Université Catholique de Louvain

LSINF1252

Systèmes Informatiques

Password Cracker

Groupe 3

22391700 - Augustin d'Oultremont
52141700 - Stanislas Gorremans

2018 - 2019

1 Introduction

Dans le cadre du cours de Systèmes Informatiques (LSINF 1252), il nous a été demandé d'implémenter un programme en C. Ce programme a pour but de cracker des mots de passes protégés par un hash, le `SHA-256`. La partie du travail qui a été demandée est celle de la parallélisation pour rendre un brute force le plus efficace possible avec les processeurs actuellement utilisés (multi-coeurs).

2 Architecture

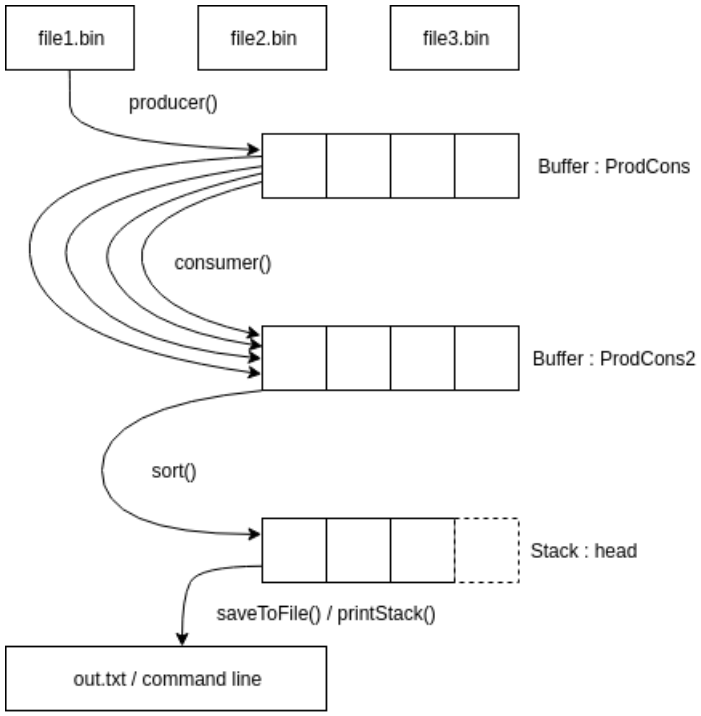

Figure 1: Visualisation de la structure de notre programme.

`producer()`, `consumer()` et `sort()` sont les fonctions utilisées par nos threads. `saveToFile()` ou `printStack()` sont utilisées en fin de programme, lorsque les différents threads ont rempli leurs rôles et que la `stack` ne contient que les meilleurs candidats. On notera qu'il n'y a qu'un thread "`producer()`" et un seul "`sort()`", car la fonction `reverseHash()` étant très lente, un seul thread de lecture et un seul thread de tri suffisent à garder un débit suffisant, et donc le premier buffer rempli et le dernier vide. Une raison supplémentaire pour l'unicité du thread qui effectue le `sort()` est que cette partie ne peut pas être parallélisée.

3 Choix d'implémentation

L'utilisation de la fonction getopt[1], conseillée par les assistants, permet d'améliorer la résistance de notre programme, en permettant l'insertion d'arguments dans un ordre quelconque.

Nous avons également décidé d'utiliser la directive `#define` du préprocesseur pour les longueurs des mots de passe et des hash. Ainsi, le code peut être facilement adapté dans le cas de consignes légèrement différentes sur ces longueurs.

Pour réaliser notre programme, nous avons utilisé un double problème producteur-consommateur. Dans les deux cas, nous utilisons des buffers (tableaux) avec autant de cases que de threads de calcul. Nous avons un thread de lecture qui remplit le premier buffer avec des `hash`, suivi de un ou plusieurs threads de calcul (spécifiés par l'utilisateur grâce à l'argument `[-t NTHREADS]`) qui effectuent l'opération de `reverseHash` par bruteforce sur les `hash` récupérés dans le premier buffer, avant de placer les candidats (résultats de la fonction `reverseHash()`) dans le second buffer. Un dernier thread est alors utilisé pour récupérer ces candidats, déterminer si ils sont meilleurs, moins bons ou égaux aux précédents et gérer la pile des meilleurs candidats. Lorsque le programme se termine et que tous les `hash` ont été traduits, on les imprime dans le terminal ou on les stocke dans un fichier (selon la présence de l'argument `[-o FICHIER_OUT]` dans l'appel à la fonction).

4 Tests

Nous avons implémenté plusieurs types de tests.

Tout d'abord, `strlenVo_test` permet de tester notre fonction `strlenVo()` qui calcule le nombre d'occurrences de voyelles ou consonnes dans un mot donné.

`getSemValue_test` permet de tester la fonction `getSemValue()`, une petite fonction qui renvoie la valeur du sémaphore passé en argument au lieu de la stocker dans le pointeur en argument.

`insertInBuffer_test` et `removeFromBuffer_test` permettent de vérifier qu'il n'y a pas de problème dans l'insertion et la récupération des valeurs dans les / des différents buffers.

5 Évaluation quantitative

Pour évaluer la vitesse de notre code, ainsi que sa parallélisation, nous l'avons lancé sur `Jabba` (qui compte 40 coeurs). Nous avons mesuré le temps d'exécution du programme, pour un nombre de threads variant de 1 à 40 pour ensuite comparer ces résultats à la loi de Amdahl (via un programme python inspiré par Antoine Gennart [2])

[1] `http://man7.org/linux/man-pages/man3/getopt.3.html`
[2] `https://github.com/gennartan/LSINF1252_project/blob/after_feedback/report/amdahl.py`

On peut remarquer que le graphe suit bien la courbe de Amdahl pour un code parallélisable à 99%, jusqu'à environ 20 threads, où les deux courbes s'éloignent. Nous supposons que c'est à ce nombre de threads que le temps de création du thread commence à avoir un impact non négligeable sur la vitesse du programme (ce temps de création n'est pas pris en compte pour la courbe de Amdahl).

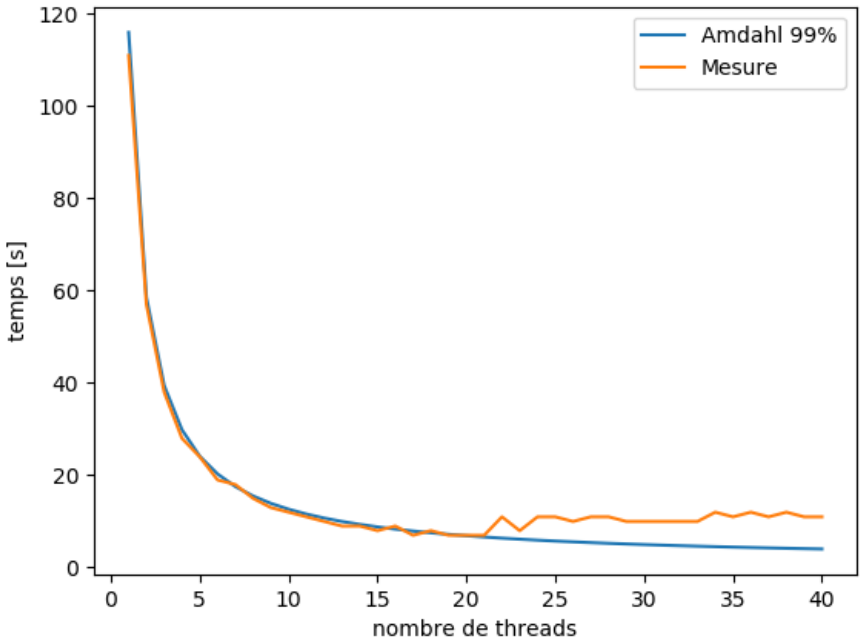

Figure 2: Comparaison entre la vitesse de notre code et la loi de Amdahl