Projet 3 - Premier rapport de performances

Cédric Kheirallah
École polytechnique de Louvain
UCLouvain
Louvain-la-Neuve, Belgique
cedric.kheirallah@student.uclouvain.be

Jacques Hogge
École polytechnique de Louvain
UCLouvain
Louvain-la-Neuve, Belgique
jacques.hogge@student.uclouvain.be

Pierre Marcipont
École polytechnique de Louvain
UCLouvain
Louvain-la-Neuve, Belgique
pierre.marcipont@student.uclouvain.be

Romain Barbason
École polytechnique de Louvain
UCLouvain
Louvain-la-Neuve, Belgique
romain.barbason@student.uclouvain.be

***Abstract*—Ce rapport est une première analyse des performances de notre programme C pour le Projet 3 en informatique de 2ème année de bachelier à l'École Polytechnique de Louvain.**

***Index Terms*—C, threads, FEC, packet loss, bytes**

I. Introduction

L'objectif de ce projet est de réimplémenter un code Python de Forward Erasure Correction (FEC) en C et d'optimiser l'utilisation de la mémoire et le temps de compilation d'une part en utilisant des threads - ce dont n'est pas capable le langage Python - et d'autre part d'améliorer les algorithmes utilisés lorsque cela est possible.
Le projet a nécessité l'utilisation de plusieurs outils tels que CUnit pour tester une à une les fonctions du programme, Valgrind afin de détecter les memory leaks et finalement Git afin de permettre à notre groupe de travailler séparément sur différentes tâches et stocker le code sur un repository Gitlab.

II. Fonctionnement du programme

Le programme de Forward Erasure Correction est un programme capable de réparer la perte de paquets de données (packet loss) dans les transmissions réseaux.
Le programme prends en entrée les fichiers binaires "endommagés" et lit les 24 premiers bytes afin d'avoir les données nécessaires à la réparation du fichiers.
Ces données sont :

- **seed** : nombre utilisé pour générer des nombres aléatoires nécessaires au calcul des coefficients de redondance, encodé sur un entier non-signé de 4 bytes (32 bits)
- **block_size** : taille d'un bloc de symboles sources qui est protégé, encodée sur un entier non-signé de 4 bytes (32 bits)
- **word_size** : taille d'un mot, encodée sur un entier non-signé de 4 bytes (32 bits)
- **redundancy** : nombre de coefficients de redondance qui protègent chaque bloc, encodé sur un entier non-signé de 4 bytes (32 bits)
- **message_size** : taille du message initial à récupérer (sans compter les 24 bytes initiaux), encodé sur un entier non-signé de 8 bytes (64 bits)

Connaissant là la taille du fichier, le programme peut trouver le nombres de blocs complets que contient le fichier. Si le dernier bloc a une taille insuffisante et est un bloc incomplet, il sera corrigé séparément.
Pour chaque bloc, on calcule les valeurs manquantes et on le retraduit en string que l'on peut injecter dans un fichier en output.
Enfin, le programme ferme le fichier binaire.

III. Structure du programme

A. main.c

Rassemble toutes les fonctions afin de les utiliser dans la correction des fichiers. On peut y lire toutes les étapes du programme dans sa méthode main().

B. src/

Dossier contenant les fichiers C utilisés par le main.c :

- **tinymt32.c** : permet de générer des nombres pseudo-aléatoires lorsqu'on lui fournit un seed (fourni de base pour notre projet)
- **system.c** : contient les méthodes liées aux calculs dans les Galois Field qu'utilise le programme main.c ainsi la méthode génératrice de coefficients

C. headers/

Dossier contenant les fichiers headers qui sont les déclarations des méthodes écrites dans leurs contreparties ".c"

D. input_binary/

Dossier contenant les fichiers binaires que le programme prend en input

E. input_txt/

Dossier contenant les fichiers textes complets initiaux (à comparer avec les sorties obtenues)

F. tests/

Dossier contenant les tests CUnit pour toutes les méthodes utilisées dans le programme

IV. Première analyse des performances

Nous comparons ici les performances de notre programme séquentielle en C avec le code Python fourni initialement.
Voici notre premier test de performances faisant la moyenne d'une vingtaine de temps d'éxécution pour chaque programme.
Matériel utilisé : Ryzen 5 4600U
OS : Ubuntu 20.04.4 LTS
Résultats :

- Python : en moyenne 1.47261415 secondes d'exécution
- C unithreadé : en moyenne 0.00826245 secondes d'exécution

Soit un temps d'exécution environ 178 fois plus rapide que le programme Python.
N'ayant pas eu le temps d'avoir un code multi-threadé fonctionnelle, nous ne pouvons pas encore faire d'analyse de performance sur plus d'un thread.

V. Conclusion

Malgré notre version multi-threadé toujours en cours de production, nous avons produit un code séquentiel 100% fonctionnelle et rapide avec succès. Les performances sont significativement améliorées, le code efficace est clairement documenté.
Enfin nous nous sommes suffisamment familiarisés avec le langage C et les outils utilisés dans ce projet pour pouvoir rapidement terminer la partie multithreadé dans les semaines qui viennent.