Calculs de distances - Multiprocessing et Multithreading

Béline Aubergeon, Thomas Doucet, Paul Petit
MS Data Science - ENSAE Paris

Février 2022

Table des matières

1 Présentation du projet

Dans le cadre d'un projet de Machine Learning (ML) sur lequel nous avons travaillé, nous devions estimer les prix d'un grand nombre de logements en France. Il nous a paru pertinent de regarder si les prix étaient impactés par la distance entre le logement et les gares (SNCF et RATP) à proximité. Pour ce faire, il était nécessaire de calculer pour chaque logement la distance entre toutes les gares de France ce qui était très chronophage et donc allongeait considérablement le temps d'exécution de notre algorithme. C'est pourquoi nous avons décidé de mettre à profit les différents sujets abordés en cours et d'implémenter deux concepts de parallélisation, le multiprocessing et le multithreading. Notre projet est codé en Python et en Cython, langages qui nous parraissent être les plus efficients pour ce type de calculs.

Avant de comparer les performances des différentes techniques de parallélisation que nous avons implémenté pour résoudre notre problème, nous allons présenter leurs avantages et inconvénients.

2 Présentation des techniques utilisées

L'augmentation croissante de la quantité de données disponibles et accumulées à laquelle nous assistons aujourd'hui a plusieurs conséquences majeures. D'un côté elle permet d'entrainer des algorithmes de ML sur davantage de données et ainsi améliorer leur prédiction. De l'autre, travailler avec des ensembles de données de plus en plus volumineux entraîne un traitement plus lent. C'est pourquoi il souvent nécessaire de recourir à la paréllisation afin de réduire les temps de calcul. Cette technique aussi appelée traitement parallèle consiste à effectuer un ensemble de calculs coordonnés en parallèles afin de maximiser l'utilisation du CPU. Pour cela, Python propose deux librairies intégrées : le multiprocessing et le multithreading.

2.1 Multiprocessing

Un processus est une instance d'un programme en cours d'exécution. Chaque processus utilise un espace mémoire différent dans lequel il stocke les instructions en cours d'exécution en plus de toutes les données utilisées pour s'exécuter.
On parle de multiprocesseur lorsqu'un système possède plus de deux processeurs et de multiprocessing lorsqu'on demande l'exécution de plusieurs processus simultanément. Le recours au multiprocessing est certes plus lent et utilise plus de mémoire (puisque tous les processus ont un espace mémoire différent), mais il

permet de gagner en temps d'exécution. De plus, puisque chaque processus est indépendant des autres, la panne d'un processus n'affectera pas l'exécution du programme car les autres se partageront son travail.

2.2 Multithreading

Un thread désigne un ensemble d'instructions en langage machine. C'est un composant d'un processus. Plusieurs threads peuvent fonctionner en parallèle sur un seul processeur en utilisant le même espace mémoire, soit celui du processus auquel il sont rattachés. Tous les threads ont donc accès au même code et aux mêmes variables déclarées. Cela implique qu'ils sont dépendants les uns des autres. Comme les threads partagent le même espace de stockage d'un même CPU, la communication des tâches est plus rapide et l'espace de mémoire utilisé est plus faible. Cependant le multithrading en Python présente un inconvenient majeur dû à la présence du Global Interpreter Lock (GIL). Ce dernier peut être interprété comme un verou utilisé par l'interpreteur Python afin de garantir la protection de tous les objets qu'il manipule. Ainsi, plusieurs threads ne sont autorisés à accéder à l'interpréteur que l'un après l'autre. C'est pourquoi même sur les systèmes multicœurs les threads ne peuvent pas fonctionner en parallèle. Un système multithread se comporte alors comme un système à un seul thread. Il est donc possible que l'utilisation de nombreux threads dans un programme conduise à un goulot d'étranglement. Le blocage de l'exécution simultanée de plusieurs threads a également comme conséquence d'entrainer la panne de tous les threads si l'un d'eux tombe en panne.

Si ces deux techniques cherchent à réduire le temps d'exécution d'un programme, il se peut qu'en Python, l'utilisation du multiprocessing au profit du multithreaing soit plus efficace à cause de la présence du GIL. En effet, chaque processeur possède son propre GIL ce qui permet à Python d'y accéder à tout moment. Nous verrons cependant qu'il est possible de contourner ce problème dans le cas du multithreading grâce à Cython.

3 Application et présentation des résultats

La base de données qui nous a été fournie et sur laquelle nous travaillons est constituée de 37 000 [1] annonces immobilières reparties sur tout le territoire français. Nous avons récupéré les latitudes et longitudes de 1 171 gares RATP et 2 874 gares SNCF afin de calculer les distances à l'aide de la formule de haversine. Cette dernière permet de déterminer la distance entre deux points d'une sphère La formule mathématique est la suivante :

$$d_{1,2} = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Avec :

- $d_{1,2}$ la distance entre les points 1 et 2
- r le rayon de la terre
- ϕ_i la latitude du point i
- λ_i la longitude du point i

Le choix de cette fonction plutôt qu'une autre se justifie par le gain de temps entre le calcul de la distance haversine par rapport au calcul de la distance avec la fonction geodesic de la librairie geopy de Python. Nous pouvons voir Figure 1 que le temps d'exécution de la fonction geodesic est considérablement supérieur

1. Les résultats présentés ci-après ont été trouvé sur une base de 5 000 annonces pour le multiprocessing et de 500 pour le multithreading (les résultats pour 5 000 sont disponibles dans le drive). Nous avons cependant testé les techniques les plus efficaces sur l'ensemble du jeu de données.

à celui de notre fonction. Celui-ci est est multiplié par 3,5 et augmente exponentielllement alors que le test n'est effectué que sur 120 logements. Nous remarquons également que la Mean Absolute Poucentage Error (MAPE) reste relativement proche de 0,14% même lorsque le nombre de logements augmente.

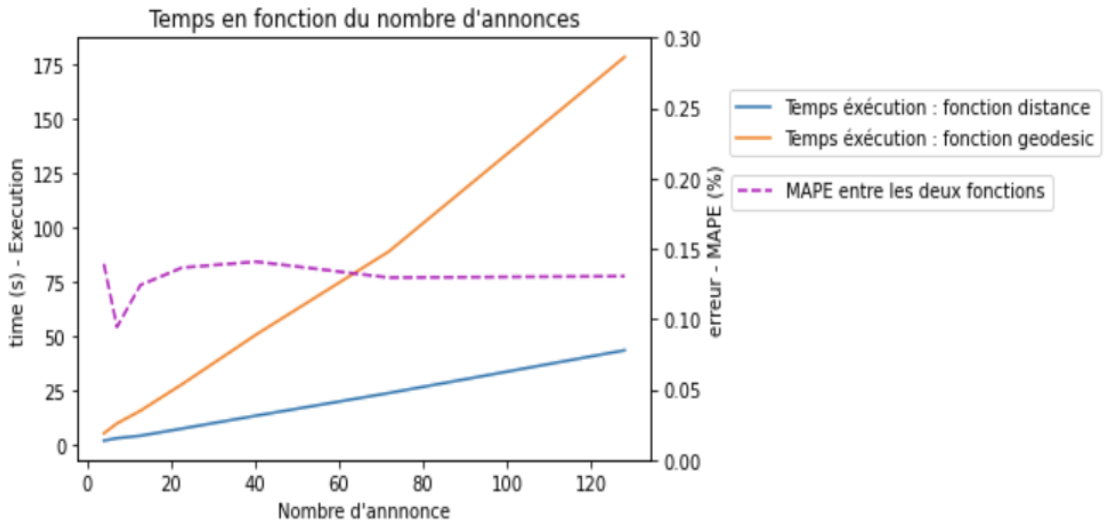

FIGURE 1 – Comparaison du calcul des distances avec geopandas et Haversine

Notre code consiste à boucler sur tous les logements en portefeuille pour regarder leur distance avec chaque gare. Cela représente 149 665 000 itérations (37 000×(1 171 + 2 874)). Nous avons limité à 5000 le nombre d'annonces pour la première partie et comparer les performances. Nous souhaitons donc faire appel à une fonction qui permettrait de diviser nos données en plusieurs groupes et d'itérer simultanément. C'est que nous permet de faire la data parallélisation.

3.1 Multiprocessing

Nous pouvons voir sur la Figures 2 une baisse drastique du temps d'exécution de notre programme lorsque nous augmentons le nombre de processeurs dans la parallélisation. Le convexité de cette courbe s'explique par le fait que plus le nombre de processeur croit, plus le nombre de données que chaque processeur doit traiter diminue. Cette diminution reste importante jusqu'à l'ajout de 16 processeurs dans notre cas, nombre à partir duquel l'augmentation des processeurs n'a plus d'effet notable sur le temps d'exécution. Attention, si on utilise un nombre de processeur supérieur au nombre de coeurs de la machine, ces processeurs seront mis en attente et l'on observera une perte de temps dans l'éxecution du programme. Nous constatons par ailleurs une augmentation du temps de communication entre les processeurs quand ils sont très nombreux. Le temps de communication est défini comme étant la différence entre le temps total et le temps pris par le processeurs le plus lent. Puisque chaque processeur doit échanger des données avec les autres, lorsque leur nombre augmente la communication se fait plus difficile. Cet effet est illustré Figure 2.
D'autre part, le temps d'exécution de chaque processeur est relativement homogène. Plus leur nombre augmente, plus le temps moyen pris par un processeur est faible. A noter toute fois que nos résultats peuvent être biaisés. En effet nous avions à disposition un cluster de l'école avec 40 CPU partagé avec d'autres utilisateurs. Il se peut donc que la performance ait été impactée par le partage des ressources.

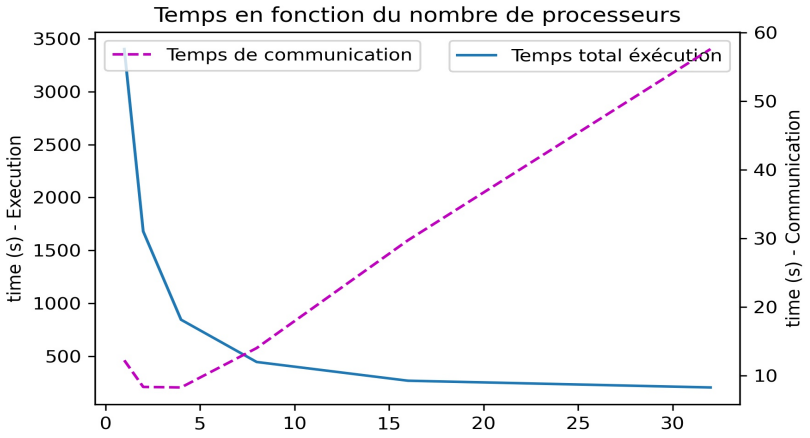

FIGURE 2 – Temps d'exécution et de communication en fonction du nombre de processeurs

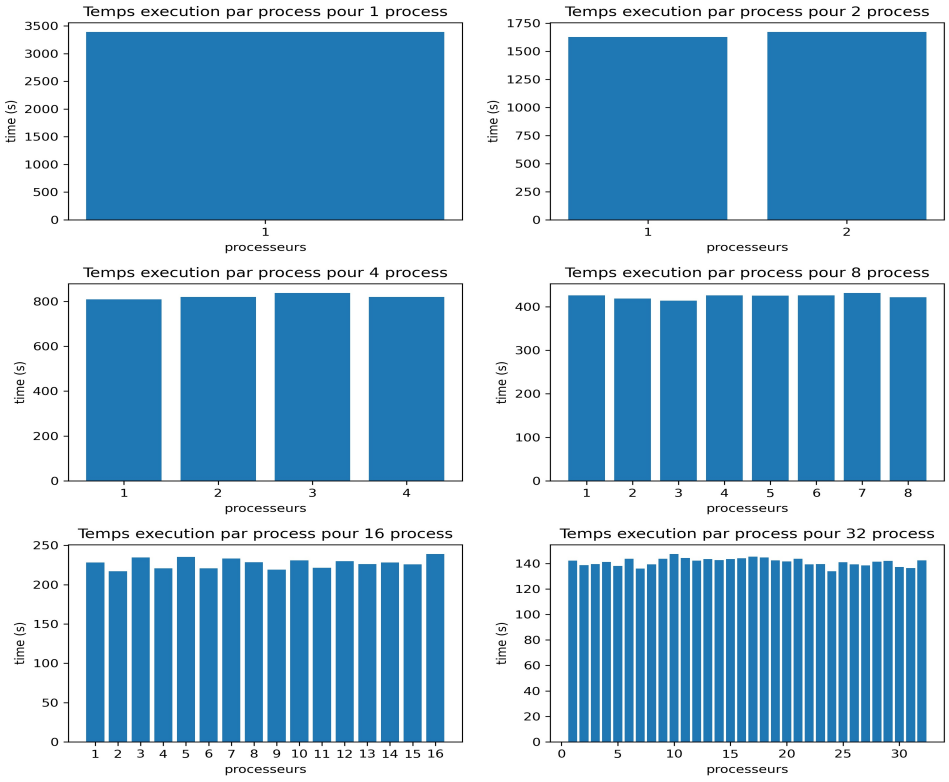

FIGURE 3 – Temps d'exécution de chaque processeur

3.2 Multithreading

Alors que notre objectif principal est de gagner en temps d'exécution, la Figure 4 nous permettent d'affirmer que ce n'est pas le cas lorsque nous implémentons du multithreading en Python. Nous voyons que plus le nombre de threads augmente, plus le temps d'exécution est elevé. Nous passons de 0,40 secondes

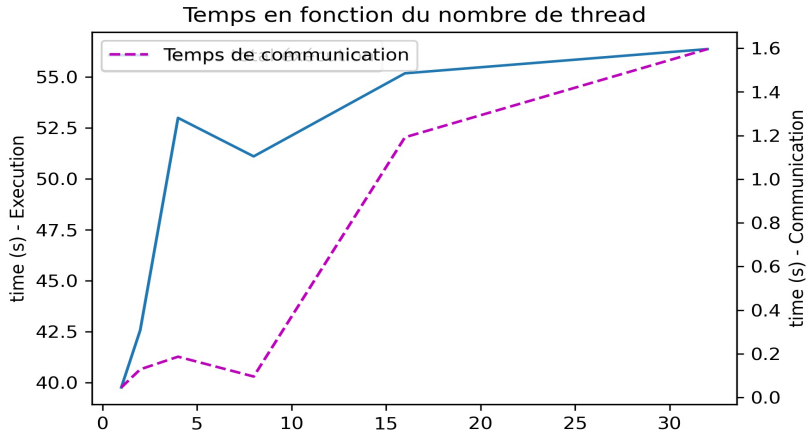

FIGURE 4 – Temps d'exécution et de communication en fonction du nombre de threads

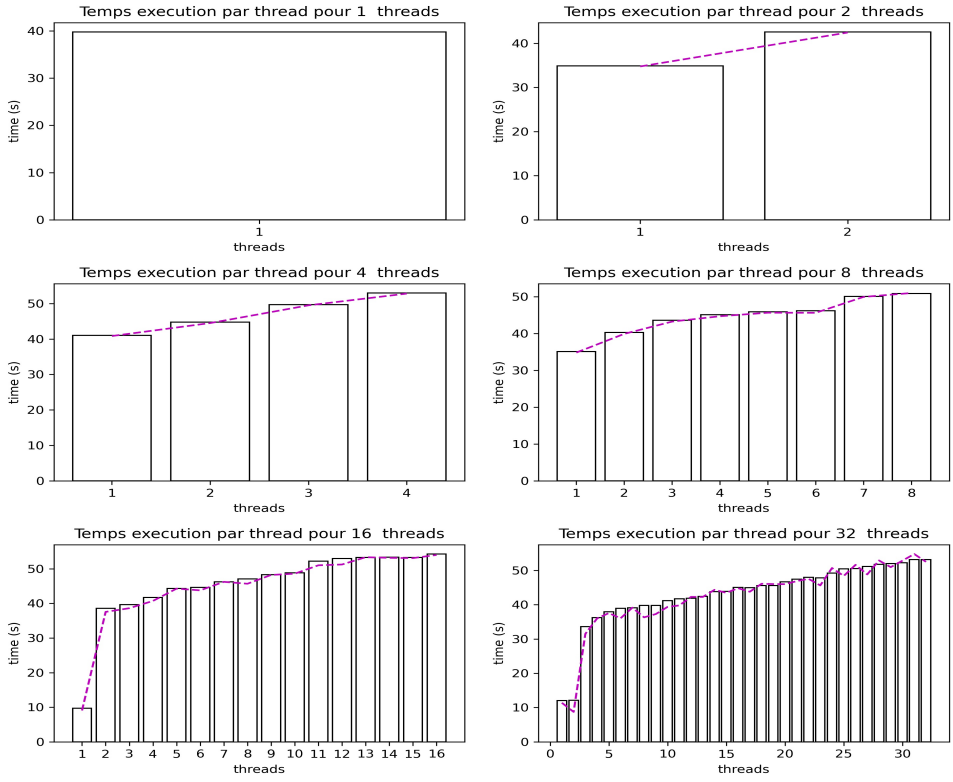

FIGURE 5 – Temps d'exécution de chaque threads

avec un seul thread à 0,57 secondes avec trente threads. Nous observons Figure 4 qu'à mesure que le nombre de threads augmente, le temps de communication augmente également. Comme nous l'avons mentionné précèdemment, il est possible qu'un nombre trop important de threads créé comme un embouteillage rendant la communication plus lente. C'est aussi ce que nous pouvons constater après lecture de la Figure 5. Les premiers threads à être exécutés sont plus rapides que les derniers. Il y a une différence de temps de 40

secondes entre le temps d'exécution du premier thread et du 32ième thread. Il est interessant de constater que même si le nombre de données traitées par thread diminue avec le nombre de threads, le temps mis par chaque thread est comprise entre 30 et 50 secondes. Il est important de mentionner que ces résultats varient avec l'ordinateur utilisé. Selon la machine, les threads se lancent en même temps ce qui créé un embouteillage ou bien ces derniers se lancent de façon séquentielle (le temps d'exécution augmente toujours mais le temps d'exécution de chaque thread est réduit selon leur nombre).

Ces résultats sont dus à la présence du GIL qui ne permet pas d'exécuter plusieurs threads en même temps. Il est cependant possible de le contourner via l'utilisation de Cython. Cython est un langage de programmation qui conserve la productivité de Python tout en autorisant l'utilisation des bibliothèques C/C++ qui permmettent l'optimisation du code numérique. L'utilisation de la fonction *nogil* en Cython autorise l'exécution d'un programme sans le GIL tant que celui-ci ne touche aucun objet Python. Cette fonction consiste à libérer le GIL pendant les opérations de blocage (liées au CPU), ce qui permet à d'autres threads Python d'exécuter leurs tâches. Nous avons implémenté notre code en Cython afin de voir si cela pouvait réduire le temps d'exécution dans le cas du multithreading.

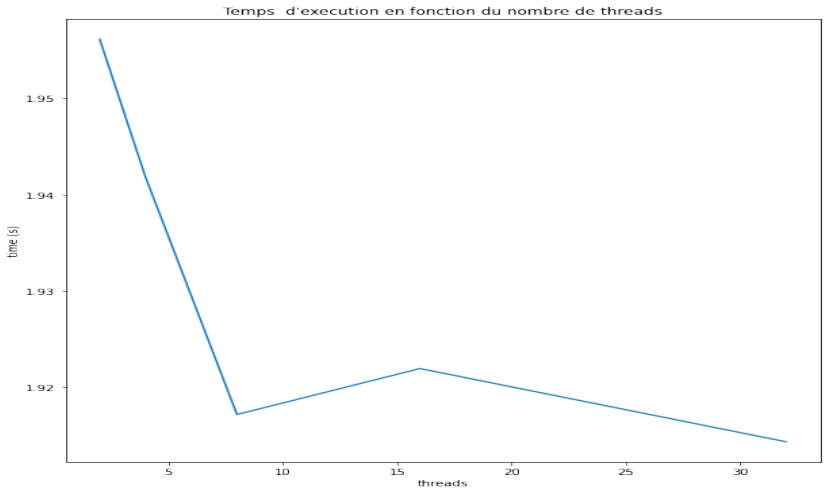

FIGURE 6 – Evolution du temps d'exécution en fonction du nombre de threads

Le gain de temps après implémentation de notre fonction en Cython est considérable. Non seulement le temps d'exécution diminue dans le cas du multithreading mais il est également bien inférieur aux résultats obtenus avec le multiprocessing. Alors que le programme est exécuté en environ 550 secondes avec 5 processeurs, il met moins de 1,93 secondes avec 5 threads.

3.3 Pour aller plus loin

Bien que nous ayons réussi à paralléliser notre code et ainsi gagné en temps de calcul, nous avons souhaité savoir s'il était possible d'optimiser d'avantage le temps d'exécution en modifiant notre fonction de calcul des distances qui est de compléxité $O(n^2)$. Nous allons nous inspirer de l'algorithme KDTree qui est utilisé notamment dans les K plus proches voisins. L'utilisation d'algorithme en arbre va nous permettre de réduire la complexité initiale en évitant de calculer des distances avec des points lointains. L'idée générale est de couper l'espace en deux de façon récursive et de s'arrêter lorsqu'il reste plus qu'un nombre restreint de points. Vous trouverez dans le notebook "Elements Logiciels Distance" la mise en place de notre fonction. A noter que ce type d'algorithme peut commettre des erreurs si le plus proche voisin se trouve de l'autre côté du plan. Afin de contrer cela, nous avons procédé à la séparation spatiale (longitude/latitude) en prenant

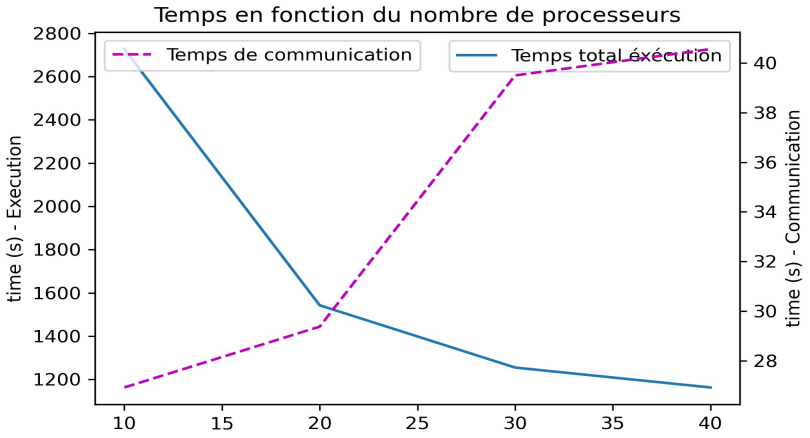

FIGURE 7 – Algorithme d'origine

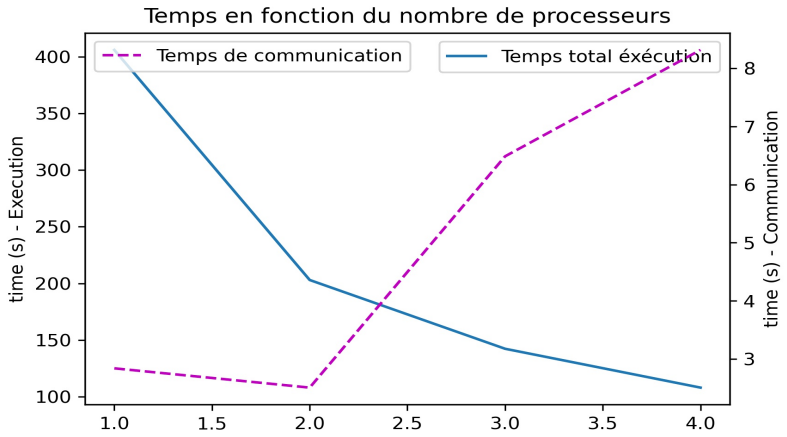

FIGURE 8 – KDTree

en compte la distance avec le point d'origine et dans un deuxième temps nous avons fixé un seuil de points après lequel nous procédons à l'ensemble des calculs des distances. L'algorithme a ensuite été parallélisé en multi-processing, et nous avons comparé les résultats obtenus avec celui d'origine (appliqués sur 30 000 annonces). Les résultats montrent une nette amélioration du temps de calcul malgré un nombre de coeurs utilisés 10 fois inférieur. En ce qui concerne les erreurs de la nouvelle méthode, on obtient un MAPE de 0.005 et 1.3% des points l'algorithme n'a pas trouvé la gare la plus proche. Il serait nécessaire d'améliorer notre algorithme pour qu'il puisse gagner en précision.

4 Conclusion

Ce projet nous a permis d'implémenter différentes techniques de parallélisation afin de diminuer significativement le temps d'exécution de notre programme tout en restant sur une interface Python qui selon

nous est plus facile d'accès. Ces problèmes d'optimisation de temps de calculs sont d'autant plus importants aujourd'hui alors que la loi de Moore, qui stipule que la puissance des CPU augmente avec les années, atteint sa limite.
Cependant, nous tenons à mentionner que nous avons du mal à comprendre pourquoi l'utilisation de Cython permet un gain de temps aussi important.