Rapport de Projet 3 : Traduction d'un algorithme de Bellman-Ford de Python en C

Ghislain LIEUKAP
ghislain.lieukap@student.uclouvain.be

Freddy NTAMWANA
freddy.ntamwana@student.uclouvain.be

Maurine PONCELET
maurine.poncelet@student.uclouvain.be

Alexandre STORDEUR
alexandre.stordeur@student.uclouvain

Krystian TARGONSKI
krystian.targonski@student.uclouvain.be

Robin WARICHET
robin.warichet@student.uclouvain.be

***Résumé*—Le but de ce rapport est d'expliquer le fonctionnement du code en C que nous avons du implémenter. Nous y aborderons la description de l'algorithme, les structures de données, ainsi que les tests. Ensuite, nous passerons à la comparaison de performances, puis les différences entre python et C, pour finir avec notre dynamique de groupe.**

I. Description de l'algorithme

L'objectif de ce projet est le suivant ; créer un programme en langage C qui implémente l'algorithme de Bellman-Ford. Son but est simple, trouver le chemin le plus court entre deux noeuds d'un graphe pondéré. Ce graphe peut contenir des noeuds isolés mais aussi des chemins de coûts négatifs, c'est pour cela que nous avons décidé d'utiliser cet algorithme comme suggéré au début du projet par les professeurs. L'algorithme de Bellman-Ford permet de trouver le plus court chemin entre un noeud source et un autre noeud dans un graphe pondéré. Pour ce faire, il va calculer les différents chemins possibles entre les deux noeuds, et supprimer au fur et à mesure les chemins trop longs jusqu'à arriver à la meilleure option, qui s'avère être le plus court chemin.

Il est important de souligner qu'il y a d'autres algorithmes plus efficaces (shortest path faster algorithm, ou SPFA, par exemple), qui auraient pu être implémentés dans notre programme afin d'optimiser encore plus le temps d'exécution, mais nous avons décidé de s'orienter plutôt dans l'optimisation de l'utilisation de mémoire. En faisant ce choix, nous nous sommes rendu compte que notre programme sera certes un peu plus lent, mais il pourra tourner sur des machines ayant moins de mémoire disponible. De plus, avec l'implémentation d'une version multithreadée, nous pouvons mitiger cette perte de performance en utilisant le nombre de threads optimal.

A. Architecture générale du programme

Comme mentionné précédemment, notre implémentation peut utiliser plusieurs threads afin de diminuer considérablement le temps d'exécution sur des graphes de grande taille. Pour ce faire, nous avons créé une architecture qui se base sur le modèle des producteurs et consommateurs. Les threads sont divisés en 3 rôles :

- Le premier thread, *dispatcher*, s'occupe de distribuer les noeuds aux threads suivants afin qu'ils sachent quelle source utiliser.
- Les prochains threads, *computers*, sont ceux qui vont exécuter l'algorithme sur la source qui leur a été asignée, ils passent ensuite le résultat de leurs calculs au dernier thread.
- Le dernier thread, *writer*, se charge de l'output. Par défaut, il affiche le résultat dans le terminal. Si un fichier de sortie est précisé, il écrit le résultat dans ce fichier.

Ces threads communiquent entre eux grâce à deux *buffers* qui sont respectivement partagés entre *dispatcher* et *computers* ainsi que *computers* et *writer*. L'utilisateur peut choisir le nombre de threads *computers* qu'il souhaite, afin de minimiser le temps d'exécution en spécifiant "-n" lors de l'exécution du programme (voir README pour plus de détails). Outre cette spécification, le nombre de threads *dispatcher* et *writer* est fixé à 1. En effet, le thread *dispatcher* et le thread *writer* étant tous deux plus rapides que les threads *computers*, notre perte de temps se créera à l'exécution de l'algorithme de Bellman-Ford par les *computers*. Cette différence est telle que nous avons jugé inutile de mettre plus d'un thread pour la distribution et plus d'un thread pour l'écriture dans le fichier/affichage du résultat. A spécifier que le programme prend en entrée seulement des fichiers binaires. Si le fichier d'entrée est mal structuré ou dans le mauvais format, le programme retournera simplement un message d'erreur.

B. Améliorations apportées par rapport au code en python

Pour l'implémentation de l'algorithme de Bellman-Ford, nous nous sommes basés sur le code python qui nous a été fourni sans y faire de modifications majeures au niveau du fonctionnement de celui-ci. Le seul changement notable apporté, est que l'algorithme coupe son exécution si un chemin de coût négatif est détecté.

II. Structures de données utilisées

En C, nous avons la liberté d'utiliser des structures de données qui sont plus adaptées à nos besoins, comparés au Python où, nous sommes fixés aux structures qui sont définies au préalable. C'est notamment grâce à cela que nous avons pu optimiser l'utilisation de la mémoire. En effet nous avons défini des structures différentes tout au long du programme afin d'être sûrs que l'on stocke uniquement les informations nécessaires et ce dans un espace qui leur est adapté.

Avec notre architecture, on stocke seulement le graphe ainsi que les informations de celui-ci tout au long de l'exécution, afin que les threads *computers* puissent y accéder de manière indépendante, sans devoir les copier à chaque fois. Ce graph est stocké dans une structure nommée *graph_t* qui est composée de deux parties :

- Un array de deux nombres ; un pour le nombre de noeuds et l'autre pour le nombre de liens du graphe.
- Une sous structure nomée *branch_t* qui stocke le graphe en lui même. Elle est composée de 3 arrays qui sont respectivement utilisés pour ; le noeud d'origine, le noeud d'arrivée et le coût entre eux.

Un point positif à cette approche, est que cette structure n'a pas besoin d'être protégée par un mutex car les threads qui y accéderont, le feront seulement pour lire des informations et non pas pour modifier ce qu'elle contient. Ensuite, le reste des structures sont allouées et relâchées dynamiquement par les threads *computers* et *writer* quand elles sont nécessaires.

Lors de l'exécution, les *computers* génèrent les structures *ford_t* et *mcost_t*. Ils les encapsulent ensuite dans la structure *thread_data_t* afin de passer seulement les informations nécessaires au dernier thread. Ces informations sont les suivantes : le noeud source, les structures *ford_t* et *mcost_t*, le chemin final ainsi que sa longueur. Ce sont celles-ci qui seront soit affichées soit enregistrées dans le fichier de sortie.

Mcost_t est utilisé pour stocker le coût maximal et le noeud correspondant, tandis que *ford_t* stocke les distances et chemins vers tous les noeuds à partir d'un noeud source.

III. Tests

Étant donné que la seule partie qui peut influencer la sortie de notre programme est le fichier d'entrée, nous avons décidé de traiter tout le programme comme une "boîte noire" lors de nos tests. Autrement dit, il suffit de vérifier que le programme se comporte comme attendu avec un input donné. En effet, cette approche est tout à fait valable si toutes les fonctions font ce qui leur est demandé. Nous avons donc fait des tests *Cunit* préliminaires pour être sûrs que nos fonctions se comportent comme attendu. Nous avons ensuite pu faire des tests *Cunit* avec des cas-limites et des fichiers d'entrée différents. Les situations suivantes sont prises en compte :

- Graphe nul
- Graphe vide
- Fichier d'entrée corrompu
- Graphe avec des poids négatifs
- Graphe général

À la suite de ces tests, nous avos pu constater que notre programme se comporte comme attendu ; c'est-à-dire qu'il retourne bien un message d'erreur dans les cas où le chemin ne peut pas être calculé, et ce pour une des différentes raisons mentionnées ci-dessus. Nous avons fait en sorte que les fonctions principales retournent NULL en cas d'erreur (que ce soit à cause de l'entrée, un problème lors de l'allocation de mémoire ou autres) et pour les différencier on renvoie un message d'erreur avec perror() pour spécifier ce qui a dysfonctionné.

IV. Analyse détaillée des performances de notre programme

A. Temps d'exécution

Nous avons testé le programme en faisant varier le nombre de threads puis comparé ces résultats entre eux et avec ceux obtenus avec le code Python. Cela nous a permis de remarquer très vite que le nombre de threads a bien un effet sur le temps d'exécution. Augmenter le nombre de threads a pour effet de diminuer le temps d'exécution mais cela seulement si ce nombre ne dépasse pas le nombre de coeurs de la machine qui exécute le programme.

Vous pouvez voir sur la figure 1, reprenant les valeurs mesurées sur le raspberry Pi, que le temps d'exécution diminue d'un facteur proche de 2. Cela se passe à chaque fois que le nombre de threads est doublé, jusqu'à ce qu'on atteigne le nombre de coeurs du raspberry Pi (ici 4). Au-delà, le temps ne change plus trop même si l'on ajoute beaucoup de threads. On observe aussi un effet inverse si on ajoute beaucoup trop de threads, le temps d'exécution commence à remonter, ce qui est probablement dû au fait qu'il faut gérer les threads supplémentaires.

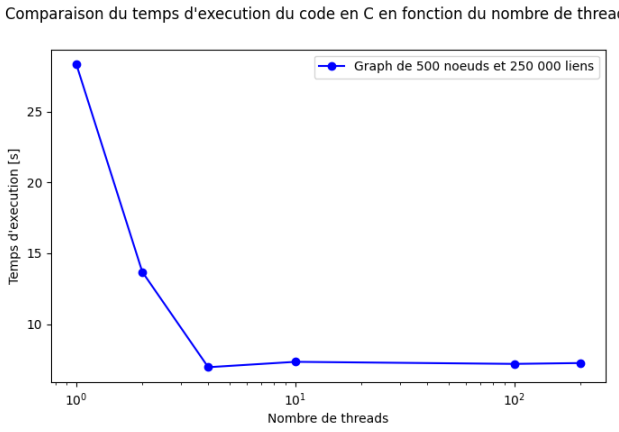

Figure 1 – Temps d'exécution en fonction du nombre de threads

Nous avons aussi comparé le temps d'exécution en variant le nombre de threads pour des graphes de tailles différentes (voir Figure 2). Ce graphe nous montre bien qu'à partir de 4 threads, le gain n'est plus distinguable et que le saut le plus important est réalisé lors du passage du Python vers C.

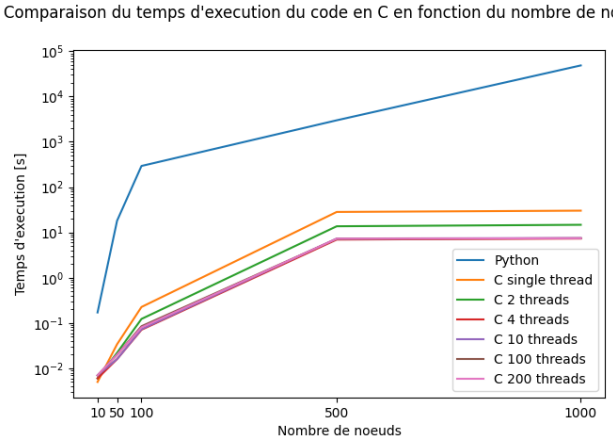

FIGURE 2 – Temps d'exécution

B. Consommation de mémoire

Notre implémentation a été pensée avec une utilisation de mémoire minimale comme expliqué plus tôt.

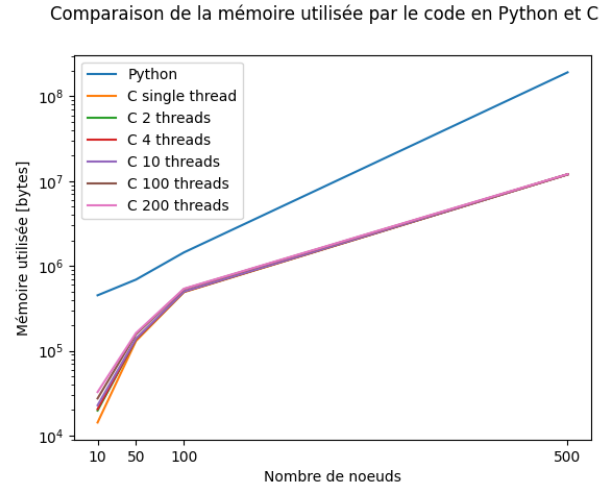

FIGURE 3 – Mémoire utilisée

Sur la figure 3, vous pouvez voir que plus le graphe est grand, plus la consommation de mémoire va se stabiliser indépendamment du nombre de threads. En effet, avec des graphes relativement petits, notre programme dépend plus de la taille du graphe en lui-même donc la mémoire utilisée est majoritairement représentée par la structure *graph_t*.

Lorsque le graphe augmente en taille, ce sera les échanges entre threads qui prendront la majorité de l'utilisation de mémoire. La figure précédente confirme que notre approche fonctionne car, avec un graphe de petite taille, notre programme prend bien moins de mémoire et ce grâce à notre structure qui stocke le graphe de manière globale. De plus, avec un graphe de grande taille, notre échange dynamique de données permet bien de minimiser l'empreinte que nous avons sur la mémoire.

En effet, à tout moment lors de l'exécution, l'empreinte maximale de notre programme sera seulement le graphe global et les deux buffers partagés entre les threads. Ces buffers ont une taille suffisante pour permettre à chaque thread d'avoir un emplacement d'écriture avec un surplus d'une place pour une question de sécurité. Ce choix a été fait afin de ne pas avoir de threads qui attendent sans rien faire.

C. Consommation énergétique

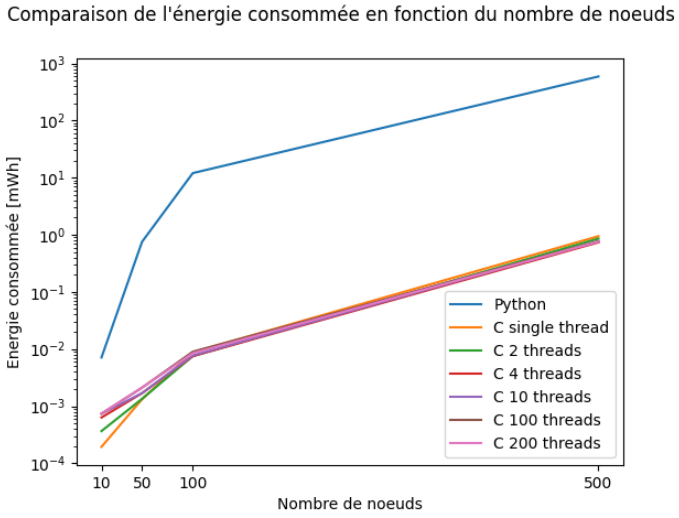

FIGURE 4 – Energie consommée

Après une prise de mesure (voir Figure 4), nous avons constaté que malgré l'utilisation instantanée quasi identique, le code C en single thread consomme moins d'énergie que celui en Python car il s'exécute plus rapidement. Dès lors, notre métrique de puissance consommée sur un temps donné (en mWh) reflète bien cette nuance. En effet en single thread, le code Python et le code en C avaient une consommation instantanée d'environ 0.704 W. On remarque aussi un comportement similaire quand on augmente le nombre de threads, la consommation instantanée augmente mais le temps d'exécution diminue (jusqu'au nombre du thread qui est égal au nombre de coeurs du processeur). Ceci revient a une puissance consommée en fonction du temps qui est similaire au final. Pour optimiser davantage la consommation énergétique, il faudrait développer le code avec le moins d'appels systèmes possible mais cela ne faisait pas partie de nos objectifs.

V. Différences entre Python et C

A. Difficulté d'utilisation

Le langage Python dispose d'un Garbage Collector, contrairement au langage C pour lequel c'est l'utilisateur qui gère la mémoire. Le langage python nous demande donc moins d'efforts en terme de gestion de mémoire. Cependant, ce garbage collector ne fonctionne pas toujours d'une manière qui serait optimale au programme qui est développé. Mais avec la gestion manuelle de mémoire avec le langage C, il est possible d'allouer exactement autant de mémoire que nécessaire et la libérer au moment le plus approprié.

B. Autres solutions

Jusqu'ici, l'image est claire, s'il faut développer un programme qui va servir juste quelque fois ou s'il n'est pas nécessaire qu'il soit le plus performant possible, alors dans ce cas en génénral les langages de plus haut niveau tels que Python ou Java devraient être utilisés. Tandis que si le programme doit être fiable, efficace et facile à utiliser sur une grande échelle, alors les langages de bas niveau devraient être majoritairement utilisés.

Ceci n'est par contre pas toujours aussi simple à faire. En effet, il y a des méthodes qui permettent d'écrire dans un langage tel que Python et tout de même avoir des performances similaires au C. L'utilisation de compilateurs spécialisés, comme Codon, peut être avantageux dans certains cas, et c'est pour ça qu'il faut toujours prendre en compte l'application qu'aura le programme. En effet, lors du développement d'un programme nous devons non seulement penser à comment le résoudre, à savoir quel algorithme utiliser et autres, mais nous devons aussi prendre en compte les contraintes externes tels que les contraintes budgetaires ou temporaires. Il se peut que développer un programme en Python et utiliser un compilateur comme Codon soit moins cher et plus rapide à développer mais avec un petit défaut de performances.

VI. Dynamique de groupe

Au début, afin de mieux se répartir le travail, nous nous sommes divisés en sous-groupes de deux, et nous sommes attribués les différentes fonctions à coder. Chaque binôme s'est ensuite organisé de son côté pour réaliser ces fonctions dans les délais convenus. Une fois plusieurs semaines passées, nous avons pu repérer les qualités de chacun et donc optimiser la répartition du travail pour que chacun soit assigné aux tâches sur lesquelles il est le plus doué.

Nous avons également organisé des réunions hebdomadaires qui nous ont permis de faire le point sur l'avancement de chaque personne dans ses tâches respectives, ainsi que de dresser une liste des choses à faire pour la semaine suivante. Ces réunions servaient aussi à discuter entre nous des éventuels problèmes que nous avons pu rencontrer lors de l'avancement autonome de chacun et ainsi s'aider les uns les autres. En plus de ça nous avons créé un groupe teams afin de pouvoir s'entraider, se donner des conseils et poser des questions lorsque chacun travaillait de son côté.

VII. Conclusion

Au terme de ce projet, nous avons atteint nos objectifs. C'est à dire traduire le programme Python donné en lagage C multithreadé. Cette traduction visait à améliorer les performances du programme ; diminuer le temps d'exécution, la consommation de mémoire et la consommation énergétique, ce qui à clairement été atteint. L'apprentissage de ce nouveau langage nous a apporté un degré d'optimisation du programme bien plus important que si on avait utilisé un langage de plus haut niveau tel que Java ou Python. On peut tout de même souligner que si notre but avait été de simplement implémenter un programme dans un court délai et qu'il soit optimal, l'utilisation de compilateurs spécialisés pour des langages plus haut niveau aurait été une bonne idée.