Compte Rendu de Projet de recherche de L3

N°2 : Récapitulatif travail réalisé & explications du raisonnement.

Florian REYNIER & Mathis CARISTAN

11/02/2016

1 Résumé des idées de la réunion

Suite à cette réunion, il y a eu peu de changements sur les objectifs précédemment définis, nous continuons donc sur le chemin que nous avions pris. Notre principale tâche est de créer un histogramme des performances des fonctions implantées, puis, de commencer à implanter de nouvelles fonctions.

2 Explications concernant la fonction xxHash

La première tâche que nous avons entrepris a été de nous documenter sur la fonction xxHash. Pour cela, la meilleure source d'informations que nous avons trouvé est un projet github implémentant la fonction en C [2]. Le projet est bien documenté, permettant une bonne compréhension.
Un premier problème de taille nous est alors apparu, car xxHash est une fonction **extrêmement** optimisée ([3] indique une vitesse 2 à 10 fois plus rapide que la plupart des meilleurs fonctions). Ré-écrire les quelques 1700 lignes de code, et adapter le travail d'optimisation pour l'utiliser en OCaml ne semble *a priori* pas un travail réalisable dans le cadre de ce projet, en particulier en considérant que notre niveau en OCaml est probablement loin d'être suffisant.
En conséquence, la meilleure solution nous a semblé de récupérer la fonction C utilisée, et de l'appeler dans une fonction OCaml grâce à la fonction "`external`" d'OCaml.
Cependant, pour pouvoir l'utiliser, nous avons tout de même du apporter quelques modifications au fichier C, notamment sur le type de retour. Il est possible de transmettre des paramètres à la fonction C, car grâce à l'utilisation de `external`, OCaml transmet à la fonction C un pointeur sur un tableau d'arguments.
En conclusion, nous pouvons utiliser la fonction xxHash de manière optimale sans que le travail ait été hors de portée.

3 Explications concernant notre fonction de hachage

La création de la fonction de hachage a été réalisée en s'aidant des remarques du livre *Types de données et algorithmes* [1].
La construction de la fonction de hachage a été faite en plusieurs étapes. Nous avons dans un premier temps décidé d'associer à chaque caractère c, une valeur $v_{|10}$. Puis, cette valeur $v_{|10}$ a été convertie en binaire pour obtenir $v_{|2}$. Un mot m était donc représenté par une suite S d'élément binaires. Pour contracter cette suite S, nous avons choisi de la découper en "sous-suites" s_i de longeur l, et d'appliquer une opération binaire o entres les s_i, pour obtenir une unique suite s_f de taille l. Cette suite était alors reconvertie en décimale pour obtenir la valeur e. Enfin, la dernière étape a consistée à appliquer la fonction f_{mult} à e (voir équation 1), afin d'obtenir $f_{mult}(e) = h$, la valeur de hachage du mot.

$$f_{mult}(e) = \lfloor ((e * \theta) \mod 1) * T \rfloor \tag{1}$$

3.1 Associer une valeur $v_{|10}$ à un caractère

Comme expliqué plus haut, la première étape a été d'associé une valeur à chaque caractère. Pour cela, nous avions plusieurs options, la première à laquelle nous avons pensé, était d'utiliser la table ASCII. Cependant,

cette méthode présente plusieurs défauts. Tout d'abord, la taille de la table ASCII est trop grande pour ce dont nous avons besoin ici, en effet, nous ne considérons que les lettres (un mot commme "`(texte)`" ne doit pas être hashé en prenant en compte les parenthèses). De plus, la table ASCII fait la différence entre les majuscules et les minuscules, ce que nous ne cherchont pas non plus. Enfin, le dernier problème est dû à la représentation des lettres accentuées en OCaml. En effet, celles-ci ne sont pas considérées comme un unique caractère, mais comme une chaîne de deux caractères spéciaux : `é` $=$ "$\backslash 195 \backslash 169$". Du coup, nous avons plutôt choisi d'associer à chaque lettre, sa place dans l'alphabet. De plus, en considérant les lettres accentuées comme non accentuées (un `é` est équivalent à un `e`), cela nous permet de résoudre le problème de l'accentuation en OCaml. Cette méthode a également l'avantage de nous laisser libre de rajouter aisément des caractères si nous le souhaitons (notamment les lettres accentuées).

3.2 Choix de l'opération primaire de la fonction de hachage

Dans [1], les opérations primaires présentées sont **l'extraction**(on ne considère arbirtrairement que certains bits d'une représention binaire d'un mot), **la compression** (voir 3.3), **la division** (reste d'une division entière d'une représentation décimale d'un mot) et **la multiplication** (modulo 1 de la multiplication d'une représentation décimale d'un mot par un paramètre). Les première opération ne donne pas de bons résultats en terme de hachage, tandis que la seconde est généralement surtout utilisée pour réduire la taille d'une chaîne de bits. Les deux dernières semblent être des bonnes opérations primaires, et nous avons choisi de retenir la multiplication pour notre fonction de hachage. Le paramètre multiplicatif θ a été choisi parmi deux valeurs qui donnent théoriquement la meilleure uniformité de distribution de clefs ([1]).

3.3 Compression

Lorsque nous avons cherché quelle valeur e associer à un mot pour la multiplication, nous avons envisagé plusieurs possibilités. La simple addition, ou multiplication, des valeurs $v_{|10}$ des lettres composant un mot peut créer trop facilement des collisions dans la table de hachage. Nous avons donc choisi à la place de représenter un mot par la "juxtaposition des valeurs de ses caractères" ("`ABCDE`" $= 12345$). Cependant, cette méthode peut facilement créer de très grandes valeurs, voire dépasser la capacité d'un INT dans certains cas. Pour palier à ce problème, nous avons choisi de convertir les valeurs $v_{|10}$ des caracères en binaire (sur 5bits), et de représenter un mot comme un tableau de bits ("`A B C D E`" $= 1 - 2 - 3 - 4 - 5 = 00001_{|2} - 00010_{|2} - 00011_{|2} - 00100_{|2} - 00101_{|2}$, donc "`ABCDE`" $= 0000100010000110010000101$). Cette nouvelle représentation assure de manière unique la représentation de chaque mot, cependant, la taille de cette ensemble est trop grande (2^{5*n} pour un mot de n lettres). Pour une table de hachage de taille 2^{16}, il n'est pas intéressant d'avoir plus de 16bits. Ainsi, nous avons cherché à compresser cette longue chaîne de bits à une plus courte. Pour cela, nous avons découpé la chaîne S d'un mot, en s_i sous chaînes de 16bits. Ensuite, en appliquant une opération binaire entre ces sous-chaînes, nous obtenons une unique chaîne finale s_f. Les opérations **AND** et **OR** ont la mauvaise propriété d'entraîner des accumulations en début et en fin de tableau respectivement, tandis que l'opération **XOR** (ou exclusif) semble être plus équitable. Une fois que nous avons obtenu cette chaîne de 16 bits, nous pouvons la reconcertir en décimal pour obtenir la valeur e du mot avec lequel nous travaillons à l'étape de multiplication.

NB : C'est lors de l'application de l'opération **XOR** que sont crées les collisions dans cette méthode. En effet, un mot correspond à une unique chaîne S et réciproquement, mais ce n'est plus nécessairement le cas après l'opération.

Références

[1] Marie-Claude Gaudel, Michèle Sorian Christine Froidevaux, *Types de données et algorithmes*. Ediscience internationnal, Paris, 1993.

[2] Yann *Cyan4973* Collet, *xxHash GitHub project page*. Website : `https://github.com/Cyan4973/xxHash`, visité le 09/02/16.

[3] Yann *Cyan4973* Collet, *xxHash - Extremely fast non-cryptographic hash algorithm*. Website : `http://www.xxhash.com/`, visité le 09/02/16.