LSINF 1252 : Système informatique
Projet : password cracker

Arnaud Lahousse 5629-15-00
Jeanne Malcourant 5698-16-00

1 Introduction

L'objectif de ce projet était de déhasher un hash de 32 bytes préalablement crypté en SHA-256. Pour ce faire nous devons optimiser nos calculs sur le processeur afin de prendre le moins de temps possible et de rentre notre programme le plus efficace possible. Cette optimisation des calculs va être possible grâce au parraléllisme des calculs que nous allons mettre en place grâce à des threads.

2 Architecture

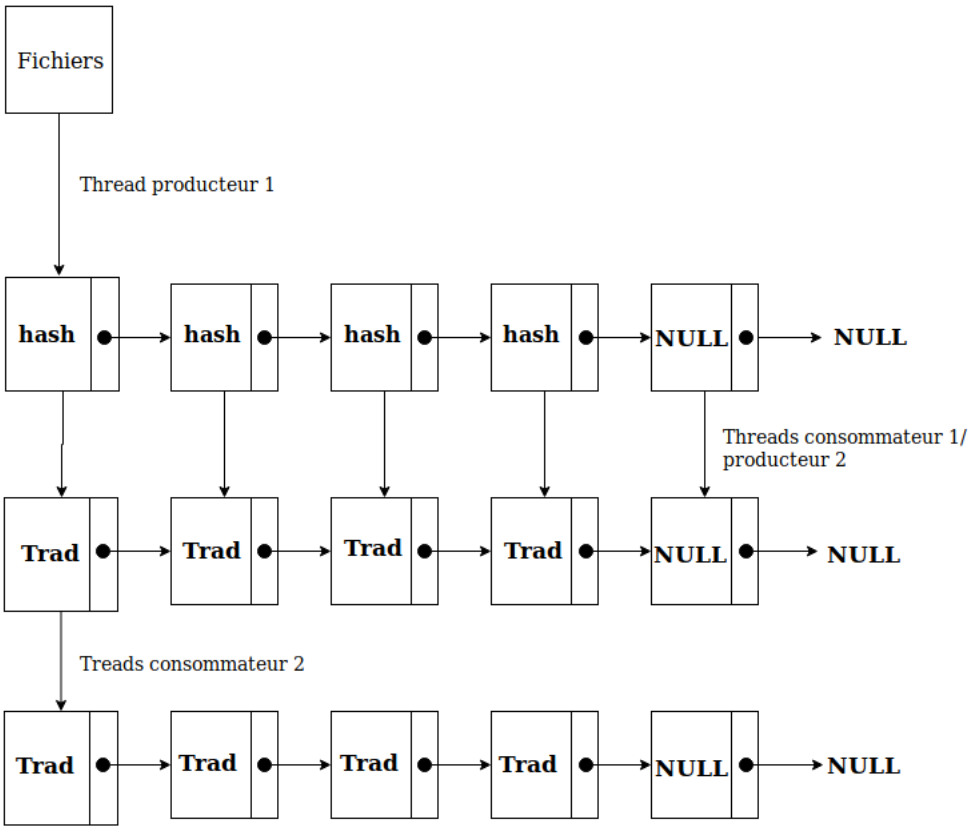
Nous avons élaboré l'architecture autours de trois buffers différents. Le premier stockera les hash, le deuxième,lui,stockera les traductions, et enfin le dernier stockera les traductions possédant le plus de consonnes ou de voyelles. Ces buffers ont la forme d'arraylists de taille vingt. Précisons que le dernier noeud contient une valeur NULL mais également que l'élément suivant pointe vers NULL. Nous avons choisi d'implémenter vingt sections; en effet, il s'agit d'une part d'un multiple de quatre (nous savons que notre code va être tester sur les quadcores ce qui va le rendre plus performant). Et d'une autre part car au-dela de vingt la différence de temps d'execution devient minime. Il est important de noté que nous avons fait deux arraylists différentes, un contenant comme valeur un hash et l'autre ccontenant une liste de character.

Pour remplir les différents buffers nous nous sommes basés sur le principe de producteur-consommateur. Notre premier producteur placera les hashs dans la première arraylist. Il sera initialisé à un seul thread, en effet cette opération demande très peu de calculs si on la compare à reversehash.

Notre premier consommateur joue lui un double rôle, à savoir celui de consommateur mais également le rôle de producteur pour le deuxième buffers. Il s'agira ici de traduire les hashs en chaines de caractères et les placer dans le buffer 2, d'ou l'appellation producteur. Ce consommateur/producteur tournera sur nthreads (fournis dans les arguments du programme) étant donné que cette fonction demande un certain temps de calculs, il est impératif de la parralliser.

Pour terminer, nous avons un second consommateur jouant lui, le rôle de producteur comme le précédent. Il remplit dans ce cas-ci notre troisième buffer avec les "traductions" des hashs en fonction de leurs nombres de voyelles.

Voici un shéma récapitulatif qui pourra éclaircir au mieux votre compréhension de notre architecture:



3 Choix de conception

Dans notre fonction consumer ainsi que consumer 2, nous avons un enchevêtrement de 2 whiles, celles-ci permettent aux deux consommateurs de ne pas finir le thread avant même que les producteurs aient eu le temps de mettre des éléments de le buffer d'où ce semblant de boucle infinie.

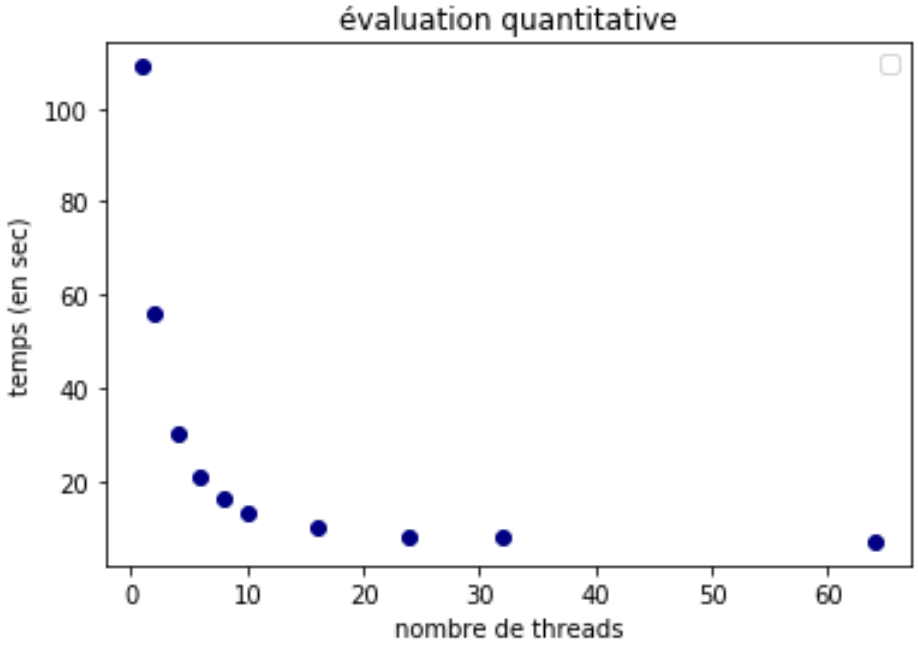
4 Strategie des tests

Afin de pouvoir gérer nos erreurs plus facilement, notre programme test a été fait en paralléle avec le "vrai" programme. Pour ce faire, nous avons commencé par tester toutes nos fonctions utilisées dans la main mais déclarées à un autre endroit :

- testInitialisation1 : cette fonction vérifiera que initialisation() ajoute un élément contenant un hash NULL dans le premier type d'arraylist.
- testInitialisation2 : cette fonction vérifiera que initialisation2() ajoute un élément contenant un hash NULL dans le deuxième type d'arraylist.
- testInsertion1 : cette fonction va vérifier que insertion() insert bien le hash dans l'arraylist
- testInsertion2 : cette fonction va vérifier que insertion2() insert bien la trad dans l'arraylist
- testSuppression1 : cette fonction va vérifier que suppression() supprime bien le premier élément de la list
- testSuppression2 : cette fonction va vérifier que suppression2() supprime bien le première élément dans l'arraylist
- consonne : vérifie que la fonction consonne renvoie bien le nombre de consonnes présent dans la chaine de caracter.
- voyelle : vérifie que la fonction voyelle renvoie bien le nombre de voyelles présent dans la chaine de caracter.

5 Evalutation quantitative

la première évalution que nous avons faites,c'est d'analyser les différents temps d'execution en fonction du nombre de thread. Pour ce faire, nous avons fais tourner notre programme sur "Jaba" avec un fichier d'input de 1000hash et voici un graphique de nos résultats :



On remarque clairement que c'est une courbe de type 1/x, cela prouve bien que d'une part que notre parallélisme fonctionne bien, et plus on augmente les threads plus notre programme s'execute rapidement. Lors de l'execution de valgrind on remmarque qu'il y a quand même une perte de mémoire à la fin de l'execution du programme. On n'a malheureusement pas pu trouver d'où venait cette fuite.

6 Conclusion

Au terme de ce projet, nous pouvons affirmer que notre programme et le parrallélisme des calculs fonctionne comme l'a prouvé la section précedente. Malgré tout, avec un peu plus de temps et de réflexion, je pense qu'il y aurait moyen d'optimiser ce programme encore plus, malgré son rendemement déjà impressionnant.