LEPL1503 - Rapport final

Groupe A-1
Nicolas Bourez
nicolas.bourez@student.uclouvain.be 28462000
Simon De Vadder
simon.devadder@student.uclouvain.be 14412000
Tim Defranne
tim.defranne@student.uclouvain.be 18872000
Benoit Lhernoud
benoit.lhernoud@student.uclouvain.be 77102000
Aurélien Livin
aurelien.livin@student.uclouvain.be 38772000
Rémi Stoelzaet
remi.stoelzaet@student.uclouvain.be 15232000

I. Introduction

Après la découverte du C et ses multiples facettes, nous avons reçu pour tâche de traduire en C un programme Python fonctionnel résolvant le problème de l'utilisation des codes correcteurs d'erreurs. Le code python étant une version séquentielle, la première étape du projet fut de l'implémenter en langage C. La seconde étape, la plus critique, fut de construire une version multithreadée de notre code. Enfin, il était demandé de pouvoir faire tourner le programme sur un Raspberry Pi et d'exploiter au mieux ses 4 coeurs.

Dans ce rapport, le lecteur découvrira la structure générale du projet. Ensuite, il aura l'occasion de comprendre plus en détail l'algorithme utilisé, les différences de notre code C avec la version python proposée, les différents tests ainsi que la partie parallélisée de notre projet. Nous finirons par une analyse des performances.

II. Structure générale du programme

Le projet est segmenté en plusieurs répertoires :

- `src`
  L'ensemble des fichiers .c nécessaires à l'exécution du programme principal.
- `headers`
  L'ensemble des structures et descriptions des fonctions utiles à notre programme.
- `test_big`
  Contient deux gros fichiers .bin (5MB) afin de tester notre version multithreadée.
- `test_basic`
  Le répertoire basique donné par nos professeurs via le programme python.
- `tests`
  L'ensemble des fichiers contenant des test unitaires. Ils nous permettent de vérifier le bon fonctionnement de nos fonctions et utilisent la librairie CUnit.

Il contient également des fichiers principaux comme :

- `main.c`
  Contient la fonction `main` permettant de faire tourner l'ensemble du programme.
- `README.md`
  Ce fichier contient toutes les informations nécessaires à la bonne exécution de notre programme.
- `output.txt`
  Le fichier dans lequel le code stocke la sortie de `main.c`.
- `Makefile`
  Ce fichier permet de simplifier des commandes souvent utilisées. Typiquement, il permet de compiler et de créer un exécutable de la fonction `main.c`, de compiler et d'exécuter nos tests, d'utiliser l'outil valgrind afin de détecter les éventuelles fuites de mémoire ou encore de supprimer les exécutables via la commande `make clean`.

III. Description du programme C

L'algorithme implémenté en C à pour but de récupérer les symboles sources perdus lors de la transmission d'un fichier binaire. Il résout cette tâche à l'aide d'un code correcteur d'erreurs utilisant la méthode RLC. La sortie est un fichier binaire contenant pour chaque fichier traité sa taille (en bytes), la taille de son message (en bytes), son nom et son contenu. Notre programme est constitué d'un fichier principal, `main.c`, utilisant plusieurs autres sous-fonctions réparties dans `block.c`, `get_file_infos.c` et `system.c`.

4 différents arguments optionnels peuvent être donnés à l'exécutable de la fonction `main.c` :

- -f
  Correspond au fichier .txt dans lequel nous allons stocker notre sortie.
- -n
  Correspond au nombre de threads (4 maximum pour le Raspberry Pi).
- -v
  Active les messages de débugage.

A. block.c :

Ce fichier contient l'intégralité des fonctions utiles aux blocs. La fonction `make_linear_system(2)` se charge de créer un système linéaire Ax = B qui possède autant d'équations que de symboles de redondances. Ce système linéaire est ensuite résolu à l'aide de la fonction `process_block` qui permettra de retrouver les symboles sources perdus.
Une fois ces opérations effectuées et les symboles retrouvés, le programme les réécrit dans notre fichier de sortie output.txt à l'aide des fonctions `write_block(3)` et `write_last_block(4)`. Cette dernière se différencie de `write_block(3)` puisqu'elle doit gérer le cas du dernier symbole du dernier bloc qui n'est pas forcément de taille identique aux autres.

B. get_files_infos.c :

Ce fichier contient la fonction `get_file_infos.c` qui s'occupe de traiter les 24 premiers caractères binaires de nos fichiers d'entrée et de les placer dans une structure `infos_t`. En effet, ils commencent toujours avec différentes informations comme la seed[1], la taille d'un bloc de symboles sources, la taille d'un mot, le nombre de coefficients de redondance et la taille d'un message initial à récupérer.

C. system.c :

Les fonctions se trouvant dans ce fichier sont les fonctions calculatoires de notre programme. Ces calculs sont opérés dans un Corps Fini (ou Galois Field). Un Galois Field (GF) contient un nombre fini d'éléments et redéfinit les opérations courantes comme l'addition ou la multiplication entre deux nombres appartenant au GF. Etant donné que nous travaillons en byte et que nous voulons que nos opérations retournent un entier dans $[0, 2^8 - 1]$ sans aucune perte de précision, nous utilisons un GF(2^8). Dans system.c, se trouve donc `gf_256_full_add_vector(3)` qui permet d'additionner deux vecteurs, `gf_256_mul_vector(3)` qui permet la multiplication de deux vecteurs et `gf_256_inv_vector(3)` qui inverse un vecteur. S'y trouve également `gf_256_gaussian_elimination(3)` qui utilise le trois foncions précédentes pour réaliser l'élimination gaussienne dans le Galois Field. A ces quatre fonctions s'ajoute `gen_coefs(3)` qui retourne une matrice de coefficients de redondance utile à la résolution du système linéaire lorsque qu'un symbole est perdu.

[1]Utilisée pour générer la suite des nombres aléatoires nécessaires au calcul des coefficients de redondance.

IV. Améliorations par rapport au fichier python

A. Améliorations liées à la nature de C

Le langage C est plus explicite et plus proche du langage machine que son homologue Python consommant beaucoup de ressources afin de fournir une abstraction de l'environnement. Là où Python allouera implicitement la mémoire de ses variables, C délègue aux développeurs la tâche d'allouer la mémoire. Cette allocation se fait à l'aide de fonctions telles que `malloc(1)` ou `calloc(2)`. Quand les variables qui y sont stockées peuvent être écrasées, les développeurs doivent s'occuper de libérer la mémoire à l'aide de la fonction `free(1)`. De plus, à chaque appel des fonctions d'allocation et d'écrasement de mémoire, il faut vérifier leur valeur de retour et implémenter au besoin les erreurs qui seront levées en cas d'échec. Enfin, C ne vérifie pas le type de donnée qu'on lui donne en argument de fonction et se contente d'exécuter le code malgré des erreurs potentielles. Nous pouvons vulgairement dire que C considère l'ordinateur comme une "grosse calculette" qui se contente d'exécuter les calculs que nous lui demandons de faire sans se poser de question. Toutes ces raisons font que le C est bien plus efficace en terme de rapidité d'exécution de calcul, d'allocation de mémoire et de consommation énergétique. A condition qu'il soit utilisé correctement et intelligemment! Toutefois, le défaut majeur de C réside dans le fait qu'il nécessite en général plus de temps de développement. Python permet donc de réduire le temps de développement comparativement au C tandis que le C réduit le temps de réponse du code. C'est donc au développeur de choisir quel langage utilisé en connaissance de cause.

B. Améliorations algorithmiques

L'utilisation de l'algorithme en C est globalement le même qu'en Python mis à part pour quelques fonctions et structures.

1) Améliorations dues aux fonctions: Dans le programme C, nous utilisons les fonctions `fopen()`, `fread()` et `fwrite()`. Ces fonctions ont la particularité d'être non seulement plus rapide que leurs homologues sans préfixe "f" mais aussi d'être "thread-safe". C'est-à-dire qu'elles passent par un pointeur de fichier (FILE*) pour réaliser leurs actions. Ce fichier est un buffer qui assure la sécurité pour les threads.

2) Améliorations dues aux structures: Différentes structures sont présentes dans le programme. Elles ne le rendent pas plus rapide mais permettent d'améliorer la lisibilité du code. Elles donnent aussi la possibilité aux fonctions d'avoir accès à plus de valeurs tout en limitant le nombre d'arguments. Enfin, elles sont surtout utiles aux threads car elles permettent de donner plusieurs arguments à la fonction qu'ils exécutent. Parmi les structures initialisées

dans la main, nous retrouvons `args_t`, `fichier_t` et `usefull_info_thread_t`. D'autres structures sont initialisées dans les différents headers, à savoir `linear_system_t`, `unknows_t` et `infos_t`.

C. Améliorations liées à l'architecture

Notre algorithme fonctionne en parallèle contrairement à la version python qui fonctionne uniquement en séquentiel. L'utilisation des threads permet d'exécuter plusieurs tâches simultanément en parallélisant l'exécution sur les multiples cœurs du processeur de notre machine. Initialiser un thread consomme une quantité non négligeable d'énergie et de temps. Le temps d'initialisation des threads est heureusement contrebalancé par le gain de temps dû à leur fonctionnement. La rentabilité des threads se fait donc dans le cas de code demandant une grande puissance de calcul.

V. Tests

A. CUnit

Afin de vérifier que chacune de nos fonctions fonctionne bien indépendamment, nous avons effectué des tests unitaires sur chacune d'entre elles. Le répertoire `tests` contient 3 fichiers permettant de tester les fonctions qu'utilise notre fonction `main`.

B. Valgrind

L'outil valgrind nous permet de détecter les fuites de mémoires potentielles dans notre code. En utilisant cet outil, nous avons pu constater que nous n'avions qu'une infime fuite de mémoire. Cela est dû au fait que la plupart de nos allocations de mémoire (pas toutes malheureusement) ont été libérées une fois que cette partie de la mémoire n'a plus d'utilité dans la fonction.

C. GDB

L'outil GDB est un débogueur qui nous a permis de trouver facilement où se situaient nos `segmentation fault` ainsi que comment les corriger.

VI. Multithreading

Notre programme mutlithreadé se base sur le principe du producteur/consommateur. Nous avons décidé d'utiliser des threads uniquement sur la partie du programme la plus gourmande en opération calculatoire, à savoir la résolution gaussienne. Nous avions d'abord essayé d'associer un thread à chaque fichier mais nous avons rapidement remarqué que cette solution revenait à faire tourner chaque fichier sur un ordinateur différent. Et donc perdait l'essence même du multithreading.

Concrètement, notre programme va exécuter 3 étapes sur chaque fichier du répertoire placé en input:

- La première étape est de créer les blocs dans un buffer de taille `N`, correspondant au nombre de blocs. Cette étape est donc monothreadée car elle ne prend que peu de temps d'exécution mais demande l'allocation d'une grande quantité de mémoire. Contrairement à un producteur/consommateur classique où la consommateur agit dès que le producteur a rempli une place du buffer, dans notre code, le consommateur doit attendre que le producteur ait terminé de remplir le buffer avec l'intégralité des blocs avant de pouvoir agir. Cela nous permet de ne pas avoir à utiliser de sémaphores mais nous oblige cependant à allouer plus de mémoire.

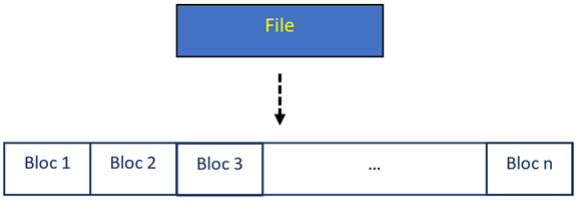

Fig. 1. étape 1

- C'est à la seconde étape qu'a lieu la création des threads. Ceux-ci vont appliquer l'élimination gaussienne à chaque bloc créé au préalable. La section critique partagée par ces threads est uniquement un itérateur sur le buffer. Cela nous permet d'avoir un message déjà dans l'ordre, et de n'avoir qu'une modification de variable dans la section critique.

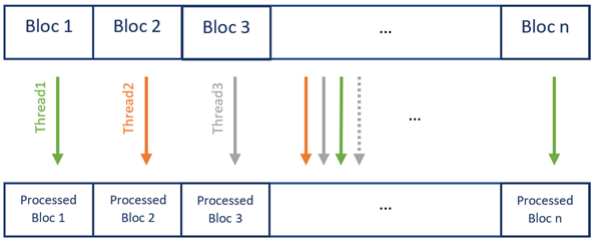

Fig. 2. étape 2

On remarque alors qu'il est possible que certains threads soient plus rapide que d'autres mais qu'ils continueront à traiter les blocs dans l'ordre grâce au pointeur.

- La troisième étape, qui comme la première n'utilise qu'un seul thread, consiste simplement à écrire le contenu de nos blocs traités dans le fichier de sortie.

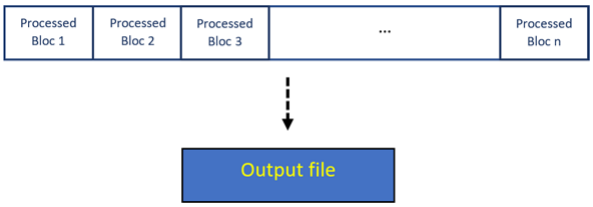

Fig. 3. étape 3

VII. Analyse des performances

Pour analyser les performances de notre programme, nous avons à notre disposition 4 fichiers binaires dans un dossier `test_basic`. Cependant la taille de ces fichiers n'est pas assez grande que pour rendre le programme multithreadé plus rapide que le séquentiel. En effet, la création de threads prend un certain temps et la rentabilité des threads ne se fait qu'à partir d'une certaine taille de fichier d'entrée. Nous avons donc rajouté 2 fichiers tests plus conséquent dans le dossier `test_big` de taille 5.05MB.

A. Rapidité

Pour traiter les 4 fichiers tests (africa.bin, big.bin, medium.bin et small.bin), le programme Python met en moyenne 1.58 secondes. Lors de l'utilisation de notre programme C fonctionnant en séquentiel, celui-ci est exécuté en 0,0055 secondes en moyenne. Comme espéré, écrire en C permet d'améliorer significativement la vitesse d'un code écrit en Python, 287 fois plus rapide en moyenne dans notre cas.

1) Monothread vs Multithread: Malheureusement, le programme multithreadé ne produit pas l'effet escompté. En effet, peu importe le taille des fichiers d'entrées, celui-est plus lent que la version monothreadée. On peut le voir sur les graphiques[2] suivants :

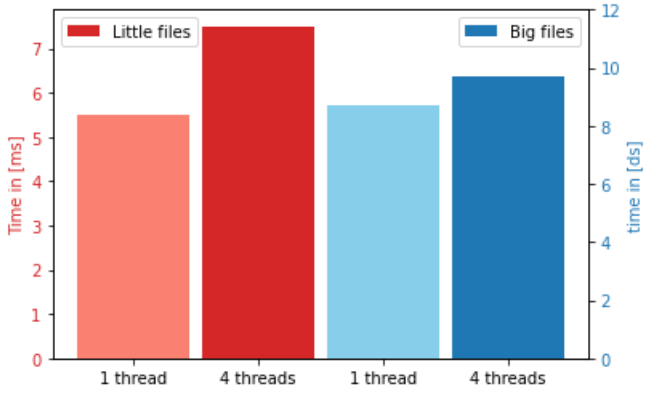

Fig. 4. Monothread vs Multithread

On remarque cependant que la différence entre le monothread et le mutlithread diminue lorsque la taille des fichiers dans le répertoire placé en input augmente.

B. Consommation de mémoire

Malgré un bon nombre de libération de mémoire via la fonction `free(1)`, de la mémoire reste perdue à la sortie de notre programme. Le lecteur peut, grâce aux graphiques suivants, se donner une idée du pourcentage de mémoire non-libérée :

[2]Les données de ce graphique sont basées sur une moyenne des différentes exécutions du programme.

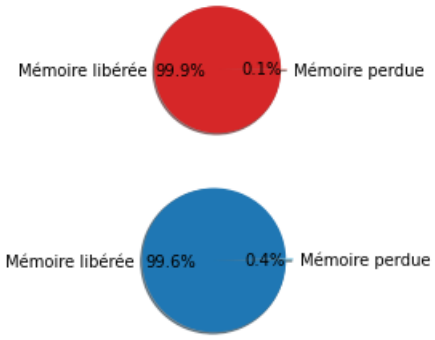

Fig. 5. Pourcentage de la perte de mémoire de petits fichiers (rouge) et de gros fichiers (bleu)

Nous pouvons remarquer que la perte de mémoire augmente en fonction de la taille du fichier, mais pas de manière significative (0.1% contre 0.4% alors que la taille du réprtoire est multipliée par $63,8$).

VIII. Conclusion

Premièrement, ce projet nous a permis de mettre en pratique nos connaissances du langage C développées durant les 6 premières semaines de cours, notamment le multithreading, l'allocation et la libération de mémoire. De plus, nous avons appris à travailler en équipe sur un projet informatique.

Pour cela, nous avons du développer nos connaissances de l'utilisation d'un git partagé. Nous avons aussi appris à utiliser un Makefile ainsi que de nombreux outils tels que Valgring, GDB ou encore CUnit. Enfin, nous avons dû apprendre à utiliser un Raspberry Pi, dispositif jusque là inconnu pour nous. Nous avons ensuite dû nous entraîner à l'exercice des Peer-reviews. Phase très enrichissante pour nous, autant dans la compréhension et la correction d'autres codes que dans les remarques avisées de l'assistant[1] que nous remercions. Ses différentes remarques nous ont permis d'améliorer et d'optimiser un code à l'époque déjà fonctionnel.

L'objectif principal de ce projet était d'implémenter un algorithme FEC (Forward Erasure Correction) en C, à partir d'un code python fonctionnel. Nous devions ensuite optimiser cette version C à l'aide de threads. La concrétisation de notre projet s'est fait lorsque nous avons testé notre algorithme sur Raspberry Pi grâce auquel nous avons pu comparer les différences d'efficacité entre l'algorithme initial en Python, notre version séquentielle et notre version multithreadée.

IX. Bibliographie

1) Syllabus du cours: *O.Bonaventure, G.Detal,C.Paasch, LEPL1503 :Introduction au langage C, Version 2021.*

[1]Suite à une erreur inconnue, nous n'avons reçu que la review d'un assistant et non de plusieurs élèves.