Un nouvel algorithme pour l'alignement de longues séquences génomiques

Stage effectué au sein de l'équipe GenScale sous la direction de Dominique Lavenier

IGOR MARTAYAN, École Normale Supérieure de Rennes, France

Additional Key Words and Phrases : pairwise sequence alignment, consensus sequence, dynamic programming, vectorization, processing-in-memory

1 INTRODUCTION

Contrairement à ce que l'on pourrait croire, l'informatique joue un rôle central dans l'analyse de séquences génomiques telles que l'ADN ou l'ARN. En effet, il n'est pas rare de devoir analyser des génomes comptant plusieurs millions, voire plusieurs milliards de bases[1], et pour traiter une telle quantité de données il est indispensable de concevoir des algorithmes efficaces.

Du fait de la taille conséquente des génomes étudiés, les technologies de séquençage divisent généralement les génomes en plusieurs séquences de taille plus petite que l'on appelle des lectures. Ces lectures sont représentées par des mots dans l'alphabet des nucléotides (A, T, C, G) avec une taille pouvant varier de quelques centaines à plusieurs milliers de bases. Se pose alors le problème de l'assemblage de génomes, incontournable en bioinformatique : comment reconstituer le génome d'origine de façon aussi fidèle que possible à partir de cet ensemble de lectures ? Et surtout, comment le faire en un temps raisonnable ?

Qui plus est, les nouvelles technologies de séquençage, dites de troisième génération, apportent elles aussi leur lot de contraintes : si elles ont l'avantage de produire des lectures plus longues que les technologies précédentes, ce qui est bien pratique pour reconstruire un génome, elles ont aussi la particularité d'avoir des taux d'erreur nettement plus élevés. Ces erreurs, qui apparaissent lors de la lecture du génome, sont classifiées en trois catégories :

- les insertions qui ajoutent des bases non présentes dans la séquence d'origine,
- les délétions qui effacent des bases initialement présentes, et
- les substitutions qui remplacent certaines bases par d'autres.

Il est donc primordial d'ajouter une phase de correction d'erreurs lors du séquençage d'un génome. La plupart du temps, on souhaite que la correction des lectures ait lieu le plus tôt possible. Pour ce faire, on produit plusieurs exemplaires de chaque lecture en répétant le séquençage du génome. On dispose alors de plusieurs séquences issues d'une même région du génome ayant chacune leur lot d'erreurs et à partir desquelles on cherche à construire une séquence "consensus" avec un taux d'erreur aussi faible que possible. C'est précisément ce problème du consensus qui nous intéresse dans le cadre de ce stage.

Même si nous ne nous y intéresserons pas ici, il est également bon de noter que de nouvelles approches proposent d'effectuer cette phase de correction *après* l'assemblage du génome. Parmi les outils adoptant cette approche, on peut notamment mentionner Racon [Vaser et al. 2017].

L'objectif initial de ce stage était de proposer un algorithme parallélisé de consensus de séquences génomiques adapté à l'architecture *processing-in-memory* développée par la société UpMem[2], que l'on présente plus en détails dans la section suivante, dans le cadre d'un projet plus large de parallélisation d'algorithmes de bioinformatique sur cette architecture. Cet objectif a évolué au fil de l'avancement du stage et mon travail a finalement consisté à concevoir un nouvel algorithme

[1] Les bases nucléiques (adénine, thymine, cytosine et guanine) sont les briques élémentaires de l'ADN, elles sont assemblées par paires pour former des chaînes de nucléotides.

[2] https://www.upmem.com/technology/

pour l'alignement de longues séquences génomiques similaires (issues de la lecture d'une même région du génome) afin d'identifier les erreurs de séquençage. Bien que cet algorithme ne soit pas spécifique à l'architecture *processing-in-memory*, il a tout de même été pensé pour répondre aux contraintes de cette architecture. J'ai ensuite été amené à écrire en C une implémentation de cet algorithme axée sur les performances et à comparer le temps d'exécution et la qualité des résultats obtenus par cette méthode avec l'état de l'art.

2 CONTEXTE

2.1 Alignement entre deux séquences

Historiquement, l'une des premières méthodes mise au point pour identifier les erreurs de séquençage est celle de l'alignement de séquences. Aligner une paire de séquences consiste à mettre en correspondance les deux séquences en indiquant les zones où elles coïncident et les zones où elles diffèrent avec les trois types d'erreurs vus précédemment (insertion, délétion, substitution).

```
ATCGG_GCAATTAAAAGGATCTGAAGCGA_AGA_CACCGTACCA_GACGTAGCGAGCCCTATTT
|||||_||||_||||||||||_|||||||_|||_|||||||_||_|||||||_||||||||||
ATCGGAGCAA_TAAAAGGATCCGAAGCGAGAGACCACCGTA_CAGGACGTAGGGAGCCCTATTT
```

Fig. 1. Exemple d'alignement entre deux séquences. Les barres verticales identifient les paires de bases identiques. Les tirets dans la première séquence indiquent une insertion tandis que ceux dans la deuxième séquence indiquent une délétion.

Pour une même paire de séquence, il existe évidemment plusieurs alignements possibles mais on souhaite généralement obtenir un alignement optimal, c'est-à-dire un alignement présentant un nombre d'erreurs minimal.

L'approche la plus courante pour aligner deux séquences consiste à calculer un score d'alignement que l'on souhaite maximiser (ou de façon duale une distance d'édition que l'on souhaite minimiser). Le score d'alignement dépend généralement de trois paramètres : une récompense α lorsque deux bases coïncident, un coût de substitution β et un coût d'insertion / délétion δ.

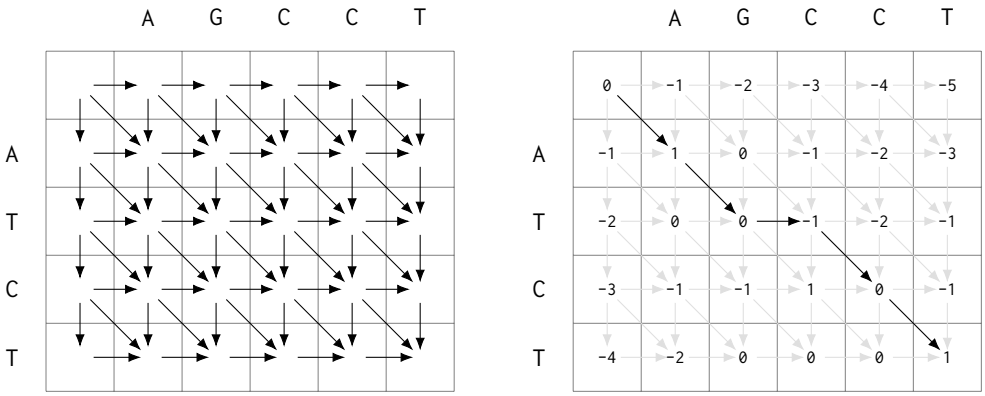
Pour trouver un alignement optimal entre deux séquences $u = u_1 \dots u_n$ et $v = v_1 \dots v_m$, on peut calculer la matrice des scores S où $S_{i,j}$ désigne le score maximal d'alignement pour les séquences $u_1 \dots u_i$ et $v_1 \dots v_j$. On peut alors établir la relation de récurrence suivante :

$$S_{i,j} = \max \left\{ S_{i-1,j} - \delta, S_{i,j-1} - \delta, S_{i-1,j-1} + m_{i,j} \right\}$$

avec

$$m_{i,j} = \begin{cases} \alpha \text{ si } u_i = v_j \\ -\beta \text{ sinon} \end{cases}$$

L'algorithme de Needleman—Wunsch [Needleman and Wunsch 1970] introduit une méthode pour calculer la matrice des scores basée sur la programmation dynamique : la matrice est calculée ligne par ligne, de gauche à droite, en utilisant la relation de récurrence établie ci-dessus (la première case de la matrice est initialisée à 0). De plus, en mémorisant pour chaque case le prédécesseur qui maximise son score, on peut reconstruire un chemin correspondant au score maximal et en déduire l'alignement associé. Cette méthode permet donc d'obtenir un alignement optimal avec une complexité spatiale et temporelle en $O(nm)$.



(a) Dépendances entre les $S_{i,j}$. (b) Chemin correspondant à un score maximal.

Fig. 2. Matrice des scores associée aux séquences AGCCT et ATCT en prenant pour paramètres $\alpha = \beta = \delta = 1$. Le chemin illustré dans la deuxième figure correspond à l'alignement AGCCT / AT_CT.

Plusieurs variantes de cet algorithme sont apparues par la suite : on peut citer entre autres l'algorithme de Smith—Waterman [Smith et al. 1981] qui a pour particularité de produire un alignement local (en alignant uniquement les régions des deux séquences qui coïncident le mieux) et l'algorithme de Gotoh [Gotoh 1982] qui permet d'introduire des fonctions de coût affines selon le nombre d'insertions ou de délétions successives.

2.2 Alignement multiple

L'alignement de deux séquences présenté dans la section précédente peut se généraliser à p séquences. On parle alors d'alignement de séquences multiples, souvent abrégé en MSA (Multiple Sequence Alignment). De la même façon que l'on peut calculer une matrice des scores pour deux séquences, on peut calculer un tenseur à p dimensions pour p séquences. Le problème d'une telle méthode est que la complexité explose : dans le cas de p séquences de taille n, la complexité spatiale et temporelle est alors en $O(n^p)$, ce qui est beaucoup trop compte tenu de la taille des données que l'on doit traiter.

Dans le cas général, il s'avère que l'alignement optimal de séquences multiples est un problème NP-complet [Wang and Jiang 1994]. Cependant, plusieurs algorithmes permettent d'obtenir de bonnes approximations en temps polynomial.

Une première approche, proposée par Carrillo et Lipman [Carrillo and Lipman 1988] puis améliorée par Altschul [Lipman et al. 1989], se ramène au cas des alignements 2 à 2 entre chaque paire de séquences pour restreindre l'espace de recherche et en déduire un alignement global.

Une autre approche, dite d'alignement progressif [Feng and Doolittle 1987], consiste à construire un arbre binaire (le *guide tree*) dans lequel les séquences les plus proches sont celles qui présentent le plus de similarité. Un alignement global est ensuite calculé en fusionnant les séquences voisines dans l'arbre. Parmi les outils utilisant l'alignement progressif, on peut notamment citer MUSCLE [Edgar 2004].

Toutes ces méthodes d'alignement multiple ont en commun de se ramener au cas des alignements par paire pour calculer un alignement global. Améliorer l'alignement par paire permet donc indirectement d'améliorer l'alignement multiple.

2.3 État de l'art

2.3.1 Alignement par paire. De très nombreux outils existent pour aligner des paires de séquences génomiques, la plupart de ces outils sont basés sur une approche par programmation dynamique telle que nous l'avons présentée plus haut. Parmi ceux-là, l'un des outils les plus performants et les plus largement utilisés aujourd'hui s'appelle `minimap2` [Li 2018]. L'algorithme d'alignement utilisé par `minimap2` reprend l'approche proposée par Gotoh [Gotoh 1982] pour l'alignement avec des fonctions de coût affines et apporte de nombreuses optimisations pour l'accélérer. L'une de ces optimisations repose sur une formulation différente de la relation de récurrence pour le calcul des scores, introduite dans un article de Suzuki et Kasahara [Suzuki and Kasahara 2018].

`edlib` [Šošić and Šikić 2017] est également un outil particulièrement performant pour l'alignement de séquences. L'une des particularités d'`edlib` est qu'il utilise comme score la distance d'édition de Levenshtein [Levenshtein et al. 1966]. Bien que les alignements produits avec ce score soient moins précis que ceux obtenus avec des scores paramétrés, le fait d'utiliser la distance de Levenshtein permet d'avoir recours à des optimisations très efficaces telles que l'utilisation de vecteurs de bits, présentée dans un article de Myers [Myers 1999].

2.3.2 Graphe de de Bruijn. Une autre méthode de plus en plus répandue pour reconstruire des séquences génomiques consiste à utiliser des graphes de de Bruijn. Pour rappel, le graphe de de Bruijn d'ordre k pour un alphabet Σ est un graphe orienté dans lequel chaque sommet correspond à un mot de taille k et chaque arête (u, v) indique que le plus grand suffixe propre de u coïncide avec le plus grand préfixe propre de v. Autrement dit, ce graphe peut être défini par :

$$V = \Sigma^k$$

$$E = \{(u, v); u_2 \dots u_k = v_1 \dots v_{k-1}\}$$

Dans le cadre de la génomique, on travaille souvent avec des sous-graphes du graphe de de Bruijn pour les k-mers, c'est-à-dire les mots de taille k sur l'alphabet $\{A, T, C, G\}$. À partir d'une séquence donnée, on peut calculer l'ensemble des k-mers contenus dans la séquence et construire le graphe de de Bruijn induit par cet ensemble de k-mers. L'intérêt d'une telle méthode est que l'on peut ensuite reconstruire la séquence en cherchant un chemin eulérien dans ce graphe [Medvedev and Pop 2021]. Certains algorithmes récents tels que CCSA [Lavenier 2021] appliquent cette méthode au problème du consensus.

2.3.3 Processing-in-memory. En complément de ces approches algorithmiques, plusieurs articles présentent des approches basées sur l'accélération matérielle pour améliorer l'alignement de séquences. Ces approches reposent généralement sur l'utilisation de cartes graphiques [Klus et al. 2012 ; Luo et al. 2013] en utilisant des librairies telles que CUDA[3], ou sur l'utilisation de FPGA (Field-Programmable Gate Array) [Benkrid et al. 2009]. L'une des limitations de ces méthodes de calcul est liée à la vitesse de transfert des données entre les différents composants : puisque l'on veut généralement aligner un grand nombre de séquences en parallèle, la quantité de données à transférer est souvent un goulot d'étranglement.

De nouvelles architectures, dites de *processing-in-memory* (PIM), ont été développées pour permettre d'effectuer des calculs directement au sein de la mémoire [Mutlu et al. 2019]. L'architecture PIM proposée par UpMem consiste à ajouter de petits processeurs directement dans la DRAM afin de minimiser le temps d'accès aux données. Une barrette de 16 Go de DRAM compte ainsi jusqu'à 256 processeurs (appelés DPU pour DRAM Processing Unit). L'intérêt d'une telle architecture est double : cela permet d'accélérer les calculs gourmands en données tout en réduisant les ressources nécessaires. Plusieurs algorithmes de bioinformatique ont déjà été portés sur l'architecture PIM,

[3]CUDA est une librairie développée par Nvidia permettant d'effectuer des calculs performants sur carte graphique.

c'est notamment le cas de BLAST [Lavenier et al. 2016a], ainsi que des algorithmes de mapping [Lavenier et al. 2016b] et de recherche de variants [Lavenier et al. 2020].

3 CONTRIBUTION

Nous proposons un nouvel algorithme pour l'alignement de longues séquences génomiques dans le cadre du problème du consensus. Cet algorithme, que nous avons baptisé `coal` pour *consensus alignment*, est accompagné d'une implémentation écrite en C, accessible en ligne à l'adresse suivante : https://github.com/imartayan/coal.

3.1 Alignement fragmenté de séquences

L'idée de ce nouvel algorithme est venue du constat suivant : dans le cadre du problème du consensus, les séquences à aligner présentent de fortes similarités et ne nécessitent donc pas autant de comparaisons qu'un alignement quelconque. En particulier, une fois que l'on a identifié deux régions des séquences qui se recouvrent, il est très probable que ces deux régions soient de tailles comparables et divergent assez peu l'une de l'autre. On peut alors tirer parti de ces propriétés pour accélérer l'alignement des deux régions.

Le principe général de notre algorithme peut être résumé comme suit :

- On parcourt simultanément les deux séquences jusqu'à trouver un k-mer qui soit commun aux deux séquences. La position de ce k-mer dans les deux séquences est marquée comme un point d'ancrage.
- On continue de parcourir les séquences jusqu'à trouver un nouveau k-mer commun aux deux séquences, qui est marqué comme second point d'ancrage. On aligne alors les séquences entre les deux points d'ancrage.
- On continue ensuite la lecture des séquences jusqu'à trouver un troisième point d'ancrage pour refaire un alignement, et ainsi de suite jusqu'à ce que l'on arrive à la fin des séquences.

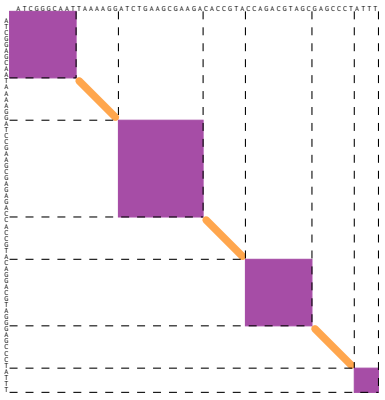


Fig. 3. Exemple de fragmentation de la matrice des scores. Les diagonales oranges correspondent aux k-mers communs aux deux séquences qui servent de points d'ancrage. Les sous-matrices violettes correspondent aux sous-séquences que l'ont doit aligner.

Algorithme 1 : Alignement fragmenté de deux séquences

```
1  fonction alignement fragmenté(s_1, s_2)
2      i_0 ← 0 // dernier point d'ancrage dans s_1
3      j_0 ← 0 // dernier point d'ancrage dans s_2
4      i ← k // indice dans s_1
5      j ← k // indice dans s_2
6      K_1 ← ∅ // dictionnaire des k-mers de s_1
7      K_2 ← ∅ // dictionnaire des k-mers de s_2
8      tant que i < |s_1| et j < |s_2| faire
9          k_1 ← s_1[i − k : i]
10         k_2 ← s_2[j − k : j]
11         ajouter (k_1, i) à K_1
12         ajouter (k_2, j) à K_2
13         si K_1 contient k_2 alors
14             i_1 ← indice associé à k_2 dans K_1
15             aligner s_1[i_0 : i_1 − k] et s_2[j_0 : j − k]
16             i_0 ← i_1
17             j_0 ← j
18             i ← i + 1
19             j ← j + G
20             K_2 ← ∅
21         sinon si K_2 contient k_1 alors
22             j_1 ← indice associé à k_1 dans K_2
23             aligner s_1[i_0 : i − k] et s_2[j_0 : j_1 − k]
24             i_0 ← i
25             j_0 ← j_1
26             i ← i + G
27             j ← j + 1
28             K_1 ← ∅
29         sinon
30             i ← i + 1
31             j ← j + 1
32         fin
33     fin
```

La constante G (pour *gap size*) qui apparaît dans l'algorithme (lignes 19 et 26) joue un rôle important : elle s'assure que les k-mers utilisés comme points d'ancrage sont espacés d'au moins G caractères. Cela permet d'éviter d'avoir à effectuer beaucoup d'alignements très rapprochés.

Cette approche présente plusieurs avantages :

- Tout d'abord, cet algorithme fait abstraction de la méthode d'alignement utilisée entre deux points d'ancrage. On peut donc combiner cet algorithme avec des méthodes d'alignement performantes qui ont déjà été développées.
- Ensuite, le fractionnement des séquences réduit considérablement la taille des problèmes à traiter. Comme la complexité des algorithmes d'alignement est souvent quadratique (ou

en tout cas plus que linéaire) en la taille des séquences, cette approche réduit donc à la fois le temps d'exécution et l'espace mémoire utilisé.
- Enfin, comme l'alignement de petites séquences est assez peu gourmand en ressources, il est tout à fait envisageable de paralléliser les alignements des sous-séquences.

Toutefois, cette approche perd tout intérêt si l'on ne parvient pas à identifier des k-mers communs aux deux séquences. Il est donc important d'ajuster la taille des k-mers en fonction du taux d'erreur des séquences utilisées.

3.2 Optimisation de l'alignement des sous-séquences

Nous avons présenté dans la section précédente un algorithme pour l'alignement fragmenté de deux séquences. Nous détaillons ici l'algorithme utilisé pour aligner les sous-séquences entre les points d'ancrage. Cet algorithme est indépendant de l'algorithme d'alignement fragmenté et peut donc être utilisé pour aligner des séquences quelconques, mais il a avant tout été conçu pour fonctionner sur de petites séquences présentant de fortes similarités.

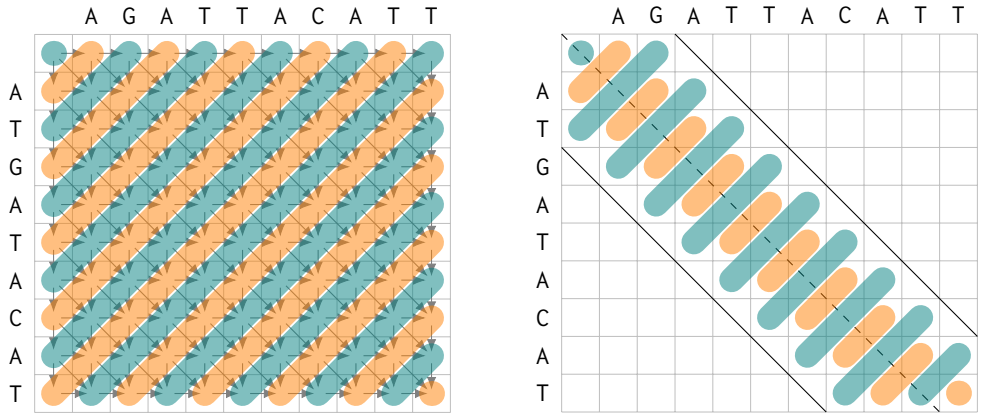
Notre méthode d'alignement reprend le fonctionnement de l'algorithme de Needleman—Wunsch [Needleman and Wunsch 1970], c'est-à-dire le calcul d'une matrice des scores par programmation dynamique à partir d'une relation de récurrence, auquel nous avons apporté plusieurs optimisations pour l'accélérer.

3.2.1 Restriction autour de la diagonale principale. La première optimisation importante que l'on peut apporter à l'algorithme de Needleman—Wunsch consiste à restreindre les zones que l'on calcule autour de la diagonale principale de la matrice. Cette idée a été introduite dans un article d'Esko Ukkonnen [Ukkonen 1985] avant d'être réutilisée dans de nombreux outils d'alignement. Dans notre cas, elle est très pertinente puisque les séquences que l'on souhaite aligner sont très similaires et il est donc très probable que le chemin correspondant à l'alignement optimal soit proche de la diagonale principale. En pratique, on fixe une largeur de bande w et on calcule uniquement les cases dans un rayon w autour de la diagonale principale.

3.2.2 Vectorisation des calculs sur les antidiagonales. La seconde optimisation porte sur l'ordre dans lequel on calcule les cases de la matrice des scores. Lorsque l'on observe les dépendances entre les différentes cases de la matrice, on peut constater que les cases qui partagent la même antidiagonale (c'est-à-dire les cases vérifiant $i + j = d$ pour un d fixé) ne présentent aucunes dépendances les unes vis-à-vis des autres. Ces cases peuvent donc être calculées simultanément sans que cela ne pose problème. Par ailleurs, si l'on combine ce découpage par antidiagonales avec la restriction de la matrice détaillée plus haut, on travaille alors avec des antidiagonales de taille fixe ($2w$ ou $2w + 1$ selon la parité).

Puisque les cases suivent toutes la même relation de récurrence, elles peuvent être calculées en effectuant les mêmes instructions (moyennant l'ajout de valeurs par défaut pour calculer correctement les cases du bord), c'est le paradigme SIMD (Single Instruction Multiple Data). L'idée est alors de vectoriser les antidiagonales de la matrice : on représente chaque antidiagonale par un grand entier dans lequel chaque bloc représente une case de la matrice. Dans notre implémentation, nous avons utilisé les jeux d'instructions SSE et AVX (dont la documentation est disponible sur le site d'Intel[4]) et nous avons représenté chaque antidiagonale par un entier de 256 bits divisé en 16 blocs de 16 bits.

[4]https://software.intel.com/sites/landingpage/IntrinsicsGuide/



(a) Dépendances entre les antidiagonales. (b) Antidiagonales dans la matrice restreinte.

Fig. 4. Schémas des antidiagonales dans la matrice des scores utilisée pour l'algorithme de Needleman—Wunsch. La première figure illustre le fait que les cases appartenant à une même antidiagonale ne présentent pas de dépendances entre elles. La seconde figure montre que les antidiagonales de la matrice restreinte sont de taille fixe et qu'elles peuvent donc être vectorisées.

3.3 Analyse de la complexité

3.3.1 Recherche des kmers communs. Pour stocker le dictionnaire des k-mers de chaque séquence, nous utilisons une table de hachage. En supposant que la fonction de hachage répartisse les collisions de façon uniforme, la recherche, l'ajout et la suppression d'éléments dans la table de hachage se fait en $O(1)$. La recherche des k-mers communs se fait donc avec une complexité temporelle linéaire en la taille des deux séquences.

3.3.2 Alignement des sous-séquences. L'alignement de deux sous-séquences de taille n se fait en $O(nd)$ où d désigne la largeur des antidiagonales. Dans le cas où les antidiagonales sont suffisamment petites pour être vectorisées, l'alignement se fait en $O(n)$.

3.3.3 Modélisation probabiliste des erreurs. On note p la proportion d'erreurs dans les séquences à traiter et on pose E_i l'évènement : il y a une erreur dans la i-ème paire de bases. Sous l'hypothèse que les erreurs sont réparties de façon uniforme dans les séquences, on a donc $\mathbf{P}(E_i) = p$. On note ensuite K_i l'évènement : le k-mer débutant à la position i ne présente aucune erreur. En supposant que les E_i sont deux à deux indépendants, on a alors

$$\mathbf{P}(K_i) = \mathbf{P}\left(\bigcap_{j=i}^{i+k-1} \overline{E_j}\right) = \prod_{j=i}^{i+k-1} \mathbf{P}(E_j) = (1-p)^k$$

On cherche maintenant à estimer le nombre de bases qu'il faut lire avant de trouver un k-mer entier sans erreurs. Pour cela, on va définir une chaîne de Markov à $k+1$ états $Q_0, \ldots, Q_k$ de façon à être dans l'état Q_i lorsque les i dernières bases lues ne comportent pas d'erreurs et à rester dans l'état Q_k une fois que l'on a lu un k-mer sans erreurs. On définit alors la matrice de transition

suivante :

$$M = \begin{bmatrix} p & \dots & p & 0 \\ 1-p & & (0) & \vdots \\ & \ddots & & 0 \\ (0) & & 1-p & 1 \end{bmatrix}$$

On définit ensuite une suite de vecteurs aléatoires $X_n \in \mathcal{M}_{k+1,1}(\mathbf{R})$ correspondant à la probabilité d'être dans un état donné après avoir lu n bases. On a donc

$$X_0 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

et pour tout $n \in \mathbf{N}$,

$$X_{n+1} = MX_n$$

Par une récurrence immédiate, on en déduit que pour tout $n \in \mathbf{N}$,

$$X_n = M^n X_0$$

La probabilité de trouver un k-mer sans erreurs en lisant n bases vaut donc

$$p_n = U_{k+1} M^n X_0$$

avec $U_{k+1} = \begin{bmatrix} 0 & \dots & 0 & 1 \end{bmatrix}$.

On pose maintenant une variable aléatoire L comptant le nombre de lectures nécessaires avant de trouver un k-mer sans erreurs. D'après les résultats précédents, on a

$$\mathbf{P}(L = n) = p_n - p_{n-1} = U_{k+1} M^{n-1} (M - I_{k+1}) X_0$$

Comme L est une variable aléatoire à valeur entière, on a

$$\mathbf{E}(L) = \sum_{n=0}^{\infty} \mathbf{P}(L > n) = \sum_{n=0}^{\infty} (1 - p_n) = \sum_{n=0}^{\infty} (1 - U_{k+1} M^n X_0)$$

Par conséquent, la taille moyenne des alignements effectués par notre algorithme vaut donc

$$\mathbf{E}(L) - k = \sum_{n=0}^{\infty} (1 - U_{k+1} M^n X_0) - k$$

4 RÉSULTATS

Pour évaluer les performances de notre algorithme par rapport à l'état de l'art, nous avons mesuré les temps d'exécution et les scores obtenus sur un grand panel de séquences génomiques fournies par Pacific Biosciences[5]. Nous avons calculé pour chaque alignement une moyenne sur 10 exécutions afin d'avoir des mesures significatives.

Nous avons choisi de comparer notre algorithme avec `ksw2`[6], l'algorithme d'alignement qui est au cœur de l'outil `minimap2`. C'est cet algorithme qui est actuellement utilisé par Pacific Biosciences lors de la phase de correction d'erreurs[7].

Nous avons également mesuré les temps d'exécution d'`edlib`, qui sont détaillés en annexe, afin d'avoir un point de comparaison avec les méthodes les plus performantes à l'heure actuelle. Cependant, puisque `edlib` calcule uniquement une distance d'édition (les coûts d'insertion / délétion / substitution ne sont donc pas paramétrables), il n'est pas pertinent de vouloir comparer les alignements obtenus.

Afin d'avoir un point de comparaison équitable, nous avons prétraité ces séquences pour ne garder que les zones où elles se recouvrent. En effet, les séquences brutes présentent parfois des décalages importants en début ou en fin de séquence ce qui biaise fortement les résultats obtenus.

Les résultats que nous avons obtenus peuvent être reproduits sur d'autres jeux de données avec la méthode suivante :

(1) On commence par regrouper les paires de séquences que l'on souhaite aligner.
(2) On effectue ensuite un prétraitement sur chaque paire de séquence afin de conserver uniquement les régions qui se recouvrent. Pour cela, on peut utiliser l'outil `cut` qui est disponible avec notre implémentation de l'algorithme[8].
(3) On mesure le temps d'exécution de `coal` et `ksw2` en veillant à utiliser les mêmes paramètres pour le calcul du score. Dans notre cas, nous avons choisi d'utiliser GNU Time[9] et de prendre comme paramètres $\alpha = 2$, $\beta = 3$, et $\delta = 2$.

Nous avons utilisé les commandes suivantes pour répéter 10 fois l'alignement d'une paire de séquences avec les paramètres souhaités :

```
./ksw2-test -R 10 -t gg2_sse -A 2 -B 3 -O 0 -E 2 <seq1.fasta> <seq2.fasta>
./coal_benchmark_10 <seq1.fasta> <seq2.fasta>
```

Les mesures des temps d'exécution ont été effectuées avec la commande suivante :

```
/usr/bin/time -f '%e' <some command> 1>>score_output_file 2>>time_output_file
```

[5]Pacific Biosciences est, avec Oxford Nanopore, l'une des deux sociétés qui développe les technologies de séquençage de troisième génération.
[6]https://github.com/lh3/ksw2
[7]https://ccs.how/how-does-ccs-work.html
[8]https://github.com/imartayan/coal
[9]https://www.gnu.org/software/time/

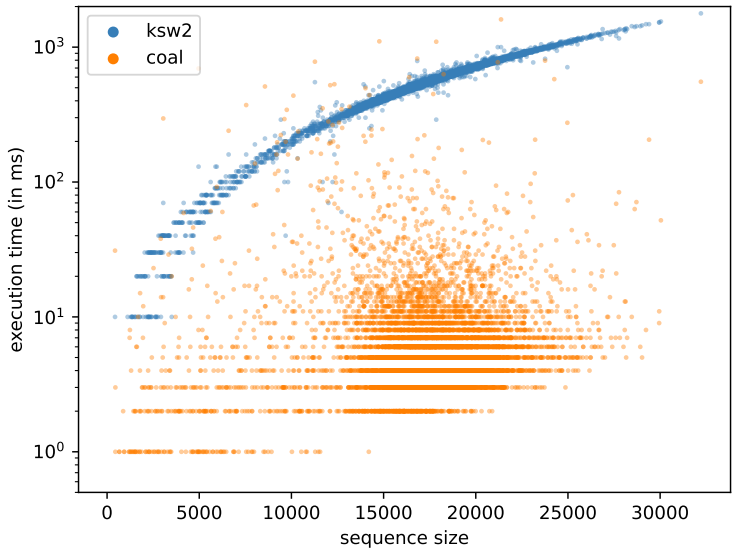


Fig. 5. Graphique en échelle logarithmique des temps d'exécution de `coal` et `ksw2` sur le jeu de données fourni par PacBio.

	taille des séquences			
	14000–16000	16000–18000	18000–20000	20000–22000
algorithme	temps d'exécution moyen (en ms)			
`ksw2`	430	531	645	776
`coal`	10.2	8.0	7.2	10.8
facteur d'accélération	× 42	× 74	× 90	× 72

Tab. 1. Tableau comparatif des temps d'exécution moyens de `coal` et `ksw2` en fonction de la taille des séquences sur le jeu de données fourni par PacBio.

	taille des séquences			
	14000–16000	16000–18000	18000–20000	20000–22000
algorithme	score moyen			
`ksw2`	18967	21955	24543	26878
`coal`	18304	21359	23931	26071
écart relatif	3.5%	2.7%	2.5%	3.0%

Tab. 2. Tableau comparatif des scores moyens de `coal` et `ksw2` en fonction de la taille des séquences sur le jeu de données fourni par PacBio. L'écart relatif désigne le rapport entre l'écart des deux scores et le score maximal.

Les mesures des temps d'exécution montrent que notre algorithme est effectivement plus rapide que `ksw2` avec une accélération de quasiment deux ordres de grandeur sur les données testées. Qui plus est, l'écart de performance semble s'accroître lorsque la taille des séquences augmentent, ce qui peut s'expliquer par une différence de complexité entre les deux algorithmes. On peut également noter que la répartition des temps d'exécution de notre algorithme est assez sporadique comparée à celle de `ksw2`. Cela s'explique principalement par le fait que la taille des sous-séquences que l'on doit aligner peut grandement varier d'une séquence à une autre, ce qui se traduit par des temps de calcul assez variables.

Par ailleurs, même si notre algorithme est plus rapide, les scores obtenus restent inférieurs à ceux de `ksw2` avec un écart moyen d'environ 3%. Il est difficile de savoir précisément ce qui cause cet écart, mais on peut tout de même émettre quelques hypothèses :

- Une première hypothèse est que les alignements effectués sur les sous-séquences sont un peu moins bons que ceux produits par `ksw2`. Comme les paramètres utilisés sont identiques, le problème viendrait donc plutôt de la restriction des matrices des scores autour de la diagonale principale. Cependant, même après avoir augmenté la largeur de bande pour le calcul des matrices, l'écart de score reste significatif, ce qui laisse donc penser que cette hypothèse est assez peu probable.
- Une seconde hypothèse, peut-être plus vraisemblable, est que le problème vient de la fragmentation des séquences effectuée par l'algorithme. En effet, il est possible que l'algorithme identifie un k-mer commun aux deux séquences et l'utilise à tort comme point d'ancrage pour l'alignement alors que ce n'est pas le cas dans l'alignement optimal. Par exemple, cela pourrait se produire si un k-mer était répété plusieurs fois de façon locale dans les deux séquences. Toutefois, ce cas de figure devrait être assez rare lorsque nous utilisons des k-mers de taille 16. De plus, augmenter la taille des k-mers ne semble pas résoudre le problème (des statistiques sur les scores obtenus en fonction de la taille des k-mers sont disponibles en annexe).

5 CONCLUSION

Ce stage nous a amené à concevoir un nouvel algorithme d'alignement de séquences qui apporte un gain de performances notable par rapport aux algorithmes existants. De plus, le fractionnement des alignements, qui peuvent donc être répartis entre différents processeurs, et la faible empreinte mémoire rendent cet algorithme particulièrement adapté à une architecture *processing-in-memory*.

Toutefois, nous avons pu constater que les scores obtenus sont encore un peu inférieurs à ceux attendus. Il serait donc intéressant de comprendre ce qui cause cette différence afin d'améliorer cette méthode ou bien d'en proposer une nouvelle.

Bien que nous ayons atteint notre objectif, il reste encore de nombreuses pistes d'amélioration :

- Si cet algorithme est amené à être utilisé plus largement, il serait pertinent d'en proposer une implémentation avec l'algorithme de Gotoh [Gotoh 1982] afin de pouvoir utiliser des fonctions de coût affines qui sont plus répandues.
- La méthode de hachage des k-mers peut très certainement être améliorée. Elle est actuellement assez rudimentaire, car elle a été pensée pour être facilement portée sur une architecture *processing-in-memory*. Dans le cas général, on gagnerait sûrement en efficacité en utilisant des méthodes de hachage plus modernes telles que Robin Hood Hashing [Celis et al. 1985] ou Cuckoo Hashing [Pagh and Rodler 2004].
- Il pourrait être intéressant d'affiner le choix des k-mers utilisés comme points d'ancrage (en privilégiant les k-mers plus rares ou plus complexes par exemple) et d'observer l'impact sur la qualité des résultats.
- La vectorisation des antidiagonales pourrait être poussée encore plus loin pour améliorer les performances. Actuellement, on se limite à vectoriser les antidiagonales de 16 cases ou moins, mais on pourrait facilement élargir cela à 32 ou 64 cases.
- Il serait intéressant de remplacer notre implémentation de l'algorithme de Needleman—Wunsch par d'autres implémentations utilisées plus largement afin de voir si les résultats obtenus sont meilleurs.

RÉFÉRENCES

Khaled Benkrid, Ying Liu, and AbdSamad Benkrid. 2009. A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 4 (2009), 561–570.

Humberto Carrillo and David Lipman. 1988. The multiple sequence alignment problem in biology. *SIAM journal on applied mathematics* 48, 5 (1988), 1073–1082.

Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. IEEE, 281–288.

Robert C Edgar. 2004. MUSCLE : a multiple sequence alignment method with reduced time and space complexity. *BMC bioinformatics* 5, 1 (2004), 1–19.

Da-Fei Feng and Russell F Doolittle. 1987. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of molecular evolution* 25, 4 (1987), 351–360.

Osamu Gotoh. 1982. An improved algorithm for matching biological sequences. *Journal of molecular biology* 162, 3 (1982), 705–708.

Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles SH Yeo, and Brian YH Lam. 2012. BarraCUDA-a fast short read sequence aligner using graphics processing units. *BMC research notes* 5, 1 (2012), 1–7.

Dominique Lavenier. 2021. Constrained Consensus Sequence Algorithm for DNA Archiving. *arXiv preprint arXiv :2105.04993* (2021).

Dominique Lavenier, Remy Cimadomo, and Romaric Jodin. 2020. Variant Calling Parallelization on Processor-in-Memory Architecture. In *BIBM 2020 - IEEE International Conference on Bioinformatics and Biomedicine*. IEEE, Virtual, South Korea, 1–4. https://hal.archives-ouvertes.fr/hal-03006764

Dominique Lavenier, Charles Deltel, David Furodet, and Jean-François Roy. 2016a. *BLAST on UPMEM*. Research Report RR-8878. INRIA Rennes - Bretagne Atlantique. 20 pages. https://hal.archives-ouvertes.fr/hal-01294345

Dominique Lavenier, Charles Deltel, David Furodet, and Jean-François Roy. 2016b. *MAPPING on UPMEM*. Research Report RR-8923. INRIA. 17 pages. https://hal.archives-ouvertes.fr/hal-01327511

Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

Heng Li. 2018. Minimap2 : pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100.

David J Lipman, Stephen F Altschul, and John D Kececioglu. 1989. A tool for multiple sequence alignment. *Proceedings of the National Academy of Sciences* 86, 12 (1989), 4412–4415.

Ruibang Luo, Thomas Wong, Jianqiao Zhu, Chi-Man Liu, Xiaoqian Zhu, Edward Wu, Lap-Kei Lee, Haoxiang Lin, Wenjuan Zhu, David W Cheung, et al. 2013. SOAP3-dp : fast, accurate and sensitive GPU-based short read aligner. *PloS one* 8, 5 (2013), e65632.

Paul Medvedev and Mihai Pop. 2021. What do Eulerian and Hamiltonian cycles have to do with genome assembly ? *PLoS Computational Biology* 17, 5 (2021), e1008928.

Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Processing data where it makes sense : Enabling in-memory computation. *Microprocessors and Microsystems* 67 (2019), 28–41.

Gene Myers. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)* 46, 3 (1999), 395–415.

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

Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

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

Martin Šošić and Mile Šikić. 2017. Edlib : a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics* 33, 9 (2017), 1394–1395.

Hajime Suzuki and Masahiro Kasahara. 2018. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC bioinformatics* 19, 1 (2018), 33–47.

Esko Ukkonen. 1985. Algorithms for approximate string matching. *Information and control* 64, 1-3 (1985), 100–118.

Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. 2017. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome research* 27, 5 (2017), 737–746.

Lusheng Wang and Tao Jiang. 1994. On the complexity of multiple sequence alignment. *Journal of computational biology* 1, 4 (1994), 337–348.

ANNEXES

A FIGURES SUPPLÉMENTAIRES

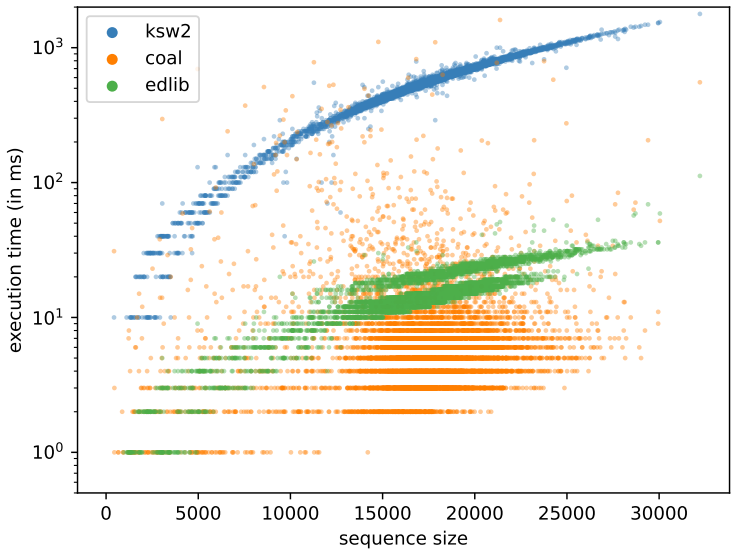


Fig. 6. Graphique en échelle logarithmique des temps d'exécution de `coal`, `edlib` et `ksw2` sur le jeu de données fourni par PacBio.

B STATISTIQUES

	taille des séquences			
	14000–16000	16000–18000	18000–20000	20000–22000
algorithme	temps d'exécution moyen (en ms)			
`ksw2`	430	531	645	776
`edlib`	12.8	14.2	16.7	20.5
`coal`	10.2	8.0	7.2	10.8

Tab. 3. Tableau comparatif des temps d'exécution moyens de `coal`, `edlib` et `ksw2` en fonction de la taille des séquences sur le jeu de données fourni par PacBio.

	taille des séquences			
	14000–16000	16000–18000	18000–20000	20000–22000
taille des k-mers	temps d'exécution moyen (en ms)			
14	7.8	6.6	5.8	8.4
16	10.2	8.0	7.2	10.8
18	14.9	10.9	10.2	14.9

Tab. 4. Tableau comparatif des temps d'exécution moyens pour différentes tailles de k-mers.

	taille des séquences			
	14000–16000	16000–18000	18000–20000	20000–22000
taille des k-mers	score moyen			
14	18312	21337	23932	26011
16	18304	21359	23931	26071
18	18308	21360	23910	26138

Tab. 5. Tableau comparatif des scores moyens pour différentes tailles de k-mers.