Rapport Projet 3

A. Antonutti, M. Delsart, E. Michiels, P. Pasture, L. Rauw, U. Reinbold

I. Introduction

Ce rapport a pour objectif de décrire notre implémentation C et les améliorations (algorithmiques et architecturales) apportées par rapport au programme Python. Nous expliciterons nos structures de données, nous ferons une description brève de nos tests unitaires et des outils utilisés. Une analyse des performances des 3 métriques principales sera également effectuée afin de comparer les versions Python et C en single thread et en multi-threads. Pour conclure, nous expliquerons la dynamique de groupe.

II. Implémentation C

A. *Description*

Notre implémentation C réalise l'algorithme de Bellman-Ford à plusieurs reprises sur un graphe afin de connaître tous les plus courts chemins les plus longs à partir de tous les noeuds.

Pour réaliser cela, nous avons divisé le programme en plusieurs modules ayant chacun un nom clair et un rôle précis.

Tout fichier source contient son propre fichier header. Cependant, il y a 4 fichiers headers qui n'ont pas de fichier source associé:

- *./include/unassociated/macro.h* ⇒ Réuni toutes les macros utilisées dans tout le code.
- *./include/unassociated/portable_endian.h* ⇒ Permet la conversion du format des nombres binaires (format "Big Endian", "Little Endian" et "Host").
- *./include/unassociated/portable_semaphore.h* ⇒ Permet l'utilisation (sur tout OS) des sémaphores pour synchroniser les threads (Pas utilisé pour notre version finale).
- *./include/unassociated/struct.h* ⇒ Regroupe dans un seul fichier header toutes les structures utilisées dans tout le code. Cela évite d'avoir des dépendances circulaires en important beaucoup de modules uniquement pour avoir accès aux structures et non aux fonctions.

Les fichiers sources sont les suivants:

- *./src/bellman-ford.c* ⇒ Algorithme et retrace les chemins (post-algorithme).
- *./src/files.c* ⇒ Lecture et écriture des fichiers binaires.
- *./src/display.c* ⇒ Affiche les erreurs et les messages pour le mode -v (si activé). Tous les printf() se trouvent dans ce fichier source.
- *./src/struct_initializer.c, ./src/struct_creater.c et ./src/struct_free.c* ⇒ Initialise, créé et libère la mémoire de toutes les structures se trouvant dans *./include/unassociated/struct.h*.
- *./src/threads.c* ⇒ Regroupe les fonctions permettant de lancer les threads et de les attendre mais aussi la fonction utilisée par les threads pour effectuer l'algorithme.
- *./tests/src/verifyOutput.c* ⇒ Permet de vérifier si deux fichiers binaires sont identiques (même taille et même contenu) [Utilisé uniquement pour les tests].
- *./tests/src/sort_bin_files.c* ⇒ Permet de remettre un fichier binaire dans l'ordre croissant des noeuds sources (sans changer la structure des blocs de résultats!) [Utilisé uniquement pour les tests finaux (= test de validation) en multithreadé].

Il y a également tous les fichiers tests qui ont exactement la même structure. Plus de détails à ce propos sont donnés dans la section IV.

B. *Complexité temporelle*

La complexité de notre implémentation est de $O(|V^2| * |R|)$ où $|V|$ est le nombre de noeuds du graphe et $|R|$ le nombre de liens du graphe. La raison de cette complexité temporelle est que nous appliquons $|V|$ fois l'algorithme de Bellman-Ford. Cela signifie que le temps d'exécution du programme varie d'autant plus avec le nombre de noeuds qu'avec le nombre de liens du graphe. Ce détail aura son importance dans la section VI.

C. *Amélioration algorithmique*

Nous avons amélioré l'algorithme de Bellman-Ford afin que celui-ci s'arrête dans le cas où lors d'une itération complète, aucun changement des coûts n'est effectué. En effet, dans ce cas, cela ne sert à rien de continuer puisqu'aucune modification de coût ne sera appliquée lors des prochaines itérations. Cette petite modification diminue de manière significative le temps d'exécution pour des graphes très volumineux et très peu connectés.

Nous avons utilisé de manière efficace un maximum de pointeurs pour privilégier le passage par référence et non par copie. Cela améliore principalement l'utilisation de la mémoire mais aussi le temps d'exécution si l'appel de fonction se fait de manière repétée.

Nous vérifions également chaque possibilité d'échec pour toutes fonctions utilisées (les appels système, nos fonctions si elles renvoient quelque chose,...). Cela permet au programme d'être plus robuste et de ne pas crash si une erreur comme cela se produit.

D. *Améliorations liées à l'architecture*

Nous avons implémenté une architecture multithreadée afin de permettre l'utilisation simultanée de plusieurs coeurs d'un processeur.

Ce processus a de nombreux avantages en terme de performances comme nous le verrons dans la section VI.

Nous avons utilisé différents threads afin de répartir le travail de la manière la plus équitable possible. Les threads s'occupent de l'algorithme de Bellman-Ford en décomposant les $|V|$ itérations en N parties, N étant le nombre de threads.

Pour rendre cela le plus optimal possible, nous avons créé une structure commune aux threads (Section III). Cela nous permet de

ne pas devoir recréer inutilement les mêmes attributs pour chaque threads.

Dans notre implémentation, les threads sont uniquement synchronisés pour écrire dans le fichier binaire de sortie et/ou afficher les résultats dans la console stdout. Ceci est réalisé à l'aide d'un seul mutex se trouvant dans la structure commune aux threads. Les threads ne communiquent jamais entre eux lors de l'exécution du programme.

Nous avions également testé de faire un producer/consumer mais le temps d'exécution ainsi que le total de mémoire allouée étaient nettement moins bons. En effet, il fallait copier chaque structure afin de la mettre dans un buffer, ce qui prenait du temps et surtout beaucoup de mémoire inutilement. Nous avions aussi tenté de lire le fichier binaire d'entrée avec N threads mais cette méthode était deux fois plus lente malgré tous nos efforts pour la rendre plus efficace.

Nos tests sur le Raspberry Pi nous ont montré qu'un nombre de 4 threads permet une optimisation des coeurs du processeur (qui sont au nombre de 4). Au-delà de ce nombre, nous n'observons plus aucun gain de temps. Nous observons au contraire, une nouvelle augmentation du temps utilisé qui est due à la création de threads supplémentaires non utilisables. Ce phénomène est observé à la section VI-A pour de petits graphes. En testant sur un ordinateur avec un processeur contenant 8 coeurs, on a pu observer que le nombre de threads optimal était de 8. Ceci nous prouve que notre architecture est bien fonctionnelle.

III. Structures de données

Plusieurs structures ont été nécessaires au bon fonctionnement de notre programme C. Celles-ci se trouvent toutes dans le fichier *./include/unassociated/struct.h* avec des descriptions très détaillées. Les trois premières structures nous ont été utiles dès le début du projet avec la version en single thread. Cependant, les deux dernières ont été ajoutées par la suite afin de supporter l'implémentation en multi-threading. Nous avons changé la structure *arg_t* initialement présente dans le squelette du code C lors du passage en multithreadé car elle ne nous était plus utile.

A. *edge_t :*

Cette structure représente l'arrête d'un graphe. Elle a comme attributs le noeud source, le noeud destination et le coût de ce lien.

B. *graph_t :*

Cette structure preprésente le graphe d'entrée. C'est en quelque sorte la traduction du fichier binaire d'entrée en une structure simple et facile d'accès avec des nombres non binaires. La structure *graph_t* n'est initialisée qu'une seule fois pour tout le programme.

Elle est composée du nombre total de noeuds et de liens compris dans le graphe ainsi que d'un pointeur vers la structure *edge_t*. Nous pouvons donc connaître toutes les informations sur une arête du graphe en effectuant "my_graph→edge[nber_lien]".

C. *outputGraph_t :*

Cette structure représente un bloc de données qui sera écrit en résultat dans le fichier binaire de sortie. Les données présentes à l'intérieur de cette structure seront directement écrites dans le fichier binaire puis réinitialisées pour l'itération suivante. Une structure *outputGraph_t* est nécessaire pour chaque thread. C'est donc la traduction des résultats de l'algorithme de Bellman-Ford (et de la recherche du plus long chemin parmis les plus courts) en une structure simple et facile d'accès.

La structure est composée de plusieurs attributs:
- Le nombre de noeuds dans le graphe d'entrée.
- Le noeud sur lequel on applique l'algorithme de Bellman-Ford en tant que noeud source.
- Le coût total du chemin le plus long parmi tous les chemins les plus courts (trouvés par l'algorithme) entre le noeud source et le noeud destination.
- La longueur de ce chemin.
- Le noeud correspondant au noeud final de ce chemin partant du noeud source. Il est appelé le noeud destination.
- un pointeur de nombre entier correspondant aux noeuds formant ce chemin (du noeud source au noeud destination).

La seule information qu'il nous manque si le flux du fichier de sortie est stdout est un pointeur de tableau nommé distance, reprenant les distances du noeud source vers tous les autres noeuds (où l'index = le noeud destination). Nous n'avons pas dû stocker cette information car elle est directement utilisée après avoir été calculée.

D. *datas_threads_t :*

Cette structure est décomposée en deux parties. La première partie a pour but de stocker les arguments entrés par l'utilisateur.

Ces arguments sont distinguables en 2 catégories:
- Arguments obligatoire :
  - Le nom du fichier binaire d'entrée ⇒ Stocke le flux de ce fichier binaire d'entrée.
- Arguments optionnels :
  - Le nom du fichier binaire de sortie ⇒ Stocke le flux de ce fichier binaire de sortie [Par défaut : stdout = sortie standart].
  - le nombre de threads de calcul pour effectuer l'algorithme de Bellman-Ford ⇒ Doit être strictement compris entre 1 et 127 [Par défaut : 4].
  - Valeur booléenne permettant (si true), ou non (si false), d'indiquer des messages de débogue, des résultats et autres dans la console stdout [Part défaut : false].

La deuxième partie est composée de deux attributs qui seront également utilisés durant tout l'algorithme par tous les threads.
Les attributs sont les suivants:
- un pointeur vers la structure *graph_t* représentant le graphe d'entrée.
- un pointeur vers un mutex qui permet la synchronisation des threads dans la section critique (écriture dans fichier binaire + affichage des résultats dans la console stdout).

Cette structure possède tous les attributs utiles et communs à tous les threads. Par conséquent, elle n'est initialisée qu'une seule fois pour tout le programme.

E. *datas_threads_algorithm_t :*

Cette structure est spécifique pour les threads effectuant la phase algorithmique du programme. Chaque thread possède sa structure composée des attributs suivants:

- un pointeur vers la structure *datas_threads_t* (commun pour tout les threads).
- un nombre entier représentant le nombre d'itération de l'algorithme que doit effectuer le thread.
- un nombre entier représentant le noeud source où le thread commence à "travailler".

F. *all_struct_t :*

Cette structure est uniquement utile pour le test final. Elle permet de stocker tous les blocs de résultats du fichier binaire de sortie ainsi que le nombre de noeuds total du graphe d'entrée afin de le trier plus tard par ordre croissant de noeud source.

Elle a donc comme attributs le nombre total de noeuds du graphe d'entrée et un pointeur de pointeurs vers la structure *outputGraph_t*.

IV. Description des tests unitaires

Les tests unitaires sont séparés par modules. Toutes fonctions de chaque module a un (ou plusieurs si nécessaire) test(s) unitaire(s) associé(s) dans le module de tests correspondant. Cela permet de vérifier chaque fonction de manière indépendante et de facilement débugger les éventuels problèmes rencontrés. Nous avons testé toutes les lignes de chaque fonction à l'exception des "blocs" créés par des erreurs d'appel système. Nous avons pu vérifier cela à l'aide d'un "Test Coverage" disponible dans l'archive de notre Projet.

Le fichier *./tests/src/main_test* permet de rassembler et de lancer tous les tests unitaires de chaque module. CUnit permet d'apercevoir de manière claire les tests qui passent et ceux qui échouent.

Nous avons également effectué un test final testant la fonction main. Nous testons des cas limites bien pensés pour couvrir tous les cas de bord possible, ainsi que des graphes aléatoires beaucoup plus volumineux pour tester la consistance du programme à évaluer de gros graphes. Les graphes sont également représentés à l'aide de graphviz et du script Python dans les dossiers *./binary_files/general_case/visualize_graph* et *./binary_files/limits_case/visualize_graph*.

Pour effectuer ce test final, nous créons les fichiers binaires de sortie associés aux différents graphes à l'aide de notre programme C et nous les comparons aux fichiers binaires de sortie renvoyés pour les mêmes graphes par le programme Python fourni. Cette comparaison est effectuée par des fonctions que nous avons crées dans le fichier *./tests/src/verifyOutput.c*.

Nous comparons aussi nos fichiers de sortie avec 1, 4 et 80 threads de calcul afin de vérifier le bon fonctionnement de notre architecture multithreadée.

Lorsque le nombre de threads est supérieur à 1, le graphe de sortie n'est pas trié par ordre croissant des noeuds sources. Nous utilisons donc des nouvelles fonctions permettant de les trier. Ces fonctions se trouvent dans le fichier *./tests/src/sort_bin_files.c*.

V. Outils utilisés

Lors de la conception de notre implémentation, nous avons utilisé divers outils informatiques.

A. *Outil de débogage*

1) Gdb: Ce débeugueur nous a permis de trouver plus facilement les erreurs dans notre programme. Il permet d'exécuter le code et d'afficher toutes les variables (et leur valeur associée) en temps réel lors de l'exécution et plein d'autres choses très utiles.

B. *Outil pour analyser l'utilisation de la mémoire (fuite,..)*

1) Valgrind: Ce package (non disponible sur MacOS) nous a permis d'éliminer toutes les fuites de mémoire directes et indirectes de notre programme ainsi que tous les bytes encore "reacheable" après la fermeture du programme. Il nous a également permis de voir la quantité de mémoire allouée lors de l'exécution du programme afin de l'améliorer.

2) Leaks: Cet outil est l'équivalent de Valgrind sur MacOS. Il a donc été utilisé par les membres du groupes étant sur Mac.

C. *Autres outils*

1) Git: Git est la plateforme collaborative que nous avons utilisée au cours du projet. Celle-ci nous a permis de travailler simultanément et sans conflit. Cet outil possède un historique des anciennes versions ce qui nous a quelques fois sauvé de la catastrophe. Il permet donc de garder une trace de chaque modification apportée, ce qui est très précieux lors d'un gros projet de groupe.

2) Jenkins: Cet outil a été utilisé afin de vérifier, à chaque push sur le GitLab, le bon fonctionnement de nos tests. Nous n'avons pas trouvé cet outil très utile sachant que nous vérifions de manière manuelle les tests en local avant de push. Lorsque nous apportions de grosses modifications dans le programme, nous ne changions pas toujours directement les tests. Par conséquent, ça ne passait pas souvent sur Jenkins alors que le programme *./main* était fonctionnel et correct car les tests n'étaient pas mis à jour.

VI. Analyse des performances

Pour les 3 métriques, nous avons effectué des tests sur 10 graphes. Il y a 5 graphes dont le nombre de noeuds est fixé à 100 et le nombre de liens est variable et 5 graphes où à l'inverse le nombre de liens est fixé à 100 et le nombre de noeuds qui varie.

Cela nous permet de voir les évolutions des différentes métriques dues à une augmentation du nombre de noeuds ou à une augmentation du nombre de liens. En effet, dans un graphe d'entrée, les deux paramètres principaux sont ces deux variables. Cependant, d'autres facteurs jouent un rôle important comme par exemple la connectivité du graphe, qui seront uniquement pris en compte dans la section VII où nous poussons les tests sur la métrique "Temps d'exécution".

Les résultats sont représentés sous forme de graphiques avec des échelles logarithmiques mais les tables sont tout de même disponibles en annexe.

Pour un des 10 graphes, nous n'avons pas pu obtenir toutes les valeurs car le temps d'exécution était beaucoup trop long (100 000 noeuds pour Python par exemple).

A. *Le temps d'exécution*

*1) **Attentes**:* En se rappelant la complexité du programme (section II-B), on peut déduire la forme de la courbe attendue :

$$log(|V^2| * |R|) = log(|R|) + 2 * log(|V|)$$

On obtient donc:

- Figure 1: $log(|V^2| * |R|) = 2 + 2 * log(|V|) \cong 2 * log(|V|)$
- Figure 2: $log(|V^2| * |R|) = 2 + log(|R|) \cong log(|R|)$

Vu que c'est un graphe en échelle logarithmique, on s'attend à une droite de pente 2 pour la figure 1 et à une droite de pente unitaire pour la figure 2.

On s'attend également à ce que le programme C soit plus rapide que le programme Python. Si on augmente le nombre de threads, le programme devrait être plus rapide jusqu'au nombre de threads optimal pour le temps qui correspond au nombre de coeurs que possède le processeur.

*2) **Résultats**:*

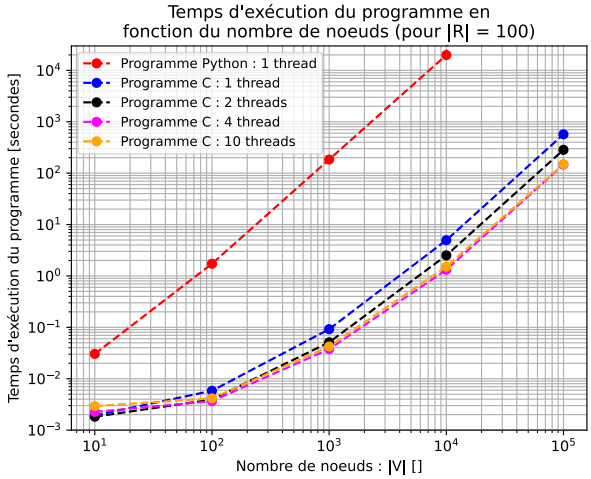

Fig. 1. Temps d'exécution des 2 programmes (C et Python) en fonction du nombre de noeuds (avec $|R| = 100$) pour différents nombres de threads

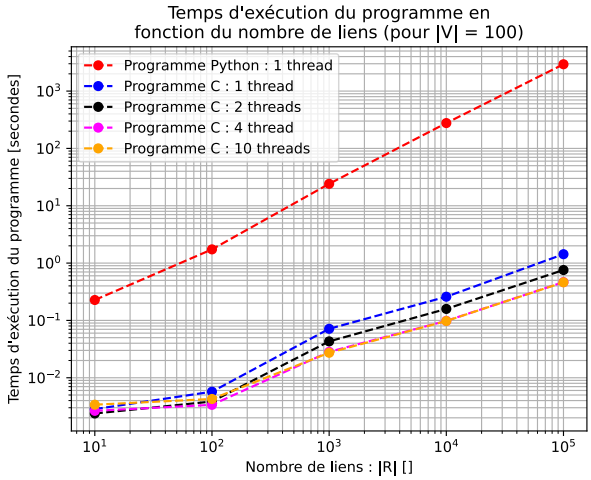

Fig. 2. Temps d'exécution des 2 programmes (C et Python) en fonction du nombre de liens (avec $|V| = 100$) pour différents nombre de threads

*3) **Interprétations**:* On observe bien toutes nos attentes sur ces figures. Pour des faibles valeurs du nombre de noeuds (resp. liens), la complexité temporelle n'est plus respectée car elle est asymptotique.

On observe également que le programme C est nettement plus rapide que le programme Python. Notre version multithread permet de diminuer le temps d'exécution lorsque nous utilisons le nombre de thread optimal. En effet, il y a également bien une limite due au processeur. Le Raspberry Pi ayant 4 coeurs monothreadés, la courbe de temps la plus optimale est celle représentant l'exécution à 4 threads, en rose.

On observe d'ailleurs que même pour des petits nombres de noeuds (resp. liens),l'utilisation de 4 threads est plus rapide que celle de 10 threads car le temps consacré à la création non nécessaire de 6 threads est non négligeable pour de petits graphes.

B. *La consommation mémoire*

*1) **Attentes**:* Nous nous attendons à ce que le programme Python consomme plus de mémoire car c'est un langage interprété et non compilé, il a donc besoin de mémoire pour "comprendre" le code.

Plus la consommation de mémoire sera élevée, plus nous nous attendons à ce que plus le nombre de threads augmente car cela nécessite de créer des threads, d'initialiser de nouvelles structures,... Au vu de l'implémentation du programme, le nombre d'itérations de calcul durant la phase algorithmique est de : $|R|*|V|*(|V|-1) \cong |R|*|V^2|$. On s'attend donc à voir une augmentation de l'utilisation de la mémoire plus importante avec l'augmentation du nombre de noeuds qu'avec l'augmentation du nombre de liens.

*2) **Résultats**:* Les mesures ont été prises avec l'outil Valgrind pour le programme C et avec les module sys et gc pour le programme Python. Ces modules permettent d'interagir avec l'interpréteur Python (sys) et le garbage collector (gc).

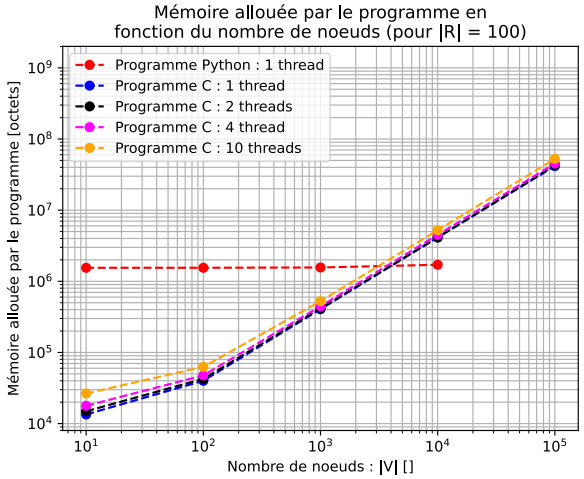

Fig. 3. Mémoire allouée par les 2 programmes (C et Python) en fonction du nombre de noeuds (avec $|R| = 100$) pour différent nombre de threads

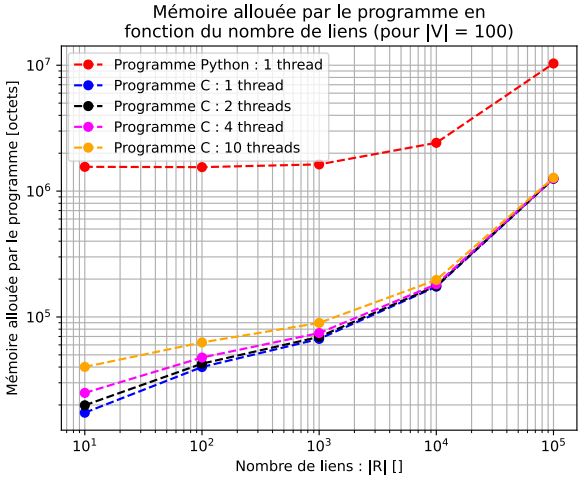

Fig. 4. Mémoire allouée par les 2 programmes (C et Python) en fonction du nombre de liens (avec $|V| = 100$) pour différent nombre de threads

*3) **Interprétations**:* Les résultats de la figure 4 confirment donc bien nos attentes. Nous pouvons également remarquer que la différence de consommation de mémoire entre des nombres différents de threads devient négligeable, proportionnellement au total de mémoire allouée, pour des graphes volumineux.

Cependant, nous ne comprenons pas les résultats obtenus à la figure 3. Nous sommes donc incapables d'expliquer ce phénomène qui nous semble bien erroné. La mémoire allouée par le programme Python devrait augmenter de la même manière que la mémoire allouée par le programme C.

C. *La consommation énergetique*

*1) **Attentes**:* En analysant l'équation et en supposant que le courant reste relativement constant durant l'exécution, on peut récrire l'équation comme (= approximation) : $E = P * t \ [Ws = J]$ où P est la puissance. On s'attend donc exactement aux mêmes courbes que celles du temps d'exécution mais multipliées par un faible facteur constant puisque la puissance varie peu.

*2) **Résultats**:* Pour effectuer ces mesures, nous avons mesuré le courant fourni par le Raspberry Pi. En connaissant la valeur de la tension aux bornes du Raspberry Pi qui est de 5.06V et le temps d'exécution du programme pour chaque graphe, nous pouvons calculer la consommation énergétique avec cette formule : $E = V * I * t = P * t \ [Ws = J]$.

Fig. 5. Exemple de consommation éléctrique

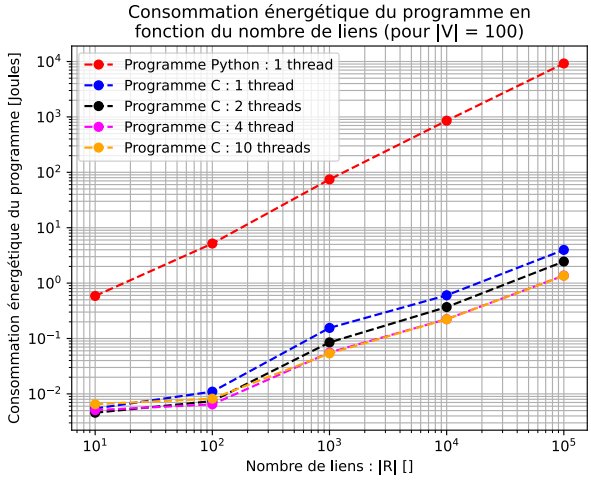

Fig. 6. Consommation énergetique des 2 programmes (C et Python) en fonction du nombre de liens (avec $|V| = 100$) pour différents nombre de threads

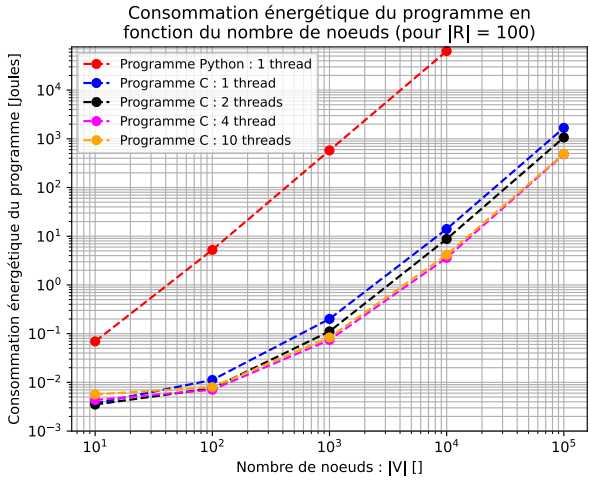

Fig. 7. Consommation énergétique des 2 programmes (C et Python) en fonction du nombre de noeuds (avec $|E| = 100$) pour différents nombre de threads

*3) **Interprétations**:* La puissance P étant, en réalité, comprise entre 1.9W et 3.5W pour tous les graphes, l'approximation est cohérente. Nous avons donc obtenu les résultats voulus.

Le programme consomme donc plus d'énergie avec l'augmentation du nombre de noeuds que du nombre de liens.

D. *Conclusion*

Nous pouvons donc conclure que le langage C est plus performant sur ces différentes métriques. Ceci n'est en réalité pas étonnant au vu des 3 grandes caractéristiques intrinsèques du langage C.

Tout d'abord, le langage C est un langage compilé (= code traduit en langage machine avant l'exécution) tandis que Python est un langage interprété (= code interprété à chaque exécution).

Ensuite, le langage C est un langage de bas niveau et donc plus proche du langage machine. Python est, quant à lui, un langage de haut niveau. Et par conséquent, il est plus facile à lire et écrire mais moins performant en temps d'exécution.

Pour finir, le langage C est un langage statiquement typé, contrairement à Python qui est un langage dynamiquement typé, ce qui permet une utilisation plus efficace des ressources ainsi qu'une gestion plus précise de la mémoire.

VII. Optimisation de la métrique : Temps d'exécution

Nous avons optimisé un maximum notre code dans l'optique d'obtenir les temps d'exécution les plus courts possibles.

Pour cela, nous avons fait en sorte que les threads :

- puissent écrire les résultats le plus rapidement possible et de manière autonome (sans devoir attendre quelque chose d'autre).
- soient le moins possible synchronisés entre eux car l'attente d'un thread ralentit le temps d'exécution global du programme.

Ces sacrifices ont comme conséquence que l'utilisation de la mémoire est plus importante car nous devons créer des structures pour chaque thread afin qu'ils puissent être autonome. Aucune communication entres les threads est autorisée. Un mutex est utilisé pour verrouiller une section critique à un seul thread à la fois.

Nous voulons mettre ici en évidence l'évolution du temps d'exécution du programme C en fonction du nombre de threads. Pour cela, nous allons reprendre les mesures de la section VI-A pour les exploiter autrement.

On a vu à la section VI-A que plus le nombre de threads est élevé, plus le temps d'exécution est court. Mais nous ne savons pas à quelle type de courbe nous attendre : droite, parabole, exponentielle,..

Voici les résultats obtenus pour 5 graphes différents dont $|V| = 100$:

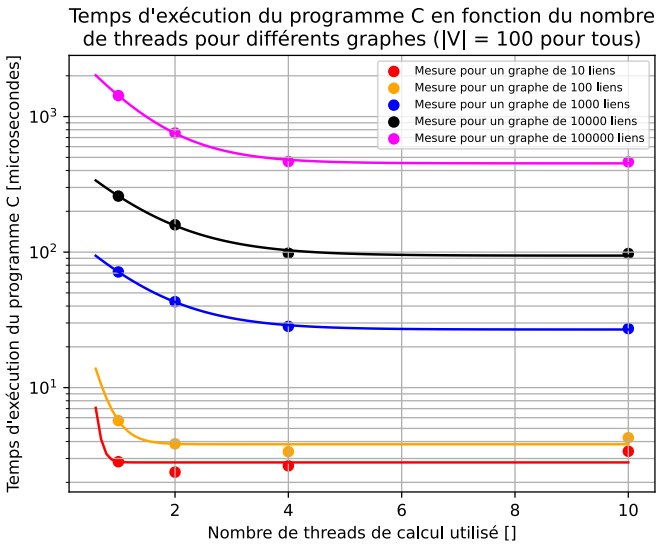

Fig. 8. Temps d'exécution du programme C en fonction du nombre de threads de calcul pour différents graphes ($|V| = 100$ pour tous)

On aperçoit donc bien que les courbes approximées sont des exponentielles négatives de type : $f(x) = a * exp(-b * x) + c$.

La pente n'est pas raide car l'axe des y est en échelle logarithmique (pour voir tous les graphes).

On peut apercevoir la limite asymptotique de la courbe qui commence à partir de x = 4 car les mesures ont été prises sur le Raspberry Pi qui contient un processeur de 4 coeurs. Cette limite correspond au paramètre c de l'approximation.

VIII. Dynamique du groupe

Lors des séances de TPs, nous avons avancé dans le projet en effectuant les différentes tâches prévues pour la semaine. Souvent, des binômes se sont formés pour échanger leurs idées.

La répartition équitable des tâches a été particulièrement compliquée tout au long du projet. Mais avons finalement obtenu un résultat plutôt concluant.

Voici la répartition des tâches principales au cours du Projet:

- Ecriture globale du code C (structures de données, différents modules,...) [Mathis D. et Adrien A.]
- Conception des tests unitaires et des tests de validations [Mathis D.]
- Conception du MakeFile [Adrien A. et Mathis D.]
- Ecriture du README [Mathis D.]
- Setup de l'outil Jenkins [Adrien A.]
- Tentative d'effectuer une architecture multithreading Producer/Consumer avec un buffer [Mathis D. et Adrien A.]
- Tentative de multithreader la lecture du fichier binaire d'entrée [Mathis D.]
- Architecture multithreading finale utilisée [Mathis D.]
- Correction des fuites de mémoires à l'aide de l'outil Valgrind [Mathis D., Adrien A. et Laura R.]
- Extensions réalisées :
  - Modification du script Python pour générer des graphes complets [Mathis D.]
  - Test Coverage [Adrien A.]
  - Création d'une image (sous le format souhaité) de la structure complète de la première partie de notre projet (sans le multithreading) [Mathis D.]
- Raspberry Pi (prise en main et setup wifi,...) [Pierre P., Edouard M. et Ulysse R.]
- Ecriture du rapport (global + analyse des performances avec mesures) [Mathis D. et Pierre P.]

IX. Annexe

A. *Tables de résultats*

Cette sous-section contient les tables des résultats obtenus à la section VI.

Les valeurs pour l'analyse de la métrique "consommation énergétique" ne sont pas reprises ici car elles sont très proches de celles pour la métrique "temps d'exécution" (elles sont multipliées par un facteur compris entre 1.5 et 3.5). Cependant, toutes les valeurs se trouvent dans l'archive de notre projet dans le fichier *./rapport/perf_analyse.py*.

TABLE I

TABLE DU TEMPS D'EXÉCUTION DES DEUX PROGRAMMES EN FONCTION DU NOMBRE DE LIENS ET DU NOMBRE DE THREADS UTILISÉS

	Programme C				Python
Nombre de liens	1 thread	2 threads	4 threads	10 threads	1 thread
10^1	2.84 ms	2.38 ms	2.65 ms	3.39 ms	227.17 ms
10^2	5.71 ms	3.85 ms	3.37 ms	4.27 ms	1.734 s
10^3	71.38 ms	43.08 ms	28.31 ms	27.27 ms	24.13 s
10^4	0.259 s	0.159 s	98.18 ms	97.63 ms	276.87 s
10^5	1.429 s	0.757 s	0.466 s	0.463 s	48.97 min

TABLE II

TABLE DU TEMPS D'EXÉCUTION DES DEUX PROGRAMMES EN FONCTION DU NOMBRE DE NOEUDS ET DU NOMBRE DE THREADS UTILISÉS

	Programme C				Python
Nombre de noeuds	1 thread	2 threads	4 threads	10 threads	1 thread
10^1	1.93 ms	1.82 ms	2.26 ms	2.94 ms	30.38 ms
10^2	5.83 ms	3.94 ms	3.66 ms	4.15 ms	1.726 s
10^3	92.12 ms	51.08 ms	37.67 ms	42.4 ms	184.205 s
10^4	4.96 s	2.51 s	1.316 s	1.517 s	5.5 h
10^5	570.19 s	284.32 s	146.94 s	149.14 s	/

TABLE III

TABLE DE LA MÉMOIRE UTILISÉE PAR LES DEUX PROGRAMMES EN FONCTION DU NOMBRE DE LIENS ET DU NOMBRE DE THREADS UTILISÉS

	Programme C				Python
Nombre de liens	1 thread	2 threads	4 threads	10 threads	1 thread
10^1	17352 bytes	19880 bytes	24936 bytes	40104 bytes	1558892 bytes
10^2	40016 bytes	42544 bytes	47600 bytes	62768 bytes	1549900 bytes
10^3	66940 bytes	69468 bytes	74524 bytes	89692 bytes	1629836 bytes
10^4	174120 bytes	176648 bytes	181704 bytes	196872 bytes	2418348 bytes
10^5	1253728 bytes	1256256 bytes	1261312 bytes	1276480 bytes	10326860 bytes

TABLE IV

TABLE DE LA MÉMOIRE UTILISÉE PAR LES DEUX PROGRAMMES EN FONCTION DU NOMBRE DE NOEUDS ET DU NOMBRE DE THREADS UTILISÉS

	Programme C				Python
Nombre de noeuds	1 thread	2 threads	4 threads	10 threads	1 thread
10^1	13452 bytes	14900 bytes	17796 bytes	26484 bytes	1548460 bytes
10^2	40016 bytes	42544 bytes	47600 bytes	62768 bytes	1549900 bytes
10^3	405208 bytes	418536 bytes	445192 bytes	525160 bytes	1564244 bytes
10^4	4133184 bytes	4133184 bytes	4497168 bytes	5225136 bytes	1708244 bytes
10^5	41613184 bytes	42814512 bytes	45217168 bytes	52425136 bytes	/