

Programmation Parallèle pour le recherche de motifs

Rapport
en vue de la validation de l'UE Initiation à la recherche

année Universitaire 2022-2023

étudiants :
Martin Gurtner
Timeo Jacques
Yohan Boucher

Encadrants :
Damien Jamet
Guihlem Gamard

Decharge de Responsabilités

L'Université de Lorraine n'entend donner ni approbation ni improbation aux opinions émises dans ce rapport, ces opinions doivent être considérées comme propres à leur auteur.

Remerciement

Nous tenons à remercier Damien Jamet et Guilhem Gamard qui ont contribue, par leur ecoute, conseils et accompagnement au bon deroulement de notre travail, et à la redaction de ce rapport.

Table des matières

- 1 Developpement 2
 - 1.1 Presentation du sujet 2
 - 1.2 Version basique de l’algorithme 3
 - 1.2.1 Fonctionnement de l’algorithme 3
 - 1.2.2 Execution de l’algorithme sur le debut du fichier de 70 millions de caractères 3
 - 1.2.3 Execution de l’algorithme sur un second fichier 4
 - 1.3 Somme Partielle 4
 - 1.4 Parallélisation de calculs 5
 - 1.4.1 Definition de la parallelisation 5
 - 1.4.2 Application à nos algorithmes 5
 - 1.5 Algorithmes plus avancées 5
 - 1.5.1 Arbre des suffixes 5
- 2 Conclusion 6
- 3 Bibliographie 7
- 4 Glossaire 7
- 5 Declaration en l’honneur sur le plagiat 8
- 6 Resume 11

Introduction

Actuellement des recherches sont regulierement effectuées sur la repetition de **motif**. Afin de comprendre le sujet, il faut tout d'abord comprendre ce qu'est une repetition, c'est-à-dire à partir de quel moment un **motif** est considère comme répété. Ce sujet est très present dans le domaine de la bio-informatique comme par exemple pour le sequençage de l'ADN. Dans le cadre de ce rapport, nous allons exclusivement nous interesser à des mots qui ont pour caractères uniquement : 0,1,2 et 3.

Pour une très longue chaîne chaîne de **caractères** donnee, on risque d'y trouver des répétitions de differents **motifs**, ces derniers pouvant être trouves par des algorithmes **parallélisables** et grâce à differents concepts, comme par exemple le **cube additif**, la somme partielle ou l'arbre des suffixes.

Un algorithme **parallélisable** est exécutable sur plusieurs coeurs simultanément afin de diminuer son temps d'exécution et le rendre plus efficace. Les **motifs** sont differentes combinaisons specifiques de caractères entre eux, ce sont leurs recherches qui nous interessent. Un **cube additif** est la succession de trois motifs de même tailles, qui lorsque l'on additionne les chiffres obtenus pour chaque caractères entre eux, donnent la même somme. Exemple : 301 220 112 est un cube additif car : $3+0+1 = 2+2+0 = 1+1+2 = 4$

Notre objectif de travail est le suivant : concevoir differents algorithmes, des plus basiques aux plus complexes pour trouver dans differentes chaines de tailles avoisinant les millions de caractères les motifs qui se repetent. Sachant que nous avons dans notre base de travail un mot de 70 millions de caractères dans lequel il n'y a aucune repetition, mot fourni par les encadrants. Pour faciliter et acclerer les algorithmes, nous cherchons à les paralleliser.

Nous avons commencé dans un premier temps par un algorithme des plus basiques, avant de passer à la somme partielle et les cubes additifs, somme partielle que nous avons cherche à paralléliser, puis nous sommes passés sur les tableaux de suffixes que nous avons également cherche à paralléliser.

1 Developpement

1.1 Presentation du sujet

Dans ce sujet, nous nous intéressons à la recherche de répétitions dans des motifs de longues tailles. Ces derniers sont des chaines de caractères uniquement composés de 0, de 1, de 2 ou de 3. Aucun autre caractère ne peut apparaitre des les motifs que nous analysons.

Notre recherche a pour but de determiner, si il y a des répétitions additives. Une repetition additive est le fait d'additionner tous les element d'un echantillon d'une chaine de caractère et de les comparer à un autre echantillon ayant le même nombre d'element. Par exemple, la chaine 030111 est une repetition additive, car $0 + 3 + 0 = 1 + 1 + 1$. Dans notre cas, nous allons nous focaliser sur des répétitions additives triples, ou cubes additifs. C'est-à-dire que nous regardons trois echantillons, un motif ayant uniquement des répétitions additives doubles ne sera ainsi pas considère comme étant une chaine ayant des répétitions. Prenons l'exemple precedent mais en y concatenant 212 (030111212). Ainsi, sur une taille d'échantillon de trois, le motif a une repetition additive double mais n'a pas de repetition additive triple : $0 + 3 + 0 = 1 + 1 + 1 \neq 2 + 1 + 2$.

Le motif que nous avons choisi précédemment est un cas simple, car la chaine de caractère etait de taille 9 et il est possible de trouver des motifs beaucoup plus long n'ayant pas de repetition additive triple. Actuellement, le plus long motif n'ayant aucune repetition additives triples a pour taille 70 880 000 caractères avec uniquement les caractères 0, 1, 2 et 3. Quelques regles sont à mentionner pour definir si une chaine de caractère a une repetition additive ou non. Tout d'abord, les trois échantillons d'une chaine de caractères doivent être tous de la même taille. Exemple avec le motif 31221 où $3 = 1 + 2 = 2 + 1$, ici nous ne pouvons pas considere qu'il s'agit d'une repetition additive car le premier echantillon n'a pas la même taille que les deux derniers. Ensuite, il est necessaire d'analyser toutes les tailles d'échantillons afin de tester toutes les combinaisons possibles et de savoir si un motif possede une repeton additive triple. Enfin, les echantillons que nous prenons d'un motif doivent tous être consecutifs. Dans le motif 030111, si le premier echantillon que nous prenons est 03, alors le deuxieme echantillon doit forcement correspondre au deux caractères suivant dans le motif 030111, soit 01. Puis le troisième échantillon sera 11.

Notre but sera d'implementer differents algorithmes qui prendront en entree, une chaine de caractère composee exclusivement de 0, de 1, de 2 ou de 3. Puis l'algorithme retournera le nombre de cubes additifs trouves dans la chaine. Nous avons choisi d'implementer ces algorithmes en C++. Nous avons choisi ce langage car il est véloce, et nous permet d'avoir un grand contrôle de nos actions, ce qui est important pour optimiser nos calculs. De plus, il existe des bibliotheques simples pour le calcul parallèle.

Nous détaillerons par la suite les differents algorithmes que nous avons implementes, en comparant leurs performances grâce à leurs temps d'execution en fonction de la taille du mot d'entrée. Dans un second temps, afin d'augmenter les performances nous utiliserons la bibliotheque OpenMP qui permet de faire de la programmation parellele. Nous justifierons le fonctionnement de la bibliothèque OpenMP dans les differents programmes que nous avons utilises afin d'effectuer la recherche sur les eventuelles répétitions additive triple dans un motif.

Ces recherches sont notamment prisees dans le domaine de la bio-informatique. En effet, les motifs que nous analysons ressemblent notamment au sequençage ADN. Ces derniers ont quatre composants qui ont pour identites les lettres A, C, G et T. Ces sequences peuvent être représentés par une suite de caractères que l'on nomme brin.



FIGURE 1 – sequençage de L'ADN

Si nous remplacons les lettres du sequençage ADN par les chiffres 0, 1 ,2 et 3 alors il y a une equivalence avec les motifs que nous analysons. Donc à partir de ces donnees, il est possible d'etudier le sequençage d'ADN avec les differents algorithmes que nous avons implementer. De ce fait, nous allons maintenant presenter le premier algorithme que nous avons implementé dans le cadre de la recherche.

1.2 Version basique de l’algorithme

1.2.1 Fonctionnement de l’algorithme

Ce tout premier algorithme est une version dite basique. En effet, cet algorithme teste toutes les combinaisons possibles concernant les sommes additives sur trois echantillons de même taille sur un motif. L’algorithme est compose de deux boucles imbriquées afin d’effectuer toutes les combinaisons possibles. La première boucle itère sur toutes les tailles d’échantillons. La longueur minimum d’échantillon pour la recherche de sommes additives triple est de 1, tandis que la longueur maximale correspond à la taille du motif divisée par 3. Il n’est pas possible de dépasser cette limite, les échantillons dépasseraient la taille du mot d’entrée. La seconde boucle réalise un décalage à droite de un caractère jusqu’au dernier caractère du motif teste. Ainsi, toutes les combinaisons auront été testées et l’algorithme retourne en réponse le nombre de sommes additives triples trouvées, qui sera de 0 si aucun cube additif n’a été trouvé.

Faisons une démonstration sur le motif 030111212 :

N° iteration	Taille des echantillons	ϵ_1	$\sum \epsilon_1$	ϵ_2	$\sum \epsilon_2$	ϵ_3	$\sum \epsilon_3$	Somme additive ?
1	1	0	0	3	3	0	0	Non
2	1	3	3	0	0	1	1	Non
3	1	0	0	1	1	1	1	Non
4	1	1	1	1	1	1	1	Oui
5	1	1	1	1	1	2	2	Non
6	1	1	1	2	2	1	1	Non
7	1	2	2	1	1	2	2	Non
8	2	03	3	01	1	11	2	Non
9	2	30	3	11	2	12	3	Non
10	2	01	1	11	2	21	3	Non
11	2	11	2	12	3	12	3	Non
12	3	030	3	111	3	212	5	Non

ϵ_i : échantillon i d’une chaîne de caractère
 $\sum \epsilon_i$: Somme additive pour un échantillon i

TABLE 1 – Tableau du fonctionnement de l’algorithme basique

Dans cet exemple, le motif contient une seule somme additive, à l’endroit des trois 1 consécutifs. Ce cube additif est donc trouvé dans le premier tour de la boucle externe, avec des échantillons de taille 1, et dans le 4e tour de la boucle interne. On montre ainsi qu’un motif ne peut avoir trois chiffres identiques consécutifs, sinon ce motif contiendrait une somme additive avec des échantillons ayant pour taille un seul caractère. En outre, nous remarquons que l’itération qui effectue le plus de calculs est celle dont la taille des échantillons est de un. En effet, plus les échantillons sont grands et moins il y a de calcul pour la boucle externe.

1.2.2 Execution de l’algorithme sur le debut du fichier de 70 millions de caractères

L’algorithme bien qu’étant parfaitement fonctionnel, présente néanmoins un défaut majeur, celui du temps d’exécution. En effet, nous avons testé l’algorithme avec différentes tailles de mot en entrée. Pour nos tests de vitesse, nous prenons des extraits du fichier de 70 880 000 caractères fournis par nos encadrants et ne présentant aucun cube additif. Pour ces tests, les tailles de mots variaient de 100 à 5 000 caractères et nous affichons le temps d’exécution de l’algorithme.

Comme nous pouvons le constater, l’algorithme est très limité par l’explosion de son temps de calcul. Si jusqu’à 500 caractères, le calcul se fait en moins d’une seconde, Ce temps augmente exponentiellement et sur un fichier de 1 000 caractères, le temps d’exécution dépasse déjà les 4 secondes. Sur un fichier de 2 500 caractères, le temps d’exécution dépasse la minute en atteignant les 70 secondes. Arrive à une taille de 5 000, le temps d’exécution approche des 10 minutes. Cet algorithme est donc limité lorsque la taille d’une chaîne de caractère devient importante. Si il peut s’avérer utile et efficace sur de petit fichier, il devient totalement inadapté pour le fichier de plus de 70 millions de caractères que nous voulons pouvoir tester en temps raisonnable. Un autre constat est que cet algorithme n’a trouvé aucune somme additive triple sur les 5 000 premiers caractères du fichier de 70 millions de caractères. Mais tester 5 000 caractères sur ce fichier volumineux reste très superficiel et ne prouve pas que ce dernier contient ou non une somme cubique.


```

Sur les 100 caractères du fichier, il y a 0 somme(s) additive(s) découvert.
Temps d'exécution de l'algorithme : 0.004313 secondes.

Sur les 250 caractères du fichier, il y a 0 somme(s) additive(s) découvert.
Temps d'exécution de l'algorithme : 0.074091 secondes.

Sur les 500 caractères du fichier, il y a 0 somme(s) additive(s) découvert.
Temps d'exécution de l'algorithme : 0.584719 secondes.

Sur les 1000 caractères du fichier, il y a 0 somme(s) additive(s) découvert.
Temps d'exécution de l'algorithme : 4.398402 secondes.

Sur les 2500 caractères du fichier, il y a 0 somme(s) additive(s) découvert.
Temps d'exécution de l'algorithme : 70.260864 secondes.

Sur les 4000 caractères du fichier, il y a 0 somme(s) additive(s) découvert.
Temps d'exécution de l'algorithme : 287.930237 secondes.

Sur les 5000 caractères du fichier, il y a 0 somme(s) additive(s) découvert.
Temps d'exécution de l'algorithme : 561.821106 secondes.

```

FIGURE 2 – Temps d'exécution de l'algorithme basique en fonction de la taille du motif

1.2.3 Execution de l'algorithme sur un second fichier

Nous avons executer ce même algorithme sur un autre motif qui est cense contenir plusieurs sommes additives triple.

```

Egalité cubique ! Taille des échantillons : 3
030
210
012
Egalité cubique ! Taille des échantillons : 3
210
012
003
Egalité cubique ! Taille des échantillons : 6
030210
012003
112200
Sur les 114 caractères du fichier, il y a 3 somme(s) additive(s) découvert.
Temps d'exécution de l'algorithme : 0.007486 secondes.

```

FIGURE 3 – Execution de l'algorithme sur un motif contenant des sommes additives triple

Contrairement au fichier precedent où sur les 5 000 premiers caractères, l'algorithme n'avait detecte aucun cube additif sur ce fichier de 114 caractères l'algorithme a repere trois additivite cubique. En effet, en realisant le calcul à la main, nous retrouverons le même resultat pour les trois echantillons de même taille. La premiere somme additive triple contient des echantillon de trois caractère. La somme de $0 + 3 + 0 = 2 + 1 + 0 = 0 + 1 + 2 = 3$. Donc il s'agit bien d'une somme additive triple. Le deuxieme additivite cubique contient lui aussi des echantillons de trois caractères. Le calcul est $2 + 1 + 0 = 0 + 1 + 2 = 0 + 0 + 3 = 3$. Le resultat est idantique à la premiere somme additive triple. Quant au dernier, la taille des echantillons est de six caractères. Nous obtenons bien une somme additive triple : $0 + 3 + 0 + 2 + 1 + 0 = 0 + 1 + 2 + 0 + 0 + 3 = 1 + 1 + 2 + 2 + 0 + 0 = 6$.

Remarque : Il n'est pas obligatoire que la resultat des sommes additives triple corresponde à la taille des echantillons.

Ainsi, nous montrons par l'exemple que cet algorithme est fonctionnel et est parfaitement utilisable pour des fichiers de très petite taille. Cependant, il ne sera pas utilisable pour des fichiers de taille plus grande. Pour ce faire, nous devons donc chercher un moyen pour ameliorer les performances de calcul de ce premier algorithme. Nous allons à present dans la prochaine etape modifier cet algorithme en implementant une nouvelle methode de calcul, la somme partielle.

1.3 Somme Partielle

L'un des principaux soucis de cet algorithme est que, à chaque tour de boucle, il recalcule des sommes qui ont déjà ete calcules. La solution adoptee est alors de calculer, en tout debut d'exécution la somme de tout les éléments du mot, dans une structure de donnees, puis de se servir uniquement de ces resultats par la suite.

Le principe de la somme Partielle est le suivant : On cree un tableau de longueur de la chaine de caractères plus 1, on initialise la 1ere case a 0. Et pour chaque case du tableau on ajoute la somme entre la precedente case, et le chiffre obtenu en sortant les caractères un par un de la chaine, etant donne qu'ici, la chaine de caractère se resume a des 0,1,2,3 il n'y a pas de conversion autre que passer du caractère au nombre.

Une fois le tableau complete on cherche des recurrences de resultats dans les differences entre differents couples de cases, toutes espacees par la meme distance, si l'on trouve un resultat identique pour deux couples

de cases, on considere qu'il y a donc une recurrence de motif entre ses memes cases, on trouve la meme combinaisons de caracteres.

1.4 Parallélisation de calculs

1.4.1 Definition de la parallelisation

De nos jours, les processeurs equipant les machines disposent de plusieurs coeurs physiques, des unites separees permettant d'executer les calculs. Par default, un processus s'execute sur un seul coeur, une seule unite de calcul. Depuis le debut des années 2000, avec la limitation de l'augmentation de la frequence d'horloge, pour augmenter la vitesse de traitement, le paradigme utilise est souvent la parallelisation. La parallelisation consiste à faire executer un processus sur plusieurs coeurs en simultanes afin de traiter les informations en parallèle. Afin que le gain soit optimale, les programmes doivent être developpes avec pour objectif d'être parallelises.

1.4.2 Application à nos algorithmes

Notre algorithme basique est compose, pour schematiser, de deux boucles, qui s'occupent des traitements. Pour rappel, la boucle dite externe itere sur toutes les tailles possibles de motifs, et la boucle dite interne ietre sur toutes les positions possibles pour chaque taille. La verification de cubes additifs sur un motif de taille n est independant de la verification sur un motif de taille $n + 1$, de même que la verification en debut de mot est independante de celle en fin de mot, pour une même taille de motif. Cet algorithme est donc parfaitement parallelisable, les iterations de ces boucles ne sont pas interdependantes. Pour faire cela, nous avons choisi d'utiliser une bilbiotheque nommee OpenMP. En effet, cette bilbiotheque est compatible avec une grande majorite des systemes d'exploitations, rapide à prendre en main et à utiliser, et suffisamment performante.

```
Durée :1.70529 secondes avec OpenMp
Durée :10.1783 secondes sans OpenMp
```

FIGURE 4 – Comparaison du temps d'execution avec et sans parallelisation sur un extrait de 100 000 caracteres

L'utilisation que nous avons faite de OpenMP est assez simple, elle consiste à utiliser une directive pre-processeur que l'on place avant la boucle à paralleliser. Nous avons ensuite cherche quel boucle etait la plus efficace à paralleliser. Par une serie de tests, sur des fichiers de differentes tailles, il s'avere que, en parallelisant la boucle externe, le gain de temps est plus consequent qu'avec la boucle externe.

Taille de l'échantillon	T(Boucle externe parallelisée)	T(Boucle interne parallelisée)
5 000 caracteres	0,0082643833s	0,0416457s
10 000 caracteres	0,02979825s	0,07109625s
50 000 caracteres	0,467458333s	0,5982698333s
100 000 caracteres	1,74364666s	2,450136667s
500 000 caracteres	48,016s	48,43975s
1 000 000 caracteres	207,021833s	186,562667

TABLE 2 – Tableau comparatif du temps d'execution en fonction de la parallelisation choisie

1.5 Algorithmes plus avancées

Nous allons à present implementer différents algorithmes qui nous permettrait d'améliorer les performance de calcul afin de trouver des sommes cubiques dans un motif. Nous pourrons effectuer des tests sur des sommes additives triple sur des fichiers de plus grande taille. Ainsi, le premier algorithme que nous voulons implementer est celui qui genere les arbres à suffixes.

1.5.1 Arbre des suffixes

Concernant les arbres à suffixes nous allons nous appuyer sur les travaux de recherches de Cedric Herpson sur le rapport suivant : *Apprentissage pour la Recherche d'Information textuelle et multimédia : Construction d'Arbres de Suffixes*. Ici, l'auteur va expliquer le fonctionnement de l'arbre à suffixe ainsi que dans quels domaines sont utilises les arbres à suffixe. Par exemple, C. HERPSON expliquent que ces arbres sont le fruits d'un travail des chercheurs qui sont en Science de la vie ou specialises dans la fouille de donnees.

Pour revenir à l’algorithme de l’arbre à suffixes, son but principal est notamment de représenter tous les suffixes d’un motif. Ainsi, la recherche dans une chaîne de caractères se ferait sur une partie du motif et non de son entiere permettant ainsi que gagner du temps de calcul. Dans ce rapport, l’auteur a représenté l’arbre à suffixes pour le mot Ananas.

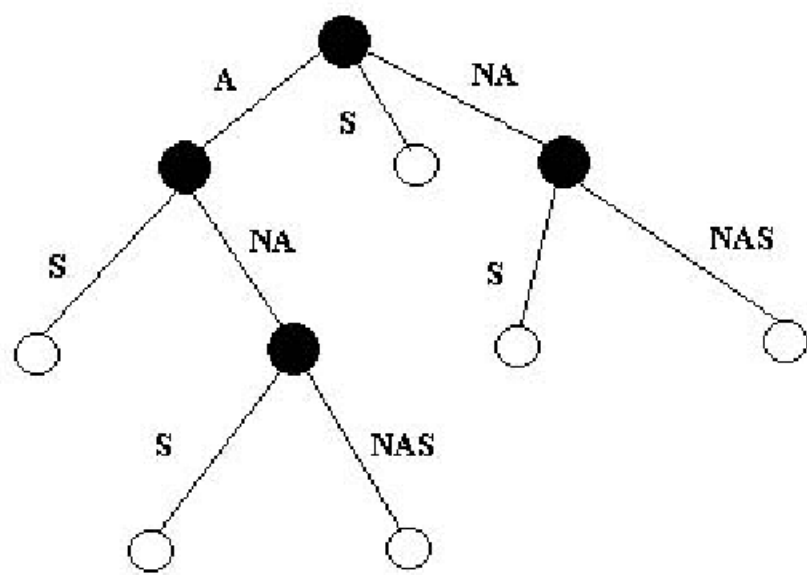


FIGURE 5 – Arbre à suffixes avec le mot Ananas

Avec le mot ananas, l’arbre à suffixe correspond à celui de l’illusatration ci-dessus. Tous les suffixes de ce mot sont représenter sur cet arbre. La lecture de ce dernier se fait depuis sa racine (point noir tout en haut de l’image) et nous descendons de noeud en noeud jusqu’à atteindre un point blanc qui représente toutes les feuilles de l’arbre. Sachant que le mot Ananas fait 6 lettres, alors nous pouvons compter sur cette arbre 6 feuilles représentant les six suffixes du mot.

Taille du suffixe	Nombre de noeud avant d’atteindre la feuille	Suffixe
1	1	S
2	2	AS
3	2	NAS
4	3	ANAS
5	2	NANAS
6	3	ANANAS

TABLE 3 – Tableau représentant tous les suffixes de l’arbre du mot ANANAS

En représentant les éléments de l’arbre à suffixe sous forme de tableau, nous constatons que les branches les plus longues de l’arbre ne menent pas systemtiquement aux suffixes les plus long. En effet, deux feuilles ont pour ascendance trois noeud. Les suffixes qui s’y trouvent sont ANAS et ANANAS qui ont respectivement 4 et 6 caractères. Or le suffixe NANAS contient cinq caractères et pourtant il a seulement deux noeuds d’ascendances.

2 Conclusion

3 Bibliographie

Références

- [1] Herpson C., *Apprentissage pour la Recherche d'Information textuelle et multimedia : Construction d'Arbres de Suffixes*, 31 mars 2008, France.
- [2] Lietard F. et Rosenfeld M., *Additive cubes are avoidable over all but one set of four numbers*, France.
- [3] Lothaire M., *Applied Combinatorics on Words*, 23 Juin 2004.

4 Glossaire

5 Déclaration en l'honneur sur le plagiat



Je soussigne

Etudiant 1 : JACQUIER Timeo

Etudiant 2 : BOUCHER Yohan

Etudiant 3 : GURTNER Martin

Régulièrement Inscrit à l'université de Lorraine

N° de carte étudiant 1 : 31908723

N° de carte étudiant 2 : 32201411

N° de carte étudiant 3 : 31709194

Année Universitaire : 2022/2023

Niveau d'études : Master

Parcours : Master 1 Informatique

N° UE : 811

Certifions qu'il s'agit d'un travail original et que toutes les sources utilisées ont été indiquées dans leur totalité. Nous certifions de surcroît que nous n'avons ni recopié ni utilisé des idées ou des formulations tirées d'un ouvrage, article ou mémoire, en version imprimée ou électronique, sans mentionner précisément leur origine et que les citations intégrales sont signalées entre guillemets

Conformément à la loi, le non respect de ses dispositions nous rends passible de poursuites devant la commission disciplinaire et les tribunaux de la République Française.

Fait à , le.....

Signature des étudiants :

Table des figures

1	sequençage de L'ADN	2
2	Temps d'execution de l'algorithme basique en fonction de la taille du motif	4
3	Execution de l'algorithme sur un motif contenant des sommes additives triple	4
4	Comparaison du temps d'execution avec et sans parallelisation sur un extrait de 100 000 caractères	5
5	Arbre à suffixes avec le mot Ananas	6

Liste des tableaux

1	Tableau du fonctionnement de l'algorithme basique	3
2	Tableau comparatif du temps d'exécution en fonction de la parallélisation choisie	5
3	Tableau representant tous les suffixes de l'arbre du mot ANANAS	6

6 Resume

Francais

Dans le cadre de l'étude de motifs dans de très longues chaines de caractères comme les séquences ADN on veut comptabiliser les répétitions dans les motifs de ces longues chaines. Pour pouvoir trouver et etudier ces differents motifs de facon efficace et le plus rapidement possible, on cherche a concevoir differents algorithmes que l'on cherche a paralléliser pour optimiser leur vitesse d'exécution malgré les longueurs des différentes chaines de caractères étudiées.

Dans ce rapport est expliqué différents algorithmes de detection de motifs des plus simples aux plus complexes, les resultats obtenus, les temps d'executions, les tailles des chaines étudiés.

English

In the context of the patterns' studies in very long strings like DNA sequences, we want to count the repetitions in this long strings' patterns. In order to find and study theses different patterns effectively and faster, we looking for conceive different algorithms that we want to parallelize for optimize their execution speed despite of the lenghts of the strings studied.

In this account is explained some different pattern detection's algorithms from the simpler to the more complex, the result obtained, the execution's times, the size of the strings studied.