



ECOLE POLYTECHNIQUE DE LOUVAIN

LEPL1503 - PROJET 3

K-Means Problem

C implementation on a Raspberry Pi

Hauret	Antoine	2492-1900
Ingenzi	Vany	8046-1900
Logeot	Bastien	8347-1800
Manzi-Mihigo	Jonathan	1063-1800
Ndayisaba Shima	Ingabire Pascale	2669-1800
Sanchez Alvarez	Kimberly	3308-1800

LOUVAIN-LA-NEUVE, 2021

Rapport

Groupe B13
Université Catholique de Louvain

Abstract—Ce rapport présente une analyse détaillée du projet pour le cours LEPL1503 dont le sujet principal était l'implémentation efficace du K-means Clustering en multi-threading sur des systèmes d'exploitation Linux.

I. Introduction

Durant la première moitié du quadrimestre, nous avons appris à programmer dans le langage de programmation C^1 . Cet apprentissage nous a permis de comprendre les possibilités et les différentes subtilités de ce langage de programmation.

A. Énoncé du projet

L'énoncé de ce projet était que pour un programme écrit en Python résoudre le K-means Clustering avec l'algorithme de Lloyd, nous devons en faire une version écrite en C . Ces derniers seront expliqués dans le chapitre suivante.

Le programme codé en C doit implémenter aussi l'algorithme de Lloyd, qui pour les mêmes données d'entrées doit arriver au même résultat que le code en Python. De plus, notre programme doit être significativement plus rapide que ce dernier, en exploitant les ressources de l'ordinateur.

Il doit aussi être capable de tourner sur plusieurs systèmes d'exploitation de la famille LINUX dont obligatoirement les suivants:

- Fedora
- Raspbian GNU/Linux 10

La possibilité d'utiliser plusieurs threads afin d'accélérer davantage le programme en parallélisant l'exécution des tâches sur les multiples cœurs de la machine choisie, est encore un avantage que nous offre C par rapport à Python.

Enfin, notre programme doit écrire le résultat du K-means Clustering dans un fichier csv.

II. K-means Clustering

Avant de parler de la réalisation du projet nous allons expliquer ce qu'est le problème K-means Clustering : Étant donné un ensemble des points et un entier k , le problème est de diviser et de rassembler les points en k groupes distincts, appelés *clusters*. Ce problème est utilisé dans des nombreux domaines allant de la compression d'image à l'analyse marketing.

¹ C est un langage de programmation impératif généraliste, de bas niveau. Inventé au début des années 1970 pour réécrire UNIX.

A. L'algorithme de Lloyd

Il existe plusieurs algorithmes pour résoudre ce problème. Dans ce projet, nous utiliserons l'algorithme de Lloyd proposé par Stuart Lloyd dans cette publication [1] en 1957.

Cet algorithme divise ces points en k clusters (ou "groupes"), réduisant le plus possible la distance entre les différents points du cluster et le point central, appelé *centroïde*. Pour mesurer la qualité de nos clusters, l'algorithme utilise une métrique de distorsion qui est définie comme la somme du carré des distances entre chaque point et le centroïde de son cluster.

La somme du carré des distances entre deux points, peut être calculée de deux manières différentes. Soient deux points $(x_0, x_1, \dots, x_{i-1})$ et $(y_0, y_1, \dots, y_{i-1})$ dans un espace de dimension i . Nous pouvons la calculer en utilisant le carré de la **distance euclidienne** :

$$\sum_{n=0}^{i-1} (x_n - y_n)^2 \quad (1)$$

ou en utilisant le carré de la **distance de Manhattan**:

$$\left(\sum_{n=0}^{i-1} |x_n - y_n| \right)^2 \quad (2)$$

Dès lors pour un *cluster* donnée, C , et son *centroïde*, p , nous pouvons calculer sa distorsion de avec la formule suivante.

$$\sum_{0 \leq i \leq |C|} \left(\sum_{p \in C_i} \text{squared_distance}(p, C_i) \right) \quad (3)$$

Où $\text{squared_distance}(p, C_i)$ est une de deux manières de calculer la somme du carré des distances vu dans les formules (1) et (2).

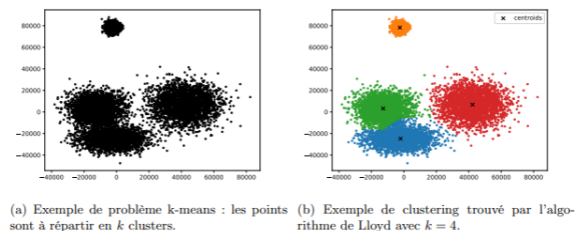


Fig. 1. La figure 1 illustre un exemple d'application de l'algorithme de Lloyd. L'algorithme de Lloyd reçoit un ensemble de points et un nombre k en entrée et répartit les points en k clusters. Tiré du PDF de l'énoncé du projet

Algorithm 1 Lloyd's algorithm.

Require: \mathcal{P} , the set of points
Require: k , the number of clusters to determine
Require: \mathcal{I} , the set of initial centroids
Require: \mathcal{C} , the set of clusters
Require: *distance*, the function computing the distance between two points

```

1:  $oldcentroids \leftarrow \emptyset$ 
2:  $centroids \leftarrow \mathcal{I}$  ▷ Initialization
3: while  $oldcentroids \neq centroids$  do
4:    $oldcentroids \leftarrow centroids$ 
5:   for  $i : 0 < i < |\mathcal{C}|$  do ▷ reset the current clusters
6:      $\mathcal{C}_i \leftarrow \emptyset$ 
7:   end for
8:   for  $p \in \mathcal{P}$  do ▷ Assign the points to their closest cluster
9:      $i \leftarrow \text{argmin}_{j} \text{distance}(p, centroids_j)$  ▷ Find the index  $i$  of the closest centroid
10:     $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{p\}$  ▷ Assign  $p$  to its closest cluster
11:   end for
12:   for  $i : 0 < i < |\mathcal{C}|$  do ▷ Compute the new centroids by doing an average
13:      $centroids_i \leftarrow \frac{\sum_{p \in \mathcal{C}_i} p}{|\mathcal{C}_i|}$ 
14:   end for
15: end while
16: return  $\mathcal{C}$ 

```

Fig. 2. L'algorithme de Lloyd en pseudo code. Tiré du PDF de l'énoncé du projet [3]

Il est très important de noter que l'algorithme de Lloyd a besoin, d'un ensemble des centroïdes initial. Cette partie est essentielle et va être cruciale pour pouvoir paralléliser l'exécution de cet algorithme efficacement.

III. Structure globale du programme

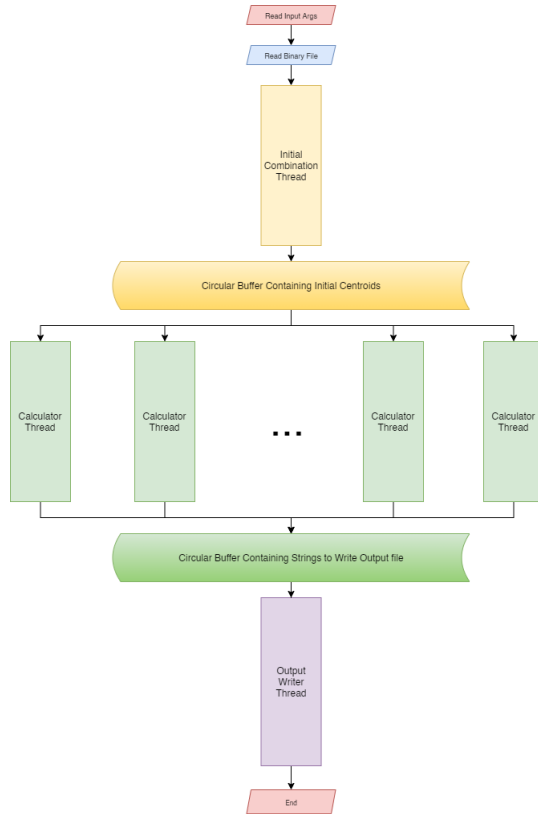


Fig. 3. L'architecture du programme. Les rectangles représentent les threads. Les parallélogrammes représentent les exécutions exécutées par le thread principal.

Tout comme les arguments, le fichier binaire d'entrées est lu par le thread principal de façon séquentielle. En effet, cette lecture est très rapide et il n'y a donc que peu d'intérêt à la paralléliser, contrairement aux autres parties qui nécessitent l'entièreté des points d'entrée, ce qui pourrait les faire attendre.

Dès lors notre programme est divisé en trois parties parallèles visibles dans la figure 3.

- Combinaisons des centroïdes .
- K-means Clustering.
- Écriture des résultats .

A. Combinaisons des centroïdes

L'algorithme de Lloyd, expliqué dans le chapitre (II), a donc besoin des centroïdes initiaux pour procéder à la suite du calcul. Nous avons attribué cette partie à un thread ayant la plus haute priorité selon la politique utilisée par le *scheduler*, il est appelé **Initial Combination Thread**. Celui-ci va calculer toutes les combinaisons de taille k , à partir d'une liste de p points, à condition que $p \geq k$.

B. K-means Clustering

Cette partie du programme va exécuter l'algorithme de Lloyd et classer l'ensemble des points. Le nombre des threads exécutant cette partie dépend de l'utilisateur. Par défaut, ce nombre est initialisé à 4. Dans la figure 3, ces threads sont appelés **Calculator Threads**.

C. Écriture des résultats

Un thread spécifique gère l'écriture des résultats calculés par les threads antérieurs dans un fichier .csv. Il est appelé **Output Writer Thread** et sera aussi prioritaire, les choix concernant la priorité vont être expliqués en détail dans un chapitre ci-dessous.

IV. Techniques de synchronisation des threads utilisées

Dans ce chapitre nous verrons toutes les techniques et structures de données utilisés pour la synchronisation des threads.

A. Priorité des threads

Il est important de souligner que les threads utilisés dans ce projet sont des threads POSIX. Nous utilisons la librairie *pthread*.

Le *scheduler* est un composant du noyau d'un système d'exploitation, il choisit l'ordre d'exécution des processus sur les processeurs de l'ordinateur [2]. Il y a plusieurs manières d'implémenter un *scheduler*, nous les appelons : Politiques du *scheduler*. Pour plusieurs politiques, il y a différents niveaux de priorités. Plus un thread a une grande priorité, plus il sera choisi par le *scheduler* pour continuer son exécution.

La priorité des threads n'est pas anodine, la raison est que nous avons besoin que ces derniers terminent leur travail le plus vite possible pour éviter l'attente des autres threads. Les deux threads qui bénéficient de cette configuration sont **Initial Combination Thread** et **Output Writer Thread**. Le premier génère des combinaisons des centroïdes qui sont *sine qua non* au calcul de l'algorithme de Lloyd et le second doit libérer de la place dans le buffer pour que les **Calculator Threads** puissent écrire le résultat de leur calculs.

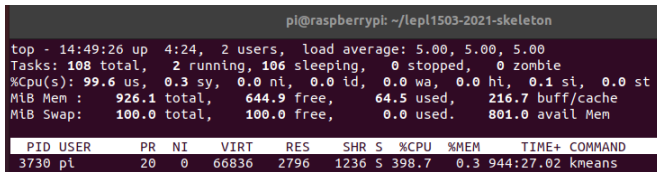
B. Nombres total des threads

Dans notre architecture le nombre total de threads n'est pas celui que l'utilisateur entre comme argument lorsqu'il exécute le programme. En effet, il faut faire attention au fait que notre programme est toujours multi-threadé car il y a toujours le **Initial combination thread** et le **Output writer thread** qui sont exécutés en parallèle. Dès lors, le nombre des threads est donné par $n + 2$, avec $n \geq 1$, où n est le nombre des threads qui exécutent l'algorithme de Lloyd. Cette décision n'impacte pas négativement le temps d'exécution, au contraire, celle-ci permet d'avoir toujours un programme rapide étant donné que le Raspberry est un système quad-core.

C. Buffers, Sémaphores et Mutex

Les buffers sont des structures de données utilisées pour stocker temporairement des données entre deux ou plusieurs threads. Les buffers peuvent prendre toute forme, que ça soit des listes dynamiques, listes chaînées etc...

Pour notre programme nous avons opté pour un buffer circulaire à taille fixe. Le Raspberry Pi 3 étant un ordinateur embarqué à petite mémoire (1 Giga Byte) nous avons voulu nous assurer que la consommation de la mémoire soit constante et minime tout au long de l'exécution de notre programme. Ce buffer nous permet de faire tourner le programme avec un très grand nombre des combinaisons, sans crainte et sans perte de vitesse pour autant. La figure 4 présente une capture d'écran d'une consommation.



```

pi@raspberrypi: ~/lepl1503-2021-skeleton
top - 14:49:26 up 4:24, 2 users, load average: 5.00, 5.00, 5.00
Tasks: 108 total, 2 running, 106 sleeping, 0 stopped, 0 zombie
%Cpu(s): 99.6 us, 0.3 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.1 st, 0.0 sr
MiB Mem : 926.1 total, 644.9 free, 64.5 used, 216.7 buff/cache
MiB Swap: 100.0 total, 100.0 free, 0.0 used, 801.0 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
 3730 pi        20   0 66836 2796 1236 S 398.7  0.3 944:27.02 kmeans

```

Fig. 4. Par exemple, pour un fichier de 5320 points de 2 dimension, nous voulons 10 clusters avec 100 points d'initialisations, nous avons 17310309456440 combinaisons possibles. Cet exemple montre que le programme ne consomme que 0.3% de MEM, qui équivaut à ~ 200 Kilo bytes et cette consommation reste constante. Cette capture a été faite 4 heures après le lancement du programme.

Ces buffers circulaires ont été implémentés en utilisant des structures et des tableaux en C. Chaque buffer possède des attributs mais les plus essentiels sont les deux sémaphores et un mutex propre à chaque buffer. Ce mutex est utilisé pour la synchronisation des threads, un sémaphore pour la gestion des threads qui viennent écrire dans le buffer en question et un autre sémaphore pour la gestion de ceux qui viennent lire dans le buffer. Le mutex sera utilisé pour chaque endroit nécessitant l'exclusion mutuelle propres aux fonctions qui concernent le buffer et uniquement ce buffers.

D. Design (Producer / Consumer Pattern)

Pour notre pattern de concurrence nous avons adopté le pattern "Consommateur/producteur". Cela signifie que pendant que certains threads produisent, d'autres consomment ce qui

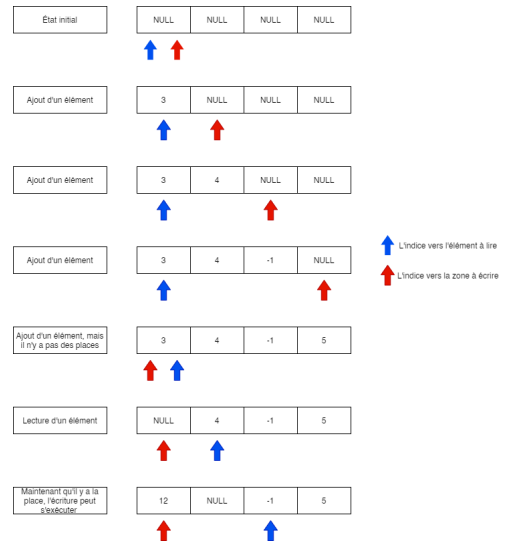


Fig. 5. Un exemple d'une suite d'exécutions sur le buffer circulaire.

à été produit. Les consommateurs et producteurs partageront dès lors un buffer circulaire.

Cette architecture est présente à deux niveaux d'étages dans notre programme, comme nous pouvons le voir sur la figure 3 :

- 1) *The Initial Combination Thread* produit les centroïdes initiaux qui seront consommés par les threads calculateurs, pour lancer l'algorithme de LLoyd.
- 2) *The Calculator Threads* produisent les outputs après avoir calculé les derniers centroïdes, et clusters, ces résultats vont être consommés par *The output writer Thread*.

Nous avons aussi ajouté une tâche pour les threads producteurs, lorsque ceux-ci ont fini de produire, ils envoient un signal aux threads consommateurs afin de réveiller tous ceux qui attendent pour la lecture tout en empêchant d'autres d'attendre pour rien.

Tous les threads travaillant sur le même buffer envoient un signal, en cas d'erreur, aux autres threads avec lesquels ils interagissent. Une fonction s'occupe de gérer cette partie et libère les threads en cas d'erreur.

V. Analyse temporaire

Afin de réaliser l'analyse temporaire, nous devons garder en mémoire le nombre total des threads qui sont exécutés dans notre programme, ceci est expliqué dans le sous-chapitre IV-B. Toutes les données utilisées pour cette analyse sont disponibles dans notre répertoire.

Pour l'analyse nous avons testé les performances de notre code sur le raspberry pi, avec différentes données. Nous avons utilisé deux ensembles de données, le premier contient 8320 points et l'autre 5320 points. Nous avons vérifié le temps que prend le programme en C pour s'exécuter à chaque variation. Les variables k , n et p varieront de la manière suivante :

- k , le nombre des clusters = [1,3,5,7].

- n , le nombre des threads qui exécutent l'algorithme de Lloyd = [1,2,3,4].
- p , le nombre des éléments parmi lesquelles il faut faire des combinaisons de centroïdes = [13,15,17,20].

Au final nous avons eu 112 exécutions différentes de notre code C, toutes exécutées en utilisant la commande *time* et le *makefile*.

A. Comparaison temporaire entre le nombre des threads

Étant donné que quelques minutes peuvent séparer les exécutions, nous avons utilisé la médiane pour pouvoir une idée de la tendance centrale. La figure 6 montre parfaitement cette dernière, et nous pouvons remarquer que lorsque le nombre des threads qui calculent l'algorithme de Lloyd est augmenté, le temps diminue significativement. Cette observation reflète effectivement la réalité lorsque nous observons les données.

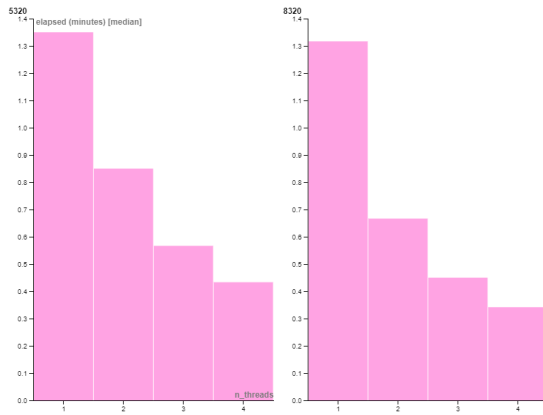


Fig. 6. La médiane temporaire en fonction du nombre de threads qui exécutent l'algorithme de Lloyd exécuté sur le fichier avec 5320 points.

B. Comparaison temporaire entre le nombre des clusters

Cette comparaison a été faite pour avoir l'aperçu de l'augmentation très significative des calculs lorsque nous augmentons le nombre des clusters. En effet, pour un ensemble de 17 points, si nous voulons 5 clusters, nous avons 3003 points mais en augmentant le nombre des clusters à 7, nous avons 6435. En diminuant le nombre de clusters à 2 nous avons 105 combinaisons pour le même ensemble. Ces résultats sont obtenus en utilisant le calcul du coefficient binomiale. La figure 7 montre cette analyse. Malgré cette variation assez importante notre code minimise l'écart de temps d'exécution, comme nous pouvons le voir cette différence est de 7 minutes dans les cas extrêmes, elle est de 4 minutes en moyenne.

C. Comparaison avec le Raspberry Pi

Voici la comparaison du temps d'exécution en moyenne après 4 exécutions, entre notre programme par rapport au programme version python avec un fichier de 2340 points.

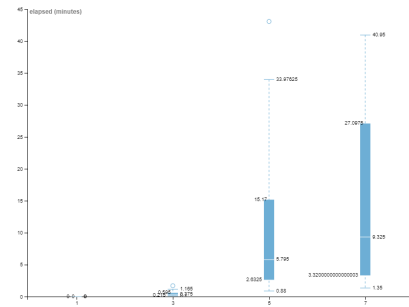


Fig. 7. La moyenne du temps d'exécutions en fonction du nombre des clusters pour les mêmes arguments, pour un fichier de 5320 points.

arguments	version Python	version C
k = 3 et p = 7	~ 56 sec	0.612 sec
k = 3 et p = 13	6 min 12 sec	4.407 sec

VI. Limitations

Nous avons travaillé de sorte à minimiser le plus possible des limitations liées à notre implémentation. Néanmoins, nous avons eu une seule limitation:

- **Fichier d'input.** Si le fichier d'entrée est plus volumineux que la mémoire RAM disponible, notre programme ne s'exécutera pas convenablement. Ce problème ne peut pas être résolu étant donné que nous devons utiliser l'algorithme de Lloyd, qui a besoin des toutes les points pour faire la classification.

Il faudra noter que si l'utilisateur utilise un nombre des threads plus grand que celui supporté par le système nous allons continuer l'exécution avec le nombre des threads acceptable.

VII. Tests Unitaires

A. CUnit Test

Le but principal de nos tests est de vérifier si les modules que nous avons définis dans headers/ et implémentés dans src/ fonctionnent comme ils le devraient. Pour cette raison, la plus part des fichiers dans src a un test éponyme. Il est important de noter qu'il est possible qu'un fichier src dépende d'un autre fichier pour être pleinement fonctionnel, un tableau des dépendances est disponible dans notre README. Nous utilisons également le code python donné pour tester la sortie de la fonction entière. Et la vitesse du main.c si nous augmentons le nombre de threads sur des systèmes multi-cœurs.

Nous avons utilisé CUnit pour pouvoir tester nos fonctions.

B. Valgrind

Nous avons utilisé Valgrind pour vérifier si notre programme ne comporte pas des fuites de mémoires. De plus, Valgrind possède aussi d'autres outils comme Helgrind et Massif qui ont contribué à vérifier le bon fonctionnement du projet. Le premier a été utilisé pour détecter les data-races possibles et pour voir la consommation de mémoire utilisée par le programme. Massif nous a permis de réduire significativement la consommation du programme.

C. CPP Check

Cpp check a été utilisé pour détecter des comportements pouvant être dangereux dans le code source.

VIII. Améliorations suite aux peer-review

La phase des peer-reviews a permis de prendre conscience de certaines erreurs. Les messages reçus ont été très positifs avec une note globale de 4.5/5 en moyenne. Voici quelques remarques importantes et les améliorations correspondantes:

- Notre README, comporte beaucoup d'humour ce qui n'était pas professionnel de notre part.
- Dans le fiche de sortie il manquait un guillemet.
- Dans certaines exécutions, Helgrind détectait de possibles data races. Ces dernières avaient lieu aux endroits où nous appellions "printf", nous avons remplacé cette fonction par "fprintf".

La phase de peer-reviews nous a permis de critiquer notre code, d'un point de vue externe. Dès lors nous l'avons amélioré comme suit :

- Avant les résultats étaient transformés en chaîne de caractères par les **Calcutor Threads** avant d'être stockés dans le buffer, afin être écrits par le thread d'écriture. Ceci ralentissait significativement le code lorsque les points d'entrées étaient assez importants. Nous avons donc choisit de supprimer cette étape, de cette façon les résultats sont castés en chaîne de caractère au moment d'écriture seulement.
- Dans la fonction "assign_vectors_to_centroids", nous appelons "realloc" à chaque fois que nous voulons ajouter un point dans un cluster. La fonction "realloc" est une fonction assez coûteuse ayant une complexité de $\Theta(n)$, où n est le nombre des bytes de la zone mémoire que nous voulons élargir. Nous avons alors implémenté cette ajout de point de la même manière que langages haut niveaux comme Python implémentent les listes dynamiques. C'est à dire que nous allouons une mémoire initiale, si elle est pleine, nous appelons realloc à nouveaux mais cette fois-ci en multipliant par deux la taille du tableau. De cette façon pour la même tâche, nous avons une fonction en $O(n)$ et il s'avère que l'analyse amortie montre que l'ajout d'un élément a une complexité de $O(1)$.
- Enfin, nous avons aussi fait très attention à la gestion des erreurs. En vérifiant tous les appels de fonctions nécessaires, en donnant un feedback correct à l'utilisateur en cas d'erreur et en restructurant le code de façon plus lisible et correcte.

IX. Conclusion

Pour conclure, nous sommes arrivés à terminer le projet et, à nos yeux, obtenir un résultat très satisfaisant. Grâce au travail collectif et l'aide de notre tuteur, nous avons su travailler efficacement et terminer le travail dans les délais imposés.

Ce fut un très bon exercice d'application de la matière de C vue précédemment et un bon premier contact avec l'utilisation de la parallélisation (threads, mutex, buffer, ect...) dans ce

langage. Chaque membre, grâce à ce projet, a pu apprendre beaucoup, tant par la théorie que par l'entraide dans le groupe.

Malgré le fait d'être majoritairement à distance, nous avons pu travailler correctement. Nous avons appris à travailler dans des circonstances particulières, nous coordonner et communiquer. Cela nous sera sûrement très utile pour le futur.

Contribution de chaque membre:

Antoine : Génération des points initiaux. Vérification du programme Uni Thread et fonctionnement avec les différents fichiers inputs. Contribution à la première version des tests unitaires, à la structure du rapport et sa rédaction. Analyse de performance entre le code C et le code Python.

Bastien : Génération des points initiaux. Contribution à la première version des tests unitaires, à la structure du rapport et sa rédaction.

Ingabire Pascale : Contribution à la rédaction des fonctions de distance et de la fonction kmeans. Contribution à la première version des tests unitaires, à la structure du rapport et sa rédaction.

Jonathan : Contribution à la structure du ReadMe et au MakeFile et leur rédaction. Contribution à la rédaction des fonctions de distance et de la fonction kmeans. Structure du code et implémentations. S'est assuré que la version finale du code tourne sur le Raspberry et tests de performances.

Kimberly : Contribution à la rédaction des fonctions de distance. Gestion des fichiers d'entrée et de sortie. S'est assurée que la version finale du code tourne sur le Raspberry et tests de performances. Implémentation et Architecture distribuée.

Vany : Organisation du groupe, répartition des tâches. Structure du code et implémentations. Implémentation et Architecture distribuée. Contribution à la première version des tests. Analyse de performance entre le code C et le code Python. Correction du code suite aux peer-reviews.

References

- [1] Stuart Lloyd. *Least Squared quantization in pcm*. IEEE transactions on information theory, 28(2):129-137, 1982.
- [2] Wikipedi. *Scheduling* [https://en.wikipedia.org/wiki/Scheduling_\(computing\)](https://en.wikipedia.org/wiki/Scheduling_(computing))
- [3] Olivier Bonaventure, Axel Legay, Mathieu Jadin, François Michel, Sébastien Strebelle et Tom Rousseaux. *Enonce du projet*. https://moodleucl.uclouvain.be/pluginfile.php/2227387/mod_resource/content/1/lepl1503_enonce_2021.pdf