LEPL1503 - Groupe V2
Code efficace multithreadé en C
Rapport de projet

BOTTON Florentin
florentin.botton@student.uclouvain.be
3008-19-00

D'INTINO SALGADO Lorenzo
lorenzo.dintino@student.uclouvain.be
8028-19-00

ACCOU Pierre
pierre.accou@student.uclouvain.be
0129-18-00

DE RO Corentin
corentin.dero@student.uclouvain.be
1310-18-00

DOERAENE Anthony
anthony.doeraene@student.uclouvain.be
0302-20-00

RIXEN Thomas
thomas.rixen@student.uclouvain.be
1463-20-00

***Résumé*—Implémentation d'un code C afin d'améliorer les performances globales du programme et sa vitesse d'exécution grâce aux threads**

I. INTRODUCTION

Ce projet vise à implémenter la récupération de symboles sources perdus lors d'une transmission dans un réseau satellite subissant des pertes. Néanmoins, l'ensemble des opérations constituant cette procédure peut prendre du temps et de l'énergie. Le langage python est capable de réaliser ce type d'opération mais être assez consommateur en temps. L'initiative principale de ce projet réside donc dans l'utilisation du langage C qui est lui plus proche de la machine. Permettant en autre une meilleure gestion de la mémoire et est donc nettement plus performant.
Les objectifs sont de ce fait multiples. D'une part l'exécution de notre code doit être bien plus rapide que sa traduction Python et d'autre part l'utilisation de la machine doit être optimisée. Cette dernière opération passe naturellement par l'utilisation de plusieurs threads mobilisant le maximum de ressources de nos machines. Tout cela en maintenant une consommation énergétique raisonnable.

II. DESCRIPTION DE L'ALGORITHME IMPLÉMENTÉ EN C ET DE SES AMÉLIORATIONS

L'algorithme que nous avons implémenté en C réalise les étapes suivantes :

1) Lecture et parsing des arguments
2) Extraction des noms des fichiers du dossier à traiter. Ceux-ci sont ensuite stocké dans l'array `filenames` de `t_args`
3) Création des threads ainsi que des sémaphores. Nous pouvons, par la suite, lancer les threads qui exécuterons chacun les instructions suivantes :
   a) Obtention du prochain fichier qui n'a pas encore été traité. Cela est fait à partir de la variable `nextFile` se trouvant dans `t_args` contenant l'index de l'array `filenames` où se trouve le prochain élément à traiter. Nous devons ici utiliser un sémaphore pour assurer que la variable est bien incrémentée.
   b) Lecture de ce fichier, création d'une structure `message_t` contenant toutes les informations relatives à ce fichier (seed, taille message, ...) et parsing du message sous forme de liste de blocs.
   c) Génération des coefficients aléatoires et vérifications des blocs afin de savoir s'ils possèdent des symboles perdus. Le cas échéant, création du système d'équation linéaire et résolution de celui-ci. Nous remplaçons ensuite les symboles perdus par ceux ayant été retrouvés grâce à la résolution.
   d) Ecriture du message dans le fichier de sortie. Nous devons ici nous assurer que deux threads n'écrivent pas en même temps, c'est pourquoi nous utilisons un sémaphore dans cette étape.
4) Attente que tous les threads aient finis grâce à un `join` et nous terminons l'exécution en détruisant les sémaphores et les threads. Nous libérons aussi le reste de la mémoire encore utilisée.

Différentes optimisations ont été réalisées afin d'améliorer cet algorithme. Nous tentons d'utiliser notamment le moins de mémoire grâce à différentes techniques :

— Nous stockons les booléens sur des `uint8_t` et n'utilisons pas les booléens de la librairie standard afin d'utiliser le moins de mémoire possible (1 byte au lieu de 4). Nous pouvons donc stocker les index des symboles perdus en divisant la mémoire utilisée par au moins 24 (booléen en python prennent 24 bytes pour False et 28 pour True, contre 1 byte pour notre programme, par booléen)
— Nous ne stockons pour chaque fichier qu'une seule fois la taille des symboles et des redondances, étant les mêmes pour tout le fichier.

Pour ce qui est des optimisations de vitesse d'exécution, nous utilisons les threads en limitant le plus possible les exclusions mutuelles entre ceux-ci afin de paralléliser au plus possible notre programme et ainsi gagner du temps d'exécution. Ainsi, les deux seules exclusions mutuelles sont lors de l'obtention de l'index du prochain fichier à traiter et l'écriture dans le fichier de sortie. Ceci permet de fortement accélérer l'exécution de notre programme et de maximiser l'impact des threads.

Nous essayons également de limiter le plus possible le nombre de malloc, ce qui permet de fortement accélérer le code et de consommer le moins de mémoire possible. Un bon exemple de cette pratique est notre implémentation de la résolution du système linéaire. Nous faisons les opérations sur les lignes en modifiant directement la matrice passée en argument, ce qui nous permet d'éviter les opérations inutiles. Nous nous assurons bien sûr de conserver l'exactitude du code en s'assurant que l'on puisse modifier cette donnée sans répercussion.

Enfin, nous rajoutons de nombreuses indications qui permettent d'afficher des messages en cas d'erreur du programme (par exemple, un échec de malloc ou d'ouverture de fichier). Ceci devrait permettre à l'utilisateur de mieux comprendre l'erreur pouvant survenir lors de l'utilisation du programme.

III. Description des tests unitaires et des outils d'analyse de code

Nous possédons, dans notre code, 4 fichiers de tests que nous avons implémentés ainsi qu'un fichier **testRun.c** qui permet de lancer les différentes suites de tests définies dans ces fichiers :

1) `test_tinymt32.c` : Ce fichier testant le fichier source `tinymt32.c` nous a été fourni lors de la remise du projet, nous ne détaillerons donc pas celui-ci.
2) `test_system.c` : Dans ce fichier consacré aux tests des fonctions de `system.c`, nous vérifions que ces fonctions retournent les bons résultats. En particulier :
   - Tests des fonctions sur les vecteurs : nous vérifions chaque fonction qui réalise des opérations sur les vecteurs (modification en place ou non) à partir de vecteurs générés aléatoirement. Pour les fonctions qui doivent retourner un nouveau vecteur, nous vérifions également que les vecteurs passés en arguments n'ont pas été modifiés lors de l'exécution de la fonction.
   - `test_gaussian_reduction_fixed` vérifie la fonction de réduction gaussienne pour un petit exemple repris du code python
   - `test_gaussian_reduction_bigger` vérifie aussi cette réduction gaussienne, mais avec un bien plus gros exemple qui est généré aléatoirement grâce à la fonction `gen_coeffs` de `system.c`. Nous générons également la matrice **b** afin que celle-ci soit une copie de la matrice **A**. Ainsi, comme chaque opération sur les lignes de **A** est censée également s'exécuter sur chaque ligne de **b** correspondante, nous pouvons vérifier que la matrice **A** et la matrice **b** ne contiennent bien à la fin que des 1 sur leur diagonale et des 0 partout ailleurs (car nous devons arriver à une matrice de Gauss-Jordan suite à la réduction). Si ce n'est pas le cas, cela veut dire que certaines opérations sur les lignes ont mal été effectuées, et donc que la fonction est erronée.
3) `test_block.c` : Ce fichier reprend les fonctions de tests pour certaines fonctions définies dans le fichier `block.c`. Les différents tests sont les suivants :
   - `testSymbolLost` : cette fonction va générer d'abord aléatoirement un nombre afin de décider si le symbole sera perdu ou non. Si celui-ci doit être perdu, alors le vecteur représentant le symbole sera rempli de 0. Sinon, celui-ci sera rempli de nombres aléatoires compris entre 0 et 256 non-inclus. Nous vérifions ensuite que la fonction fonctionne bien en lui passant ce vecteur. Nous répétons ensuite cette procédure plusieurs fois, pour tester les deux cas (perdus ou non).
   - `testBlockLost` : cette fonction permet de tester si notre implémentation détecte bien les blocs ayant des symboles perdus. Elle possède un fonctionnement analogue à la fonction de test `testSymbolLost` , juste en rajoutant l'index de perte dans un tableau contenant le résultat attendu.
4) `test_message.c` : Ce fichier contient des tests sur les méthodes de `message.c` ainsi qu'un test sur la bonne exécution de notre programme en entier.
   - `test_Message` Ce test compare le fichier obtenu par notre programme avec celui obtenu par le programme Python. Il va donc pour chaque fichier traité :
     a) obtenir les informations (seed, nom du fichier)
     b) rechercher le message correspondant à ce fichier dans le fichier obtenu grâce au programme Python
     c) comparer les deux messages afin de vérifier s'ils sont identiques
   - `test_readdir` : Ce test vérifie que la fonction `readDir` retourne la bonne liste avec les noms des fichiers contenus dans un répertoire.
   - `test_makeBlockList` : Ce dernier test a pour but de vérifier l'exactitude de la fonction `makeBlockList` . Nous utilisons pour cela deux fichiers annexes : `makeblocklist_expected.txt` et `makeblocklist.bin`. Le premier contient l'entièreté du message tel qu'il est censé être reconnu. Le deuxième quant à lui, contient le message encodé envoyé sur le réseau. Ainsi, nous pouvons vérifier qu'après transformation en blocs, l'entièreté du message initial est bien conservé et inchangé.

IV. Métriques de comparaison de performance

A. Vitesse d'exécution

L'exécution du code C en multi et single threads sur un Raspberry Pi nous a permis d'observer une différence de vitesse d'exécution impressionnante. En effet, pour une exécution avec 200 fichiers (6,6Mo) donnés en input, le temps d'exécution du programme en single-thread est de **2549.5ms**. Avec le même nombre de fichiers en input, mais cette fois en exécutant le programme en multi-threads (4 threads) nous avons obtenu un temps d'exécution de **814.67ms**. Soit plus de 3 fois plus rapide. Quant à la comparaison avec le programme python, après avoir fait des tests de vitesse pour 5, 10, 20, 50 fichiers respectivement de taille totale 156ko, 312ko, 624ko et 1,58Mo, nous pouvons dire que notre programme avec 4 threads est, à peu près, 2400 fois plus rapide que son équivalent python. Nous pouvons bien voir sur les graphes suivants la tendance linéaire de notre programme. Le temps d'éxécution croît linéairemenent par rapport au nombre de fichier (ou de la taille totale des fichiers). Nous pouvons tout de même remarquer que la croissance linéaire est plus lente avec 4 threads. En effet, la pente de la droite est moins accentué que pour celle de la fonction d'éxécution avec 1 thread.

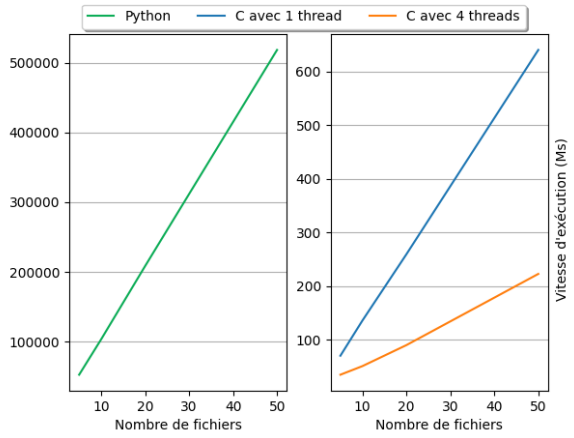

Figure 1. Comparaison des temps d'exécution (échelle logarithmique pour la figure de droite)

B. Mémoire utilisée

Nous avons mesuré la mémoire utilisée par le programme python grâce au package memory-profiler [1] qui nous a permis de générer les résultats utilisés sur le graphique suivant. Nous avons ensuite utilisé valgrind pour mesurer la mémoire utilisée en C.

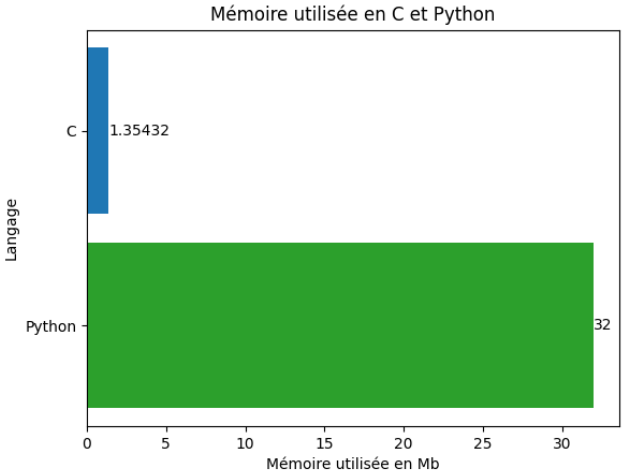

Figure 2. Comparaison de la mémoire utilisée

Nous pouvons directement observer une nette amélioration quant à la quantité de mémoire utilisée par notre programme. En effet, seulement à peu près 4.2% de la mémoire utilisée par le programme python est utilisée lors de l'exécution de notre code étant donné que le programme python consomme 32Mo de mémoire et notre programme seulement 1.35Mo.

C. Consommation énergétique

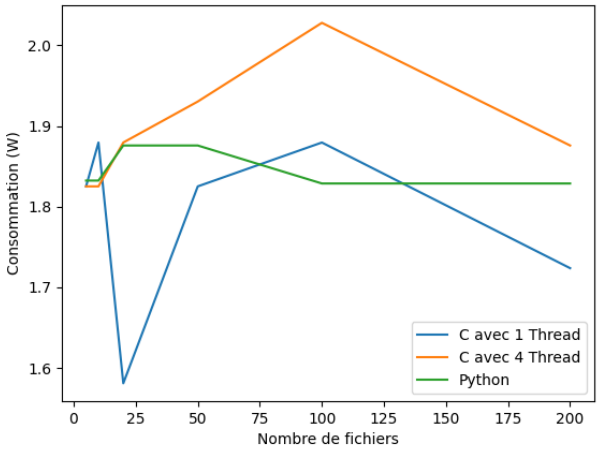

Figure 3. Consommation énergétique

La consommation énergétique de notre programme est plus élevé lors du pic de consommation que l'équivalent python. Mais, il ne faut pas oublier que celui-ci est beaucoup plus rapide et donc consomme en définitive moins d'énergie car devant s'exécuter pendant moins de temps. Il est tout de même intéressant de remarquer que l'utilisation de threads augmente légèrement la consommation électrique (de l'ordre de 0.2W). Ces mesures peuvent potentiellement contenir de légères erreurs de mesures dû au matériel, nous nous en

excusons d'avance. Néanmoins, celles-ci ne devraient pas changer drastiquement les résultats obtenus.

V. Améliorations suite au peer-review

Suite aux peer-reviews, diverses améliorations ont été insérées dans notre projet.
Premièrement, nos fonctions readdir et makeBlockList ont toutes deux été testées. Ces deux fonctions ont donc été logiquement ajoutées dans test_message.c . Bien entendu, nous avons vérifié que ces dernières ne causaient pas de memory leak supplémentaires avec leur implémentation.
Deuxièmement, nous avons également veillé à la documentation de notre projet. De nouveaux commmentaires ont été insérés et d'autres ont été modifiés. Cela rendra certainement la lecture du projet plus agréable.
Troisièmement, la commande pour run de notre projet a elle aussi été modifiée puisqu'elle comportait une petite faute de syntaxe.
Finalement, nous avons également bien pris en compte la remarque sur le manque d'optimisation de nos threads. Cependant, cette dernière a été laissée sans suite. En effet, il nous semble normal que lorsque le projet s'exécute sur 4 fichiers on ne voit que très peu la différence entre la version séquentielle et parallélisée du projet. A plus grande échelle, cette différence est bien marquée et a été retestée par nos soins.

Remerciements à : Olivier BONAVENTURE, Axel LEGAY, Louis NAVARRE, Christophe CROCHET, Magali LEGAST, Matthieu PIGAGLIO, Tom ROUSSEAUX ainsi qu'à notre tuteur Antoine DE LA VALLÉE pour l'aide, les conseils et le matériel apporté qui ont permis l'aboutissement de ce projet

Références

[1] Memory profiler : package python d'analyse de mémoire utilisée (https://pypi.org/project/memory-profiler/), par Fabian Pedregosa and Philippe Gervais, inspiré du travail de Robert Kern