Parallélisation de la recherche de patterns dans une séquence

Raphael Montaud & Hugo Dobbelaere

***Résumé*—Le but de ce projet est de proposer des méthodes de parallélisation dans le cadre de la recherche de patterns dans une séquence. Nous proposons ainsi plusieurs implémentations en MPI (calcul à mémoire distribuée), OpenMP (calcul à mémoire partagée) et en CUDA (calcul vectoriel).**

I. Introduction

Le but de ce projet est de retrouver un pattern (une petite séquence) dans une séquence de taille plus conséquente et de compter le nombre d'occurrences de ce pattern dans la séquence. Cela s'applique notamment au domaine de l'ADN. Plutôt que de chercher le pattern exact dans la séquence ADN s, nous allons autoriser une marge d'erreur, c'est à dire que nous allons chercher tout mot t tel que la distance d'édition entre notre pattern p et t est inférieure à un seuil r que nous nous posons à l'avance. La distance d'édition entre deux mots correspond au nombre minimal d'opérations pour transformer le mot t en p, les opérations pouvant être des additions, délétions ou substitutions. Cette distance est appelée distance de Levenshtein. Cette distance peut se calculer en temps quadratique de la longueur des mots à l'aide d'un algorithme de calcul dynamique. Il a de plus été prouvé que, si on admet que P !=NP, le temps quadratique est le meilleur temps que l'on peut espérer. Si on note m la taille de la séquence ADN s et n la taille du pattern que nous cherchons, la complexité totale s'écrit donc $c = m * n^2$.

La figure 1 montre l'évolution du temps de calcul en secondes en fonction de la taille de la séquence considérée (en nombre de KB). On peut voir que cette évolution est linéaire comme dans la formule théorique. Ensuite la figure 2 montre l'évolution du temps de calcul en fonction de la taille du pattern au carré. Cela confirme encore la relation linéaire entre les deux grandeurs.

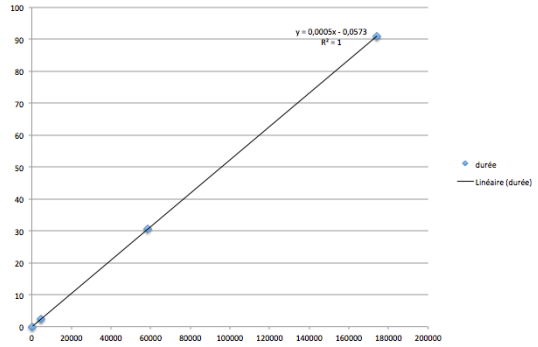

FIGURE 1. Temps de calcul vs taille du fichier en KB

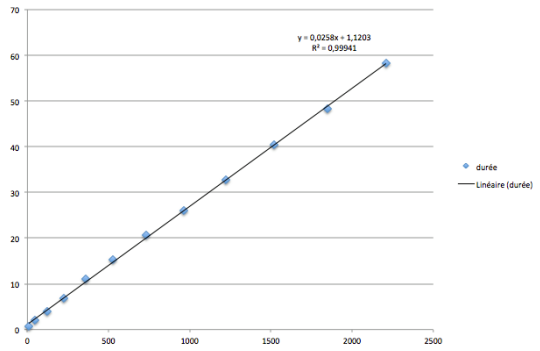

FIGURE 2. Temps de calcul vs taille du pattern au carré

Dans ce problème, deux types de parallélisme sont envisageables :

- le parallélisme sur les patterns. Si l'on souhaite retrouver plusieurs patterns dans notre séquence, alors on peut séparer notre problème sous la forme 1 pattern = 1 tâche.
- le parallélisme sur la séquence. Si notre pattern p est de taille n, nous allons calculer la distance de Levenhstein $L(p,t)$ pour tout t où t appartient à une partition de la séquence s. Nous pouvons donc diviser notre problème sous la forme 1 tâche = le calcul de $L(p,t)$ pour un mot t.

II. Méthodes de parallélisation

A. Premier type de parallélisme : les patterns

Ce type de parallélsime ne pose pas de problèmes particuliers. Nous proposons une implémentation en MPI de la parallélisation. Il suffit de répartir équitablement les patterns à retrouver entre les rangs MPI puis de les calculer et de récupérer de communiquer les résultats. Si on considère k patterns $(p_1, ..., p_n)$ de tailles $(n_1, ..., n_n)$ alors la complexité totale du problème est $c = m * \sum_i n_i^2$. Dans le meilleur des cas, si la répartition des patterns est parfaitement équitable entre l rangs MPI, le speed-up est égal à l. Cependant si la répartition des tailles n'est pas idéale, par exemple dans le cas de trois patterns de tailles 5, 10 et 25 avec 3 rangs MPI, l'accélération ne sera que de 1,2 au lieu de 3. Il faut donc prendre bien soin de répartir équitablement les tâches, donc trouver un recouvrement $(I_1, ..., I_l)$ de $(1, ..., k)$ qui minimise $\max_j \sum_{i \in I_j} n_i^2$. Une solution gloutonne à ce problème consiste à trier les patterns par ordre décroissant et à chaque itération, le rang MPI avec le moins de complexité accumulée ($\sum_{i \in I_j} n_i^2$) se voit confier le plus grand pattern restant. La figure

3 montre l'évolution de l'accélération quand le nombre de patterns augmente. A chaque itération nous avons ajouté un pattern de taille aléatoire entre 2 et 7. On peut voir que quand on augmente le nombre de patterns, on parvient à mieux équilibrer mais même dans le cas de plus de 20 patterns, la répartition peut être mauvaise et l'accélération de l'ordre de la moitié de sa valeur idéale. Pour des raisons de temps de calcul, nous n'avons pas pu continuer les expériences mais l'accélération devrait tendre vers 4 lorsque le nombre de patterns tend vers l'infini.

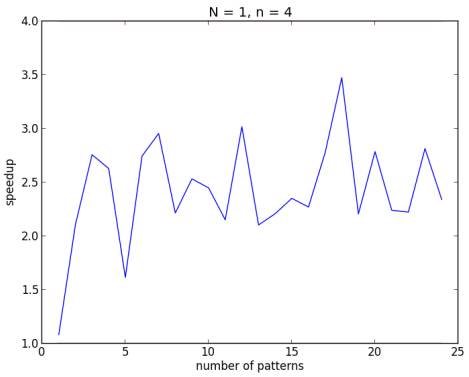

FIGURE 3. Speed-up vs nombre de patterns

Nous voyons donc que cette méthode de parallélisation sur les patterns est simple à implémenter, cependant, elle reste limitée puisque dans le cas d'un faible nombre de patterns, la répartition des tâches ne sera pas idéale. De plus l'exemple que nous présentons est dans le cas de seulement 4 rangs MPI mais plus le nombre de rangs MPI augmente et plus il est difficile de bien répartir les tâches.

B. Deuxième type de parallélisme : calcul de la distance de Levenshtein

Dans les applications de ce problème aux séquences ADN, les patterns recherchés sont généralement de tailles raisonnables, tandis que les séquences ADN sont immenses (des centaines de milliers de caractères). Il y a donc environ m opérations qui peuvent être effectuées en parallèle. Ce type de parallélisme offre donc plus de souplesse dans la répartition des tâches et a l'avantage d'avoir un speed-up théorique idéal.

C. Implémentation MPI

Dans un premier temps, nous proposons une implémentation MPI de ce type de parallésition. Chaque rang MPI va se voir attribuer une quantité q de distances de Levenshtein à calculer. Pour ce travail, nous séparons la séquence ADN en plusieurs morceaux équitables, autant que de rangs MPI. Les morceaux de séquences doivent "déborder" en partie. En effet, pour qu'un rang MPI puisse faire ses q calculs de distance de Levenshtein, il aura besoin d'un morceau de taille $q + n_i - 1$ où nous rappelons que n_i est la taille du pattern p_i. Il faut donc que chaque rang MPI puisse accéder à un morceau de taille $q + \underset{i}{max}(n_i) - 1$. Le rang 0 lit donc le fichier et envoie aux autres rangs les morceaux qui leurs sont attribués en utilisant la fonction Scatter. En effet, la fonction Scatter permet de ne communiquer que des parties du fichier, ce qui réduit le coût en communications MPI par rapport à un Broadcast du fichier complet. Chaque rang fait son calcul de distances de Levenshtein et calcule le nombre de correspondances pour chaque pattern. Les résultats sont ensuite aggrégés par le rang 0 qui reçoit les résultats de chacun. Nous testons notre implémentation et comparons les résultats pour plusieurs patterns et séquences par rapport à ceux de l'implémentation basique, ce qui nous permet de confirmer que notre implémentation est correcte. La figure 4 présente les résultats d'accélération pour différentes tailles de fichiers. On observe que l'accélération pratique est dans ce cas inférieure à l'accélération théorique (qui devrait être égale au nombre de rangs MPI). Cela peut venir du surcoût imposé par la communication de la séquence ou à l'attente du rang 0 par rapport aux autres rangs.

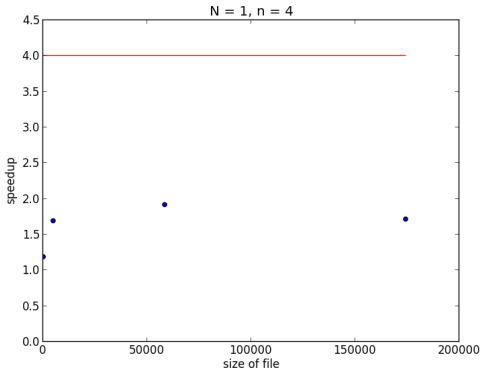

FIGURE 4. Speed-up vs taille du fichier en KB

D. Implémentation MPI et openMP

Nous proposons maintenant de greffer un mutliprocessing à mémoire partagée sur l'implémentation présentée en II-C. Etant donné que chaque rang MPI dispose en mémoire d'un morceau de séquence et qu'il doit faire q calculs de distances de Leveinshtein, nous pouvons paralléliser ces calculs en mémoire partagée. Pour cela il suffit de répartir les tâches de la boucle entre les process openMP à l'aide de la fonction schedule. Il faut cependant faire attention à l'opération d'incrémentation du nombre de "match" qui doit être atomique. Cependant, comme ce verrou ralentit l'exécution, nous utilisons la fonction "reduction" qui s'avère être plus rapide car non bloquante. A nouveau, nous présentons dans la figure 5 l'accélération pour différentes tailles de fichiers. Etant donné que ces calculs ont été effectués pour 4 rang MPI avec 4 coeurs chacuns, l'accélération théorique maximale est de 16. À nouveau l'accélération pratique est à un facteur 2 de l'accélération théorique, les mêmes raisons que précédemment pouvant s'appliquer.

E. Cuda

Afin de profiter des GPU dont nous disposons, il est nécessaire de proposer une implémentation de parallélisation en

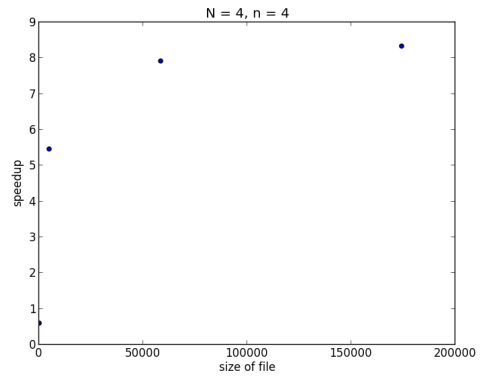

FIGURE 5. Speed-up vs taille du fichier en KB

CUDA. Il faut donc reformater le code et changer la signature de la fonction de calcul de distance de Levenshtein, celle-ci prend maintenant en argument un morceau de la séquence de taille $1024 + n$, elle traite un calcul correspondant au rang du processeur qui le gère et stocke la distance de Levenshtein calculée dans un tableau de résultats qui est passé en argument. Les résultats d'accélérations pour différentes tailles de fichiers sont présentés dans la figure 6.

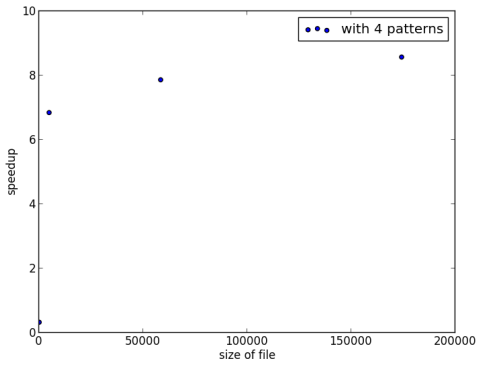

FIGURE 6. Speed-up vs taille du fichier en KB

F. Cuda et MPI

Afin de faire les calculs sur plusieurs GPU en parallèle, il faut proposer une implémentation hybride CUDA et MPI. Comme précedemment, chaque rang MPI reçoit un morceau de la séquence à l'aide de la fonction Scatter. Puis chaque rang MPI utilise son GPU pour paralléliser les calculs sur son morceau de séquence. Dans la figure 7 nous présentons les résultats d'accélération pour différentes tailles de fichier, dans le cas où nous utilisons 4 rangs MPI, chacun disposant d'un GPU. Ainsi, de toutes les implémentations essayées, c'est bien celle-ci qui donne les meilleurs résultats. En effet, le calcul sur GPU se prêtant très bien au calcul de la distance de Levenhstein, le même calcul étant exécuté un grand nombre de fois.

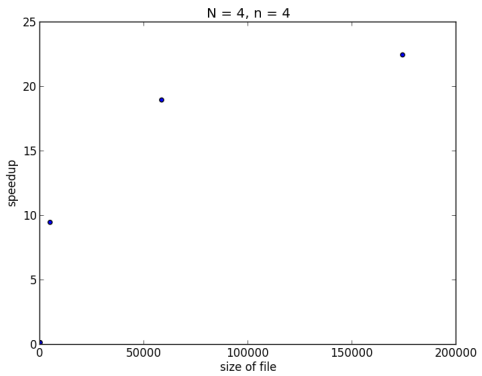

FIGURE 7. Speed-up vs taille du fichier en KB

TABLE I
COMPARAISON DES DIFFÉRENTES IMPLÉMENTATIONS

implémentation	**Temps de calcul en secondes**	**Accélération**
Normale	1767	1
MPI+OpenMP	17	104
MPI+CUDA	8	221

III. CONCLUSION PARTIELLE ET AMÉLIORATIONS POSSIBLES

Enfin nous pouvons conclure sur les méthodes à préférer selon les propriétés du cluster mis à disposition. Premièrement, il est généralement préférable de paralléliser sur la séquence plutôt que sur les patterns, à moins que la taille de la séquence soit faible et le nombre de patterns très grand. En effet, même si le nombre de patterns est très grand, il est moins coûteux en termes de communication MPI de transmettre des morceaux de la séquence plutôt que la séquence entière. Ensuite, il est très avantageux d'utiliser les GPUs, il faut donc tous les exploiter. Enfin, une piste d'amélioration pour exploiter le plein potentiel d'un cluster est d'évaluer les temps de calcul pour des processeurs et celui pour un GPU afin d'équilibrer au mieux les tâches dans le cadre d'une implémentation hybride MPI-OpenMP-CUDA et donc réduire encore le temps de calcul.

Pour conclure, voici un tableau présentant les temps de calculs selon les méthodes. Nous nous autorisons un cluster de 30 noeuds, chacun ayant 8 coeurs et un GPU. L'objectif est de calculer le plus vite possible, à l'aide de ce cluster, le nombre de matches sur une séquence ADN de 250Mo pour une dizaine de patterns, de tailles variant entre 10 et 40.

On peut donc constater que la meilleure implémentation est celle exploitant Cuda. A titre indicatif, l'accélération maximale théorique pour une implémentation hybride MPI+OpenMP sur ce cluster est de 240. Il y a donc un certain nombre de pertes, probablement causés par des problèmes de verrous et de communication MPI. Ce tableau nous indique de plus qu'il serait intéressant de coder une implémentation hybride MPI+OpenMP+Cuda, comme décrit plus haut. En effet, étant donné que 8 processeurs vont deux fois moins vite qu'un GPU, on peut se dire qu'en répartissant les tâches entre les deux, on pourrait réduire le temps de calcul de $\frac{2}{3}$ par rapport au temps de calcul MPI+Cuda. Cela donnerait donc 5 secondes de calcul

ce qui est une amélioration non négligeable.

RÉFÉRENCES

[1] P. Carribault INF 560 : Calcul Parallèle et Distribué *Ecole Polytechnique*
[2] Documentation Open MPI https://www.open-mpi.org/doc/
[3] Page wikipedia, distance de Leveinshtein https://en.wikipedia.org/wiki/Levenshtein_distance