Rapport du projet 3 - Groupe W4

Dewilde Alexandre
UCLouvain
Louvain-la-neuve, Belgique
alexandre.dewilde@student.uclouvain.be

Olyvia Hirwa Mihigo
UCLouvain
Louvain-la-neuve, Belgique
raissa.hirwamihigo@student.uclouvain.be

Balde Lamarana
UCLouvain
Louvain-la-neuve, Belgique
lamarana.balde@student.uclouvain.be

Barry Sounounou
UCLouvain
Louvain-la-neuve, Belgique
sounounou.barry@student.uclouvain.be

Ebona Noah Patrick Junior
UCLouvain
Louvain-la-neuve, Belgique
patrick.ebonanoah@student.uclouvain.be

***Résumé*—Ce rapport présente une analyse détaillée du projet 3 pour le groupe W4 dans le cadre du cours LEPL1503 dont le but était l'implémentation d'un programme de code correcteur de données en C multi-threadé sur base d'un programme python mono-threadé.**

I. Introduction

La tache demandé était de produire un programme C sur base d'un code python [2], qui résout le problème de l'utilisation de codes correcteurs dans un protocole réseau.
Le programme C devait être significativement plus rapide que le programme Python [2] en utilisant plusieurs threads pour paralléliser l'exécution sur les multiples coeurs du processeur de la machine.

II. Description du programme C

Le programme C est basé sur le code python [2] mais avec une architecture multi-threadé, ainsi que des améliorations algorithmiques et des optimisations.

A. Amélioration de l'architecture : Producer/Consumer Pattern Design

L'architecture du programme est structuré de la manière suivante :

Producer → *Consumers and Producers* → *Consumer*, avec deux buffers partagés (voir fig.1).

Description de l'architecture :

- Le premier *Producer* que l'on nomme *folder_producer*, s'occupe de lire le dossier avec les fichiers d'entrées pour le programme, et ajoute ceux-ci dans le premier buffer partagé. Il n'y a qu'un producer de ce type afin de ne pas rentrer en conflits lors de la lecture du dossier entre différents threads ; de plus la tâche réalisée par ce producer représente une faible partie du temps d'exécution du programme comparé aux *producers* s'occupant des calculs.
- Ensuite il y a N *producers* de calcul, qui sont les consumers du *folder_producer*, ils consomment le premier buffer partagé, en retire un fichier, lisent le fichier et effectue les calculs sur chaque fichier ; une fois leurs tâches accomplies, ils ajoutent dans le second buffer partagé, la traduction du fichier.
- Et pour finir, il reste un consumer qui s'occupe d'écrire le résultat dans le fichier de sortie. Il n'y a qu'un seul consumer de ce type, bien que *fwrite* soit *thread-safe* [4], il n'est pas intéressant dans avoir plusieurs car la majorité du temps pris par le consumer est l'écriture dans le fichier d'output ce qui est lock lors d'une écriture, ceci afin d'éviter des conflits d'écriture.

L'architecture du programme C est une amélioration de l'architecture du programme python [2] qui lui est mono-threadé, l'architecture du projet a donc complètement été repensée pour profiter pleinement des cores à dispositions du CPU sur lequel le programme est exécuté.

B. Amélioration algorithmique et optimisations

Différentes améliorations algorithmiques et optimisations ont été faites, mais le projet a été réalisé en gardant en tête que *Premature Optimization Is the Root of All Evil* [3]. Le projet a été développé de manière simple, sans faire d'optimisations prématurées qui pourraient réduire la lisibilité du code et pourraient introduire des bugs futurs au cours du développement. C'est seulement une fois le projet fonctionnel et les tests bien rodés que des optimisations et *micro optimizations* ont été réalisées, tout en gardant un code lisible ; cela est important pour la maintenabilité du code.

1) Diminution du nombre d'opérations I/O: Les opérations de lecture et d'écriture sont très coûteuses et ont un grand effet sur le temps d'exécution, dans le code C le nombres d'écritures a été drastiquement réduit pour réduire le temps d'exécution. Le programme python [2] écrit byte par byte, ce qui est très lent ; dans la version du programme C cela été remplacé par une écriture du message en une seule écriture.

2) Diminution du nombre d'allocations mémoire: Les allocations mémoires sont quelque chose de coûteux en terme de temps, de plus certains systèmes embarqués peuvent avoir leurs mémoires ram limitées ; pour palier à cela, le programme a été repensé pour limiter le nombre d'allocations et réduire l'utilisation de mémoire.

Comme dans ces cas :

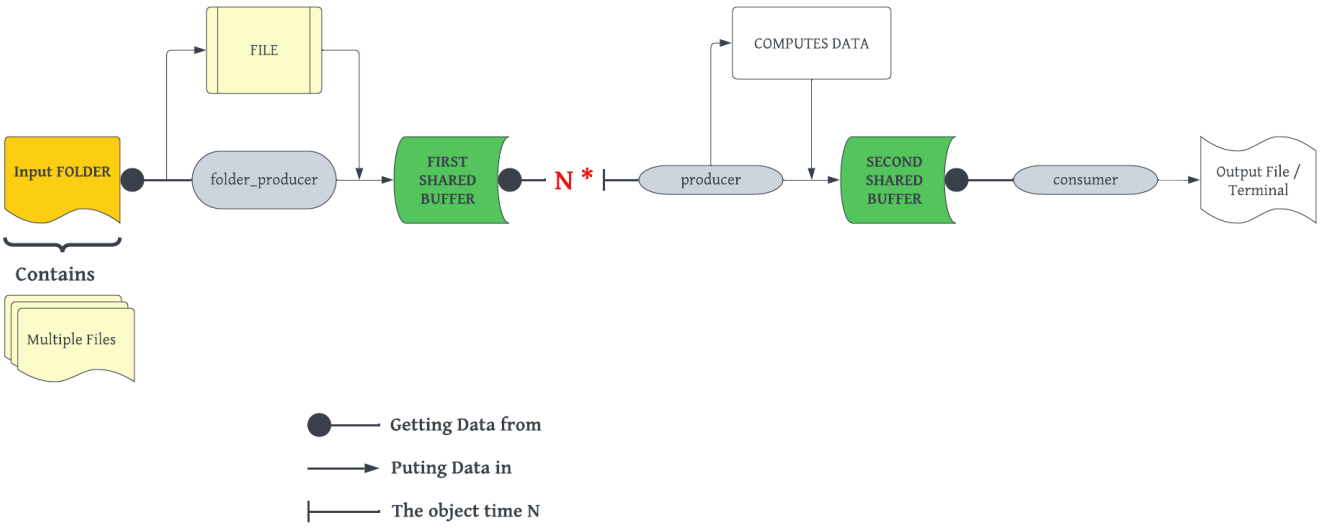

FIGURE 1. Architecture du projet

— Pas de nouvelles allocations pour le contenu des blocs : le contenu de chaque bloc (message et symboles de redondances) n'est pas une copie mais une référence vers la même zone mémoire du contenu du fichier grâce à l'arithmétique des pointeurs, il n'y a donc pas de données dupliquées.
— Fonctions en place : un autre moyen pour diminuer les allocations a été de remplacer certaines fonctions qui allouaient de la mémoire supplémentaires par des fonctions en place, ceci a été fait principalement par des fonctions utilisées dans la méthode de gauss (voir fig 2).

3) Optimisations du compiler: Pour améliorer les performances, les optimisations du compiler sont utilisées à l'aide de l'argument *-O3* [5] à la compilation.

III. TESTS

Les tests ont été créés dès le début du projet, de nombreux tests ont été développé à l'aide du framework de test unitaire CUnit [6], ceux-ci ont été cruciaux. Le projet a été orienté avec la philosophie *Test-driven development* afin d'avancer dans le projet tout en s'assurant que chaque bouts développés soient corrects et d'éviter de se retrouver avec des bugs vers la fin du projet

A. Test des fonctions

Pour la plupart des fonctions, il y a des tests, cela permettra d'informer directement une personne modifiant une fonction et créant un erreur.

Par exemple, pour la fonction d'élimination de gauss, qui était une partie critique des premières semaines, une centaine de système linéaire (avec pivots non nul comme *gen_coeffs()* génère) ainsi que leurs solutions ont été générées ; le tout pour s'assurer que la fonction d'élimination fonctionnait correctement.

Exemple d'améliorations par des opérations en place

Extrait du code python [2]

```python
def gf_256_full_add_vector(v1, v2):
    rep = v1.copy()
    for i in range(len(v1)):
        rep[i] = \\
            np.bitwise_xor(v1[i], v2[i])
    return rep
```

Extrait du code C

```c
void inplace_gf_256_full_add_vector(
    uint8_t *symbol_1,
    uint8_t *symbol_2,
    uint32_t symbol_size)
{
    for (uint32_t i = 0;
        i < symbol_size;
        i++)
    {
        symbol_1[i] ^= symbol_2[i];
    }
}
```

FIGURE 2. Amélioration par des fonctions en place

B. Test sur l'ensemble du programme

Des tests sur l'ensemble du programme ont aussi été réalisés. Des fichiers générés à l'aide de *make_input.py [2]*, et leurs solutions définies grâce au programme python [2] ont été utilisés pour s'assurer que le programme fonctionne correctement.

C. Valgrind [8]

Valgrind a été utilisé tout au long du développement, cet outil a permis de détecter et corriger de nombreuses erreurs mémoires et surtout de détecter les memory leaks. Cela a permit d'arriver à un projet final sans fuite et erreurs de mémoire.

D. Cppcheck [10]

Cppcheck a été utilisé et a permis de détecter des erreurs d'overflow dans le projet.

E. Jenkins [11]

Jenkins a aussi été utilisé sur le repository git en exécutant les tests à chaque commits. Cela permettait de s'assurer qu'un commit ne "cassait" pas le projet ou, en cas de problème, permettait de le résoudre.

IV. Comparaisons des programmes

A. Temps d'exécution

Pour comparer les programmes un script bash, qui calcul le temps d'exécution pour une commande donnée avec n itérations (argument du script) et stocke les n temps d'exécution dans un fichier csv, a été utilisé. Les résultats ont été ensuite interprétés par un programme python qui génère des moyennes et des graphiques à l'aide de la librairie matplotlib [12]. Les tests ont été réalisés sur le raspberry pi.

1) Programme python vs Programme c: Comparer le programme python et C à l'aide d'un graphique n'est pas intéressant visuellement tellement la différence est grande, un tableau a plus de sens :

Fichiers	Python	C 1 thread	C 4 threads
20 africa.bin	155.2484s	0.0908s	0.0477s
40 africa.bin	308.1105s	0.1847s	0.0881s
dossier samples	21.2247s	0.1840s	0.1237s
fichiers données [1]	5.2891s	0.0984s	0.1011s
1 fichier hamlet	5.2526s	0.0977s	0.1061s

FIGURE 3. Comparaisons des vitesses d'exécutions sur le raspberry pi 3B

On remarque une grosse différence entre le programme python et le programme C, jusqu'à 1500x plus rapide pour 40 fichiers africa.bin, la différence est plus marquée lorsque le nombre de fichiers est grand. On remarque aussi lorsqu'il y a peu de fichiers, le programme avec plusieurs threads est légèrement plus lent ; ce qui s'explique par le fait que plusieurs producers sont lancés et ne sont pas utilisés.

1. Fichiers fournis avec le programme python [2]

B. Programme C en fonction du nombre de threads

Le nombre de threads de calculs utilisés par le programme joue aussi sur le temps d'exécution. Pour observer ce phénomène, différents tests ont été réalisés :

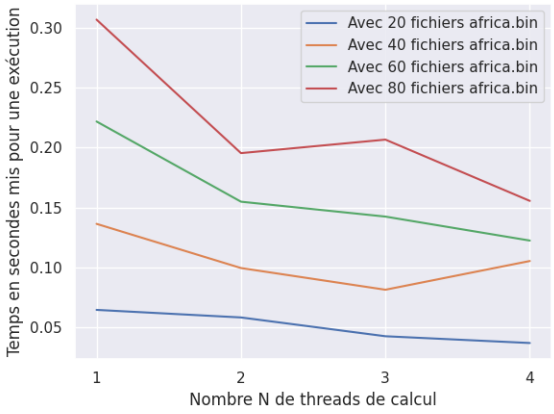

FIGURE 4. Runtime par nombre de threads de calcul sur le PI 3B

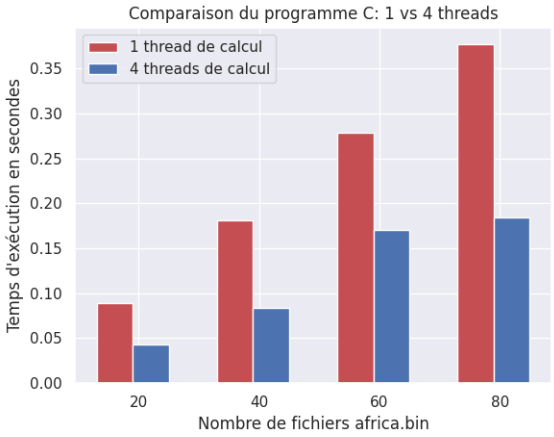

FIGURE 5. Runtime, avec 1 thread contre 4 threads sur le PI 3B

C. Consommation de mémoire

Pour la comparaison de mémoire, l'outil massif [9] de la suite valgrind [8] a été utilisé. Celui-ci analyse l'utilisation du heap par le programme en fonction du temps. Voici les pics de la mémoire utilisé sur le heap :

Fichiers	Python	C
20 africa.bin	7 Mio	0.308Mio
40 africa.bin	7 Mio	0.544Mio
dossier samples	7 Mio	0.182 Mio
fichiers données[1]	7 Mio	0.179Mio
1 fichier hamlet	7 Mio	0.595Mio

FIGURE 6. Comparaisons des pics d'utilisation du heaps

On constate que le programme python [2] utilise beaucoup plus de mémoire que le programme C, jusqu'a 10x moins, en

fonctions des fichiers. Observer la stack n'est pas utile car elle représente une faible partie de la mémoire par apport au heap.

D. Consommation électrique

Pour calculer la consommation électrique des programmes, on peut utiliser la consommation moyenne d'un raspberry de ce modèle, (ici le Raspberry Pi 3 Model B). Selon le site pidramble [7], ce modèle consomme sur un core 2.4 w et 3.7 w sur 4 core. On peut donc estimer la consommation grâce au temps d'exécution de la figure 6

Fichiers	Python	C 1 thread	C 4 threads
20 africa.bin	0.1 Wh	0.00006 Wh	0.00005 Wh
40 africa.bin	0.2 Wh	0.00012 Wh	0.00009 Wh
dossier samples	0.01414 Wh	0.00012 Wh	0.00012 Wh
fichiers données[1]	0.0035 Wh	0.00006 Wh	0.0001Wh

FIGURE 7. Comparaison de la consommation sur le raspberry pi 3B

On constate que la diminution de la consommation est significative. Pour un système embarqué sur batterie, cela peut augmenter nettement la durée d'utilisation sans charge de l'appareil, ce qui est non négligeable.

V. CONCLUSION

En conclusion, le projet final multi-threadé montre de gros progrès en terme de rapidité, d'utilisation de mémoire et de consommation électrique, ce qui peut se révéler crucial pour un système embarqué. Le projet qui avait été demandé a parfaitement été réalisé en respectant les contraintes imposées.

.

RÉFÉRENCES

[1] O. Bonaventure, A. Legay,L. Navarre, T. Rousseaux, C. Crochet, M. Pigaglio and M. Legast, "Énoncé du projet – Code efficace multi-threadé en C". https://moodle.uclouvain.be/pluginfile.php/334805/mod_resource/content/3/lepl1503_project_statement.pdf.

[2] https://github.com/louisna/lepl1503-2022-pyfec

[3] D. Knuth, *The Art of Computer Programming*

[4] https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_concurrency.html

[5] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[6] https://github.com/jacklicn/CUnit

[7] https://www.pidramble.com/wiki/benchmarks/power-consumption

[8] https://valgrind.org/

[9] https://valgrind.org/docs/manual/ms-manual.html

[10] https://github.com/danmar/cppcheck

[11] https://www.jenkins.io

[12] https://matplotlib.org/