x = *abracadabra $* et un symbole sentinelle $ ajouté à la fin de la chaîne. L'algorithme se déroule de haut en bas. Au début de chaque étape de doublage h, nous énumérons les clés $V\ [I\ [i] + h]$ utilisées pour trier les suffixes I [i] dans chaque groupe non trié. Nous démontrons le contenu de V, L et comment il est mis à jour à travers différentes étapes.

	i	0	1	2	3	4	5	6	7	8	9	10	11
h	x_i	a	b	r	a	c	a	d	a	b	r	a	\$
	$I[i]$	**11**	0	3	5	7	10	1	8	**4**	**6**	2	9
	$V\ [I[i]]$	0	5	5	5	5	5	7	7	8	9	11	11
	$L[i]$	-1	5					2		-2		2	
1	$V\ [I[i]+h]$		7	8	9	7	0	11	11			5	5
	$I[i]$		**10**	0	7	**3**	**5**	1	8			2	9
	$V\ [I[i]]$		1	3	3	4	5	7	7			11	11
	$L[i]$	-2		2		-2		2		-2		2	
2	$V\ [I[i]+h]$			11	11			4	1			8	0
	$I[i]$			0	7			**8**	**1**			**9**	**2**
	$V\ [I[i]]$			3	3			6	7			10	11
	$L[i]$	-2		2		-8							
4	$V\ [I[i]+h]$			8	0								
	$I[i]$			**7**	**0**								
	$V\ [I[i]]$			2	3								
	$L[i]$	-12											
	$I[i]$	11	10	7	0	3	5	8	1	4	6	9	2

Table III.7: exécution de l'algorithme LS(version de base) avec la chaîne d'entrée $x =$ *abracadabra \$* et un symbole sentinelle \$ ajouté à la fin de la chaîne.

Dans l'exemple, une fois le groupe trié, nous mettons en gras le suffixe qu'il héberge pour indiquer que sa position est fixée dans I et pour aider le lecteur à visualiser et à suivre le résultat final. Pour comparer le suffixe d'ordre h lexicographiquement, nous utilisons le symbole suivant $\vee$.
Au début de l'étape $h = 1$, nous avons trois groupes non triés 5,7 et 11. Pour trier les suffixes du groupe 5, nous devons les comparer en fonction des deux premiers symboles: ab $\vee$ ac $\vee$ et $\vee$ ab $\vee$ a \$. Comme les suffixes sont déjà triés en fonction de leurs premiers symboles, la tâche se limite à comparer $b \vee c \vee d \vee b \vee \$$. Nous appelons BMQS et fournissons les touches $V\ [I\ [i]+1]$ 7, 8, 9, 7, 0 et les valeurs correspondantes 0, 3, 5, 7, 10. Le résultat du tri est 0, 7, 7, 8, 9, ce qui place les suffixes dans l'ordre suivant: 10, 0, 7, 3, 5. Nous marquons les positions de division définies par les clés uniques 0, 8, 9 et traitons la prochaine groupe non trié. En traitant les groupes non triés 7 et 11, nous remarquons que les suffixes de ces groupes génèrent des clés $V\ [I\ [i]+1]$ égales et ne peuvent pas être distingués lexicographiquement à ce stade. L'algorithme passe à l'étape suivante en mettant à jour V et L en conséquence.
Au début de l'étape $h = 2$, on a:

$$I = \overset{11}{\$}\,\Big|\,\overset{10}{a}\,\Big|\,\overset{0}{a}\ \overset{7}{a}\,\Big|\,\overset{3}{a}\,\Big|\,\overset{5}{a}\,\Big|\,\overset{1}{b}\ \overset{8}{b}\,\Big|\,\overset{4}{c}\,\Big|\,\overset{6}{d}\,\Big|\,\overset{2}{r}\ \overset{9}{r},$$

et les groupes sont disposés comme suit

0 |1 | 3 |4 |5 | 7 |8 |9 | 11
$ |a |a a |a |a |b b |c |d |r r ⇒ $V =$ abr acadabr a$.
(V = 3 7 11 4 8 5 9 3 7 11 1 0)

Tous les groupes, sauf 3, 7 et 11 sont triés. Les clés $V\ [I\ [i] + 2]$ pour les suffixes du groupe 3 sont $V\ [0 + 2] = 11$ et $V\ [7 + 2] = 11$. Comme les clés ne sont pas uniques, l'algorithme passe au groupe non trié suivant. . Les clés pour les suffixes du groupe 7 sont $V\ [1 + 2] = 4$ et $V\ [8 + 2] = 1$. Les clés pour les suffixes du groupe 11 sont $V\ [2 + 2] = 8$ et $V\ [9 + 2] = 0$. Dans cette passe, les groupes 7 et 11 sont entièrement triés, car les clés des suffixes de ces groupes sont uniques. Les positions de division sont 1, 2, 8 et 9. Elles sont utilisées pour mettre à jour V et L en conséquence. Au début de l'étape $h = 4$, on a:

$I =$ 11 | 10 | 0 7 | 3 | 5 | 8 | 1 | 4 | 6 | 9 | 2
$ |a |a a |a |a |b |b |c |d |r |r,

et les groupes sont disposés comme suit

0 |1 | 3 |4 |5 |6 |7 |8 |9 |10 |11
$ |a |a a |a |a |b |b |c |d |r |r ⇒ $V =$ abr acadabr a$.
(V = 3 7 11 4 8 5 9 3 6 10 1 0)

Seul le groupe 3 est laissé non trié. L'algorithme passe au groupe 3 et calcule les clés $V\ [I\ [i] + 4]$. La clé pour le suffixe 0 est $V\ [0 + 4] = 8$ et la clé pour le suffixe 7 est $V\ [7 + 4] = 0$. Le suffixe 7 vient avant le suffixe 0 en fonction des clés. L'algorithme marque les positions de division 0 et 7, les mises à jour V et L. Tous les groupes sont maintenant triés, l'algorithme se termine et génère I.

III.3 Algorithmes Récursifs

L'algorithme *skew* (également appelé *DC3*) [17] est un *SACA* à temps linéaire récursif pour les alphabets entiers développé par **Karkkainen** et **Sanders** [3]. Les **SACA** récursives comportent trois étapes: (1) Construire un sous-ensemble de suffixes de taille 2/3 ou moins. Trier récursivement ce sous-ensemble. (2) Construisez un sous-ensemble des suffixes restants et triez-le en utilisant le résultat de (1). (3) Fusionner les deux sous-ensembles en un seul. Pour illustrer cette approche, nous décrivons la méthode de **Karkkainen** et **Sanders**.
Soit $T\ [0,\ n) = t_0 t_1 \cdots t_{n-1}$ soit la chaîne d'entrée de n caractères sur l'alphabet = 1, 2, $\cdots$, σ et que $t_j = \$$ pour $j \geq n$ soit un caractère spécial (appelé sentinelle) plus petit que tout autre caractère de l'alphabet. L'algorithme de biais suit les étapes suivantes:

1. Construisez un tableau de suffixes commençant aux positions $S_{12} = i$: i mod $3 \neq$ 0. Définir S_1 $S_1 = i$: i mod $3 = 1$ et $S_2 = i$: i mod $3 = 2$.

2. Trier S_{12} uniquement:
   (a) Construire des chaînes

   $R_1 = [t_1 t_2 t_3]\ [t_4 t_5 t_6] \cdots\ [t_{max} S_1 t_{max} S_1 + 1\ t_{max} S_1\ + 2]$,
   $R_2 = [t_2 t_3 t_4]\ [t_5 t_6 t_7] \cdots\ [t_{max} S_2 t_{max} S_2\ + 1\ t_{max} S_2\ + 2]$,

   où $[t_i t_{i+1} t_{i+2}]$ est un triple des trois premiers caractères des suffixes S_1 et S_2. Soit $R = R_1 R_2$ la concaténation de R_1 et R_2.

   (b) Encode chaque triple de R avec une valeur correspondante de $R_1 R_2$.

   (c) Obtenez une désignation lexicographique pour chaque triple de R en effectuant un tri de base sur celui-ci et en mappant un rang sur sa position dans R.

   (d) Si la dénomination lexicographique n'est pas unique, exécutez récursivement l'algorithme *skew* en passant les rangs de R en entrée.

3. Trier S_0:
   (a) Pour chaque i mod 3 = 0, générez un tuple ($T\ [i]$, $ISA_{12}\ [i\ 1]$), où $T\ [i]$ est un caractère unique à la position i et $ISA_{12}\ [i\ +\ 1]$ indique le rang dans S_{12}. du suffixe $i\ +\ 1$.

   (b) Appliquez *Radixsort* aux n-uplets ($T\ [i]$, $ISA_{12}\ [i\ +\ 1]$).

4. Fusionner S_0 et S_{12}:
   (a) $S_i j \in S_1$, comparez alors les nuplets formés ($T\ [j]$, $ISA_{12}\ [j\ +\ 1]$) (T [i], ISA_{12} $[i\ +\ 1]$).

(b) $S_i j \in S_2$ alors comparez les triples formés (T [j], T [j + 1], ISA_{12} [j + 2]) (T [i], T [i + 1], ISA_12 [i + 2]).

L'exécution de l'algorithme est illustrée à l'aide de la chaîne

$$T[0,10] = \overset{0}{a}\,\overset{1}{b}\,\overset{2}{r}\,\overset{3}{a}\,\overset{4}{c}\,\overset{5}{a}\,\overset{6}{d}\,\overset{7}{a}\,\overset{8}{b}\,\overset{9}{r}\,\overset{10}{a},$$

où le tableau de suffixe final sera SA = (10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2).

Étape 1. Nous construisons S_1,S_2 etS_{12}: S_1 = 1, 4, 7, 10, S_2 = 2, 5, 8 et S_12 = 1, 4, 7, 10, 2, 5, 8.

Étape 2. Pour trier S_12, nous formons les chaînes de triples suivantes: R_1 = [*bra*] [*cad*] [*abr*] [*a*$$] et R_2 = [*rac*] [*ada*] [*bra*]. En concaténant R_1 et R_2, nous obtenons:

$$R = \underset{1}{[bra]}\ \underset{4}{[cad]}\ \underset{7}{[abr]}\ \underset{10}{[a\$\$]}\ \underset{2}{[rac]}\ \underset{5}{[ada]}\ \underset{8}{[bra]}.$$

Radix triant et générant des rangs lexicographiques:

$$\underset{1}{[a\$\$]}\ \underset{2}{[abr]}\ \underset{3}{[ada]}\ \underset{4}{[bra]}\ \underset{4}{[bra]}\ \underset{5}{[cad]}\ \underset{6}{[rac]}.$$

Mapper les rangs aux triples dans R:

$$R = \underset{4}{[\texttt{bra}]}\ \underset{5}{[\texttt{cad}]}\ \underset{2}{[\texttt{abr}]}\ \underset{1}{[\texttt{a\$\$}]}\ \underset{6}{[\texttt{rac}]}\ \underset{3}{[\texttt{ada}]}\ \underset{4}{[\texttt{bra}]}.$$

Soit T_12 = (4, 5, 2, 1, 6, 3, 4) un tableau de rangs lexicographiques de R. Etant donné que le rang 4 apparaît deux fois, les rangs ne sont pas distincts, l'algorithme est appliqué récursivement en retournant le tableau (4, 5, 1, 0, 6, 2, 3), dont les positions se rapportent aux positions de T_{12}. En les mappant sur des positions dans T, nous obtenons S_{12} = (10, 7, 5, 8, 1, 4, 2), qui est trié de manière unique.

Étape 3. Pour trier S_0, nous formons des n-uplets conformément au protocole, puis nous les comparons. Tuples composés:

$$\begin{array}{llll} 0\text{: } \texttt{abracadabra} & \rightarrow & (\texttt{a}, \mathrm{ISA}_{12}[1]) & \rightarrow (\texttt{a}, 5) \\ 3\text{: } \texttt{acadabra} & \rightarrow & (\texttt{a}, \mathrm{ISA}_{12}[4]) & \rightarrow (\texttt{a}, 6) \\ 6\text{: } \texttt{dabra} & \rightarrow & (\texttt{d}, \mathrm{ISA}_{12}[7]) & \rightarrow (\texttt{d}, 2) \\ 9\text{: } \texttt{ra} & \rightarrow & (\texttt{r}, \mathrm{ISA}_{12}[10]) & \rightarrow (\texttt{r}, 1) \end{array}$$

En exécutant Radixsort, nous obtenons
T [0] < T [3] < T[6] < T [9] car (a, 5) ¡(a, 6) <(d, 2) <(r, 1). Ainsi triés S_0 = (0, 3, 6, 9).

Étape 4. Pour illustrer la procédure de fusion, nous construisons la **Table III.8**. et la remplissons de n-uplets et de triples, comme défini dans le protocole d'algorithme.

S_0				S_{12}						
0	3	6	9	10	7	5	8	1	4	2
(a, 5)	(a, 6)	(d, 2)	(r, 1)	(a, 0)	(a, 4)			(b, 7)	(c, 3)	
(a, b, 7)	(a, c, 3)	(d, a, 4)	(r, a, 0)			(a, d, 2)	(b, r, 1)			(r, a, 6)

Table III.8: tableau, qui incluons tous les n-uplets et triplets possibles utilisés pour la comparaison dans une routine de fusion.

Dans notre exemple courant, nous avons T [10, 12) < T [0, 12) depuis (a, 0) ¡(a, 5). Nous avons aussi que T [9, 12) < T [2, 12) puisque (r, a, 0) <(r, a, 6). Dans le tableau suivant, nous incluons tous les nuplets et triplets possibles utilisés pour la comparaison dans une routine de fusion.

La routine de fusion produit un tableau de suffixe final SA = (10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2). Chaque étape sauf l'appel récursif s'exécute dans le temps O (n). L'appel récursif est exécuté sur une chaîne T_{12} de longueur $\lceil 2n$ / 3]. Ainsi, le temps d'exécution

de l'algorithme est défini par la récurrence $T(n) = T(2n / 3) + \mathrm{O}(n)$. Selon le théorème principal [65], la solution à cette récurrence est $T(n) = \mathrm{O}(n)$, qui est linéaire comme prévu.

III.4 Algorithmes de tri induit

La technique de tri induit déduit l'ordre des suffixes non triés de l'ensemble des suffixes déjà triés qui sont classés en fonction de certains critères. Le tri induit Les *SACA* sont efficaces et rapides dans la pratique. Par exemple, l'implémentation de **Okanohara** et **al**. [66] est basé sur le tri induit,ainsi *libdivsufsort* [67] considéré par [68, 69] comme le SACA le plus rapide fonctionnant en mémoire principale. Pour illustrer cette approche, nous décrivons la méthode de **Nong** et **al**. [61], qui incorpore une combinaison de sous-chaînes *LMS* pour réduire le problème et un tri pur pour faciliter la propagation de l'ordre des suffixes.

Soit $X[1, n] = x[1]\, x[2] \cdots x[n]$ est la chaîne d'entrée de n caractères sur un alphabet indexé $\sum$ et que le dernier caractère de x soit la sentinelle telle que $x[n] = \$$. Suffixes de type S et de type L. Un suffixe $X[i, n]$ est classé comme suffixe de type S (plus petit) si $X[i, n] < X[i + 1, n]$, sinon si $X[i, n]$¿ $X[i + 1, n]$ il est classé comme suffixe de type L (plus grand). Dans le cas où $X[i, n] = X[i + 1, n]$, alors $X[i, n]$ est classé comme type de $X[i + 1, n]$. Un suffixe composé uniquement de la sentinelle, tel que $X[n, n] = \$$, est classé dans le type *S*. Suffixes et sous-chaînes de type *LMS*. Un suffixe $X[i, n]$, pour $1 < i \leq n$, est classé comme suffixe du type *LMS* (type S le plus à gauche) si $X[i]$ est un suffixe du type S et $X[i - 1]$ est du type L suffixe. Une sous-chaîne $X[i, j]$ est classée dans la sous-chaîne de type *LMS* si $X[i, n]$ et $X[j, n]$ sont tous deux des suffixes de type *LMS* et le type de suffixe $X[k, n]$, $i < k < j$ n'est pas du type *LMS*. Un suffixe composé uniquement de la sentinelle, tel que $X[n, n] = \$$, est également considéré comme une sous-chaîne de type LMS.

L'ordre des sous-chaînes de type *LMS* est déterminé en les comparant lexicographiquement. Dans le cas où les caractères des deux sous-chaînes sont égaux, nous rompons le lien en vérifiant le type de suffixe correspondant. Les suffixes de type S ont une priorité plus élevée que les suffixes de type L.

Préfixes de type *LMS*. Un préfixe $X[1, i]$ est classé en tant que préfixe de type *LMS* s'il se compose d'un seul suffixe de type *LMS* ou un suffixe $X[j, n]$ est classé en tant que suffixe *LMS*-type, où j est la première position après i. Si le suffixe $X[i, n]$ est classé comme suffixe de type *S*, un préfixe *LMS* $X[1, i]$ est également de type *S*. De même, si le suffixe $X[i, n]$ est classé comme suffixe de type L, un préfixe *LMS* $X[1, i]$ est également de type L.

```
1  function SAIS (X, SA, n, σ);
   Input  : X is the input string, SA is an empty suffix array, n is the
            length of X, σ is the size of indexed alphabet Σ.
   Output: SA containing sorted lexicographically suffixes (their
            starting positions) of X.
2  Define four integer arrays: t(n), X_1(n), P_1(n), B(n);
3  Classify all the S-type and L-type suffixes and store them in t;
4  Classify all the LMS-substrings and store them in P_1;
5  Induced-sort all the LMS-substrings with the aid of P_1 and B;
6  Create X_1 using the names of LMS-substrings;
7  if each character in X_1 is unique then
8  |  SA_1[X_1[i]] = i for all i;
9  else
10 |  Recursively call SAIS(X_1, SA_1, n_1, σ_1);
11 end
12 Induce SA from SA_1;
13 return
```

Figure III.2: Algorithme de *SAIS*.

La mise en œuvre de **Nong** et **al**. est décrit dans l'algorithme de la **Figure III.2** ci-dessus, nous fournissons un exemple d'exécution de cet algorithme en utilisant la chaîne d'entrée $X = abracadabra\$$. Nous supposons que X est un tableau de caractères indexé à zéro qui se termine par une sentinelle \$. Initialement, nous balayons X, classifions les suffixes de type S / L et stockons le résultat dans le type array t. Nous marquons tous les suffixes de type *LMS* par et procédons à l'exécution de l'algorithme en trois étapes.

Étape 1. Nous classons les suffixes 3, 5, 7 et 11 comme étant du type *LMS*. Ensuite, nous identifions et étiquetons les casiers comme suit. Nous appelons un compartiment contenant des caractères identiques consécutifs i comme compartiment i. L'algorithme regroupe les suffixes dont le premier caractère est identique en 6 compartiments associés aux caractères \$, a, b, c, d et r, comme illustré aux lignes 5 et 6. Nous définissons toutes les valeurs de *SA* sur une valeur négative et balayons X pour attribuer à tous les éléments. Suffixes de type *LMS* dans les compartiments correspondants. Les suffixes 3, 5 et 7 du type *LMS* commençant par le caractère identique a, ils sont tous placés dans le compartiment a. Le suffixe *LMS* 11 est placé dans le compartiment \$. À ce stade, tous les préfixes *LMS* de longueur un sont triés.

Étape 2. Aux fins d'illustration, nous marquons avec ^ la tête du casier qui est en cours de numérisation. De plus, nous utilisons le symbole @ pour indiquer quel élément de SA est en cours de traitement. Lorsque nous traitons le premier élément de *SA*, qui est *SA* [0] = 11 (ligne 9), nous utilisons cette valeur pour découvrir que le suffixe 10 commence par le caractère a et est de type L. Ensuite, nous l'ajoutons au godet a et décalons la tête d'un pas en avant. Nous répétons ce processus jusqu'à la fin de l'*AS*. À ce moment-là, tous les préfixes *LMS* de type L dans *SA* sont triés (ligne 18). Lorsque ^ pointe vers l'espace entre deux compartiments, l'un d'eux, ou les deux, est plein.

Étape 3. Maintenant, nous utilisons les préfixes de type L triés pour induire l'ordre de tous les préfixes LMS dont la longueur est supérieure à un. Nous marquons la limite droite de chaque compartiment avec ^ et analysons SA de droite à gauche. Lorsque nous traitons le dernier élément de SA, $SA\ [11] = 2$ (ligne 21), nous utilisons cette valeur pour découvrir que le suffixe 1 commence par le caractère b et est de type S. Ensuite, nous l'ajoutons au panier b et décalons d'un cran vers la gauche. Nous répétons cette procédure jusqu'au début de la SA. Lorsque le balayage SA est terminé, tous les préfixes LMS sont classés en fonction de l'ordre déduit des préfixes de type L triés (ligne 33). Ensuite, pour réduire le problème et appliquer une approche diviser et conquérir, nous divisons X en un tableau plus petit X_1 et le remplissons avec les noms générés pour les sous-chaînes LMS comme suit. Nous cartographions les suffixes 3, 5, 7 et 11 en 2, 3, 1 et 0 (ligne 35). Puisque chaque valeur de X_1 est unique, nous calculons SA_1 directement à partir de $X1$. L'algorithme induit SA à partir de SA_1 et renvoie le résultat.

```
00 Index:    00 01 02 03 04 05 06 07 08 09 10 11
01 X:         a  b  r  a  c  a  d  a  b  r  a  $
02 t:         S  S  L  S  L  S  L  S  S  L  L  S
03 LMS:                *     *     *           *
04 Etape 1:  ______________________________________________
05 Bucket:     $         a            b      c    d     r
06       SA: {11} {-1 -1 07 03 05} {-1 -1} {-1} {-1} {-1 -1}
07 Etape 2:  ______________________________________________
08       SA: {11} {-1 -1 07 03 05} {-1 -1} {-1} {-1} {-1 -1}
09            @^    ^                ^       ^    ^    ^
10           {11} {10 -1 07 03 05} {-1 -1} {-1} {-1} {-1 -1}
11             ^   @   ^             ^       ^    ^    ^
12           {11} {10 -1 07 03 05} {-1 -1} {-1} {-1} {09 -1}
13             ^       ^ @           ^       ^    ^       ^
14           {11} {10 -1 07 03 05} {-1 -1} {-1} {06} {09 -1}
15             ^       ^    @        ^       ^      ^     ^
16           {11} {10 -1 07 03 05} {-1 -1} {-1} {06} {09 02}
17             ^       ^       @     ^       ^      ^       ^
18           {11} {10 -1 07 03 05} {-1 -1} {04} {06} {09 02}
19             ^       ^             ^         ^    ^       ^
20 Etape 3:  ______________________________________________
21       SA: {11} {10 -1 07 03 05} {-1 -1} {04} {06} {09 02}
22             ^                 ^      ^    ^    ^      @^
23           {11} {10 -1 07 03 05} {-1 01} {04} {06} {09 02}
24             ^                 ^   ^       ^    ^    @  ^
25           {11} {10 -1 07 03 05} {08 01} {04} {06} {09 02}
26             ^                 ^ ^         ^   @^       ^
27           {11} {10 -1 07 03 05} {08 01} {04} {06} {09 02}
28             ^             ^     ^        @^    ^       ^
29           {11} {10 -1 07 03 05} {08 01} {04} {06} {09 02}
30             ^          ^        ^   @     ^    ^       ^
31           {11} {10 -1 00 03 05} {08 01} {04} {06} {09 02}
32             ^       ^           ^ @       ^    ^       ^
33           {11} {10 07 00 03 05} {08 01} {04} {06} {09 02}
34             ^    ^              ^         ^    ^       ^
35       X1:  2  3  1  0
```

Figure III.3: Exemple d'exécution de l'algorithme *SAIS*, en utilisant la chaîne d'entrée X = $abracadabra\$$, avec les trois étapes.

Chapitre IV

Implémentations

IV.1 Introduction

Dans le cadre de notre travail , on présente une implémentation d'un algorithme parallèle pour la construction de tableau de suffixes -également la *BWT*- inspirée d'une approche différente de celles du tri induit. Dans la première section de ce chapitre, on discute notre algorithme naïf pour la construction de la table de suffixes. Puis en deuxième section de ce chapitre, on explique le principe de notre algorithme principale pour la construction de la Transformée de **Burrows-Wheeler** ainsi que les différentes étapes de ça réalisation . Il consiste brièvement à diviser le texte initial à des parties (dites pivots) quasi-égales pour faciliter le calcule de la Transformée .

IV.2 Construction de la table de suffixes

L'approche naïve pour construire un tableau de suffixes comporte à utiliser un algorithme de tri basé sur la comparaisons des sous-chaine. Pour cela on a utilisé la fonction *sort* fournie par *C++* .
Le principe est simple et préalablement déclaré, il consiste à ordonner tous les suffixes d'une chaîne initiale suivant l'ordre alphabétique des suffixes. Cela réelement un nombre de comparaison O(n log n) , mais chaque comparaison prend un temps linéaire de O(n) , donc le temps d'exécution global pour cette approche est de O(n^2 log n)[1].

[1] https://en.wikipedia.org/wiki/Suffix_array#Correspondence_to_suffix_trees

```
20 // Construire le tableau de suffixes
21 int *ConstTabSuff(char *txt, int n)
22 {
23         struct suffix suffixes[n];
24
25
26         for (int i = 0; i < n; i++)
27         {
28                 suffixes[i].index = i;
29                 suffixes[i].suff = (txt+i);
30         }
31
32         // Ordoner le suffixes avec la fonction sort
33         sort(suffixes, suffixes+n, cmp);
34
35         // Enregistrer les indexes des suffixes ordonés
36         int *suffixArr = new int[n];
37         for (int i = 0; i < n; i++)
38                 suffixArr[i] = suffixes[i].index;
39
40
41         return suffixArr;
```

Figure IV.1: Fonction *ConstTabSuff*(*char* , *int*) illustrant la méthode naïve pour la construction de la table suffixes.

A fin de bien comprendre la fonction présentée dans la **Figure IV.1** on a l'exemple suivant:

Exemple 1. Soit le texte initial:*abracadabra*

Suffixes	Suffices triés
0 abracadabra	10 a
1 bracadabra	7 abra
2 racadabra	0 abracadabra
3 acadabra	3 acadabra
4 cadabra	5 adabra
5 adabra	8 bra
6 dabra	1 bracadabra
7 abra	4 cadabra
8 bra	6 dabra
9 ra	9 ra
10 a	2 racadabra

Table IV.1: tri alphabétique des suffixes de la chaine *abracadabra*.

Donc la table de suffixe pour abracadabra sera : 10,7,0,3,5,8,1,4,6,9,2.
Avec un test simple sur l'implémentation illustrée dans la **Figure IV.1**, on obtient les même résultats (**Figure IV.2**).

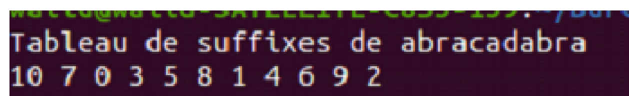

Figure IV.2: Affichage après l'exécution de l'algorithme sur la chaine *abracadabra*.

IV.3 Construction de la BWT

Pour le développement de notre algorithme , on s'inspirait de [70].**Liu** et **al** proposa un algorithme parallèle nommé *CX1* pour la construction de *BWT* mais sur *GPU* non pas *CPU*.
Soit $\sum = c_1, c_2, \cdots, c_\sigma$ un alphabet fini et ci est lexicographiquement moins que c_j pour $1 \leq i < j \leq \sigma$. donnant $S = s_0 s_1 \cdots s_{m-1} \in \sum$, on note la sous-chaîne $s_i s_{i+1} \cdots s_j$ par $S[i,j]$,pour $1 \leq i < j \leq m$-1 . Une sous-chaîne de type $S[0,i]$ est dite préfixe , tandis que une sous-chaîne de type $S[i,m$-1] est dite suffixe de *S*. Généralement durant la construction de BWT d'une chaîne d'entrée, On ajoute souvent un symbole de fin \$ qui est lexicographiquement plus petit que chaque symbole ou caractère de $\sum$. La *BWT* d'un suffixe est le caractère juste avant la position de départ du suffixe, et pour un suffixe commençant à 0, la *BWT* est défini comme symbole de fin \$.
Soit R=S_1 , S_2 ,$\cdots$, S_n une collection de n sous-chaînes .Chacune des sous-chaînes Si est de taille m+1 avec $S_i[k] \in \sum$ $(0 \leq k \leq m)$ et $S_i[m] = \$_j$, avec $\$_i < \$_j$ pour $0 \leq i < j \leq n$. Le *BWT* de *R* est similaire à celui d'une seule chaîne. Premièrement, nous trions de manière lexicographique les suffixes de toutes les chaînes de *R*. Chaque chaîne a m+1 suffixes, ainsi la liste triée contient $n(m+1)$ suffixes différents. Une fois que la

liste triée L est générée, nous pouvons définir la BWT de R, noté BWT_R, sous la forme d'une chaîne de longueur $n(m+1)$, telle que $BWT_R\ [i]$ soit la BWT du suffixe $L\ [i]$, pour $0 \leq i \leq n(m+1)$.

IV.3.1 Méthode

On trie tous les suffixes des sous-chaînes dans R pour construire la BWT de R. Toutefois, étant donné un grand nombre de caractères ou de séquences génomiques, il est impossible de trier tous les suffixes en mémoire. Considérons que la construction BWT d'un milliard de séquences à une longueur de 100 caractères/séquence. Chaque séquence a 100 suffixes non vides d'une longueur comprise entre 1 et 100. Si nous énumérons explicitement tous les suffixes du milliard de séquence, il y aura environ 5 billions de caractères. Il faut au moins 1250 gigaoctets pour stocker tous ces caractères si on donne un exemple de l'*ADN* (2 bits-par-caractère). Évidemment, une telle exigence dépasse de loin la capacité de mémoire de la plupart des PC ou des serveurs de base. Pour être efficace sur le plan de mémoire, nous adoptons une stratégie similaire au tri des suffixes par blocs proposé dans [70, 71]. Notre approche comporte principalement deux étapes: le partitionnement et le tri. Le déroulement des opérations est illustré à la **Figure IV.3** ci-dessous.

Le partitionnement

Comme nous l'avons mentionné plus haut, il n'est pas pratique d'énumérer tous les suffixes d'une grande collection de sous-chaînes en mémoire. Diviser la liste des suffixes en partitions plus petites est une approche évidente, qui a été utilisée dans CX1. Nous adoptons la même stratégie. Au début on choisit p positions (suffixes) aléatoires dans le texte initial , donc on aura *p+1* partitions. Puis on effectue un tri lexicographique simple sur les suffixes obtenus après le partitionnement. Par la suite, on parcourt le texte entier, chaque suffixe lexicographiquement compris entre deux pivots *i* et *i+1* choisis pour le partitionnement, doit appartenir à la même partition. Ainsi, l'ordre des suffixes des différentes partitions est absolument déterminé par la partition à laquelle ils appartiennent, et nous pouvons trier chaque partition séparément et concaténer simplement les partitions pour former la liste triée de tous les suffixes.

Pour **l'allocation mémoire** on ajoute un vecteur de bits (*bit_vect*) initialisé à -1 pour indiquer qu'il est vide. *bit_vect* se comporte comme suit:

- $bit_vect = 0 :\ S_i \notin P_j$ (S_i signifie le i-ème suffixe et P_j signifie la j-ème partition)
- $bit_vect = 1 :\ S_i \in P_j$

Tri des suffixes

Pour accélérer le processus de chargement, nous chargeons les sous-chaînes en parallèle comme suit. Considérons le scénario du chargement d'un fichier avec n sous-chaînes à l'aide de p processeurs. Nous pouvons également partitionner n sous-chaînes en p parties, chaque partie contenant n/p sous- chaînes. Les parties p sont ensuite chargées

simultanément dans la mémoire principale des processeurs . Ensuite, nous effectuons le processus de partitionnement sur chaque processeur (ou cœur) simultanément. Ensuite, les suffixes de les fichiers temporaires seront triés en parallèle.
Pour cela, on a utilisé *quicksort.*
Quicksort divise d'abord un grand tableau en deux sous-tableaux plus petits : les éléments bas et les éléments hauts. *Quicksort* peut alors trier récursivement les sous-tableaux. Les étapes sont les suivantes :

1. Choisir un élément, appelé pivot, dans le tableau.

2. Partitionnement : réorganiser le tableau de façon à ce que tous les éléments ayant des valeurs inférieures au pivot viennent avant le pivot, tandis que tous les éléments ayant des valeurs supérieures au pivot viennent après lui (des valeurs égales peuvent aller dans les deux sens). Après cette séparation, le pivot est dans sa position finale. C'est ce qu'on appelle l'opération de partition.

3. Appliquer récursivement les étapes ci-dessus au sous-tableau d'éléments avec des valeurs inférieures et séparément au sous-tableau d'éléments avec des valeurs supérieures.

Le cas de base de la récursivité est constitué de tableaux de taille zéro ou un, qui sont en ordre par définition, de sorte qu'ils n'ont jamais besoin d'être triés. Les étapes de sélection du pivot et de partitionnement peuvent être effectuées de plusieurs manières différentes ; le choix de schémas d'implémentation spécifiques affecte grandement les performances de l'algorithme. Le principe de *quicksort* est illustré dans la **Figure IV.4**

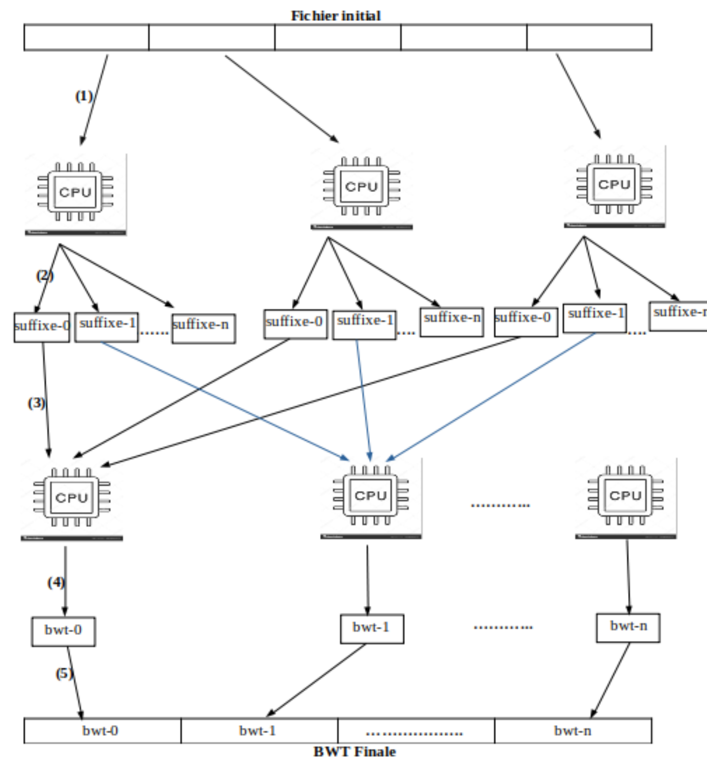

Figure IV.3: Déroulement des opérations de l'algorithme. **(1)** plusieurs processeurs chargent différentes parties des sous-chaînes contenues dans un fichier initial; **(2)** Les suffixes sont partitionnés dans des parties qui leurs convient; **(3)** plusieurs processeurs chargent et trient les suffixes qui appartiennent aux mêmes partitions; **(4)** les codes BWT de chaque partition sont sortis dans des fichiers temporaires; **(5)** concaténation des fichiers temporaires pour obtenir le BWT final.

```
partitionner(tableau T, entier premier, entier dernier, entier pivot)
    échanger T[pivot] et T[dernier]  // échange le pivot avec le dernier du tableau
, le pivot devient le dernier du tableau
    j := premier
    pour i de premier à dernier - 1 // la boucle se termine quand i = (dernier-1).
        si T[i] <= T[dernier] alors
            échanger T[i] et T[j]
            j := j + 1
    échanger T[dernier] et T[j]
    renvoyer j

tri_rapide(tableau T, entier premier, entier dernier)
        si premier < dernier alors
            pivot := choix_pivot(T, premier, dernier)
            pivot := partitionner(T, premier, dernier, pivot)
            tri_rapide(T, premier, pivot-1)
            tri_rapide(T, pivot+1, dernier)
```

Figure IV.4: Principe de l'algorithme *quicksort*

Le partitionnement peut être fait en temps linéaire, en place. La mise en œuvre la plus simple consiste à parcourir le tableau du premier au dernier élément, en formant la partition au fur et à mesure : à la *i-ème* étape de l'algorithme ci-dessous, les éléments $T[0],\cdots,T[j\text{-}1]$ sont inférieurs au pivot, tandis que $T[j],\cdots,T[i\text{-}1]$ sont supérieurs au pivot.

Cet algorithme de partitionnement rend le tri rapide non stable : il ne préserve pas nécessairement l'ordre des éléments possédant une clef de tri identique. On peut résoudre ce problème en ajoutant l'information sur la position de départ à chaque élément et en ajoutant un test sur la position en cas d'égalité sur la clef de tri.

Le partitionnement peut poser problème si les éléments du tableau ne sont pas distincts. Dans le cas le plus dégénéré, c'est-à-dire un tableau de n éléments égaux, cette version de l'algorithme a une complexité quadratique. Plusieurs solutions sont possibles : par exemple se ramener au cas d'éléments distincts en utilisant la méthode décrite pour rendre le tri stable, ou bien utiliser un autre algorithme.

L'élagage des résultats:

D'après la définition de *BWT*, nous pouvons voir qu'une liste strictement triée des suffixes n'est pas notre objectif final. Une fois les suffixes triés, nous prenons une autre mesure pour collecter le code BWT de chaque suffixe. Le BWT peut donc être calculé en temps linéaire en construisant d'abord un tableau de suffixe du texte, puis en déduisant la chaîne BWT:

$BWT[i]{=}S[A[i]\text{-}1]$.

IV.3.2 Parallélisme

Notre approche peut être fortement parallélisé, soit dans la partie de séparation (partitionnement), soit dans la partie de tri. Notez que pour ne pas manquer certains suffixes, la partie tri ne doit pas être invoquée tant que la partie partitionnement n'est pas terminée. Étant donné que la collection de sous-chaînes est répartie également, le temps de partitionnement de chaque pièce ne diffère que très légèrement, de sorte que le temps d'attente est assez court.

Dans la partie partitionnement, l'accélération est presque linéaire par rapport au nombre de processeurs employés. Cependant, il existe un souci que quand trop de processeurs écrivent sur le même disque dur, la performance d'E/S sera limitée par la vitesse d'écriture du disque dur. Une solution possible est d'utiliser plus de disques durs pour éviter les bourrages d'écriture(JAM). Dans la partie de tri, le degré de parallélisme est aussi élevé que le nombre de partitions des suffixes, plus il y a de parties, plus il y a de parallélisme. Dans le but d'avoir de meilleures performances dans notre algorithme on a pu inclure les trois directives spécifique à **OpenMP** :

- *#pragma omp parallel num_threads(numThreads)*: il permet de spécifier le nombre de threads qui doivent exécuter le bloc suivant;

- *#pragma omp single nowait*: un seul thread sera autorisé pour exécuter le bloc structuré. Le reste des threads attendent à la barrière implicite à la fin de la construction unique, sauf si *nowait* est spécifié. Donc si *nowait* est spécifié, alors le reste des threads exécutent immédiatement le code après le bloc structuré;

- *#pragma omp task*: utilisé pour identifier le bloc du code qui sera exécuté en parallèle avec le bloc en dehors de la région.

IV.3.3 Coût mémoire

Pour mesurer l'espace de travail total alloué par le programme, on a utilisé *malloc_count* [72], un outil d'analyse pour mesurer la quantité de mémoire allouée d'un programme au moment de son exécution. L'outil fonctionne en interceptant les fonctions standard *malloc*(), *free*(), *realloc*(), *calloc*() etc, et en ajoutant des statistiques de comptage simples à chaque appel.
Donc chaque appel à *malloc*() ou aux autres fonctions est transmis aux niveaux inférieurs, et le *malloc*() régulier est utilisé pour l'allocation de pointeurs.
Les différentes allocations et libérations de la mémoire dans notre programme sont comme suit :

- Le vecteur de bits temporaire (*bit_vect*) occupe 5Mo;
- Un pivot occupe 4.4Mo;
- La bwt occupe 9.4Mo;
- La table de suffixes temporaire prend 11.3Mo d'espace;
- La fonction *realloc*() pour la réallocation de la table de suffixe occupe 11Mo;
- La fonction *free*() de *bit_vect*, pivots et bwt libère environ 27Mo.

Donc au total l'espace utilisé par le programme sera : 5 + 4.4 + 9.4 + 11.3 + 11 - 27 = 14.1Mo
Et comme perspective on peut gagner encore plus d'espace mémoire en écrivant la BWT générée directement sur disque ainsi que le vecteur de bits (*bit_vect*) car il sont des données qui ne seront pas modifiées par la suite .

IV.4 Conclusion

Dans ce chapitre on a présenté nos algorithmes qu'ils ont pour but -principalement- de réduire la consommation mémoire suivant l'approche de **partitionnement** du texte, tout en utilisant le parallélisme (multithread) afin de restreindre le temps d'exécution de ces algorithmes.

Chapitre V

Tests Et Résultats

V.1 Introduction

Cette partie du mémoire a pour but de mettre en œuvre notre implémentation. Premièrement, on présente les différents équipements matériel et logiciels utilisés pour la construction de notre système de test. Par la suite, on présente les tests faits sur notre algorithme ainsi que les différents résultats obtenus. Et on fini par le calcul de l'efficacité de l'algorithme précédemment présenté dans le **chapitre IV**.

V.2 Environnent d'implémentation

V.2.1 Équipements Matériels

IBNBADIS – Moyens de calcul intensif du CERIST

Le calcul intensif est devenu aujourd'hui un des critères de crédibilité scientifique d'un établissement de recherche. Le **CERIST** vise le leadership du calcul intensif en Algérie afin de répondre aux besoins des applications scientifiques académiques et industrielles. Le CERIST met à la disposition de la communauté scientifique algérienne la plateforme de Calcul Haute Performance (**IBNBADIS**) qui leur fourni la puissance de calcul nécessaire pour les expérimentations large échelle de leurs travaux de recherche. Cette plateforme est destinée à des utilisateurs de divers domaines et appartenant à diverses institutions algériennes. On a utilisé pour nos calculs et tests le **Cluster IBNBADIS** fourni par l'organisme d'accueil CERIST. IBNBADIS est une plateforme de calcul haute performance à 32 nœuds de calcul, chacun composé de deux processeurs *Intel*(*R*) *Xeon*(*R*) *CPU E*5-2650 2.00*GHz*. Chaque processeur est doté de 8 cœurs, ce qui fait un total de 512 cœurs sur tout le Cluster. La puissance théorique crête de IBNBADIS est d'environ 8 **TFLOPS**.

Les logiciels installés sur IBNBADIS sont :

- *SLURM* pour la gestion des ressource et l'ordonnancement des jobs.
- Les compilateurs *C*/*C*++ et Fortran.
- *MPI* (Message Passing Interface).

V.2.2 Équipements Logiciels

L'environnement **Linux** (**Ubuntu**) (**version 18.04**) a été choisi pour l'implémentation et la validation de notre implémentation. **Ubuntu** [73] est une des nombreuses distributions du système d'exploitation **GNU/Linux**, souvent abrégée en Linux. Elle est inspirée d'une autre distribution (l'une des plus célèbres, nommée **Debian**) et est sponsorisée par la société **Canonical Ltd**. Ubuntu a été créé avec pour caractéristiques principales d'être:

- Conviviale : Cette distribution est donc l'une des plus faciles à l'utilisation.
- Simple : un seul outil pour chaque tâche qui fait ce qu'on demande.
- Libre : Comme un logiciel libre est le plus souvent gratuit et le code ouvert, l'avantage est que chacun peut participer à son développement.
- Gratuite : La gratuité fait partie intégrante de la philosophie d'Ubuntu. Tout le monde doit pouvoir avoir accès à un système d'exploitation et à des applications performantes sans être obligé de payer.

De plus, **Ubuntu** est considéré comme une distribution stable et sécurisée pour le développement, et supporte le parallélisme, c'est pourquoi nous avons décidé de l'utiliser dans notre travail.

V.2.3 Compilation

Le premier algorithme implémenté c'est l'algorithme naïf pour la construction de la table de suffixes. Il est écrit en *C++*. Le deuxième algorithme c'est notre algorithme principale pour la construction de la Transformée de Burrows-Wheeler, il a été développé en langage *C*. On a choisi *C/C++* car ceux sont des langages de programmation qui offrent une marge importante de contrôle sur la gestion de la mémoire.

La commande : **g++ -o exec main.cpp -O3** permet d'exécuter le premier algorithme;

tandis que la commande : **gcc str_quicksort_omp.c test_parallel_bwt1.c parallel_bwt.c malloc_count.c -ldl malloc_count.h -fopenmp -Wall -O3 -msse4.2 -o test_parallel_bwt** exécute le deuxième algorithme.

g++ : compilateur spécifique pour les programme *C++* ;
gcc : compilateur spécifique pour les programme *C* ;
-O3 : option d'optimisation de performances ;
-Wall : demande au compilateur d'afficher tous les messages de prévention (lorsque quelque-chose est ambiguë, par exemple) ;
-ldl : pour la compilation des fichiers d'entêtes ;
-msse4.2 [74]: c'est une technologie d'Intel (jeux d'instructions) qui permet d'améliorer les performances des fonctions de comparaison des chaînes de caractères.

V.3 Métriques de Performance

La mesure de performance et l'inclusion de cette mesure dans la boucle de développement ne sont pas habituels en informatique, en dehors du *HPC* et d'une partie du **Big Data**. Ce sont pourtant des concepts essentiels pour aboutir à des codes rapides et large échelle.

Les métriques de performance donc consistent à un ensemble de mesures pouvant être utilisées pour quantifier la qualité d'un algorithme. Pour les algorithmes séquentiels, les métriques temps et espace sont suffisantes. Pour les algorithmes parallèles, le scénario est un peu plus compliqué. Outre le temps et l'espace, des métriques telles que l'accélération et l'efficacité sont nécessaires pour étudier la qualité d'algorithme parallèle. De plus, lorsqu'un algorithme ne peut pas être complètement parallélisé, il est utile d'avoir une estimation théorique de l'accélération maximale possible. Dans ces cas, les lois d'**Amdahl** [75] et de **Gustafson** [76] deviennent utiles pour une telle analyse.

V.3.1 Accélération d'un traitement (Speedup)

Le speedup est la métrique la plus connue pour caractériser la performance d'une parallélisation. Sa définition initiale est simple et permet de chiffrer le gain obtenu par un algorithme et une implantation parallèles :

$$S(p) = T(1)/(p)$$

Où $T\ (1)$ et $T\ (p)$ sont respectivement les temps de traitement sur 1 et p ressources. Il peut s'agir de cœurs de calculs dans un processeur, de processeurs dans une machine à mémoire partagée, de nœuds (PCs) dans un **Cluster**, de disques dans un système de stockage massif $\cdots$

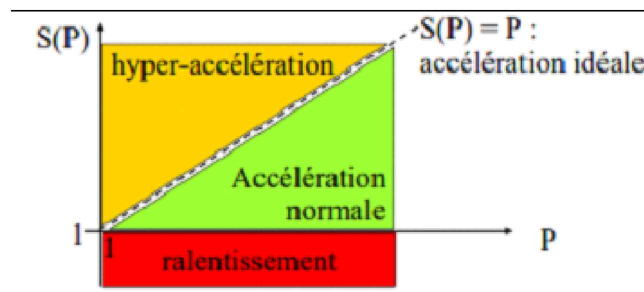

Figure V.1: Les trois types de l'accélération possibles.

Quand on peut mesurer un speedup S(p), on doit normalement obtenir : $1 < S(p) \leq p$, en espérant une valeur de $S(p)$ la plus proche possible de p (voir **Figure** V.1). Un *Speedup* inférieur à 1 correspond en fait à un ralentissement, et donc normalement à une parallélisation ratée. Mais dans le cas du traitement de très gros problèmes, on peut être amené à raisonner différemment. Un *Speedup* $S(p)$ supérieur à p correspond à une Hyper-Accélération, et possède toujours une explication. Par exemple, en utilisant plus de PC on aura utilisé plus de cœurs de calcul, mais aussi plus de mémoire, plus de cache, plus de disques, et on aura finalement cumulé plusieurs gains.

L'allure classique d'une courbe de speedup d'un problème de taille fixée est celle de la **Figure** V.2. Le début de la courbe est souvent proche de l'idéal ($S(p) = p$), puis s'en

écarte, et tangente un plafond horizontal. La présence d'un plafond est en fait inévitable, même sur une machine idéale dès qu'une partie du problème n'est pas parallélisable (voir la section qui suit sur la loi de *Amdhal*), et aussi à cause du coût non négligeable des communications dans les machines réelles. Si l'on augmente encore le nombre de processeurs utilisés, on peut même obtenir une chute du *Speedup* car les nouveaux processeurs ne seront presque pas utilisés et les temps de communications entre plus de processeurs deviendront plus importants. Il faut donc (généralement) utiliser un nombre de ressources de calcul raisonnable pour la taille de problème traité, ce que tente de mesurer l'efficacité (voir ci-après).

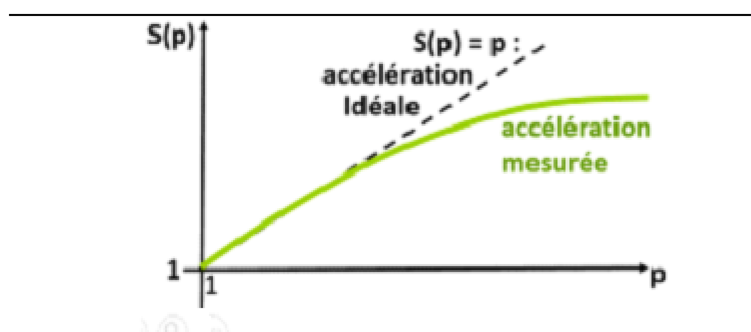

Figure V.2: l'allure classique d'une accélération expérimental.

.Loi de Amdahl (1967)

L'accélération d'un programme utilisant plusieurs processeurs en parallèle est limitée par le temps nécessaire pour la fraction série du problème. Si un problème de taille W a un composant série Ws l'accélération du programme est la suivante:

$$T_p = \frac{W - W_s}{p} + W_s$$

$$S = \frac{W}{(W - W_s)/p + W_s}$$

$$S \leq \frac{W}{W_s} \quad \text{as} \quad p \to \infty$$

Si Ws = 20%, W-Ws = 80%, alors :

$$S = \frac{1}{0.8 / p + 0.2}$$

$$S \leq \frac{1}{0.2} = 5 \quad \text{as} \quad p \to \infty$$

La loi de **Amdahl** implique que le calcul parallèle n'est utile que lorsque le nombre de processeurs est petit, ou lorsque le problème est parfaitement parallèle.

.Loi de Gutafson (1988)

Tout problème suffisamment important peut être efficacement parallélisé avec une accélération.

$S = p - \alpha\ (p\text{-}1)$

où p est le nombre de processeurs, et α est la partie série du problème. **Gustafson** a proposé un concept de temps fixe qui conduit à une accélération à grande échelle pour les problèmes de plus grande taille. Fondamentalement, nous utilisons des systèmes plus grands avec plus de processeurs pour résoudre des problèmes plus importants.

Le temps d'exécution du programme sur un ordinateur parallèle est $(a+b)$ où a est le temps séquentiel et b est le temps parallèle. La quantité totale de travail à effectuer en parallèle varie linéairement avec le nombre de processeurs. Ainsi b est fixé comme p est varié. La durée d'exécution totale est de $(a + p \times b)$ donc l'accélération est $(a{+}p \times b)/(a{+}b)$. Définir $\alpha = a/(a + b)$, la fraction séquentielle du temps d'exécution, d'où :

$S = p - \alpha\ (p\text{-}1)$

.Efficacité

E_p est l'efficacité d'un algorithme utilisant un nombre p de processeurs, elle indique à quel point les processeurs sont utilisés. E_p=1 est l'efficacité maximale et signifie une utilisation optimale des ressources. Une efficacité maximale est difficile à atteindre dans une solution mise en œuvre (c'est une conséquence de la réalisation difficile d'une accélération linéaire parfaite). Aujourd'hui, l'efficacité est devenue aussi importante que l'accélération, sinon plus, car elle mesure la qualité du matériel utilisé et elle indique quelles mises en œuvre doivent avoir la priorité lors de la compétition pour des ressources limitées.

V.4 Étude du Comportement de l'Algorithme de Construction de la BWT

Pour l'étude de l'efficacité de notre algorithme, on a utilisé le corpus de **Pizza & Chili** [77] dans les domaines suivants :

- **Traitement du texte :** on a utilisé un fichier qui est la concaténation de fichiers texte anglais sélectionnés parmi les collections *etext*02 à *etext*05 du projet de **Gutenburg** [78]. de taille 11*Mo*.
- **Bio-informatique :** on a utilisé un fichier de séquences d'*ADN* de gènes séparées par de nouvelles lignes (sans description, seulement le code *ADN* nu) obtenue à partir des fichiers 01*hgp*10 à 21*hgp*10, plus 0*xhgp*10 et 0*yhgp*10, du Projet **Gutenberg** [78] de taille 8*Mo*.
- **Wikis :** on a utilisé la bible de **KING JAMES** qui comporte plus de 10 000 lignes.

V.4.1 Étude du temps et de l'espace de Construction de l'Algorithme

Pour étudier le comportement de l'algorithme de construction de la *BWT*, on a choisi de faire des tests sur les 3 fichiers (*KING JAMES*, *ADN*, anglais). Les résultats obtenus seront comparés avec l'implémentation de **Okanohara** et **al**. [66] nommée *DBWT*, l'une des implémentations, basée sur le tri induit, les plus rapides existantes. On a varié le doublé *nsamples,nthreads* tel que *nsamples* représente le nombre de partitions et *nthreads* représente le nombre de *threads* utilisés. Les résultats des tests faits sur la bible de **KING JAMES** sont représentés dans le **Tableau** V.1 si-dessous.

nsamples,nthreads	**Temps d'exécution(secondes)**	**Espace mémoire (Mo)**
4,1	160	21
4,2	80	19
4,4	41,2	24
4,8	21	21,8
4,16	11,1	20,7
4,32	11,1	20
4,64	11,2	25
16,1	161	15,4
16,2	82	16,9
16,4	42,8	18,4
16,8	**22,4**	**21,8**
16,16	12,7	20,9
16,32	12,6	19,6
16,64	12,5	20,6
64,1	166	13,3
64,2	87	13,6
64,4	48,2	13,7
64,8	28,6	13,7
64,16	18,4	13,5
64,32	18,4	13,3
64,64	18,3	14

Table V.1: Le temps et l'espace nécessaires pour la construction de l'algorithme par rapport au nombre de partition et le nombre de threads pour le fichier de **KING JAMES**.

Les résultats obtenus après l'exécution des tests sur l'*ADN* sont représenté dans la **Tableau** V.2 (on a fixé le nombre de threads à 64 pour de meilleures performances sur les grands fichiers)

nsamples,nthreads	**Temps d'exécution(secondes)**	**Espace mémoire (Mo)**
128,64	44,6	20
256,64	66	19,5
512,64	66	19,4
1024,64	66	19,9

Table V.2: Le temps et l'espace nécessaires pour la construction de l'algorithme par rapport au nombre de partition et le nombre de threads pour le fichier d'*ADN*.

Le **Tableaux** V.3 représente les résultats des test faits sur le fichier du langage anglais.

nsamples,nthreads	**Temps d'exécution(secondes)**	**Espace mémoire (Mo)**
128,64	60	25,2
256,64	87	23,4
512,64	87	23,8
1024,64	88	24

Table V.3: Le temps et l'espace nécessaires pour la construction de l'algorithme par rapport au nombre de partition et le nombre de threads pour le fichier du langage anglais.

On a intégré l'outil de comptage de l'utilisation mémoire *malloc_count* à *DBWT* pour qu'on puisse mesurer l'espace consommé par la *DBWT*. Les résultats de calculs sur les trois fichiers (**KING JAMES**, **ADN**, **Anglais**)sont illustrés dans le **Tableau** V.4

Fichier	**Temps d'exécution(secondes)**	**Espace mémoire (Mo)**
KING JAMES	0,8	30.1
ADN	1,1	28.3
ANGLAIS	2,1	37

Table V.4: Temps d'exécution et espace utilisé par *DBWT* pour les trois fichiers (**KING JAMES, ADN, ANGLAIS**).

.Comparaison des résultats les résultats illustré dans le Tableau V.1 de notre algorithme et le Tableau V.4 de *dbwt*, montrent bien qu'avec la variation de l'espace et du temps en fonction des paramètres *nsamples* et *nthreads*, *nsamples* augmentant est sensé réduire l'espace et *nthreads* augmentant est sensé diminuer le temps d'exécution en testant sur le fichier de KING JAMES. *dbwt* a un temps idéal pour la construction de la Transformée (Tableau V.4) et un espace de calcul raisonnable ce qui montrent son efficacité.

.Interprétation des résultats d'après les comparaisons faites, on voit bien qu'on a pu rapprocher dans le temps -légèrement- de *DBWT*, l'une des plus rapides implémentations qui existent. En terme d'espace, les résultats obtenus par notre algorithme sont meilleurs que ceux obtenus par la *DBWT*, cela grâce à la grande marge de partitionnement appliqué sur notre algorithme.

Pour mieux comprendre les résultats soumis dans les tableaux si-dessus, on les a schématisé en graphe illustratif. Le graphe de la **Figure** V.3 montre que le temps de construction de l'algorithme. Pour le fichier de **KING JAMES** le temps diminue au fur et à mesure que le nombre de threads augmente jusqu'à l'arrivé au nombre maximal de

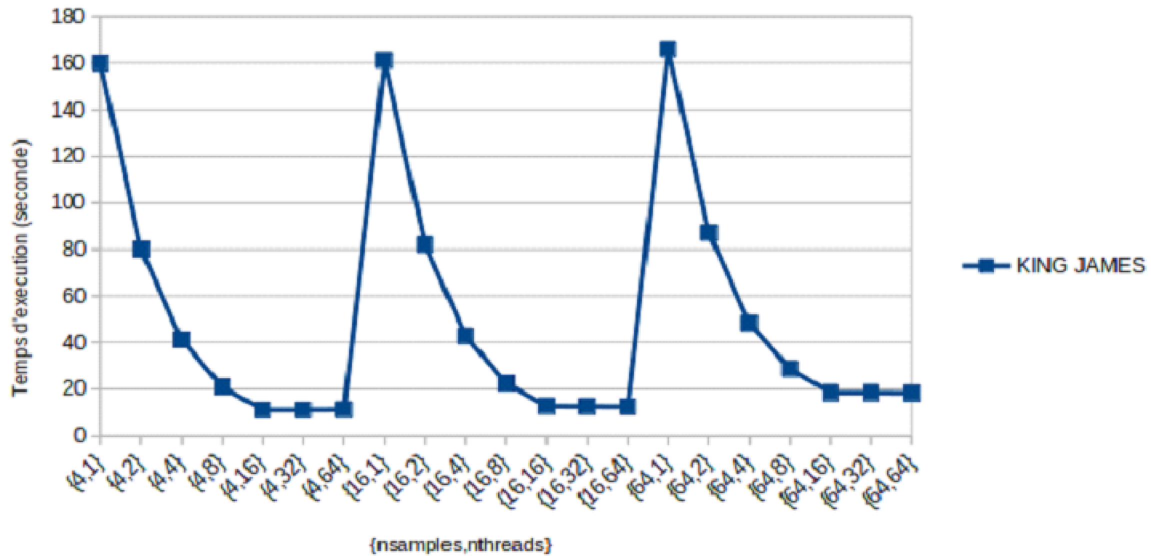

Figure V.3: Le temps de construction de la BWT pour le fichier de **KING JAMES**.

cœurs logiques de la machine (le Cluster) qui est 16 cœurs.

L'histogramme si-dessous représente le coût mémoire consommé par notre algorithme et l'algorithme de $DBWT$. (voir **Figure** V.4) On voit bien la consommation supérieure de la mémoire par dbwt par rapport à notre algorithme ce qui signifie l'efficacité en mémoire qu'on a pu aboutir par l'approche de partitionnement du texte intégral.

V.4.2 Calcul de l'Accélération (Speedup)

Le Graphe de la Figure V.5 si-dessous représente la courbe de l'accélération de notre algorithme de construction de la BWT exécuté sur le fichier de test **KING JAMES**. Nous remarquons que le speedup a été quasiment linéaire jusqu'à l'atteinte du nombre de cœurs logiques de la machine de test (16 cœur).

.Calcul de l'efficacité

On calcule la moyenne des efficacités :

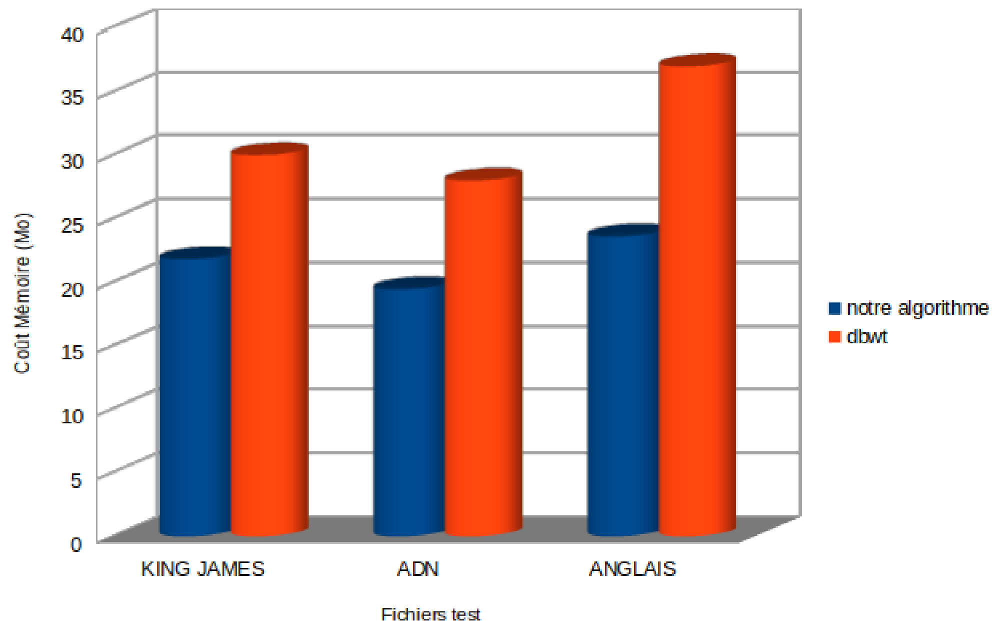

Figure V.4: Histogramme de la consommation mémoire des trois fichiers par notre algorithme et dbwt .

$Efficacite_{moyenne}$ =

$$\frac{\sum_{p=1}^{N}\left(\frac{S(p)}{p}\right)}{N}$$

$Efficacite_{moyenne} = 5{,}49\ /\ 7 = 0{,}78 = 78\%$

L'efficacité de notre algorithme est dite importante en rapprochant de 1.
Nous pouvant interpréter l'accélération (voire l'efficacité) rapide de l'algorithme comme il revient au grand nombre de threads lancés au moment de l'exécution Ce qui démontre que l'accélération était presque idéale.

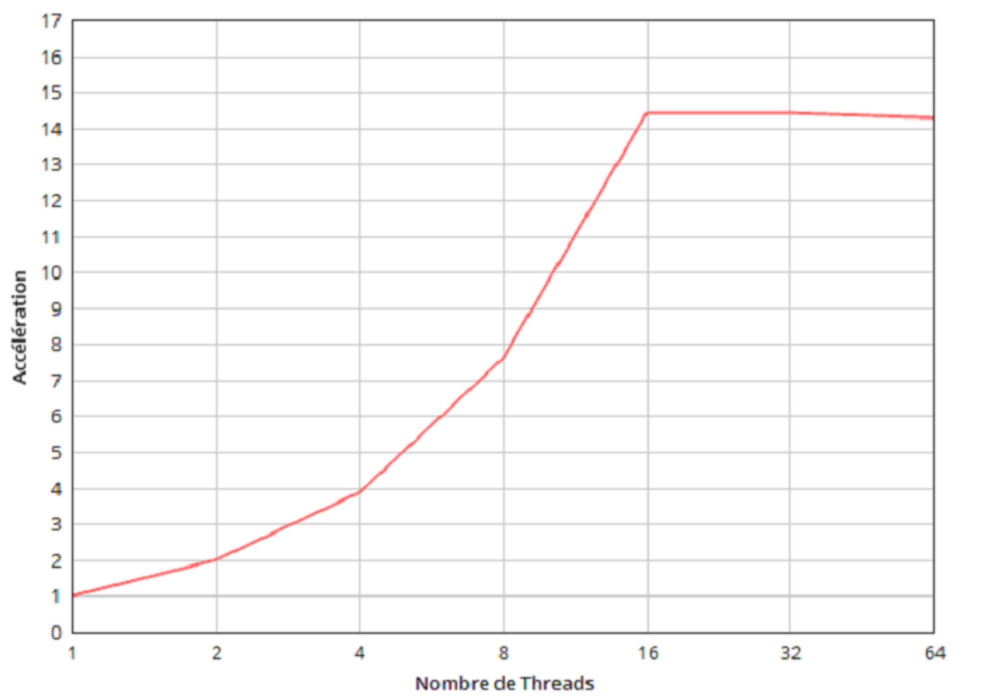

Figure V.5: Représentation de la courbe de l'accélération par rapport au nombre de threads lancés.

V.4.3 Conclusion

Dans ce chapitre on a fait une étude illustrative et comparative de l'algorithme de construction de la **Transformée de Burrows-Wheeler** présenté dans le **Chapitre IV**. Ceci nous a permet d'évaluer et de valider notre travail en faisant des test sur un Corpus de différents domaines.
Ces tests faits sur le Cluster **IBNBADIS** de l'organisme d'accueil nous ont donné des résultats très acceptable en espace mémoire et même en temps d'exécution. Les résultats obtenus nous ont permet de réduire le coût mémoire à l'aide du partitionnement du texte initial. D'autre part le temps d'exécution est réduit gràce au degré important du parallélisme appliqué à notre algorithme.

Conclusion Générale

Dans ce travail réalisé on s'est intéressé au problème de compression de données immenses. La Transformée de Burrows-Wheeler est l'un des moyens les plus efficace pour la résolution de ce problème. Elle consiste à regrouper les lettres identiques dans un texte pour avoir un taux de compression plus haut. Dans ce but, on à implémenté deux algorithmes. Le premier à pour but de construire un Tableau de Suffixes, d'une façon naïve, qui est une approche largement utilisée pour la construction de la BWT , tandis que le deuxième algorithme c'est notre algorithme principale pour la construction de la Transformée basée sur la table de suffixes.

Dans un premier temps, on a développé un algorithme naïf pour la construction de la table de suffixes. Un algorithme de tri basé sur la comparaison des sous chaînes. il consiste à ordonner tous les suffixes d'une chaîne initiale suivant l'ordre alphabétique des suffixes. En effet cet algorithme est coûteux que ce soit en mémoire ou son temps de construction (il ne passe pas vraiment à l'échelle), c'est pourquoi on lui a pas mené à des grands tests.

Le deuxième algorithme c'est l'algorithme principale efficace visant à construire la Transformée de Burrows-Wheeler passant par la construction d'un tableau de suffixes. L'approche derrière notre algorithme est de divisé le texte initial à des partitions, ce qui va nous servir à économisé l'espace de travail grâce à la grande marge de partitionnement nous proposons. Après on possède à l'étape de tri des suffixes qui mène à ordonner les suffixes obtenus pour chacune des partitions pour pouvoir calculer la BWT partielle spécifique à chaque partition. La BWT finale est calculé en concaténant les BWT partielles.

Il est bien connu que les architectures multicœurs sont beaucoup plus efficaces pour accélérer les calculs plutôt que les accès mémoires. C'est la raison pour laquelle nous avons réalisé un parallélisme sur la partie de tri des suffixes pour réduire le temps spécifique au tri.

Les tests effectués sur les différents jeu de données, montre bien que les résultats aboutis par notre algorithme sont largement satisfaisantes en terme du coût mémoire et acceptable en terme du temps de calcul.

Comme future perspective, nous suggérons :

- on peut gagner encore plus d'espace mémoire en écrivant la BWT générée directement sur disque ainsi que le vecteur de bits (bit_vect) car il sont des données qui ne seront pas modifiées par la suite .
- Et pour économiser encor plus de temps d'exécution on pourra paralléliser la partie de partitionnement et autre instructions de quicksort.

Références

[1] M BURROWS. A block-sorting lossless data compression algorithm. *SRC Research Report, 124*, 1994.

[2] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.

[3] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

[4] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

[5] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[6] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

[7] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. *Software: Practice and Experience*, 33(11):1035–1049, 2003.

[8] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2002.

[9] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *International Workshop on Algorithms in Bioinformatics*, pages 449–463. Springer, 2002.

[10] Kayhan Erciyes. *Distributed and sequential algorithms for bioinformatics*, volume 23. Springer, 2015.

[11] Louis F Williams Jr. A modification to the half-interval search (binary search) method. In *Proceedings of the 14th annual Southeast regional conference*, pages 95–101. ACM, 1976.

[12] Veli Mäkinen. Compact suffix array—a space-efficient full-text index. *Fundamenta Informaticae*, 56(1-2):191–210, 2003.

[13] S Baxter. *The bzip2 and libbzip2 official home page version 1.0.2*,2004. `http://www.sourceware.org/bzip2/`. Accessed: 2019-09-2.

[14] Charles Antony Richard Hoare. Algorithm 64: quicksort. *Communications of the ACM*, 4(7):321, 1961.

[15] Donald Knuth. Sorting and searching. *The art of computer programming*, 3:513, 1998.

[16] Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *acm Computing Surveys (CSUR)*, 39(2):4, 2007.

[17] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages, and Programming*, pages 943–955. Springer, 2003.

[18] Michael Burrows and David J Wheeler. A method for the construction of minimum-redundancy codes. 40(9):1098—-1101, 1952.

[19] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1):75–81, 1976.

[20] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[21] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[22] Ricardo A Baeza-Yates and Gaston H Gonnet. A new approach to text searching. In *ACM SIGIR Forum*, volume 23, pages 168–175. ACM, 1989.

[23] Michael J Fischer and Michael S Paterson. String-matching and other products. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1974.

[24] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[25] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[26] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[27] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[28] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.

[29] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242. Society for Industrial and Applied Mathematics, 2002.

[30] Rasmus Pagh. Low redundancy in static dictionaries with o (1) worst case lookup time. In *International Colloquium on Automata, Languages, and Programming*, pages 595–604. Springer, 1999.

[31] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

[32] Felipe A Louza, Guilherme P Telles, Simon Gog, and Liang Zhao. Computing burrows-wheeler similarity distributions for string collections. In *International Symposium on String Processing and Information Retrieval*, pages 285–296. Springer, 2018.

[33] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 636–645. Society for Industrial and Applied Mathematics, 2004.

[34] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[35] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56. Springer, 2005.

[36] Julian Seward. bzip2 and libbzip2. *avaliable at http://www. bzip. org*, 1996.

[37] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *bioinformatics*, 25(14):1754–1760, 2009.

[38] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[39] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 225–232. Society for Industrial and Applied Mathematics, 2002.

[40] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly fm-index. In *International Symposium on String Processing and Information Retrieval*, pages 150–160. Springer, 2004.

[41] Veli Mäkinen and Gonzalo Navarro. Run-length fm-index. In *Proc. DIMACS Workshop:"The Burrows-Wheeler Transform: Ten Years Later"(Aug. 2004)*, pages 17–19, 2004.

[42] Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *International Symposium on String Processing and Information Retrieval*, pages 229–241. Springer, 2007.

[43] Jeff Gilchrist. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, volume 16, pages 559–564, 2004.

[44] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. *Computer Physics Communications*, 182(1):266–269, 2011.

[45] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating system concepts. 9th, 2012.

[46] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.

[47] William Gropp, William D Gropp, Argonne Distinguished Fellow Emeritus Ewing Lusk, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[48] Juan Jose Costa, Toni Cortes, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Running openmp applications efficiently on an everything-shared sdsm. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 35. IEEE, 2004.

[49] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Programming distributed memory sytems using openmp. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[50] Charles JP Lebre. Fork-lift truck with synchronized variable travelling and lifting surfaces, July 10 1984. US Patent 4,458,786.

[51] Joe R Brown, Melissa M Garber, and Steven F Venable. Artificial neural network on a simd architecture. In *Proceedings., 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 43–47. IEEE, 1988.

[52] Richard M Karp, Raymond E Miller, and Arnold L Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 125–136. ACM, 1972.

[53] N Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[54] Dong K Kim, Junha Jo, and Heejin Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In *International Workshop on Experimental and Efficient Algorithms*, pages 301–314. Springer, 2004.

[55] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 251–260. IEEE, 2003.

[56] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 68–70. ACM, 2012.

[57] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Annual Symposium on Combinatorial Pattern Matching*, pages 55–69. Springer, 2003.

[58] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[59] Dror Baron and Yoram Bresler. Antisequential suffix sorting for bwt-based data compression. *IEEE Transactions on Computers*, 54(4):385–397, 2005.

[60] Michael A Maniscalco and Simon J Puglisi. An efficient, versatile approach to suffix sorting. *Journal of Experimental Algorithmics (JEA)*, 12:1–2, 2008.

[61] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 Data Compression Conference*, pages 193–202. IEEE, 2009.

[62] Y Mori. *sais version 2.4.1*. `https://sites.google.com/site/yuta256/sais`. Accessed: 2019-09-2.

[63] Mrinal Deo and Sean Keely. Parallel suffix array and least common prefix for the gpu. In *ACM SIGPLAN Notices*, volume 48, pages 197–206. ACM, 2013.

[64] Jon L Bentley and M Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993.

[65] Chee Yap. A real elementary approach to the master recurrence and generalizations. In *International Conference on Theory and Applications of Models of Computation*, pages 14–26. Springer, 2011.

[66] Daisuke Okanohara and Kunihiko Sadakane. A linear-time burrows-wheeler transform using induced sorting. In *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009, Proceedings*, pages 90–101, 2009.

[67] Y Mori. *sais version 2.4.1*, 2015. https://github.com/y-256/libdivsufsort. Accessed: 2019-09-2.

[68] Johannes Fischer and Florian Kurpicz. Lightweight distributed suffix array construction. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 27–38. SIAM, 2019.

[69] Juha Kärkkäinen, Dominik Kempa, Simon J Puglisi, and Bella Zhukova. Engineering external memory induced suffix sorting. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 98–108. SIAM, 2017.

[70] Chi-Man Liu, Ruibang Luo, and Tak-Wah Lam. Gpu-accelerated bwt construction for large collection of short reads. *arXiv preprint arXiv:1401.7457*, 2014.

[71] Juha Kärkkäinen. Fast bwt in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.

[72] T Bingmann. malloc_count—tools for runtime memory usage analysis and profiling, 2015.

[73] Mark G Sobell. *A practical guide to Ubuntu Linux*. Pearson Education, 2015.

[74] Hwancheol Jeong, Sunghoon Kim, Weonjong Lee, and Seok-Ho Myung. Performance of sse and avx instruction sets. *arXiv preprint arXiv:1211.0820*, 2012.

[75] Srinivasarao Krishnaprasad. Uses and abuses of amdahl's law. *Journal of Computing Sciences in colleges*, 17(2):288–293, 2001.

[76] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[77] Paolo Ferragina and Gonzalo Navarro. The pizza & chili corpus, 2007.

[78] Marie Lebert. *Le Projet Gutenberg (1971-2009)*. Project Gutenberg, 2010.