Rapport Projet 3 - Le problème des k-means

Samuel de Meester de Ravestein
École polytechnique de Louvain
UCLouvain
Louvain-la-Neuve, Belgique
samuel.demeester@student.uclouvain.be

Nicolas Jeanmenne
École polytechnique de Louvain
UCLouvain
Louvain-la-Neuve, Belgique
nicolas.jeanmenne@student.uclouvain.be

Sébastien Mary
École polytechnique de Louvain
UCLouvain
Louvain-la-Neuve, Belgique
sebastien.mary@student.uclouvain.be

Loïc Spigeleer
École polytechnique de Louvain
UCLouvain
Louvain-la-Neuve, Belgique
loic.spigeleer@student.uclouvain.be

Gilles Maes
École polytechnique de Louvain
UCLouvain
Louvain-la-Neuve, Belgique
gilles.maes@student.uclouvain.be

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

***Abstract*—Ce rapport a pour but de présenter et expliquer le déroulement de notre projet en informatique en 2e année de bachelier à l'UCLouvain. Ce projet vise à implémenter une résolution multithreadée du problème *k-means* dans le langage C.**

***Index Terms*—project, k-means, C, multithread**

I. Introduction

Pour la première fois dans notre bachelier, nous avons été confronté à un problème d'optimisation. Le problème consistait d'abord à transcrire une implémentation utilisant Python de l'algorithme de Lloyd en langage C. L'intérêt de cette transcription est que le langage C est beaucoup plus rapide que Python. Pour gagner encore en performance, il fallait également que l'implémentation en C soit multithreadée contrairement à celle en Python. L'algorithme de Lloyd [1] résout un problème bien connu communément appelé le problème de *clustering*. Pour mener à bien ce projet, il a fallu s'accommoder d'outils fortement utiles tels que Git [2], Valgrind [3] et CUnit [4].

II. Présentation de l'algorithme

A. L'algorithme Lloyd

Le problème de *clustering*, nommé *k-means*, vise à subdiviser un ensemble de points en **k** groupes distincts appelés cluster. Chaque cluster est donc un sous-ensemble de points ayant un centre nommé centroïde. L'objectif de l'algorithme de Lloyd est de minimiser la somme des distances entre chaque point et son centroïde le plus proche.

Pour ce faire, l'algorithme effectue deux opérations à chaque exécution de l'algorithme :

1) Assignation de chaque point à son centroïde le plus proche.
2) Calcul des nouveaux centroïdes en calculant la moyenne des coordonnées des points d'un cluster.

Ce processus est répété jusqu'à ce que l'algorithme converge vers un état stable (un minimum local). L'algorithme s'arrête donc lorsque les clusters restent identiques entre deux itérations successives. Mais nous n'avons pas la garantie de trouver une solution optimale, c'est pour cette raison qu'il est important d'exécuter l'algorithme plusieurs fois avec des centroïdes de départ différents.

III. Structure du programme

A. Choix des structures

1) point_t: Cette structure représente un point. Elle est composée d'un vecteur reprenant les coordonnées du point ainsi que l'indice du centroïde le plus proche. Avec cela, nous avons décidé de ne pas utiliser de structure cluster car chaque point est lié à son centroïde le plus proche via la structure *point_t*.

2) kMeans_t: Cette structure contient toutes les informations nécessaires au bon déroulement d'une itération de l'algorithme de Lloyd. Ainsi, lorsque nous avons besoin d'avoir toutes les informations, notamment dans les tests, seule cette structure est nécessaire.

3) data_t: Cette structure reprend toutes les informations contenues dans le fichier binaire fourni en entrée. Les coordonnées des points repris dans la structure ne sont jamais recopiées puisque ceux-ci ne changent jamais au cours des itérations de l'algorithme de Lloyd. Pour ce faire, à chaque création d'un point (*point_t*) nous reprenons ses coordonnées via un pointeur provenant de data_t. Cela permet un gain de temps.

B. Fichier main

Ce fichier est le chef d'orchestre du programme. Le but du fichier est de permettre au programmeur de constater toutes les étapes clés suivies par le programme. Pour ce faire, nous avons jugé pertinent d'y laisser en grande partie l'implémentation multithread du programme. De cette manière, aucun appel à une fonction importante n'est caché depuis le fichier main.

C. Répertoire headers

Ce répertoire reprend tous les fichiers headers contenant toutes les déclarations de fonctions utilisées dans le répertoire src avec leurs spécifications. Il y a également un fichier header en plus (*kMeansStruct.h*) qui reprend les différentes structures employées au cours de l'exécution de l'algorithme de Lloyd.

D. Répertoire src

Tous les fichiers .c nécessaires au bon déroulement du programme (autre que le *main.c*) sont repris dans ce répertoire.

- Le fichier *manageArgs.c* s'occupe de la gestion des arguments passé au programme.
- Ensuite, le fichier *readBinaryFile.c* prend en charge la lecture du fichier binaire donné en entrée.
- Le fichier *generateStartingCentroids.c* exécute les différents préparatifs avant le lancement de l'algorithme de Lloyd. Il calcule donc principalement les différentes séries de centroïdes de départ.
- Les fichiers *distance.c* et *kMeans.c* quant à eux, implémentent l'algorithme de Lloyd.
- Il ne reste plus qu'à écrire les résultats dans un fichier CSV à l'aide du fichier *writeOutputFile.c*.
- Le fichiers *manageHeap.c* permet de libérer les allocations mémoires et fermer les fichiers nécessaires alors que le fichier *buffer.c* permet la création du buffer utilisé par les threads.

E. Répertoire input_binary

Ce répertoire est uniquement là pour permettre d'accueillir les différents fichiers binaires donnés en entrée au programme.

F. Répertoire output_csv

Ce répertoire permet de recueillir le fichier CSV de sortie du programme.

G. tests_files

L'ensemble des fichiers utilisés par les tests sont contenus dans ce répertoire. Tout d'abord, nous avons réparti notre code en différents répertoires notamment src et headers, de cette façon notre code est plus lisible. De plus, nous avons ajouté un répertoire contenant les fichiers binaires et un répertoire pour les fichiers de sorties. Nous avons choisi de mettre ces répertoires dans le répertoire tests_files pour les tests pour ne pas avoir de conflits. Ces fichiers sont uniquement utilisés par les tests.

Comme nous avons également implémenté les fonctions pour faire des graphiques de performances, nous avons un répertoire test_performances contenant le nécessaire. Un répertoire contenant les fichiers Bash et un répertoire de fichiers Python sont également présents. Ces derniers répertoires sont utilisés par nos tests comparant le fichier de sortie produit par l'implémentation en Python avec le fichier provenant de l'exécution en C. Dans le répertoire bash, on trouve également les scripts exécutant nos tests Valgrind et Helgrind disponible avec les commandes *make valgrind* et *make helgrind*.

IV. Tests

Les tests unitaires et tests généraux sont lancés via la commande *make tests*.

A. Tests unitaires

Nous avons implémenté de nombreux tests unitaires avec CUnit tout au long de notre implémentation pour vérifier l'exactitude de nos fonctions, que ce soient les fonctions de distances, de distorsion, de l'algorithme complet, des générations de centroïdes de départ ou encore les fonctions de lecture et d'écriture des fichiers. Pratiquement toutes nos fonctions ont une fonction test associée. Pour réaliser ces tests, nous avons isolé les fonctions Pythons correspondantes afin d'obtenir les résultats attendus pour des inputs donnés. Ces tests étaient rigoureux et nous ont permis de vérifier la justesse du programme au fur et à mesure que de nouvelles fonctions étaient ajoutées. Cela nous a permis de gagner un temps considérable en plus d'avoir l'assurance que notre code fonctionnait.

B. Tests généraux

Nous n'avons pas implémenté uniquement des tests unitaires, nous avons également repris l'implémentation en Python dans notre code pour comparer des fichiers CSV. En effet, nous avons créé des scripts Bash qui nous permettent d'exécuter plusieurs fois les programmes Python et C en faisant varier les paramètres. Nous comparons le fichier de sortie provenant de l'exécution en Python avec le fichier de sortie provenant de l'exécution en C afin de vérifier si notre implémentation en C est correcte. Cet ajout, nous a semblé pertinent pour vérifier l'exactitude de notre fonction d'écriture mais également du programme tout entier de manière systématique.

C. make valgrind

La commande *make valgrind* permet de détecter des potentielles fuites de mémoires. Dans notre cas, nous compilons kmeans et nous testons celui-ci avec des paramètres statiques. Nous faisons cette opération avec et sans le paramètre "-q" afin d'être certain qu'aucune fuite de mémoire n'a lieu.

D. make helgrind

Comme son nom l'indique, cette commande vérifie qu'il n'y a pas de race conditions, *deadlock* et autres problèmes de concurrence dans notre programme. Elle utilise un paramètre dynamique, le nombre de threads, qui passe de 1 à 4 à 8. On teste également avec et sans le paramètre "-q". De cette manière, on s'assure qu'aucun problème de concurrence ne survient peu importe le nombre de thread.

E. make valgrindForTests

Cette commande a le même mode opératoire que la commande *make valgrind*. Elle détecte des potentielles fuites de mémoires pour les tests unitaires afin d'être sûr qu'aucune fuite n'ait lieu au cours de l'exécution des tests.

F. make performances

Afin d'avoir une mesure quantitative de nos performances, nous avons la commande *make performances*. Celle-ci, au moyen d'un script Bash et de Matplotlib en python, permet de tracer des graphes concernant le temps d'exécution de notre programme en fonction du nombre de threads donné en paramètre. Malheureusement, cette commande ne fonctionne pas sur Raspberry (elle est opérationnelle sur UDS) puisque la fonction *time* n'accepte aucun argument sur *Raspbian*.

G. Jenkins

Jenkins est un outil d'intégration : celui-ci nous informe, pour chaque *push* effectué sur Gitlab, si les *commits* correspondants passent les tests établis sur celui-ci. Cet outil est donc très utile pour nous faire gagner du temps et savoir si chaque *commit* est bon ou pas simplement en regardant si le *build* Jenkins a réussi ou échoué.

V. Implémentation multithread

A. Gestion de la concurrence

Pour l'implémentation multithread, nous avons opté pour une architecture producteurs-consommateurs. L'algorithme de Lloyd exécute de nombreuses fois un partitionnement des différents points avec à chaque fois des centroïdes de départ différents. Ces itérations sont indépendantes les unes des autres et peuvent donc être aisément mises en parallèle. Ce sont nos producteurs. Le nombre de thread producteur est déterminé par l'utilisateur.

Chaque producteur se voit assigner plusieurs instances du problème. Ces instances sont identifiables par un indice correspondant à une suite de centroïdes de départ reprise elle-même dans une liste. Les threads producteurs ont alors en charge la résolution des instances reprises dans le sous-tableau de suites de centroïdes de départ qui leur est attribué.

Il faut ensuite retranscrire chaque résultat dans un fichier CSV. Cette tâche ne peut pas être parallélisée et a été attribuée à notre unique thread consommateur.

1) Utilisation d'un buffer: Le buffer est le canal de communication entre les threads producteurs et le thread consommateur. Nous avons décidé d'implémenter un buffer de taille équivalente au nombre de thread producteur pour les raisons suivantes:

- Si le thread consommateur consomme les éléments sur le buffer plus lentement que les producteurs en produisent, il ne sert à rien d'enregistrer tous ces éléments. Autant se limiter à un buffer de petite taille et faire attendre les threads producteurs.
- Par contre, si le thread consommateur consomme les éléments plus rapidement que les producteurs en produisent, on veut impérativement éviter le ralentissement des threads producteurs. Le cas critique est celui où les producteurs finissent le calcul d'une instance de *k-means* au même moment. Afin d'éviter une quelconque attente supplémentaire pour les threads producteurs, il est nécessaire pour le buffer d'avoir une taille suffisante pour accueillir chacun des résultats des threads producteurs.

C'est pourquoi nous avons décidé de donner au buffer la même taille que le nombre de thread producteur.

2) Deux sémaphores: Un thread producteur peut vouloir déposer son résultat sur un buffer n'ayant plus aucune place disponible. Inversement, le thread consommateur peut tenter de récupérer un résultat sur un buffer vide. Ces problèmes sont évités grâce à l'utilisation des deux sémaphores suivants:

- full : renseigne les threads producteurs sur la présence d'au moins une place disponible sur le buffer.
- empty : renseigne le thread consommateur sur la présence d'au moins un élément à consommer sur le buffer.

3) Un mutex: Le mutex empêche l'utilisation du buffer par plus d'un thread en simultané. Sans cet élément, deux threads producteurs pourraient, par exemple, déposer leur résultat en même temps sur un même emplacement du buffer. Ce qui n'est évidemment pas souhaité.

VI. Analyse graphique des performances

Nous avons effectué nos tests sur un fichier contenant 50 000 points comprenant 5 dimensions. La métrique d'intérêt pour mesurer le bon fonctionnement de notre implémentation multithreadé est le *real time*.

A. Sur Raspberry

Nous pouvons voir que sur le Raspberry, le temps d'exécution (real time) est pratiquement divisé par 2 si l'on double le nombre de threads. La limite minimum du temps d'exécution est logiquement atteinte à partir de 4 threads. Notre commande *make performances* n'est pas fonctionnel sur Raspberry puisque la commande *time* n'a pas d'argument sur Raspian. Nous avons donc dû reprendre les résultats manuellement dans un fichier texte et ensuite à partir de ce fichier générer le graphe suivant:

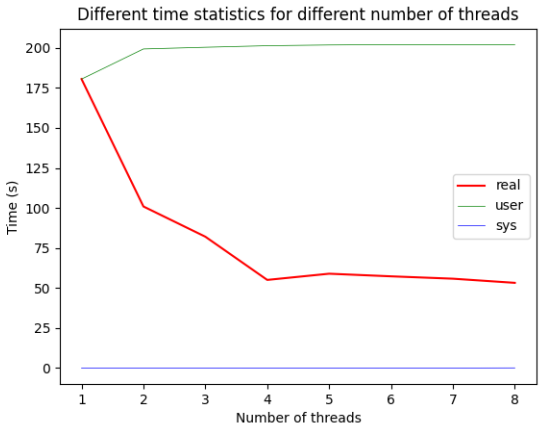

Fig. 1. Graphe de l'évolution du temps d'exécution en fonction du nombre de threads sur Raspberry

B. Sur PC

Lorsque nous lançons le programme sur nos PC personnels, il montre des temps largement inférieurs. Le gain de performance est visible jusqu'à 8 threads. Voici un exemple de graphe généré en lançant la commande *make performances*.

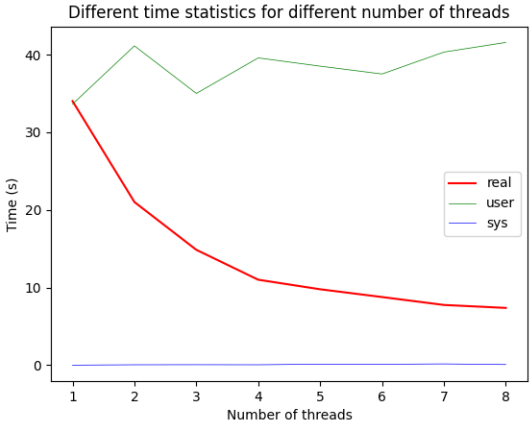

Fig. 2. Graphe de l'évolution du temps d'exécution en fonction du nombre de threads sur un PC grand public

VII. Contribution de chacun

A. Gilles Maes

Personnellement, je me suis surtout occupé de la lecture du fichier binaire et des tests associés. J'ai également codé le programme Python et le script Bash permettant de mesurer les performances de notre projet en les affichant sur un graphique.

B. Loic Spigeleer

J'ai contribué au projet de différentes manières, j'ai surtout écrit et optimisé la fonction d'écriture dans le fichier CSV et effectué des tests sur cette même fonction.J'ai également fortement contribué à l'ajout des threads pour notre architecture producteurs-consommateurs. La partie d'écriture étant la partie critique de l'algorithme au niveau des performances. Enfin, j'ai également aidé dans plusieurs fonctions à régler les problèmes rencontrés.

C. Nicolas Jeanmenne

Au niveau du code en C, je me suis occupé d'écrire la partie concernant la compilation, les tests unitaires, Valgrind et Helgrind dans le Makefile. J'ai également découpé les tests unitaires en plusieurs fichiers sources avec leur *header* associé et corrigé quelques erreurs de *link* avec le compilateur. Au niveau global du projet, j'ai mis en place l'outil d'intégrations Jenkins pour permettre de savoir si nos *commits* passaient les tests. Concernant les tests, je me suis aussi occupé de vérifier que notre code s'exécutait correctement sur Raspberry Pi.

D. Pierre Denoël

Pour ma part, j'ai tenté de faire l'écriture du fichier CSV avant que Loïc prenne la relève. Je me suis ensuite occupé de la clarté de notre code : l'homogénéisation des noms de variables, fichiers et fonctions et la descriptions des fonctions si besoin. J'ai aussi effectué des tests sur UDS et participé à l'écriture du README.

E. Samuel de Meester

Je me suis personnellement concentré sur l'implémentation de l'algorithme de Lloyd. Mais j'ai également implémenté la fonction permettant la génération des différentes suites de centroïde de départs. À côté de cela, je me suis aussi investi dans la partie test pour être certain que l'implémentation de l'algorithme soit correcte. J'ai beaucoup aimé ce projet donc j'étais assez impliqué dans la dynamique de groupe. Je portais mon aide à droite et à gauche et cela m'a donné une bonne vue d'ensemble du projet utile lors de nos discussions de groupe. J'ai donc également participé à l'implémentation de l'architecture producteurs-consommateurs. Pour finir, j'ai aussi contribué à l'utilisation du langage Bash dans nos tests afin de comparer le résultat obtenu par notre implémentation en C avec le résultat obtenu en Python.

F. Sébastien Mary

En ce qui me concerne, j'ai touché à la plupart des différentes parties du projet, que ce soit au niveau de la structure du projet ou de détails techniques d'implémentation. Je me suis moins occupé de la partie threads et synchronisation pour me concentrer sur le design et la cohérence du projet. Concrètement, j'ai écrit et corrigé plusieurs fonctions, tests ainsi que le README. Je me suis également appliqué à ce que notre projet réponde aux spécifications demandées.

VIII. Acquis d'apprentissage

Par ce projet, nous avons évidemment pu améliorer nos capacités à utiliser le langage C, mais l'expérience nous a appris plus encore.

Tout d'abord, l'organisation du travail dans un grand groupe. Avec une équipe de 6 personnes, il n'est pas toujours évident de se coordonner. Il a pourtant bien fallu savoir diviser le travail, parfois en sous-équipes pour les tâches importantes, ou individuellement. Mais ce n'est pas tout. Comme nous apprenons chacun de notre côté et que nous trouvons tous différentes solutions et idées, la communication doit être constante. Sans une communication assidue rendue possible par de nombreuses réunions pour partager nos découvertes, il y aura forcément des membres de l'équipe à la traîne. Pour que tout le monde puisse avancer ensemble, il faut communiquer, écouter et travailler.

Ensuite, nous avons vite compris l'intérêt d'une bonne maîtrise de Git. Ses différentes fonctionnalités facilitent fortement la construction d'un projet avec de nombreuses composantes et personnes. Par exemple, la possibilité de créer

une branche permettant le développement d'une extension périlleuse. Cela permet de ne jamais perdre le travail de quelqu'un, garder une branche ne contenant que les parties du code fonctionnelles et déjà testées, travailler simultanément sur un même fichier, etc. En résumé : la maîtrise de cet outil est essentielle à l'efficacité du groupe.

En outre, nous avons aussi appris à utiliser un outil de tests unitaires pour C qui se nomme CUnit [4] et l'outil d'intégration Jenkins. Nous avons aussi appris à utiliser Valgrind [3]. Nous avons pris soin de tester notre code tout au long de son écriture et cela nous a permis d'être assez peu bloqué sur des erreurs sans en trouver l'origine. Une habitude à garder peu importe la taille du projet.

Avec ce projet, nous avons également appris des rudiments de Bash ou encore découvert le Makefile.

IX. Conclusion

En conclusion, nous avons transcrit le problème de *clustering* dit *k-means* du langage Python en langage C. Nous avons, par conséquent, amélioré les performances de l'algorithme de Lloyd de manière significative. Ce projet nous aura appris à travailler en groupe et à travailler proprement avec une structure claire correctement documentée.

De plus, le projet nous aura permis de nous familiariser avec des outils de collaboration et d'intégration tels que Gitlab et Jenkins. Enfin, nous avons implémenté des ajouts nous permettant de nous initier au langage Bash. Ces ajouts permettent d'obtenir aisément une comparaison des temps d'exécution pour différents nombres de threads. Ils permettent également de vérifier si une fuite de mémoire a lieu ou si un problème lié à l'utilisation de la programmation concurrente se produit. En fin de compte un projet riche en apprentissage !

References

[1] (2021) Lloyd's algorithm. [Online]. Available: https://en.wikipedia.org/wiki/Lloyd%27s_algorithm

[2] (2021) git. [Online]. Available: https://git-scm.com/

[3] (2021) Valgrind. [Online]. Available: https://www.valgrind.org/

[4] A. Kumar and J. St.Clair. (2005) Cunit. [Online]. Available: http://cunit.sourceforge.net/doc/index.html