الجـــمهوريـــة الجزائريـــة الديمـــقراطيـــــة الشـعبيـة
ⵜⴰⵣⴰⵡⴰⵜ ⵜⴰⵎⴰⵔⴰⵖⵉⵜ ⵜⴰⵣⵣⴰⵢⵔⵉⵜ ⵜⴰⵡⵙⵙⵓⵔⵜ
République Algérienne Démocratique et Populaire
وزارة التعليـــم العـــالي والبـــحث العلمـــي
ⵓⵖⵏⵉⵔ ⵏ ⵜⵓⵙⵎⵙⴰ ⵜⵏⵉⵔ ⴰ ⵜⵙⴰⵏⵉ ⵜⵓⵓⵙⵙ
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
ⵓⵖⵓⵎⵓⵔ ⵓⵖⵏⵓⵍ ⵜⵏⵉⵔ ⵏ ⵜⵓⵙⵏⵓⵔⵉⵜ
École nationale Supérieure d'Informatique

2CSSIQ1

Module : High Performance Computing (HPC)

TP 01
Casse de mot de passe SHA-256 par méthode de force brute avec Pthreads

Réalisé par :

- LARDJANE Ikram
- NOUALI Asmaa

Année universitaire 2024/2025

1- Introduction générale :

Nous vivons aujourd'hui dans l'ère numérique, où chaque aspect de notre vie est encapsulé en une multitude de bits dans nos machines. Ce qui certes représente un gain de temps et d'espace sans précédent. Mais qui pose aussi un défi majeur : la sécurité de nos données sensibles et importantes est constamment mise à l'épreuve.
Face à des menaces de plus en plus sophistiquées, comprendre et tester la sécurité des systèmes est devenu crucial. Dans ce contexte, nous proposons dans ce TP une approche de cryptanalyse par force brute pour évaluer la robustesse des mots de passe cryptés en SHA-256 en utilisant la bibliothèque Pthreads pour paralléliser les opérations et améliorer l'efficacité du processus.

2- Présentation de l'algorithme SHA-256 :

Le SHA-256 est une norme de hachage (issue du SHA-2, Secure Hash Algorithm), un standard du gouvernement fédéral des États-Unis permettant de transformer une donnée binaire en une empreinte de 64 caractères hexadécimaux qui la caractérise de manière quasiment unique.

Les étapes essentielles de l'algorithme SHA-256 sont les suivantes :

1) **Prétraitement :**
- Convertir le message d'entrée en une suite de bits.
- Ajouter un bit '1' à la fin de la séquence binaire, suivi de bits '0' pour atteindre une longueur totale congruente à 448 modulo 512.
- Ajouter une représentation binaire de 64 bits de la longueur initiale du message (en bits) pour atteindre une longueur totale de 512 bits.

2) **Initialisation des constantes :**
- Utiliser huit variables initiales de 32 bits, dérivées de la partie décimale des racines carrées des huit premiers nombres premiers.

3) **Découpage en blocs :**
- Diviser le message pré-traité en blocs de 512 bits, chaque bloc étant ensuite divisé en 16 sous-blocs de 32 bits.

4) **Expansion des mots :**
- Étendre les 16 sous-blocs en une séquence de 64 mots (ou sous-blocs) de 32 bits. Cette expansion utilise des opérations de décalage et des XOR pour générer les nouveaux mots.

5) **Processus de hachage :**
- Initialiser les huit variables de travail (A, B, C, D, E, F, G, H) avec les valeurs initiales.
- Pour chaque mot du bloc, appliquer une série d'opérations logiques (rotations, XOR, additions modulo 2^32) et de constantes spécifiques pour transformer les valeurs des variables de travail.

- À la fin de chaque bloc, ajouter les valeurs de travail aux variables initiales pour actualiser les huit valeurs de hachage.

6) Concaténation du résultat :

- Une fois tous les blocs traités, combiner les huit variables de hachage pour obtenir le résultat final, un hash de 256 bits représenté en hexadécimal sur 64 caractères.

⇒ Cet algorithme est couramment utilisé pour sécuriser les mots de passe. Cependant, en cas de mot de passe faible ou de longueur limitée, il est possible de casser le hash par une attaque de force brute, qui consiste à tester toutes les combinaisons possibles pour trouver celle correspondant au hash cible.

3- Solution proposée :

Dans ce TP, nous allons considérer un mot de passe de **5 caractères alphabétiques en minuscules**. Le processus complet de cryptanalyse peut être divisé en 4 étapes, comme indiqué dans le pseudo-code séquentiel ci-dessous :

```
Générer toutes les combinaisons possibles de mots de passe pour une longueur donnée;
Pour chaque combinaison de caractères possible jusqu'à la longueur maximale du mot de passe :
   Calculer le hash SHA-256 de la combinaison actuelle;
   Si le hash calculé correspond au hash cible :
        Afficher la combinaison comme mot de passe trouvé;
        Terminer le programme;
   FinSi
Fin Pour
```

3-1- Pertinence de la parallélisation :

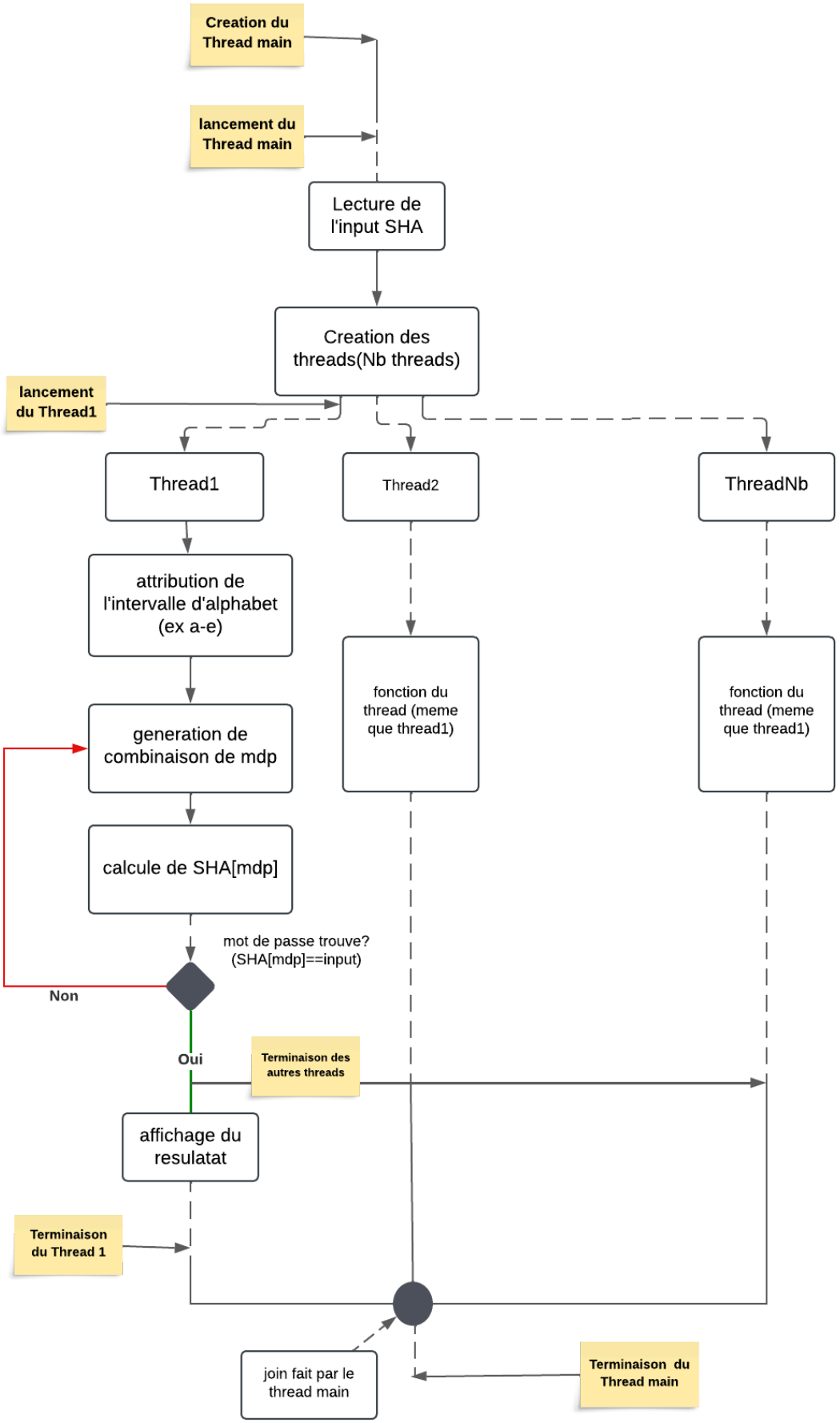
La version séquentielle de la solution est **coûteuse** en temps, ce qui souligne l'importance de paralléliser le programme afin d'accélérer le processus de recherche du mot de passe.
En effet, chaque combinaison de mot de passe peut être testée **indépendamment**. C'est pourquoi en répartissant le travail entre plusieurs threads, nous pouvons réduire considérablement le temps de calcul nécessaire pour explorer toutes les combinaisons possibles.

3-2- Stratégie de parallélisation :

La stratégie de parallélisation utilisée dans ce code repose sur la **distribution des intervalles d'alphabet parmi plusieurs threads**. Chaque thread est responsable de générer et de tester une portion spécifique de l'espace de recherche des mots de passe, calculée comme le nombre de lettres

de l'alphabet divisé par le nombre de threads. En cas de segment d'alphabet restant, celui-ci est attribué dynamiquement au premier thread qui termine son travail.
Les threads continuent à opérer jusqu'à ce qu'un d'entre eux découvre le mot de passe correspondant au hachage cible. À ce moment-là, les autres threads sont immédiatement interrompus, ce qui permet d'éviter des traitements inutiles. Cette approche réduit les temps d'attente et optimise la recherche, car tous les threads collaborent en parallèle, jusqu'à ce que le mot de passe soit trouvé ou que toutes les lettres aient été traitées.

Schéma de la solution parallèle :



3-3- Implémentation avec Pthreads :

Pour implémenter notre solution avec pthreads, plusieurs fonctions ont été utilisées :

- **pthread_create** : Cette fonction est employée pour créer des threads, chacun exécutant une fonction spécifique, appelée la fonction du thread.
- **pthread_join** : Elle est utilisée pour s'assurer que le programme principal attend la terminaison de tous les threads avant de continuer son exécution.
- **pthread_mutex_lock** : Cette fonction verrouille l'accès à la variable partagée current_position, qui est utilisée pour suivre la progression de l'intervalle dans la recherche du mot de passe. Cela permet d'éviter les conflits d'accès lorsque plusieurs threads tentent de modifier cette variable en même temps.
- **pthread_mutex_unlock** : Elle déverrouille l'accès à la variable partagée current_position, permettant ainsi à d'autres threads d'accéder à cette variable en toute sécurité.
- **pthread_mutex_init** et **pthread_mutex_destroy** : Ces fonctions sont respectivement utilisées pour initialiser et détruire le mutex, garantissant ainsi que les ressources nécessaires pour la synchronisation entre les threads sont correctement gérées.

3-4- Analyse des résultats :

Nous avons exécuté le programme pour différents nombres de threads (4, 8 et 16) et avons mesuré les temps d'exécution, que nous comparons ici à l'exécution séquentielle :

Exemple d'exécution:

Exemple 1 :

Input : Hash cible = "e8222f0195a7b1869be1766f7e128fcdb2271aa3a0641b61006f0b8fcf2c60a2"

Expected output: Mot de passe cherché = "tphpc"

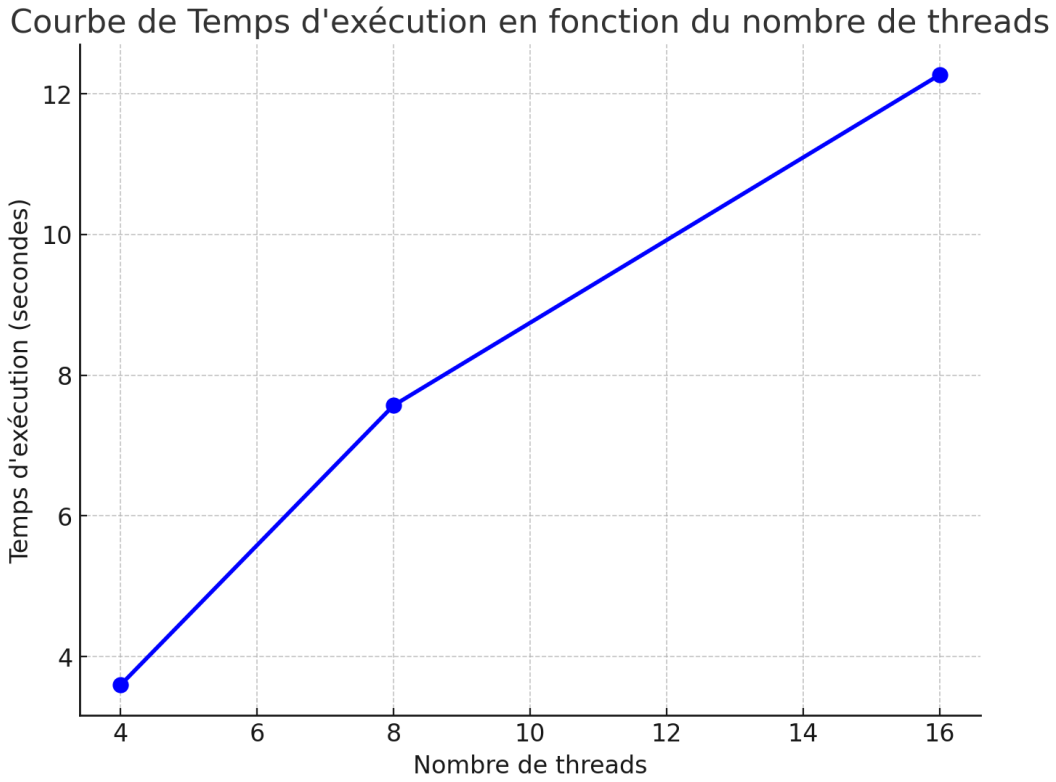
Nombre de threads	Temps d'exécution séquentiel(s)	Temps d'exécution parallèle (s)	Accélération
4 threads	11.13	3,596	3,10
8 threads		7,57	1,47
16 threads		12,27	0,91

Interpretation:

En examinant les performances du programme parallélisé selon différents nombres de threads (4, 8, et 16), nous observons une nette accélération par rapport à l'exécution séquentielle. Cela démontre l'impact de la parallélisation dans le cadre d'un problème hautement concurrentiel comme la casse de mot de passe:

- Avec 4 threads, le temps d'exécution est significativement réduit par rapport au mode séquentiel.
- À 8 threads, nous constatons un gain d'accélération encore plus prononcé. Ce résultat suggère une distribution optimale de la charge de travail pour cette configuration spécifique.
- Cependant, avec 16 threads, nous observons une légère dégradation des performances (temps_sequentiel<temps_parallèle). Cela peut être attribué à une surcharge due à la gestion des threads et à la synchronisation, qui commence à dominer les gains de parallélisation lorsque le nombre de threads devient trop élevé pour la taille de la tâche.

Représentation graphique :



Exemple 2 :

Input : Hash cible = “486ea46224d1bb4fb680f34f7c9ad96a8f24ec88be73ea8e5a6c65260e9cb8a7”

Expected output: Mot de passe cherché = “world”

Nombre de threads	Temps d’exécution séquentiel(s)	Temps d’exécution parallèle (s)	Accélération
4 threads	12,32	10,69	1,15
8 threads		7,87	1,56
16 threads		7,13	1,72

4- Conclusion :

En guise de conclusion, "Diviser pour mieux régner" a été la devise de ce TP. Au premier abord, mener une attaque par force brute sur l’algorithme SHA-256 semblait complexe et lourd. Cependant, en exploitant le potentiel de la parallélisation grâce à Pthreads, nous avons significativement optimisé notre approche et réduit le temps nécessaire pour tester toutes les combinaisons possibles.

Ce TP met en lumière l'importance de la parallélisation dans la cryptanalyse, en particulier pour les attaques de type force brute, où une approche collaborative peut transformer un défi en une tâche réalisable, augmentant ainsi l’efficacité et la rapidité des résultats.