الجـــمهوريـــة الجزائريـــة الديمـــقراطيـــــة الشـعبيـة
ⵜⴰⵙⴷⴰⵡⵜ ⵜⴰⵣⵣⴰⵢⵔⵉⵜ ⵜⴰⵎⴰⵙⵙⴰⵡⵜ ⵜⴰⵖⵔⴰⵎⵉⵢⵜ
République Algérienne Démocratique et Populaire
وزارة التعليـــم العـــالي والبـــحث العلمـــي
ⵓⵢⵏⵉⵖ ⵏ ⵜⵓⵙⵎⵉⵔⵉⵏ ⵜⵏⵉⵖ ⴰ ⵜⵓⴰⵔⵉ ⵜⵓⵙⵙⵏⵉ
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
ⵜⴰⵙⴷⴰⵡⵜ ⵜⴰⵏⴰⵢⵜ ⵜⴰⵎⵙⵙⵓⵔⵜ ⵏ ⵜⵓⵙⵙⵏⵉ
École nationale Supérieure d'Informatique

Rapport du TP

Programmation parallèle avec Pthreads

2 ème année Cycle Supérieur (2CS)

Option : Système Informatique (SQ)

Thème :

Optimisation du calcul de la "Longest Common Subsequence (LCS)" par la parallélisation

Réalisé par :

- Benameur Tarek
- Touil Nihel

Introduction

La nécessité de la parallélisation devient cruciale face à l'explosion des données et à la complexité croissante des programmes. La parallélisation permet de diviser un problème complexe en plusieurs sous-tâches exécutables simultanément, ce qui accélère le traitement et optimise l'utilisation des ressources dans un système multicœur. Cela est indispensable dans des domaines tels que la simulation, l'analyse de données et l'intelligence artificielle, où le volume de calculs peut être considérable.

Pthreads (POSIX Threads) est une bibliothèque standardisée qui permet la création et la gestion de threads (unités de traitement indépendantes) dans les environnements UNIX et Linux. Elle fournit des outils de synchronisation et de coordination entre threads, permettant aux programmeurs d'exploiter pleinement les architectures multicœurs pour exécuter plusieurs parties d'un programme en parallèle.

Dans ce travail pratique (TP), nous nous concentrerons sur la parallélisation d'un problème complexe ou nécessitant des calculs intensifs. Nous commencerons par définir le problème et décrire l'algorithme utilisé pour le résoudre en version séquentielle. Ensuite, nous évaluerons les avantages potentiels de la parallélisation. Si cela s'avère pertinent, nous élaborerons une stratégie de parallélisation en précisant comment procéder, quel type de parallélisme appliquer, et en mettant en évidence les points de synchronisation et de communication entre les threads. Cette stratégie sera illustrée par un schéma. Enfin, nous implémenterons la solution avec Pthreads et établirons des mesures de comparaison entre différentes combinaisons de métriques pour évaluer les performances.

Git, l'outil de contrôle de version le plus utilisé par les développeurs, utilise l'algorithme LCS (Longest Common Subsequence), qui **détermine la plus longue sous-séquence commune entre deux séquences textuelles,** pour comparer et mettre à jour les versions des fichiers. Cet algorithme permet à Git de déterminer quelles parties de deux fichiers sont identiques et lesquelles diffèrent, facilitant ainsi le suivi des modifications.

Figure : Github's Diff Highlighted text

Détection des similarités : ou l'on identifie les sections communes entre deux versions d'un fichier. En repérant ces séquences partagées, Git se concentre sur les parties ayant réellement changé lors de l'affichage des différences.

Résolution des conflits de fusion : En cas de modifications concurrentes sur un même fichier dans deux branches, LCS permet de détecter les zones communes et divergentes. Git peut ainsi fusionner automatiquement les modifications compatibles et signaler les conflits sur les sections modifiées de façon contradictoire.

Optimisation du stockage : Grâce à la connaissance des différences précises, Git ne stocke que les changements entre les versions, plutôt que des copies complètes. LCS aide à réduire l'espace nécessaire en enregistrant uniquement les modifications.

Fonctionnement de LCS

Le problème de la Longest Common Subsequence (LCS) consiste à déterminer la longueur de la plus longue sous-séquence commune entre deux chaînes de caractères, s1 et s2. Si aucune sous-séquence commune n'existe, le résultat est 0.

Une sous-séquence est une séquence obtenue à partir d'une chaîne originale en supprimant certains caractères, sans modifier l'ordre des caractères restants. Par exemple, pour la chaîne "**ABC**", les sous-séquences possibles incluent "", "**A**", "**B**", "**C**", "**AB**", "**AC**", "**BC**" et "**ABC**".

Prenons un exemple pour illustrer le fonctionnement de l'algorithme :

- Pour les chaînes s1 = "**abcdefghi**" et s2 = "**cgi**", la plus longue sous-séquence commune est "**cgi**".
- En revanche, si nous prenons s2 = "**ecgi**", la plus longue sous-séquence commune reste "**cgi**" ou "**egi**". En effet, "**ecgi**" ne peut pas être considérée comme une LCS, car bien qu'elle contienne des caractères de s1, l'ordre d'apparition n'est pas respecté, et la continuité des caractères n'est pas maintenue.

Cette solution peut être abordée de deux manières : soit par une méthode récursive, soit par une approche de programmation dynamique. Nous allons d'abord examiner la méthode récursive, qui est souvent considérée comme plus intuitive.

A. Approche Récursive

L'idée est de comparer les derniers caractères de s1 et s2. Lors de la comparaison des chaînes s1 et s2, deux cas se présentent :

1. **S'il y a correspondance** : effectuer l'appel récursif pour les chaînes restantes (chaînes de longueurs m-1 et n-1) et ajouter 1 au résultat.
2. **Sinon** : effectuer deux appels récursifs. D'abord pour les longueurs m-1 et n, et ensuite pour m et n-1. Prendre le maximum des deux résultats.

Pseudo-code:

```
Fonction LCS(s1, s2, m, n)
   //Si l'une des chaînes est vide, la longueur de la LCS est 0
   Si m = 0 ou n = 0 alors
      Retourner 0
   Fin Si
   Sinon
    Si s1[m+1] = s2[n+1] alors
```

```
        // Les derniers caractères des sous-chaînes sont égaux
        Retourner 1 + LCS(s1, s2, m+1, n+1)
       // Inclure ce caractère dans la LCS et récursivement    calculer pour les sous-chaînes
restantes
      Sinon
        // Les derniers caractères ne correspondent pas
        // Récursivité pour deux cas :
        // 1. Exclure le dernier caractère de s1
        // 2. Exclure le dernier caractère de s2
        // Prendre le maximum de ces deux appels récursifs
        Retourner max(LCS(s1, s2, m, n+1), LCS(s1, s2, m+1, n))
    Fin Si
Fin
```

Prenons les chaînes :

- s1 = "AXYT"
- s2 = "AYZX"

En déroulant les étapes, nous remarquons vite qu'il y a des calculs redondants, comme le met en exergue l'exemple sur la figure suivante.

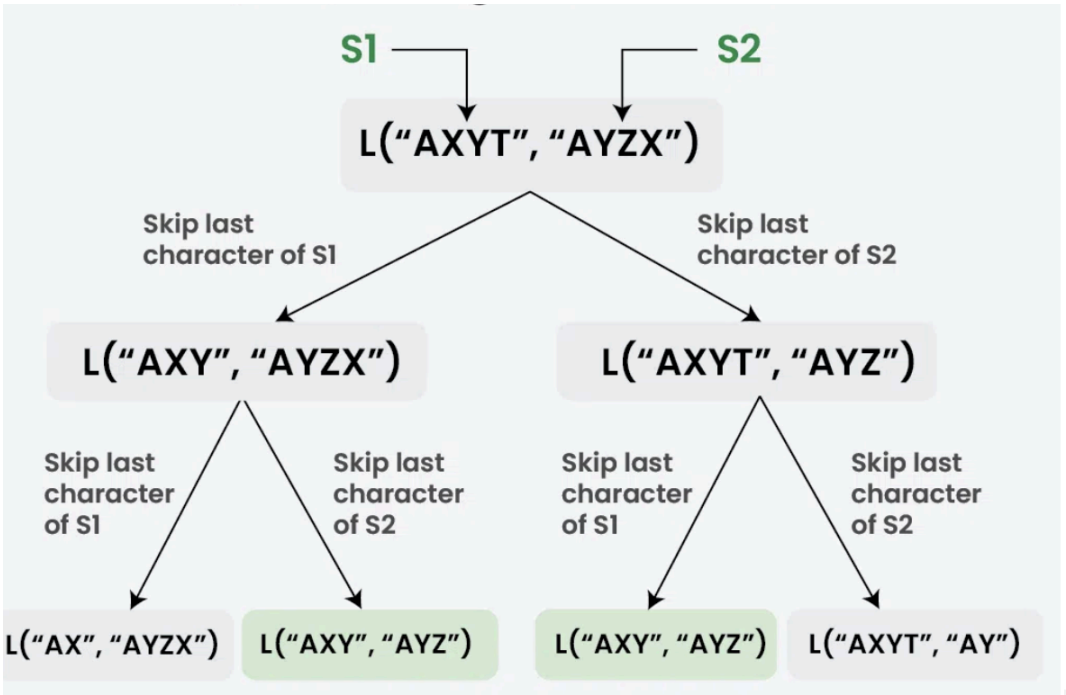


Figure: Sous-problèmes Chevauchants dans la LCS

La méthode récursive présente donc nombreux inconvénients :

- **Redondance des calculs :** La méthode récursive peut entraîner des calculs redondants pour les mêmes sous-problèmes, augmentant considérablement le temps d'exécution, surtout pour de grandes chaînes.
- **Complexité temporelle élevée :** La complexité temporelle est exponentielle, O(2^min(m,n)), ce qui la rend inefficace pour des chaînes longues.
- **Consommation élevée de mémoire :** Les appels récursifs peuvent consommer une grande quantité de mémoire sur la pile, ce qui peut mener à un dépassement de pile (stack overflow) pour des chaînes très longues.

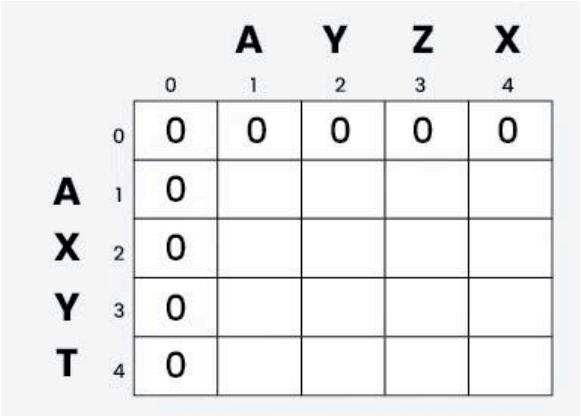
B. Approche par Programmation Dynamique (Top Down Approach)

Une meilleure approche consiste à mémoriser les résultats déjà obtenus en utilisant des tables. Il y a deux paramètres qui varient dans la solution récursive, allant de 0 à m et de 0 à n. Nous créons donc un tableau 2D dp de taille (m+1) x (n+1).

Pseudo-code:

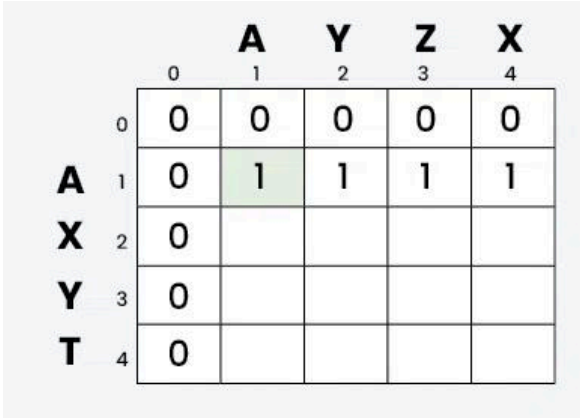
```
Fonction LCS_Tabulation(s1, s2)
    m ← longueur de s1
    n ← longueur de s2
    // Créer un tableau dp de taille (m+1) x (n+1)
    dp ← tableau de taille (m+1) x (n+1)

    // Initialiser les premières lignes et colonnes
    Pour i de 0 à m faire
        dp[i][0] ← 0  // Si s2 est vide
    Fin Pour
    Pour j de 0 à n faire
        dp[0][j] ← 0  // Si s1 est vide
    Fin Pour
    // Remplir le tableau dp en utilisant la formule récursive
    Pour i de 1 à m faire
        Pour j de 1 à n faire
            Si s1[i-1] = s2[j-1] alors
                dp[i][j] ← dp[i-1][j-1] + 1  // Correspondance
            Sinon
                dp[i][j] ← max(dp[i-1][j], dp[i][j-1]) // Pas de correspondance
            Fin Si
        Fin Pour
    Fin Pour
    // Retourner la longueur de la LCS
    Retourner dp[m][n]
Fin
```

déroulons l'algorithme ci-dessus sur le même exemple:



		A	Y	Z	X
	0	0	0	0	0
A	0	0			
X	0	0			
Y	0	0			
T	0	0			

1. L'initialisation de la première ligne et de la première colonne à 0 représente les cas de base de la récurrence.

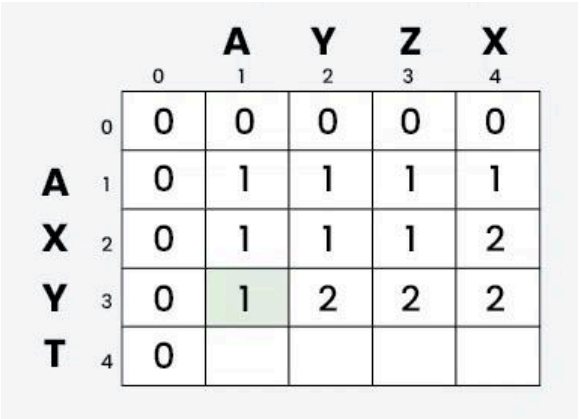


		A	Y	Z	X
	0	0	0	0	0
A	0	0	1	1	1
X	0	0			
Y	0	0			
T	0	0			

2. Pour **dp[1][1]**, nous avons **dp[1][1] = 1 + dp[0][0]** car les caractères sont égaux.
Pour **dp[1][2]**, comme **A != Y**, nous avons **dp[1][2] = max(dp[0][2], dp[1][1])**. De même, pour les autres indices de la ligne 1, nous pouvons remplir le tableau comme indiqué.

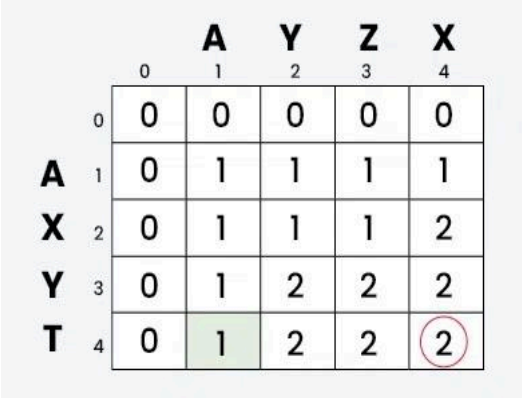
		A	Y	Z	X
	0	0	0	0	0
A	0	1	1	1	1
X	0	1	1	1	2
Y	0	0			
T	0	0			

3. Pour **dp[2][1]**, comme **X != A**, nous avons **dp[2][1] = max(dp[1][1], dp[2][0])**. De même, pour les autres indices de la ligne 2, nous pouvons remplir le tableau comme indiqué.



		A	Y	Z	X
	0	0	0	0	0
A	0	0	1	1	1
X	0	0	1	1	2
Y	0	0	1	2	2
T	0	0			

4. For **dp[3][1]**, as Y != A, **dp[3][1] = max(dp[2][1], dp[3][0])**
Similarly, for other indices of row 3, we can create the table as shown.



		A	Y	Z	X
	0	0	0	0	0
A	1	0	1	1	1
X	2	0	1	1	2
Y	3	0	1	2	2
T	4	0	1	2	2

5. For **dp[4][1]**, as T!= A, **dp[4][1]** = **max(dp[4][0], dp[3][1)**
Similarly, for other indices of row 4, we can create the table as shown.

Avantages de cette méthode

- **Efficacité temporelle** : Complexité de O(n · m), bien plus rapide que la récursion exponentielle.
- **Élimination de la redondance** : Mémorisation des sous-problèmes déjà résolus, évitant les calculs répétés.
- **Accès facile** : Utilisation d'une matrice pour stocker et récupérer rapidement les résultats.
- **Récupération de la solution** : Permet de reconstruire la LCS facilement

Pertinence de parallélisation du programme

Le problème du LCS (Longest Common Subsequence) est un défi computationnel, particulièrement visible avec de grandes chaînes de caractères, comme dans des projets de code source volumineux ou des analyses génétiques. Dans ces contextes, le temps de calcul pour remplir la matrice 2D peut devenir prohibitif avec des approches séquentielles. La parallélisation devient donc essentielle.

En répartissant la charge de travail entre plusieurs threads, la parallélisation réduit considérablement le temps d'exécution. Par exemple, lors de comparaisons de longues séquences d'ADN, l'exécution parallèle peut transformer des heures de traitement en minutes, rendant les workflows plus efficaces. Ce besoin d'efficacité est d'autant plus critique avec de grandes données, où l'optimisation est maximisée.

Dans l'approche dynamique, bien que le calcul des anti-diagonales doive être effectué successivement, les cellules au sein de chaque anti-diagonale peuvent être traitées indépendamment. Cela permet d'optimiser l'utilisation des ressources processeur tout en minimisant les contraintes de synchronisation. Ainsi, la parallélisation améliore non seulement la vitesse de calcul, mais renforce également la scalabilité du programme, permettant une meilleure adaptation aux variations de charge de travail.

Stratégie de parallélisation

En premier lieu, nous avons pensé à **paralléliser les données** pour résoudre le problème. Par exemple, prenant s1 = "**XHBHDNABCDEF**" et s2 = "**CDEF**", diviser `s1` en deux sous-chaînes s11 = "**XHBHDN**" et s12 = "**ABCDEF**" nécessiterait des matrices distinctes pour chaque sous-chaîne comparée à s2. Cela introduit plusieurs problèmes :

1. **Calculs redondants** : Chaque sous-chaîne aurait besoin de sa propre matrice de calcul, générant des résultats partiels et redondants qui ne reflètent pas l'ensemble de `s1`.
2. **Complexité de synchronisation** : Les sous-chaînes pourraient ignorer des correspondances entre segments, nécessitant une synchronisation complexe pour combiner les résultats.
3. **Augmentation des besoins en mémoire** : La gestion de matrices distinctes pour chaque sous-chaîne alourdit l'utilisation de la mémoire.

Face à ces contraintes, nous avons opté pour une stratégie de **parallélisme de tâches** en utilisant un calcul basé sur les anti-diagonales.

En choisissant cette approche et en utilisant une méthode de programmation dynamique avec une matrice 2D pour calculer la longueur de la plus longue sous-séquence commune (LCS), nous avons observé que pour des chaînes longues (par exemple, pour comparer différentes versions de code source dans Git), le calcul devient intensif. Chaque cellule (i,j) de la matrice dépend de (i−1,j)(i−1,j) imposant des dépendances entre calculs. (car si les caractères ne sont pas égaux, on prend le max de la cellule en haut et celle de gauche)

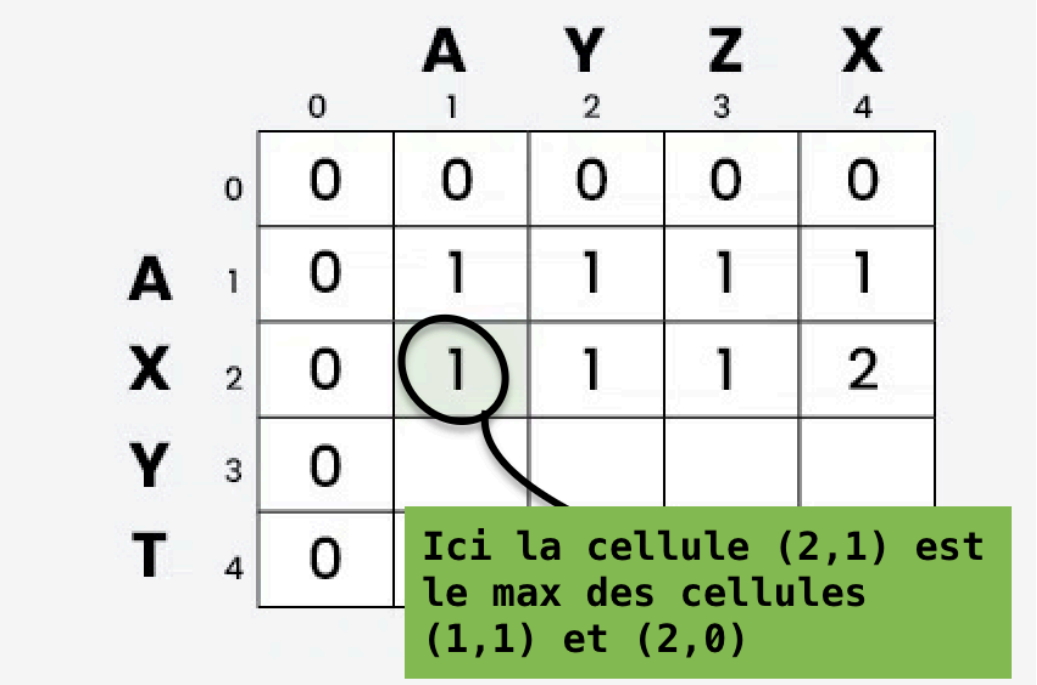


Figure: Explication des dépendances

En parcourant la matrice par **anti-diagonales**, nous pouvons calculer toutes les cellules d'une même diagonale indépendamment, une fois la diagonale précédente terminée. Cette stratégie permet de paralléliser le calcul des cellules au sein de chaque anti-diagonale. **Chaque thread est ainsi associé au calcul de certaines cellules.**

Pour synchroniser les threads, il est crucial de garantir que la diagonale (d−1) soit entièrement calculée avant de passer à la diagonale (d). Nous utilisons des barrières ou des conditions de synchronisation à la fin de chaque diagonale afin que tous les threads puissent progresser ensemble vers la suivante.

Le choix de cette méthode de parallélisation nous permettra d'effectuer des calculs simultanés au sein de la matrice, accélérant ainsi le processus de calcul de la longueur de la plus longue sous-séquence commune, et par conséquent de trouver cette sous-séquence.

Schéma de la parallélisation

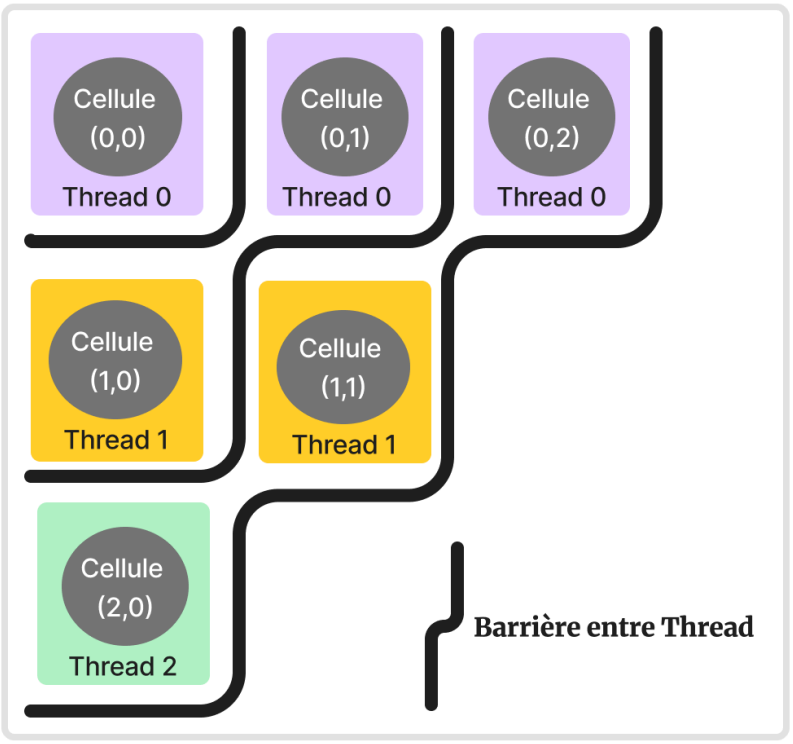


Figure: Représentation du travail partagé entre les différents threads.

Les threads ne s'envoient pas de messages explicites, mais ils partagent une matrice, qui est une variable globale commune. Chaque thread y accède en respectant les barrières de synchronisation.

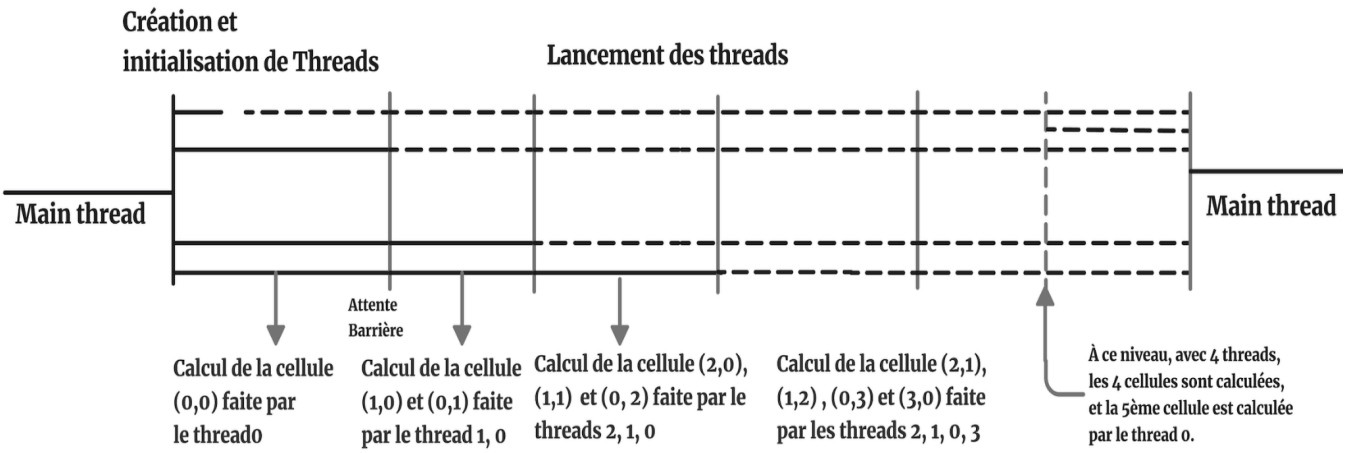


Figure: Schématisation des threads (4 threads)

Mesure des Performances de la Solution Parallèle

Cette figure présente les temps d'exécution (en secondes) pour le calcul de la Longest Common Subsequence (LCS) en mode séquentiel (S) et parallèle (P2, P4, P8, P16) pour différentes tailles de données.

	5	25	150	500	1000	2500	5000	10000	15000	20000
S	0.000001	0.000005	0.000114	0.001161	0.005869	0.036	0.177	0.726	2.045	3.908
P2	0.003290	0.007651	0.001729	0.106933	0.227117	0.735	2.200	6.291	14.569	32.100
P4	0.003978	0.003229	0.070078	0.306204	0.649584	1.641	4.154	10.795	21.212	41.650
P8	0.604000	0.023000	0.127000	0.428000	0.895000	2.199	5.082	12.279	25.035	48.122
P16	0.008103	0.052390	0.164691	0.515706	1.082979	2.628	5.911	14.440	27.942	49.894

Figure: Temps d'exécution séquentiel et parallèle pour différentes tailles de données dans le calcul du LCS (Longest Common Subsequence)

Analyse des comparaisons de performances

Ce graphique montre le temps d'exécution du calcul de LCS en fonction de la taille des données, en comparant une exécution séquentielle (notée S) avec des exécutions parallèles utilisant respectivement 2, 4, 8, et 16 threads. On constate que l'exécution parallèle permet d'accélérer le calcul, et cet effet est particulièrement marqué pour les grandes tailles de données où le travail est réparti efficacement entre les threads. Cependant, lorsque le nombre de threads augmente, les gains en performance diminuent progressivement. Cela est particulièrement visible pour les petites tailles de données, où l'ajout de threads entraîne plus de coûts en gestion et en synchronisation que d'améliorations en vitesse d'exécution. Cette perte d'efficacité est due au fait que chaque thread introduit un **surcoût de gestion** qui devient plus prononcé avec l'augmentation du nombre de threads.

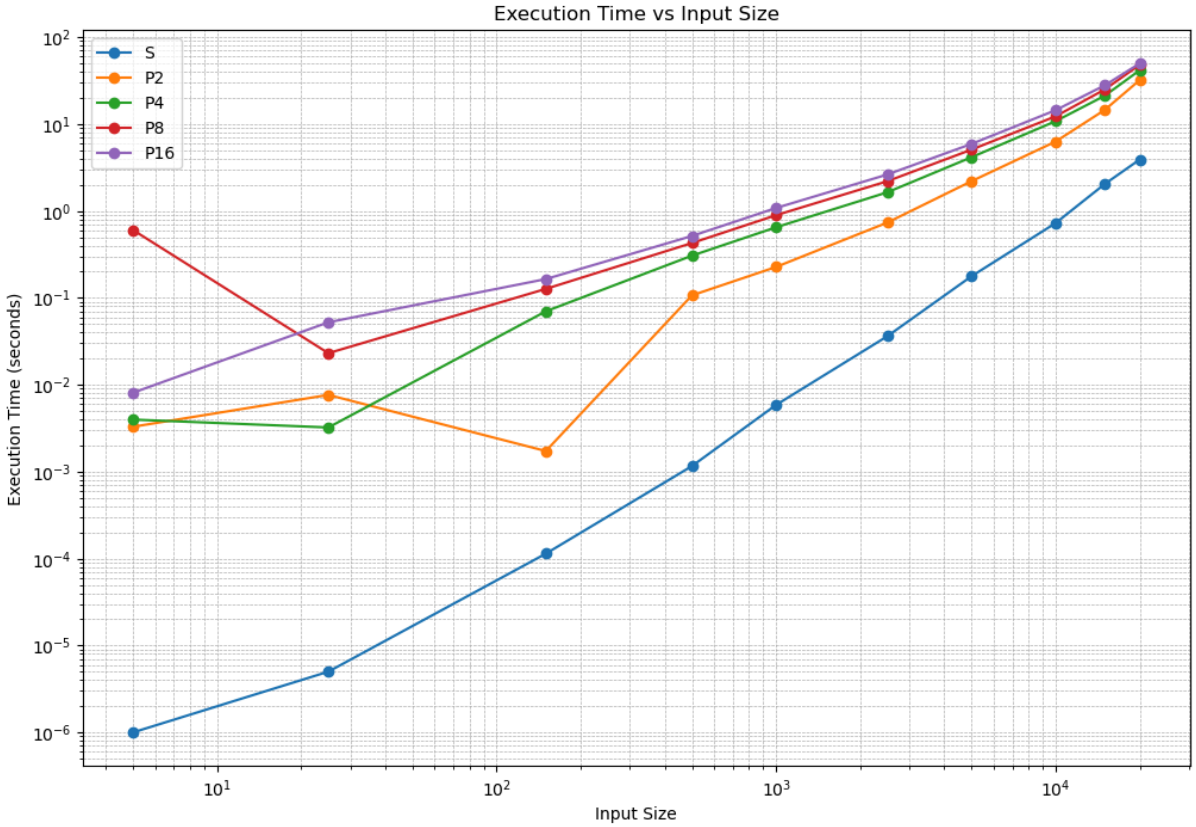


Figure: Temps d'exécution du LCS en séquentiel et parallèle