Aurélien Duval
Valentin Fries
Maxime Pelte

5ESGI 2017-2018
Architecture Logicielle
Programmation Parallèle

Shacker

SHA-256 bruteforce

Octobre 2017

Outils

- Valgrind / Callgrind
- Memcheck
- gprof : ajouter le flag *-pg* lors de la compilation puis exécuter la commande : gprof ./a

Roadmap

- Implémentation du SHA-256 (utilisation de bibliothèques)
- Implémentation du bruteforce séquentiel
- Parallélisation du calcul des hash
- Création d'un pool de threads (quantité paramétrable) au démarrage de l'application pour calculer en parallèle plusieurs tailles de clé

Actuellement les 2 premières étapes sont effectuées, et nous travaillons à la parallélisation des traitements. C'est justement sur ce point que nous rencontrons certains problèmes (voir **Benchmarks**).

Benchmarks

Pour une phrase de 4 caractères, le hash est trouvé en approximativement **0.8 secondes** de manière séquentielle.

Nous avons essayé d'implémenter un algorithme parallélisé sur 4 threads (nombre de coeurs physiques de nos machines), mais à notre étonnement le temps d'exécution a grimpé à environ **2.4 secondes**.

Il est possible que ce ralentissement soit dû à notre gestion des mutex : en effet, nous verrouillons l'écriture dans la queue de résultats, ce qui peut provoquer un goulot d'étranglement si certains threads sont plus rapides que d'autres, provoquant un dead lock pour ces derniers.

Nous avons envisagé l'utilisation de la bibliothèque OpenMP, qui permettait d'obtenir des performances intéressantes mais en abstrayant toute l'implémentation des threads ; nous ne l'utiliserons donc pas pour ce projet.

La génération du hash SHA-256 peut aussi être optimisée. Nous utilisons pour cela la bibliothèque OpenSSL, qui initialise un contexte à chaque génération de hash. Nous pensons donc effectuer un PoC visant à partager un contexte unique entre tous nos threads pour voir si cela impacte favorablement les performances, ou si au contraire cela interfère avec le fonctionnement d'OpenSSL.

Décembre 2017

Livrable attendu

projet.zip
|-- src/
| |-- CMakeList.txt
| |-- main.cpp
| \-- …
|-- lib/
| \-- ...
\-- doc/
  \-- Shacker.pdf

Procédure :

- unzip projet.zip
- cd projet
- mkdir build
- cd build
- cmake ../src
- make
- ./main $NB_THREADS $HASH

Sortie attendue : **uniquement** le mot de passe en clair.

Stratégie utilisée pour la parallélisation

Pour un alphabet "*abcdefghijklmnopqrstuvwxyz*" et 4 threads configurés au démarrage du programme, la stratégie de parallélisation est la suivante :

- On initialise un tableau d'entiers en guise d'incrémenteur :
  - Ce tableau représente le nombre de threads traduit dans la base correspondant au nombre de lettres dans le dictionnaire (ici 26 car le dictionnaire est composé de 26 lettres) ;
- On initialise le buffer contenant le mot de passe à tester avec la valeur correspondant à son identifiant de thread traduit dans la base correspondant au nombre de lettres dans le dictionnaire ;
- Tant que le mot de passe n'a pas été trouvé :
  - On teste le mot de passe contenu dans le buffer
    - Optimisation : le comparateur utilise un pointeur *long long* pour comparer directement par blocs de 64 bits, représentant 8 fois moins de tours de boucles par rapport à 1 char et évitant la traduction en string de la sortie du sha256 (*unsigned char**) ;

- On applique l'incrémenteur sur le buffer par le biais d'une addition élément à élément.

Calcul du speedup théorique

Nombre de coeurs	**Temps en secondes** *(pour "AAAAA")*	**Speedup**	**Speedup théorique** *(97% du code estimé)*
1	97.1967	1	1
2	50.7891	1.91	1.94
4	26.5705	3.66	3.67
8	27.4383	3.54	6.61

Réalisé sur un processeur Intel Core I7-6700HQ @ 2.60 GHz (4 coeurs physiques, 8 threads)

Sources

- https://www.pugetsystems.com/labs/articles/Estimating-CPU-Performance-using-Amdahls-Law-619/
- https://www.vcalc.com/wiki/vCalc/Amdahl%27s+Law

Difficultés rencontrés

- [Résolu] Beaucoup de doublons rencontrés au début du développement (passages multiples de l'algorithme de bruteforce sur la même possibilité) ;
- [Résolu] Pour les mots de passe courts (1-2 caractères), le premier threads trouvait le résultat avant l'initialisation de tous les threads ce qui provoquait un deadlock ;
- [Résolu] Parallélisation suivant le modèle producteur / consommateur n'offrant aucun gain de performances ;
- [Résolu] Parallélisation par bandes ayant des erreurs d'arrondis lors de l'exécution de divisions ;
- [Résolu] Parallélisation par sauts ayant des problèmes de dispersion si le nombre de threads dépasse la taille du dictionnaire.

Explication des flags de compilation CMake

Trois profils de compilation sont définis dans le fichier **CMakeLists.txt** :

- Profil par défaut : `set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fdiagnostics-color=auto -fmax-errors=2 -std=c++11") # Ce profil est automatiquement propagé aux deux autres profils de compilation, qui partageront alors ses arguments`

- Profil de *debug* : `set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -Wall -g -DDEBUG -O0 -no-pie -pg")`
- Profil de *release* : `set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -DRELEASE -O3")`

Les différents flags utilisés sont les suivants :

- `-g` : active les messages de debug
- `-Wall` : active tous les messages de warning disponibles
- `-fdiagnostics-color=auto` : active la colorisation de la sortie console de `g++`.
- `-fmax-errors=2` : permet de définir le nombre d'erreurs après lequel le compilateur doit arrêter ses traitements.
- `-std=c++11` : défini le standard auquel le code source doit se conformer (ici, on définit le standard `C++11`).
- `-DDEBUG` : définit la macro `DEBUG` avec la valeur 1 (utilisée dans le code source).
- `-DRELEASE` : définit la macro `RELEASE` avec la valeur 1 (utilisée dans le code source).
- `-no-pie` : désactivation du mode *Position Independent Executable*.
- `-pg` : génération d'informations pour le profilage par `gprof`.
- `-O0` : pas d'optimisation du code source, permettant de debugger le programme.
- `-O3` : optimisation de la taille du code et du temps d'exécution (au détriment du temps de compilation). Activation des flags de compilation suivants :
  `-fauto-inc-dec -fcprop-registers -fdce -fdefer-pop -fdelayed-branch -fdse -fguess-branch-probability -fif-conversion2 -fif-conversion -finline-small-functions -fipa-pure-const -fipa-reference -fmerge-constants -fsplit-wide-types -ftree-builtin-call-dce -ftree-ccp -ftree-ch -ftree-copyrename -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-fre -ftree-sra -ftree-ter -funit-at-a-time -fthread-jumps -falign-functions -falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fexpensive-optimizations -fgcse -fgcse-lm -findirect-inlining -foptimize-sibling-calls -fpeephole2 -fregmove -freorder-blocks -freorder-functions -frerun-cse-after-loop -fsched-interblock -fsched-spec -fschedule-insns -fschedule-insns2 -fstrict-aliasing -fstrict-overflow -ftree-switch-conversion -ftree-pre -ftree-vrp -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-vectorize and -fipa-cp-clone`