République Algérienne Démocratique et Populaire
Université des Sciences et de la Technologie HOUARI BOUMEDIENE
Faculté d'Informatique

Parallélisation de l'algorithme d'alignement local Smith-Waterman

Rédigé par
— MOUHOUB Massinissa
— ZIGADI Sadji Amine

Table des matières

1 Présentation de l'algorithme Smith-Waterman **3**
1.1 Définition 3
1.2 Étapes principales de l'algorithme 3
1.2.1 Initialisation de la matrice de score 3
1.2.2 Remplissage de la matrice 4
1.2.3 Recherche de l'alignement 4
1.2.4 Pseudo-code 4
1.3 Déroulement d'un exemple 4
1.3.1 La matrice initiale 6
1.3.2 Remplissage de la matrice 6
1.3.3 Interprétation des résultats 7
1.4 Complexité temporelle et spatiale de l'algorithme 7
1.4.1 Complexité temporelle 7
1.4.2 Complexité spatiale 8

2 Parallelisation de l'algorithm Smith-Waterman **9**
2.1 Analyse du script séquentiel 10
2.2 Script parallèle 11

3 Analyse des performances **13**

Introduction

La bio-informatique est un domaine clé qui lie les sciences biologiques à l'informatique, permettant de résoudre des problèmes complexes dans le traitement des données biologiques. L'alignement de séquences d'ADN est une tâche essentielle dans ce domaine, et l'algorithme Smith-Waterman est l'un des plus utilisés pour l'alignement local de séquences. Cet algorithme est particulièrement important pour la recherche de similarités entre différentes séquences génétiques, et il est basé sur la programmation dynamique pour trouver la solution optimale.

Ce projet se concentre sur la parallélisation de l'algorithme Smith-Waterman, en utilisant des techniques de programmation parallèle sous Python. L'objectif est d'améliorer l'efficacité de cet algorithme en réduisant le temps d'exécution, en particulier pour les grandes bases de données biologiques. Pour cela, nous analyserons la complexité temporelle et spatiale de l'algorithme, identifierons les goulots d'étranglement à l'aide d'outils de profiling, et proposerons un modèle de parallélisation adapté à une architecture multicoeurs.

Le rapport détaillera les différentes étapes de l'implémentation, y compris la présentation de l'algorithme Smith-Waterman, son analyse de complexité, la proposition d'un modèle de parallélisation, ainsi que l'évaluation des performances entre la version séquentielle et la version parallèle.

Chapitre 1

Présentation de l'algorithme Smith-Waterman

L'algorithme Smith-Waterman est une méthode d'alignement local utilisée en bioinformatique pour trouver des régions similaires entre deux séquences d'ADN, ARN ou protéines. Contrairement à l'algorithme Needleman-Wunsch, qui effectue un alignement global, Smith-Waterman se concentre sur l'identification des sous-séquences optimales tout en permettant des écarts et des substitutions.

1.1 Définition

L'algorithme Smith-Waterman identifie les régions d'alignement local optimales entre deux séquences en maximisant un score d'alignement basé sur une matrice de substitution et des pénalités pour les écarts (gaps). Il est particulièrement utile lorsque seules certaines parties des séquences présentent une similarité significative.

1.2 Étapes principales de l'algorithme

Les étapes de l'algorithme sont résumées ci-dessous. Voir 1.1 pour mieux visualiser.

1.2.1 Initialisation de la matrice de score

Une matrice $\mathbf{H}$ de dimensions $(m+1) \times (n+1)$ est créée, où m et n sont les longueurs des deux séquences respectivement. Les valeurs initiales de la première ligne et de la première colonne sont toutes définies à zéro.

1.2.2 Remplissage de la matrice

Chaque cellule $H(i, j)$ est calculée en fonction des scores suivants :

— $H(i-1, j-1) + score(x_i, y_j)$; en sachant que la fonction $score(x_i, y_j)$ retourne une valeur en fonction de si x_i et y_j sont similaire ou non (match ou missmatch).

— $H(i-1, j) - gap\_penalty$; avec $gap\_penalty$ une valeur définit qui représente le coût associé à l'introduction ou à l'extension d'une lacune (ou "gap") dans un alignement.

— $H(i, j-1) - gap\_penalty$.

$H(i, j)$ prend la valeur maximale parmi ces options.

1.2.3 Recherche de l'alignement

Le score maximal dans la matrice indique la fin de la région d'alignement local optimal. À partir de la cellule ayant le score maximal, on remonte dans la matrice en suivant les cellules ayant contribué au score jusqu'à atteindre une cellule avec un score de 0.

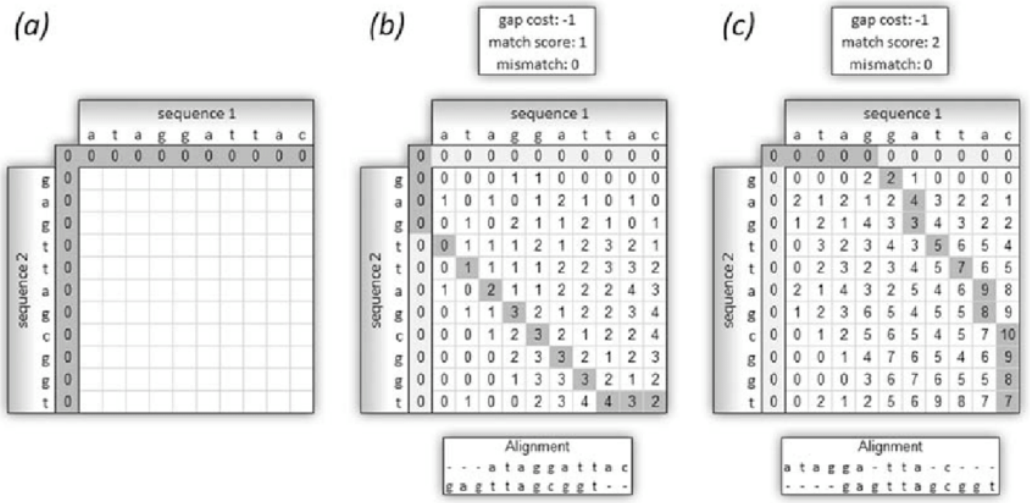

FIGURE 1.1 – Étapes de déroulement de l'algorithme Smith-Waterman.

1.2.4 Pseudo-code

1.3 Déroulement d'un exemple

Séquences

Algorithm 1 Alignement local avec Smith-Waterman

Require: Deux séquences `seq1` (longueur m) et `seq2` (longueur n)
Ensure: Alignement local et score d'alignement

Initialiser une matrice H de taille $(m+1) \times (n+1)$ avec des zéros.
Définir les pénalités : `match`, `mismatch`, et `gap`.
for $i \leftarrow 1$ à m **do**
    for $j \leftarrow 1$ à n **do**
        $match\_score \leftarrow H[i-1][j-1] + score(seq1[i], seq2[j])$
        $delete\_score \leftarrow H[i-1][j] - gap\_penalty$
        $insert\_score \leftarrow H[i][j-1] - gap\_penalty$
        $H[i][j] \leftarrow \max(0, match\_score, delete\_score, insert\_score)$
    end for
end for
Trouver la valeur maximale dans H et sa position $(i_{\max}, j_{\max})$.
Effectuer un *traceback* à partir de $(i_{\max}, j_{\max})$ pour reconstruire l'alignement :
while $H[i][j] > 0$ **do**
    if $H[i][j] == H[i-1][j-1] + score(seq1[i], seq2[j])$ **then**
        Ajouter $seq1[i]$ et $seq2[j]$ à l'alignement.
        $i \leftarrow i-1$, $j \leftarrow j-1$
    else if $H[i][j] == H[i-1][j] - gap\_penalty$ **then**
        Ajouter un gap à `seq2`.
        $i \leftarrow i-1$
    else
        Ajouter un gap à `seq1`.
        $j \leftarrow j-1$
    end if
end while
return L'alignement et le score maximal.

— Séquence 1 : `GATTACA`.
— Séquence 2 : `GCATGCU`.

Paramètres

— Match : +2.
— Mismatch : -1.
— Gap : -2.

1.3.1 La matrice initiale

On commence avec une matrice remplie de zéros :

		G	C	A	T	G	C	U
	0	0	0	0	0	0	0	0
G	0							
A	0							
T	0							
T	0							
A	0							
C	0							
A	0							

TABLE 1.1 – Étape 1 : Initialisation de la matrice

1.3.2 Remplissage de la matrice

On remplit le reste de la matrice en utilisant la méthode mentionnée précédemment, voir sous-section 1.2.2. On obtient ceci :

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

TABLE 1.2 – Étape 2 : Remplissage de la matrice

1.3.3 Interprétation des résultats

Après remplissage, on trouve la région avec le score maximal et on effectue une remontée pour déterminer l'alignement local optimal, on remonte jusqu'à trouver un 0. On peut trouver plusieurs alignements locaux, c'est le cas dans notre exemple.

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

FIGURE 1.2 – Résultat des alignements locales

Dans l'exemple traité, on a les deux alignements suivants :
— Alignement locale 1 : `AT`.
— Alignement locale 2 : `CA`.

1.4 Complexité temporelle et spatiale de l'algorithme

1.4.1 Complexité temporelle

Afin de trouver la complexité temporelle de l'algorithme de Smith-Waterman, on va calculer la complexité de toutes les étapes individuellement, ensuite on déduit la complexité totale.

— **Initialisation de la matrice** Boucle imbriqué, la première a une complexité de n (n la taille de la première séquence) et m (m la taille de la deuxième séquence) alors on aura $O(n \times m)$.
— **Remplissage de la matrice** Même chose que l'initialisation de la matrice, et on a l'étape de la valeur maximale en plus qui coutera une complexité linéaire ($O(4) = O(1)$, 4 parcequ'on compare entre 4 valeurs ; diagonale, horizontale, verticale et le 0) on aura $O(n \times m)$.
— **Interprétation des résultats** On a une boucle tant que qui va s'executer au pire la taille minimale entre les deux séquences, alors on aura

$O(min(n, m))$.
Alors la complexité temporelle est de $O(n \times m)$.

1.4.2 Complexité spatiale

Pour l'algorithme, on utilise une seule matrice de taille $n \times m$, alors la complexité spatiale est de $O(n \times m)$.

Chapitre 2

Parallelisation de l'algorithm Smith-Waterman

Tout d'abord, on présente le script *Python* séquentiel de l'algorithme Smith-Waterman.

```
1  def smith_waterman(seq1, seq2, match_score, mismatch_score,
       gap_penalty):
2      rows = len(seq1) + 1
3      cols = len(seq2) + 1
4      H = [[0 for _ in range(cols)] for _ in range(rows)]
5
6      max_score = 0
7      max_positions = []
8
9      for i in range(1, rows):
10         for j in range(1, cols):
11             value_1 = H[i-1][j-1] + (match_score if seq1[i-1] ==
                   seq2[j-1] else mismatch_score)
12             value_2 = H[i-1][j] + gap_penalty
13             value_3 = H[i][j-1] + gap_penalty
14
15             H[i][j] = max(0, value_1, value_2, value_3)
16
17             if H[i][j] > max_score:
18                 max_score = H[i][j]
19                 max_positions = [(i, j)]
20             elif H[i][j] == max_score:
21                 max_positions.append((i, j))
22
23     print("Scoring Matrix:")
```

```
24      for row in H:
25          print(" ".join(map(str, row)))
26
27      def traceback(i, j, align1, align2, alignments):
28          if i == 0 or j == 0 or H[i][j] == 0:
29              alignments.append((align1[::-1], align2[::-1]))
30              return
31
32          current_score = H[i][j]
33          diagonal = H[i-1][j-1]
34          up = H[i-1][j]
35          left = H[i][j-1]
36
37          if current_score == diagonal + (match_score if seq1[i-1] ==
                seq2[j-1] else mismatch_score):
38              traceback(i-1, j-1, seq1[i-1] + align1, seq2[j-1] +
                    align2, alignments)
39
40          if current_score == up + gap_penalty:
41              traceback(i-1, j, seq1[i-1] + align1, "-" + align2,
                    alignments)
42
43          if current_score == left + gap_penalty:
44              traceback(i, j-1, "-" + align1, seq2[j-1] + align2,
                    alignments)
45
46      all_alignments = []
47      for pos in max_positions:
48          traceback(pos[0], pos[1], "", "", all_alignments)
49
50      unique_alignments = list(set(all_alignments))
51
52      return unique_alignments
```

Listing 2.1 – Script de Smith-Waterman séquentiel

2.1 Analyse du script séquentiel

Afin de savoir quels sont les tâches à paralléliser dans le code séquentiel, on a déroulé plusieurs tests en utilisant le module *cProfile*.

Avec deux séquences de taille 108, on obtient les résultats voir fig 2.1.

Depuis fig 2.1, on remarque que l'étape qui prends le plus de temps dans

```
Profiling Results:
         22381521 function calls (3157581 primitive calls) in 13.992 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.014    0.014   13.992   13.992 profiling.py:13(main)
        1    1.495    1.495   13.978   13.978 main.py:26(smith_waterman)
19223941/1   12.399    0.000   12.476   12.476 main.py:31(traceback)
  3145799    0.078    0.000    0.078    0.000 {method 'append' of 'list' objects}
        1    0.005    0.005    0.006    0.006 main.py:1(initialize_and_fill_matrix)
    11664    0.001    0.000    0.001    0.000 {built-in method builtins.max}
      109    0.000    0.000    0.000    0.000 main.py:4(<listcomp>)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

FIGURE 2.1 – Résultat du profiling de deux séquences de 108 caractères.

l'algorithme est la fonction `trackback`, qui permet de trouver tous les alignements locales entre les deux séquences. Alors on va créer un script parallèle qui parallélise cette fonctionnalité.

2.2 Script parallèle

Voici la version parallèle :

```
1     def traceback_worker(args):
2         i, j, seq1, seq2, H, match_score, mismatch_score,
              gap_penalty = args
3         alignments = []
4
5         def traceback(i, j, align1, align2):
6             if i == 0 or j == 0 or H[i][j] == 0:
7                 alignments.append((align1[::-1], align2[::-1]))
8                 return
9
10            current_score = H[i][j]
11            diagonal = H[i-1][j-1]
12            up = H[i-1][j]
13            left = H[i][j-1]
14
15            if current_score == diagonal + (match_score if seq1[i-1]
                  == seq2[j-1] else mismatch_score):
16                traceback(i-1, j-1, seq1[i-1] + align1, seq2[j-1] +
                      align2)
17
18            if current_score == up + gap_penalty:
```

```
19                traceback(i-1, j, seq1[i-1] + align1, "-" + align2)
20
21            if current_score == left + gap_penalty:
22                traceback(i, j-1, "-" + align1, seq2[j-1] + align2)
23
24        traceback(i, j, "", "")
25        return alignments
26
27
28    def smith_waterman(seq1, seq2, match_score, mismatch_score,
          gap_penalty):
29        H, max_score, max_positions =
              initialize_and_fill_matrix(seq1, seq2, match_score,
              mismatch_score, gap_penalty)
30
31        args = [(pos[0], pos[1], seq1, seq2, H, match_score,
              mismatch_score, gap_penalty) for pos in max_positions]
32
33        with multiprocessing.Pool(processes=2) as pool:
34            results = pool.map(traceback_worker, args)
35
36        all_alignments = [alignment for result in results for
              alignment in result]
37        unique_alignments = list(set(all_alignments))
38
39        return unique_alignments, max_score
```

Ici la fonction `traceback_worker` c'est la fonction principale qui permet de paralléliser le programme. Chaque processus va exécuter cette fonction en parallèle et va générer l'alignement.

Chapitre 3

Analyse des performances

Après plusieurs tests, on a trouvé les résultats suivants :

	seq	par			
		2	4	6	8
10	0.000384	0.061976	0.057896	0.073779	0.07761
20	0.000354	0.054117	0.071049	0.077192	0.099325
50	0.001179	0.05806	0.072039	0.078288	0.093207
100	0.006348	0.065681	0.074773	0.063099	0.09565
150	0.017743	0.065865	0.087802	0.089017	0.100491

TABLE 3.1 – Résultats sur plusieurs taille de séquences.

On trouve que plus la taille de la séquence devient plus grande plus le parallélisme devient plus intéressant.

On calcule l'accélération et l'efficacité pour la taille de la séquence de 150 caractères :

	Temps	Accélération	Efficacité
Seq	0.017743	/	/
2	0.065865	0.27	0.135
4	0.087802	0.2	0.05
6	0.089017	0.19	0.03
8	0.100491	0.17	0.02

TABLE 3.2 – Accélération et efficacité

On remarque que la parallélisation n'est pas trés intéressant pour le cas de Smith-Waterman.

Conclusion

La parallélisation de l'algorithme Smith-Waterman devient particulièrement intéressante lorsqu'il s'agit de traiter plusieurs alignements locaux simultanément, ce qui est fréquent dans les applications bio-informatiques. En utilisant une architecture multicoeurs, nous avons réussi a déduire que le temps d'exécution de l'algorithme devient plus intéressant pour le cas ou on a plusieurs alignements locaux, en exploitant les possibilités offertes par la programmation parallèle. Cette amélioration permet de traiter plus rapidement de grandes quantités de données biologiques, rendant l'algorithme plus adapté aux exigences des bases de données génétiques modernes. En somme, la parallélisation apporte des gains de performance significatifs, notamment dans des scénarios avec plusieurs alignements locaux à effectuer.