

Université des Sciences et de la Technologie Houari Boumediene
Faculté d'Informatique



ACP

TP : Rapport projet TP final
Programmation parallèle en python appliquée à la
bio-informatique

Fait par :

Nom et prénom : ABDELMALEK BENMEZIANE

Matricule : 171731046778

Spécialité : M2 BIOINFO

Section : A

Contents

1	Algorithme de Smith Watermann	1
1.1	Présentation de l'algorithme	1
1.1.1	Les étapes de l'algorithme	1
1.1.2	Le pseudo code	3
1.2	La complexité temporelle et spatiale de l'algorithme	4
1.3	Le profiling du programme	5
1.3.1	Les 3 parties	5
1.4	Le modèle de parallélisation sur une architecture multicoeurs	8
1.4.1	Justification	8
1.5	Implémentation du modèle parallèle	8
1.6	Les mesures de performance	8
1.6.1	Remarque	9

List of Figures

1.1	Formule de calcul	2
1.2	Conditions de la construction de l'alignement	2
1.3	Pseudo code -1-	3
1.4	Pseudo code -2-	3
1.5	Pseudo code -3-	4
1.6	Pseudo code -4-	4
1.7	Bibliothèque time	5
1.8	Première partie	5
1.9	Deuxième partie	6
1.10	Troisième partie	7
1.11	Résultat	7
1.12	Temps séquentiel	8
1.13	Temps parallèle	9

Chapter 1

Algorithme de Smith Watermann

1.1 Présentation de l'algorithme

L'algorithme de Smith-Waterman est un algorithme d'alignement de séquences utilisé notamment en bioinformatique. Il a été inventé par Temple F. Smith et Michael S. Waterman en 1981.

L'algorithme de Smith-Waterman est un algorithme optimal qui donne un alignement correspondant au meilleur score possible de correspondance entre les acides aminés ou les nucléotides des deux séquences. Le calcul de ce score repose sur l'utilisation de matrices de similarité ou matrices de substitution.

L'algorithme est par exemple utilisé pour aligner des séquences de nucléotides ou de protéines, notamment pour la prédiction de gènes, la phylogénie ou la prédiction de fonction.

1.1.1 Les étapes de l'algorithme

- **L'initialisation d'une matrice M de $m+1$ colonnes et $n+1$ lignes**
La ligne et la colonne sont remplies de 0 parce que nous avons posé qu'il n'y avait pas de pénalités pour des gaps initiaux ou finals.
- **Le calcul de la matrice M des scores d'alignement partiel**

C'est une matrice $m \times n$, dont chaque coefficient $M(i, j)$ donne le score de l'alignement des i premiers acides aminés (ou nucléotides) de A avec les j premiers acides aminés de B. Ce calcul se fait itérativement, à partir des scores des alignements partiels plus courts. Une fois la matrice remplie, le coefficient du coin inférieur droit de la matrice, $M(m, n)$, donne le score de l'alignement optimal complet entre A et B.

$$M(i, j) = \max \begin{cases} 0 \\ M(i-1, j-1) + D(A_i, B_j) \\ M(i-1, j) + \Delta \\ M(i, j-1) + \Delta \end{cases}$$

Figure 1.1: Formule de calcul

• **La construction de l'alignement à partir de la matrice M**

En partant du coin inférieur droit, on remonte dans la matrice pour déterminer par quel chemin on a obtenu le score optimal. Ce chemin correspond à l'alignement optimal.

- si on a $M(i, j) = M(i-1, j-1) + D(A_i, B_j)$, alors on aligne A_i avec B_j et on remonte à $M(i-1, j-1)$;
- si on a $M(i, j) = M(i, j-1) + \Delta$, alors on aligne B_j avec un trou et on remonte à $M(i, j-1)$;
- si on a $M(i, j) = M(i-1, j) + \Delta$, alors on aligne A_i avec un trou et on remonte à $M(i-1, j)$;
- si on a $M(i, j) = 0$, l'alignement local est terminé.

Figure 1.2: Conditions de la construction de l'alignement

1.1.2 Le pseudo code

```
# Helper function to create a matrix filled with zeros
function createMatrix(rows, cols):
    matrix = []
    for i from 0 to rows:
        row = []
        for j from 0 to cols:
            row.append(0)
        matrix.append(row)
    return matrix

function smithWaterman(sequence1, sequence2, matchScore, mismatchScore, gapPenalty):
    m = length(sequence1)
    n = length(sequence2)

    # Initialize the scoring matrix
    scoreMatrix = createMatrix(m + 1, n + 1)

    # Initialize the traceback matrix to store the direction of the optimal alignment
    tracebackMatrix = createMatrix(m + 1, n + 1)

    maxScore = 0
    maxPosition = (0, 0)
```

Figure 1.3: Pseudo code -1-

```
function smithWaterman(sequence1, sequence2, matchScore, mismatchScore, gapPenalty):
    m = length(sequence1)
    n = length(sequence2)

    # Initialize the scoring matrix
    scoreMatrix = createMatrix(m + 1, n + 1)

    # Initialize the traceback matrix to store the direction of the optimal alignment
    tracebackMatrix = createMatrix(m + 1, n + 1)

    maxScore = 0
    maxPosition = (0, 0)

    # Fill the scoring matrix and traceback matrix
    for i from 1 to m:
        for j from 1 to n:
            match = scoreMatrix[i-1][j-1] + (sequence1[i-1] == sequence2[j-1] ? matchScore : mismatchScore)
            delete = scoreMatrix[i-1][j] + gapPenalty
            insert = scoreMatrix[i][j-1] + gapPenalty
            scoreMatrix[i][j] = max(0, match, delete, insert)

    # Update the traceback matrix
    if scoreMatrix[m][n] > 0:
```

Figure 1.4: Pseudo code -2-

```

for i from 1 to m:
    for j from 1 to n:
        match = scoreMatrix[i-1][j-1] + (sequence1[i-1] == sequence2[j-1] ? matchScore : gapPenalty)
        delete = scoreMatrix[i-1][j] + gapPenalty
        insert = scoreMatrix[i][j-1] + gapPenalty
        scoreMatrix[i][j] = max(0, match, delete, insert)

        # Update the traceback matrix
        if scoreMatrix[i][j] == 0:
            tracebackMatrix[i][j] = 0
        elif scoreMatrix[i][j] == match:
            tracebackMatrix[i][j] = 1
        elif scoreMatrix[i][j] == delete:
            tracebackMatrix[i][j] = 2
        elif scoreMatrix[i][j] == insert:
            tracebackMatrix[i][j] = 3

        # Update the maximum score and position
        if scoreMatrix[i][j] > maxScore:
            maxScore = scoreMatrix[i][j]
            maxPosition = (i, j)

# Traceback to find the optimal local alignment

```

Figure 1.5: Pseudo code -3-

```

maxScore = scoreMatrix[i][j]
maxPosition = (i, j)

# Traceback to find the optimal local alignment
alignment1 = ""
alignment2 = ""
i, j = maxPosition
while i > 0 and j > 0 and scoreMatrix[i][j] > 0:
    if tracebackMatrix[i][j] == 1: # Match
        alignment1 = sequence1[i-1] + alignment1
        alignment2 = sequence2[j-1] + alignment2
        i -= 1
        j -= 1
    elif tracebackMatrix[i][j] == 2: # Delete
        alignment1 = sequence1[i-1] + alignment1
        alignment2 = '-' + alignment2
        i -= 1
    elif tracebackMatrix[i][j] == 3: # Insert
        alignment1 = '-' + alignment1
        alignment2 = sequence2[j-1] + alignment2
        j -= 1

return alignment1, alignment2, maxScore

```

Figure 1.6: Pseudo code -4-

1.2 La complexité temporelle et spatiale de l'algorithme

Si on analyse le pseudo code on déduit que :

- Le complexité temporelle :
 $O(m + n) + O(m * n) + O(m * n) = O(m * n)$
- La complexité spatiale : $O(m * n)$

1.3 Le profiling du programme

On divise la fonction `smith_watermann` sur 3 parties sur lesquelles on va faire le profiling en utilisant la bibliothèque **time**.



Figure 1.7: Bibliothèque time

1.3.1 Les 3 parties

```
def smith_waterman(sequence1, sequence2, match_score=2, mismatch_score=-1, gap_penalty=-2):
    t1_start = perf_counter()
    m = len(sequence1)
    n = len(sequence2)

    # Initialize the scoring matrix
    score_matrix = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the traceback matrix
    traceback_matrix = [[0] * (n + 1) for _ in range(m + 1)]

    max_score = 0
    max_position = (0, 0)
    t1_stop = perf_counter()
    print("First part: ", t1_stop - t1_start)
```

Figure 1.8: Première partie


```

t2_start = perf_counter()
# Fill the scoring matrix and traceback matrix
for i in range(1, m + 1):
    for j in range(1, n + 1):
        match = score_matrix[i - 1][j - 1] + (match_score if sequence1[i - 1] == sequence2[j - 1] else
        delete = score_matrix[i - 1][j] + gap_penalty
        insert = score_matrix[i][j - 1] + gap_penalty
        score_matrix[i][j] = max(0, match, delete, insert)

        # Update the traceback matrix
        if score_matrix[i][j] == 0:
            traceback_matrix[i][j] = 0
        elif score_matrix[i][j] == match:
            traceback_matrix[i][j] = 1
        elif score_matrix[i][j] == delete:
            traceback_matrix[i][j] = 2
        elif score_matrix[i][j] == insert:
            traceback_matrix[i][j] = 3

        # Update the maximum score and position
        if score_matrix[i][j] > max_score:
            max_score = score_matrix[i][j]
            max_position = (i, j)

t2_stop = perf_counter()
print("Second part: ", t2_stop - t2_start)

```

Figure 1.9: Deuxième partie

```

t3_start = perf_counter()
# Traceback to find the optimal local alignment
alignment1 = ""
alignment2 = ""
i, j = max_position
while i > 0 and j > 0 and score_matrix[i][j] > 0:
    if traceback_matrix[i][j] == 1: # Match
        alignment1 = sequence1[i - 1] + alignment1
        alignment2 = sequence2[j - 1] + alignment2
        i -= 1
        j -= 1
    elif traceback_matrix[i][j] == 2: # Delete
        alignment1 = sequence1[i - 1] + alignment1
        alignment2 = '-' + alignment2
        i -= 1
    elif traceback_matrix[i][j] == 3: # Insert
        alignment1 = '-' + alignment1
        alignment2 = sequence2[j - 1] + alignment2
        j -= 1

t3_stop = perf_counter()
print("Third part: ", t3_stop - t3_start)

```

Figure 1.10: Troisième partie

Le résultat du profiling est :

```

First part:  1.0199961252510548e-05
Second part: 7.539999205619097e-05
Third part:  4.999979864805937e-06

```

Figure 1.11: Résultat

On remarque que la boucle qui calcule la matrice prend plus de temps, alors cette partie qu'on va la paralléliser.

1.4 Le modèle de parallélisation sur une architecture multicoeurs

Le modèle de parallélisation est le **modèle de programmation pour machine à mémoire partagées (à accès aléatoire parallèle)**.

1.4.1 Justification

- Le code est un programme qui a plusieurs instructions à exécuter et utilise plusieurs données.
- La source de parallélisme est le parallélisme de tâches.
- Le type de mémoire est mémoire partagée.
- L'architecture multicoeurs cible est MIMD avec mémoire partagée.

1.5 Implémentation du modèle parallèle

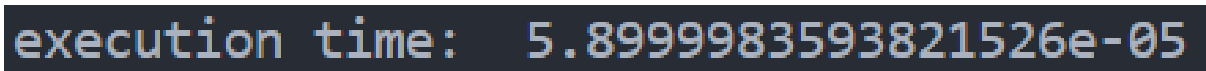
On va implémenter l'algorithme de smith watermann en langage python en utilisant les bibliothèques de la programmation parallèle.

Le script est dans le fichier "**smith_watermann.py**"

1.6 Les mesures de performance

- Accélération: $S(p) = T_{seq}/T_{par}$
- Efficacité: $E(p) = S(p)/p$

Le temps séquentiel (avant la parallélisation):



```
execution time: 5.8999983593821526e-05
```

Figure 1.12: Temps séquentiel

Le temps parallèle (après la parallélisation):

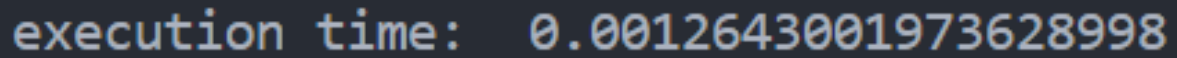
A terminal window with a dark background and light-colored text. The text reads "execution time: 0.0012643001973628998".

Figure 1.13: Temps parallèle

- On applique la formule de l'accélération, on aura :

$$S(p) = T_{seq}/T_{par}$$

$$S(p) = 0,000058999983593821526/0,0012643001973628998 = 0,0466660603$$

- On applique la formule de l'efficacité, on aura:

$$E(p) = S(p)/p$$

Nombre de processeurs (p)	2	3	4	5	6	7
Efficacité	2,3%	1,6%	1,6%	1%	1%	1%

1.6.1 Remarque

L'efficacité se diminue par l'augmentation du nombre de processeurs utilisés dans le programme parallèle.