Projet 3 - Rapport Final

Djimbi Claudel Gourari Yasmina Martin Antoine Mat Hugo Winant Guillaume

I. Introduction

Dans le cadre de notre projet, nous nous sommes penchés sur l'élaboration d'un programme en langage C permettant de répondre à une problématique, en nous focalisant sur l'optimisation des performances grâce à l'utilisation de la programmation multi-thread. Notre objectif principal était de concevoir une version hautement performante de l'algorithme de Bellman-Ford afin d'accélérer le calcul des chemins les plus courts au sein de vastes réseaux de noeuds donnés sous forme de fichiers binaires.

Pour écrire notre programme en C, nous avons d'abord étudié de manière approfondie l'algorithme de Bellman-Ford, puis examiné une implémentation de l'algorithme en Python. Le langage C à l'avantage d'être plus proche de la machine, ce qui rend l'exécution du programme plus rapide (notamment pour les gros graphes nécessitant plus de calculs). De plus, il nous permet de gérer la mémoire par nous-mêmes et donc d'en optimiser la consommation.

Nous avons ensuite réalisé des tests rigoureux pour vérifier la validité des résultats obtenus et évaluer les performances de notre implémentation.

Enfin, la dernière étape consistait à transformer notre code mono-thread en une version multi-thread, exploitant ainsi le potentiel de parallélisme offert par les processeurs modernes. En répartissant la charge de travail sur plusieurs threads, nous avons considérablement accéléré le temps d'exécution de notre programme. Nous avons également effectué des comparaisons entre les versions Python, mono-thread et multi-thread afin de mesurer les améliorations obtenues et analyser l'impact de la parallélisation sur les performances.

Ce rapport expose donc notre méthodologie et présente les résultats obtenus, mettant en évidence les problèmes rencontrés et les décisions prises pour les résoudre.

II. Implémentation de l'algorithme

Notre programme a été conçu pour prendre en entrée un fichier binaire contenant les données d'un graphe orienté et pondéré pouvant contenir des poids négatifs. Il exécute ensuite l'algorithme de Bellman-Ford afin de trouver le plus long plus court chemin depuis chaque noeud et écrit les résultats dans un fichier de sortie binaire.

Nous nous sommes assurés que notre code puisse s'exécuter à la fois sur Linux et MacOS.

A. structures de données

Afin de rendre notre code lisible et compréhensible, nous l'avons structuré à l'aide de différents dossiers et fichiers :

1) "main.c": Il s'agit de la fonction principale du programme. Elle s'occupe de créer et détruire les threads du programme. C'est à partir d'elle que nous appelons les autres fonctions.

2) Dossier src: Ce dossier contient les fichiers sources. "graph.c" reprend toutes le fonctions permettant de résoudre le problème du plus court chemin. "fichier.c" contient les fonctions permettant de lire le fichier d'entrée et de libérer l'espace alloué. "execution.c" comporte toutes les fonctions permettant d'exécuter le programme en multi-thread quand les threads ont été créés (calcul des chemins + écriture du fichier de sortie).

3) Dossier include: Ce dossier contient les fichiers headers qui comprennent la signature des fonctions et la définition de nos structures. 2 fichiers permettent aussi de fournir une compatibilité MacOS.

4) MakeFile: Le MakeFile consiste en des raccourcis permettant à l'utilisateur d'effectuer certaines opérations grâce à des commandes simples. Par exemple, il sert à compiler facilement les différents fichiers afin de créer un fichier exécutable permettant par la suite d'exécuter le programme principal. Aussi, le MakeFile permet de lancer directement notre suite de tests unitaires et de performance ainsi que de nettoyer les fichiers temporaires.

5) "tests.c": Ce fichier contient les tests unitaires testant les fonctions implémentées localement et les tests globaux.

6) "timer.c": Ce fichier contient les tests de performance relatifs au temps d'exécution.

7) "script.sh": Ce fichier permet de lancer les tests de performance (temps et consommation de mémoire).

Pour ce qui est de nos structures au sens pur du terme, nous en avons implémenté 3 : une pour les liens des graphes, une pour stocker les résultats de l'algorithme de bellman ford et une dernière pour permettre au threads producteurs et consomateur de stocker et lire résultats dans un buffer commun.

B. Optimisation

Initialement, nous utilisions de nombreuses structures au sein de notre programme. Cependant, nous nous sommes vite rendus compte qu'elles occupaient énormement d'espace. Or, ces structures contenaient des éléments en commun. Pour optimiser la consommation de mémoire, nous avons donc trouvé un équilibre grâce aux trois structures actuelles qui

permettent de stocker toutes les informations nécessaires sans redondances.

III. Architecture multithread

Pour réaliser un code multi-threadé afin d'optimiser le temps d'exécution de notre programme, l'utilisation ingénieuse des threads et mutex était primodiale. Nous avons implémenté un problème producteurs-consommateurs comportant un seul thread consommateur et au minimum un thread producteur (minimum 2 threads pour le fonctionnement du code). Le producteur-consommateur consiste en un buffer sous forme de tableau contenant les données permettant à plusieurs threads de travailler en parallèle. En effet, n threads vont exécuter l'algorithme sur plusieurs sources en même temps et vont retranscrire les résultats dans le buffer. Si le buffer contient des informations, le thread consommateur va extraire les données du buffer et les écrire dans un fichier binaire. Nous avons initialisé 2 mutex différents afin de dissocier l'accès au noeud source avec lequel calculer bellman-ford et l'accès au buffer où sont stockés les résultats de sorte à pouvoir simultanément faire ces deux actions. Ceci augmente considérablement la vitesse de calcul tout en gardant une harmonie entre les threads.

IV. Tests unitaires et outils d'analyse

Au fur et à mesure de l'avancée du projet, nous avons développé et amélioré notre suite de tests en utilisant la libraire CUnit. Ceux-ci ont pour but de nous assurer du bon fonctionnement des différentes parties de notre code.

Nous avons tout d'abord créé des tests unitaires pour toutes les fonctions que nous avons dû implémenter et adapter sur base du code python qui nous avait été donné. Les fonctions testées sont donc les suivantes : bellman_ford, get_file_infos, read_graph, get_max et get_path. Nous avons également testé différents cas limites pour les fonctions bellman_ford, get_max et get_path tels que le cas d'un graphe comprenant un cycle négatif ou encore un graphe contenant des noeuds isolés.

Finalement, nous avons créé un test global qui permet de vérifier le fonctionnement global du programme en comparant les résultats avec ceux du code python pour un même graphe. Pour ce test, nous avons légèrement modifié la partie main de la fonction sp.py donnée initialement de telle sorte que lorsqu'elle est appelée avec un certain graphe, le programme renvoie un fichier binaire contenant les résultats associés à ce graphe. Le test global va lancer le fichier python modifié ainsi que notre programme en C et va comparer les résultats afin de s'assurer que pour un graphe de taille quelconque, on obtienne bien les mêmes résultats qu'avec le code python. Ces tests nous ont aidé à nous assurer du bon fonctionnement individuel de chaque fonction et du fonctionnement global de notre programme.

Concernant les outils d'analyse utilisés, nous avons lié Jenkins à notre GitLab au début de notre projet afin de vérifier que chaque commit passe bien. Nous avons également fréquemment eu recours au logiciel Valgrind qui nous a permis de développer un programme sans fuites de mémoire. L'outil gdb nous a finalement été utile afin de déboguer efficacement notre code.

V. Analyse de performances

Nous avons réalisé plusieurs tests pour mesurer et comparer les performances de notre algortithme suivant différents facteurs. Pour ce faire, nous avons considéré les variables suivantes :

variables contrôlées :

- taille du graphe (donné en nombre de noeuds et d'arêtes)
- version du programme (python, C mono-thread, C multi-thread)
- machine (Raspberry pi, ordinateur "HP OMEN")

variables dépendantes :

- temps (en secondes)
- espace (en kilo octets)
- puissance (en Watt)

Dans chaque sous-section, nous allons comparer la variable en question avec les différentes variables contrôlées listées ci-dessus.

Notez que lorsqu'une variable est testée, les 2 autres sont mises constantes dans la légende des graphes. (ordi correspond à ordinateur, graphe 1000 correspond à un graphe de 1000 noeuds et 1000 arêtes, c multi 1 correspond à la version multi-thread avec 1 thread producteur).

A. Temps d'exécution

La figure 1 nous indique que le langage C est bien plus performant que le Python. On voit également sur la figure 2 que notre implémentation multi-threadée est plus efficace que la mono-threadée.

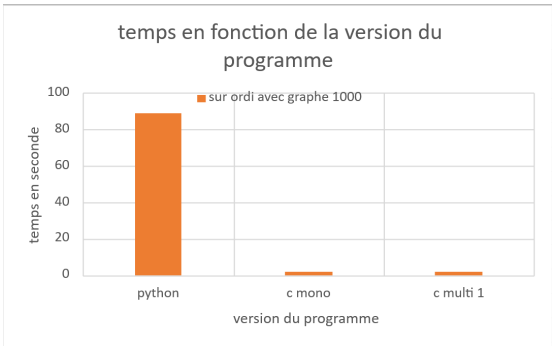

Fig. 1: On remarque un écart colossal entre les temps d'exécution des versions en C (mono et multi-thread) et de la version en Python.

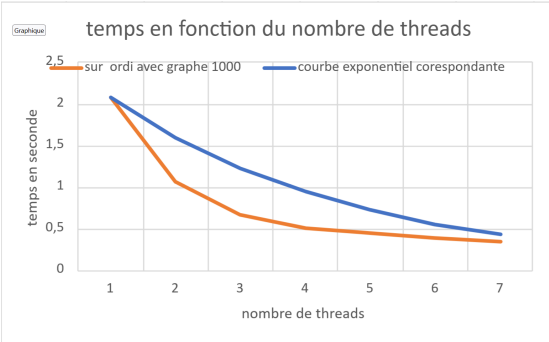

Fig. 2: On observe que le temps d'exécution en fonction du nombre de threads (orange) s'apparente à une exponentielle décroissante (bleu)

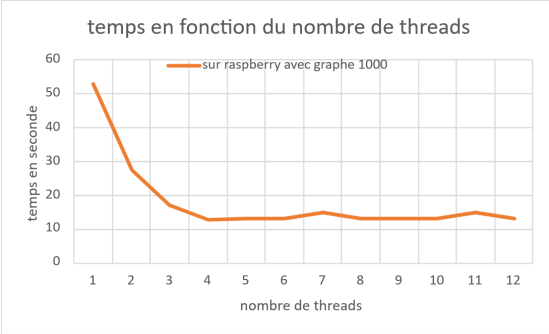

Fig. 3: Même analyse que 2 mais testé sur Raspberry avec 4 threads. Notez que après 4 threads, le temps d'exécution stagne. Ceci est sûrement dû au fait que le Raspberry pi ne contient que 4 coeurs.

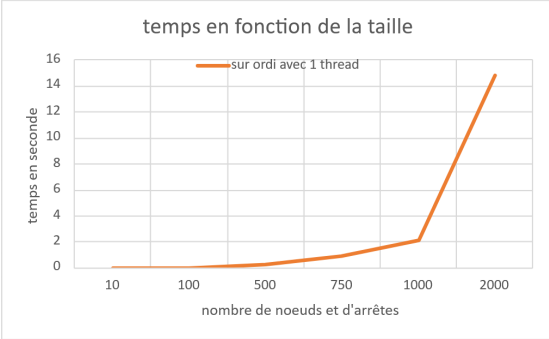

Fig. 4: Ici, on voit que le temps est de plus en plus long en fonction de la taille du graphe, mais le caractère exponentiel reste à déterminer. Notez que l'axe en x n'est pas linéaire.

B. Consommation de mémoire

Nous observons sur la figure 6 que la mémoire RAM maximale utilisée en Python est bien plus importante que celle utilisée par le langage C. De plus, la mémoire consommée en multi-thread est légèrement plus importante qu'en mono-thread (dû à l'implémentation des threads, sémaphores et mutex). Nous ne remarquons pas de différence entre l'utilisation avec 1 ou 8 threads, comme on pourrait s'y attendre.

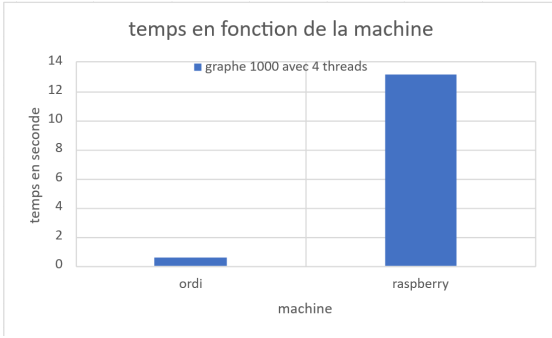

Fig. 5: On remarque ici que le Raspberry est bien plus lent qu'un ordinateur classique pour exécuter le programme.

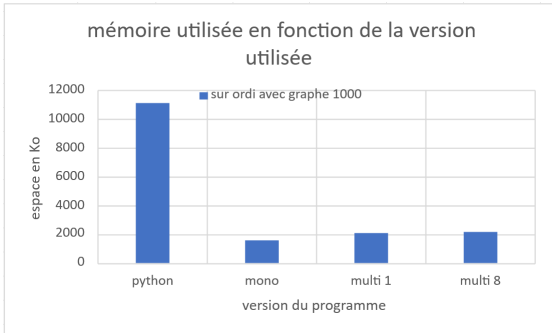

Fig. 6: On peut clairement voir que C optimise bien mieux la mémoire que python. Notez aussi que la version multithread est légèrement plus élevée que la version monothread.

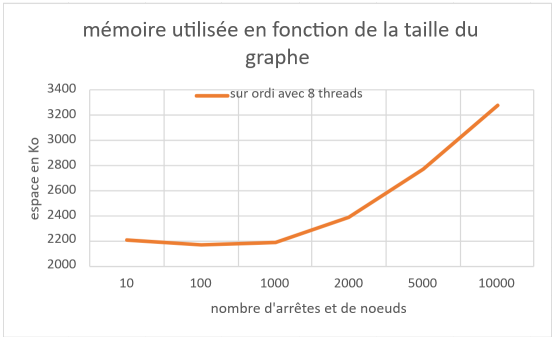

Fig. 7: plus la taille du graphe augmente, plus la mémoire allouée est importante. Notez que l'axe en x n'est pas linéaire.

C. Consommation d'énergie

On voit sur la figure 8 que, de manière équivalente à la consommation de mémoire, notre code multi-threadé est plus énergivore que la version mono-threadée. Nous concluons que ces différences sont aussi dûes aux initialisation des threads, des mutex et des sémaphores. Ceci dit, l'énergie consommée reste inférieure à celle de la version Python.

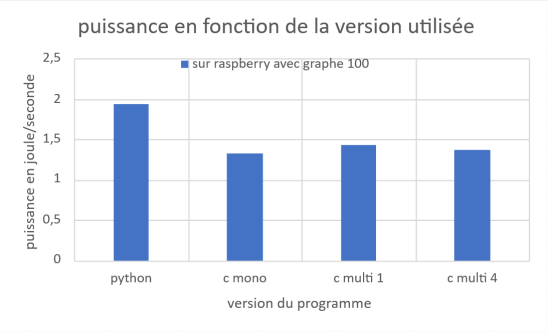

Fig. 8: Python est plus énergivore que C. Le mutlithread l'est aussi légèrement plus que le monothread.

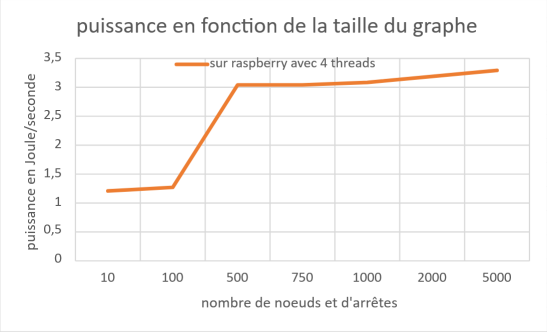

Fig. 9: La demande en énergie croit avec la taille du graphe fourni.

VI. Tests de performances

Pour réaliser les graphes ci-dessus, nous avons eu besoin de connaître le temps d'exécution du programme et sa consommation de mémoire. Pour le timer, nous avons utilisé le programme squelette en C des Professeurs que nous avons réadapté à notre programme. Ensuite, nous avons écrit un script en bash qui mesure la RAM maximale utilisée du fichier exécuté. C'est la commande bash "command time" qui permet de la mesurer. Le script, une fois lancé, va exécuter les tests de mémoire et le timer. Les résultats sont stockés dans des fichiers .csv et .txt. Nous avons du faire un timer en C au lieu d'utiliser le timer en bash "time" car ce dernier était loin d'être assez précis pour les tests effectués.

VII. Dynamique de groupe

Concernant la dynamique de groupe pour la deuxième partie du projet, les tâches à réaliser étaient les suivantes :

- S'assurer que notre version mono-threadée soit correcte
- Implémenter l'usage de threads à notre application
- Apprendre à manier le Raspberry pi et écrire les scripts permettant de lancer les tests de performances dessus
- Créer de nouveaux tests
- Gérer les fuites de mémoire
- Améliorer et mettre à jour la fichier README
- Rédiger le rapport

Nous avons dans un premier temps prêté attention aux remarques faites par nos pairs lors des peer reviews. Après avoir établi une liste des points importants relevés par ceux-ci, nous avons pu considérablement améliorer notre version mono-threadée.

Nous avons aussi fait le point sur les tâches à effectuer pour la suite du projet et nous sommes fixés des deadlines chaque semaine.

Nous avons réalisé qu'en plus d'adopter une bonne organisation, il était important de mieux communiquer entre nous. C'est pourquoi nous avons organisé de nombreuses réunions et discuté quasiment quotidiennement pour faire le point sur l'avancement du projet et nous entre-aider lors des éventuels bugs et problèmes rencontrés. Aussi, l'outil GitLab nous a permis de partager notre code et de l'organiser de façon efficace.

Concernant la répartition des tâches et des rôles joués par chacun, nous nous sommes assignés des tâches respectives à effectuer par chacun, par exemple, l'implémentation de certaines fonctions, l'optimisation du programme, la réalisation de nouveaux tests unitaires, la création des scripts et graphes pour les tests sur Raspberry, la gestion des fuites de mémoire, le débogage de certaines fonctions etc. En cas de problème, nous le communiquions au reste groupe afin de trouver des solutions ensemble.

Dans l'ensemble, le groupe a eu une bonne dynamique. Cependant, l'un des membres du groupe, Nicolas Rachid, n'a pas du tout contribué au projet. Il n'a ni manifesté sa présence lors des séances de TP, ni montré son implication de quelconque manière. Il nous a dit avoir prévenu les professeurs de sa volonté d'abandonner le projet. Nous avons nous aussi envoyé un mail de notre coté à Monsieur Navarre concernant la non participation au projet de Nicolas.

VIII. Conclusion

En conclusion, nous observons que le langage C est bien plus efficace que le Python. Il est évidemment moins facile à maitriser mais nous accorde plus de liberté, notamment pour gérer la mémoire. Au vu des résultats exposés ci-dessus, il n'est pas possible de douter de l'amélioration considérable du temps d'exécution. Nous pouvons voir que le multithreading est un outil remarquable à utiliser dans son programme en C pour optimiser davantage les performances. Malgré les difficultés rencontrées, notamment lors de l'optimisation du code, la compatibilité avec MacOS et la gestion des threads, ce projet nous a permis de développer nos compétences en algorithmique, en programmation en langage C et en optimisation des performances. Nous avons réussi à concevoir une implémentation fonctionnelle de l'algorithme de Bellman-Ford tout en exploitant les avantages de la programmation multithread. Nous sommes fiers des résultats obtenus et espérons que notre travail pourra vous satisfaire suffisamment.

References

[1] https://www.geeksforgeeks.org/
[2] https://stackoverflow.com/
[3] Manpages Linux et Syllabus du cours LEPL1503