# Clustering : calcul matriciel et vectoriel – quels sont les bénéfices de la programmation concurrentielle?

Par Kevin Mwanangwa, Laurier Lavoie-Giasson et Chris David

## Introduction : quel a été le but des observations?

Le but de ce travail était de concevoir un algorithme de clustering de type Kmeans. Un cluster est un amas d’objet, gravitant autour d’un point central appelé centroïde. Dans le cadre de notre projet, il s’agissait d’un amas de points, chaque point représentant un mot. Le programme doit donc regrouper nos mots en clusters dans l’espoir d’en ressortir des informations, des liens entres les mots se trouvant dans le même groupe.

* La pertinence des liens entres les mots est-elle augmentée proportionnellement au nombre de clusters total ?
* Si oui, est-ce-que diviser les longs calculs entre plusieurs threads d’un processeur accélère le clustering ?
* Quel est le point ou ajouter des clusters cesse d’affecter significativement les résultats ?

Voilà les trois questions autour desquelles gravitaient nos réflexions et nos procédures tout au long de ce projet.

## La programmation en parallèle est-elle possible en Python?

L’expérimentation a permis de répondre à plusieurs interrogations concernant la programmation en parallèle. Est-il très long de démarrer des threads pour calculer? Est-il simple de le faire? Les threads seront-ils exécutés en parallèle, ou seront-ils tous exécutés les uns après les autres? Comment gère-t-on l’accès aux données pour empêcher que deux threads n’accèdent/ne modifient la même adresse mémoire simultanément.

## Premier blocage : le GIL (*Global Interpreter Lock*)

Le premier obstacle à la programmation concurrentielle rencontré est le *Global Interpreter Lock*. Qu’est-ce que le *GIL*?

«In CPython, the global interpreter lock, or GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe. (However, since the GIL exists, other features have grown to depend on the guarantees that it enforces.) »

Python.org

En effet, comme expliqué sur le site web officiel du langage Python, le GIL est un mutex qui limite l’accès aux objets Python par un seul thread à la fois, ce qui empêche plusieurs threads d’exécuter des opérations en même temps. Cela rend théoriquement la programmation concurrente impossible en CPython. Cependant, durant la petite recherche, l’équipe a fait la découverte d’un module très intéressant qui, supposément, permet de programmer au-delà de cette limite.

## Contourner le GIL : découverte du module multiprocessing

Grâce au module multiprocesing, contourner ce fameux obstacle fut nettement plus accessible qu’initialement anticipé. En fait, ce module contient absolument toutes les classes nécessaires à la programmation en parallèle, ce qui fait de la lecture de la documentation une tâche remarquablement rapide.

Multiprocessing contient diverses classes qui permettent de contourner le GIL, en proposant de la programmation concurrente basée sur les processus, et non sur les threads. Chaque processus a sa propre mémoire sur laquelle il détient l’accès exclusif. Le module prévoit que les processus doivent avoir des moyens de se synchroniser ainsi que de communiquer entre eux. C’est pourquoi il contient les classes suivantes, qui ont été nécessaires au fonctionnement de notre programme:

* multiprocessing.sharedctypes.Array

Cette classe permet de partager un tableau de taille N entre des processus. Ce tableau « shared C type », peut donc ensuit être utilisé par la librairie numpy afin de partager une série de données entre plusieurs processus. Pour manipuler ce tableau en tant que matrice numpy, il suffit d’y accéder avec la fonction numpy.from\_buffer(tableau).reshape((n,m)), qui retourne une matrice de taille (n sur m). Par la suite, la matrice peut être utilisée individuellement comme n’importe quelle matrice numpy.

* multiprocessing.Process

Cette classe est la classe principale qui sert à démarrer les processus

* multiprocessing.Manager

Cette classe est un wrapper pour obtenir des variables partagées diverses. Dans le cas de programme, la méthode manager.dict() a été utilisée, afin d’obtenir un dictionnaire partagé entre les processus. Un dictionnaire qui permet de réunir/regrouper ensuite les résultats des calculs individuels de chaque processus.

## Quels sont les gains en performance?

Cette série de test nous a permis de d’arriver à ce résultat :

Tout d’abord, on remarque que le plus grand écart se situe entre 20 clusters et 50 clusters. C’est dans cette zone que les clusters se précisent. Avant 20, il y en a trop peu, il est donc facile et donc moins long de calculer les distances, non seulement parce que le nombre de calculs est plus bas, mais aussi parce cette dispersion crée des amas larges qui ne bougent pas beaucoup. Entre 20 et 50, on remarque des mouvements de plus en plus fin et précis, il arrive d’avoir par exemple, moins de 10 changements pendant plusieurs itérations consécutives. C’est aux alentours de 100 où l’on a remarqué qu’ajouter des clusters apporte moins de résultats. Il est vrai que les itérations sont plus longues à calculer, mais le nombre élevé de clusters réduit le nombre d’itérations avant de compléter l’opération, il est plus facile de trouvé son cluster final au début des calculs par exemple.

De plus, on remarque également que les bienfaits du multithreading se font sentir proportionnellement à la difficulté/au nombre de calculs. On remarque également que le saut de 1 thread à deux est le plus accessible et le plus significatif.

## Conclusion

Malgré notre temps et nos ressources limitées, nous avons réussi à tester et à prouver nos hypothèses initiales. Il est vrai que la direction de ce projet s’est légèrement éloignée du but premier, le Clustering. Ce concentrer sur les multithread nous a cependant quand même permis de faire les deux en même temps en réduisant considérablement le temps nécessaires aux calculs de nous différents tests.