Projet Multiprocessing : Anagrammes

Question 1 : Ecrire une première version naïve, la plus simple possible.

De façon naïve, on peut d'abord penser à parcourir tout le dictionnaire pour chaque mot afin de trouver les anagrammes (complexité en $O(n^2)$). En utilisant des structures de données basiques et très couteuses comme des listes, on se doute bien que cette approche sera beaucoup trop lente.

Question 2 : Réfléchir à des optimisations structurelles capables de réduire la complexité de l'algorithme. Réfléchir aussi à des optimisations plus locales, toujours intéressantes pour gagner du temps ou de la mémoire.

On peut songer à utiliser des ensembles et des dictionnaires, qui sont plus efficaces que les listes. On peut aussi penser à une optimisation plus structurelle de l'algorithme : plutôt que de parcourir tout l'ensemble des mots pour chaque mot afin d'en détecter les anagrammes, on peut parcourir une fois le dictionnaire et stocker chaque mot dans un dictionnaire dont la clé est composée des lettres du mot trié par ordre alphabétique. Ainsi, les anagrammes auront la même clé car ils sont constitués des mêmes lettres, et ils seront donc dans une même liste. Par exemple, les mots "car" et "rac" seront tous deux situés dans le dictionnaire à la clé "acr".

Question 3 : Paralléliser l'algorithme et l'implémenter à l'aide du module multiprocessing.

Pour paralléliser l'algorithme, on doit d'abord répartir les mots du dictionnaire à traiter par chaque nœud. La fonction *dispatcher* effectue ce travail en prenant soin de ne pas attribuer à deux nœuds différents des mots de même longueur, étant donné que la recherche des anagrammes se fait de façon indépendante dans chaque nœud. Les résultats sont ensuite renvoyés dans une queue.

On constate une bonne amélioration avec l'ajout des optimisations décrites ci-dessus et l'implémentation du multiprocessing. On arrive à un temps d'exécution de l'ordre de 0.8 seconde avec 4 processus.