1 Introduction et présentation de l'algorithme

Nous avons fait le choix de travailler sur l'article [4], qui est une méthode d'extraction des mots-clefs d'un texte à partir d'une analyse de graphe. Cet article nous a paru intéressant a bien des égards, puisqu'il fait le lien entre deux domaines (NLP et théorie des graphes) de façon originale, et comporte une série de méthodes qu'il serait intéressant de mettre en oeuvre de façon plus efficace.

De façon simple, l'algorithme procède de la façon suivante:

1. encoder le texte en graphe non-dirigé, en faisant défiler une fenêtre sur ce texte et reliant les mots y apparaissant. Ceci donne, par exemple, le graphe ci-dessous :

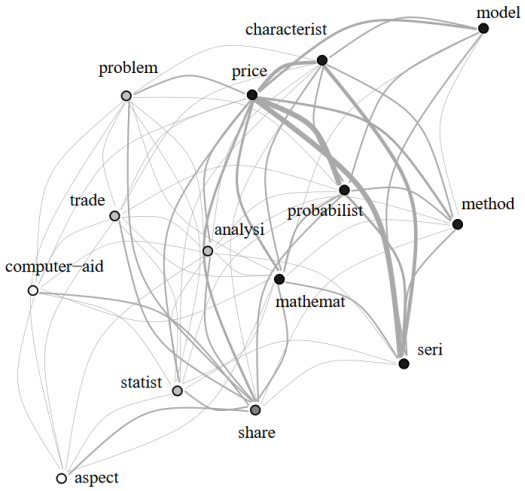

Figure 1: *Encodage d'un texte (filtré) en graphe*

2. extraite les mots-clefs à partir du graphe

   (a) effectuer la décomposition en k-core du graphe: identifier au sein du graphe initial les sous-graphes (k-cores) au sein desquels tous les noeuds ont au moins k-arcs **reliés à des noeuds intérieurs à ce même k-core**. Comme nous le verrons par la suite, ce problème est loin d'être immédiat.

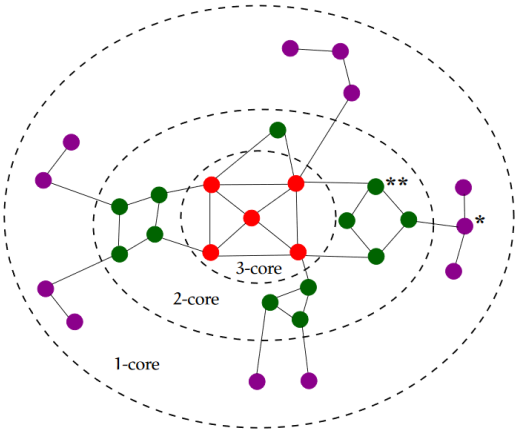

Figure 2: *Illustration de la décomposition en k-cores d'un graphe*

   (b) calculer la densité de chaque k-core, qui est définie comme:

$$\text{densite(sous-graphe)} = \frac{|\text{nombre d'arcs}|}{|\text{nombre de noeuds}| \times (|\text{nombre de noeuds}| - 1)} \quad (1)$$

   (c) une fois les densités calculées, déterminer, à l'aide du critère du coude, un seuil au-delà duquel les k-cores aux densités plus faibles que ce seuil sont considérées comme contenant des mots peu informatifs.

   (d) *output* les mots-clefs issus des k-cores ayant une densité supérieure au seuil déterminé à l'étape précédente.

Des variantes de ce modèle sont proposées dans l'article (décomposition en K-truss, identification du meilleur k-core à partir de CoreRank, etc...). Nous avons fait le choix de nous concentrer sur le modèle détaillé ci-dessus, afin de pouvoir analyser en détail les manière de rendre les algorithmes plus efficaces. Nous pensons par ailleurs que les techniques utilisées peuvent, avec un peu d'adaptation, être adaptées à la totalité des variantes proposées dans l'article.

A l'aide des éléments du cours, ainsi que d'autres qui nous ont paru intéressant, nous avons travaillé pour implémenter cette méthode de la façon la plus efficace possible. Nous nous sommes concentrés sur la deuxième partie (une fois le graph extrait du texte), ce qui représentait la partie la plus intéressante, en plus de nous permettre une analyse plus approfondie (possibilité de générer rapidement des *corner cases* en créant des graphes synthétiques, indépendants d'un texte).

Étant donné que le problème contient une multitude de fonctions, nous nous sommes concentrés à rendre plus efficaces deux fonctions caractéristiques posant des problématiques très différentes :

1. la fonction de détection des *k-core*, qui nécessite d'interagir avec un objet NetworkX et n'est pas facilement parallélisable étant donné que sa boucle modifie en permanence des variables partagées.
2. la fonction permettant de trouver le "coude", exemple de *petite fonction*, qui ne manipule pas d'objets complexes mais peut prendre du temps dans des cas extrêmes. Le défi est alors de mieux faire, via Cython et des calculs en parallèle, que l'utilisation de fonction très optimisées comme celles sur numpy.

2 Algorithme de détections des k-cores

Le code pour reproduire les résultats ci-dessous se trouve dans le notebook `1_k_core_decomp.ipynb`.

2.1 Optimisation algorithmique

La partie la plus lente étant l'extraction de k-cores, nous avons concentré d'abord nos efforts sur son optimisation algorithmique.

Algorithm 1 Décomposition en k-core naïve

Input: graphe G=(V,E)
Output: dictionnaire avec le k-core auquel chaque noeud appartient

1: **while** $|V| > 0$ **do**
2: $v \leftarrow$ element de p avec la plus faible valeur (au sens de dict(clef, valeur))
3: $c[v] \leftarrow p[v]$
4: voisins $\leftarrow$ voisins de v selon V
5: $V \leftarrow V \setminus \{v\}$
6: $E \leftarrow E \setminus \{(u, v) | u \in V\}$
7: **for** $u \in$ voisins **do**
8: $p[u] \leftarrow$ max($c[v]$,$degré(u)$)

Notons $|E|$ le nombre d'arcs. Cette décomposition est en O($|E|^2$) pour un graphe sparse et O($|E|^3$) pour un graphe dense. Face à cela, nous avons vu qu'en 2002 [1] propose un algorithme en O($|E|$).
L'idée générale de l'algorithme est la suivante :

Algorithm 2 Décomposition en k-core (Batagelj, Zaversnik, 2002)

Input: graphe G=(V,E)
Output: dictionnaire avec le k-core auquel chaque noeud appartient

1: Calculer le degré de tous les noeuds
2: Classer les noeuds par ordre croissant de leur degré (V devient ordonné)
3: **for** $v \in V$ (ordonné) **do**:
4: core[v]=degré[v]
5: **for** $u \in$ (voisins de v) **do**:
6: **if** $degré[u] > degré[v]$ **then**
7: $degré[u] = degré[u] - 1$
8: réordonner V

L'article propose une variante plus efficace de l'algorithme, que NetworkX a implémente dans son code source et dont nous nous sommes inspirés pour une fonction (comme mentionné en commentaire dans le code). L'idée générale, que [1] détaille et illustre, est d'utiliser une variante de *bin-sort* pour réordonner V à l'étape 8 étant donné que les valeurs des degrés des noeuds sont comprises entre 0 et $n - 1$.

Afin de simuler la vitesse d'exécution de ces algorithmes, nous avons simulé des graphes à l'aide de la fonction *duplication divergence graph* de la librairie *NetworkX*. Celle-ci nous permet de générer des graphes avec des k-cores de taille très variable, comme c'est souvent le cas en analyse de texte. Ceci nous permet d'effectuer des analyses proches du problème initial sans pour autant avoir à choisir des textes, etc...

Les résultats (l'échelle logarithmique à été choisie pour une meilleure visibilité) sont les suivants :

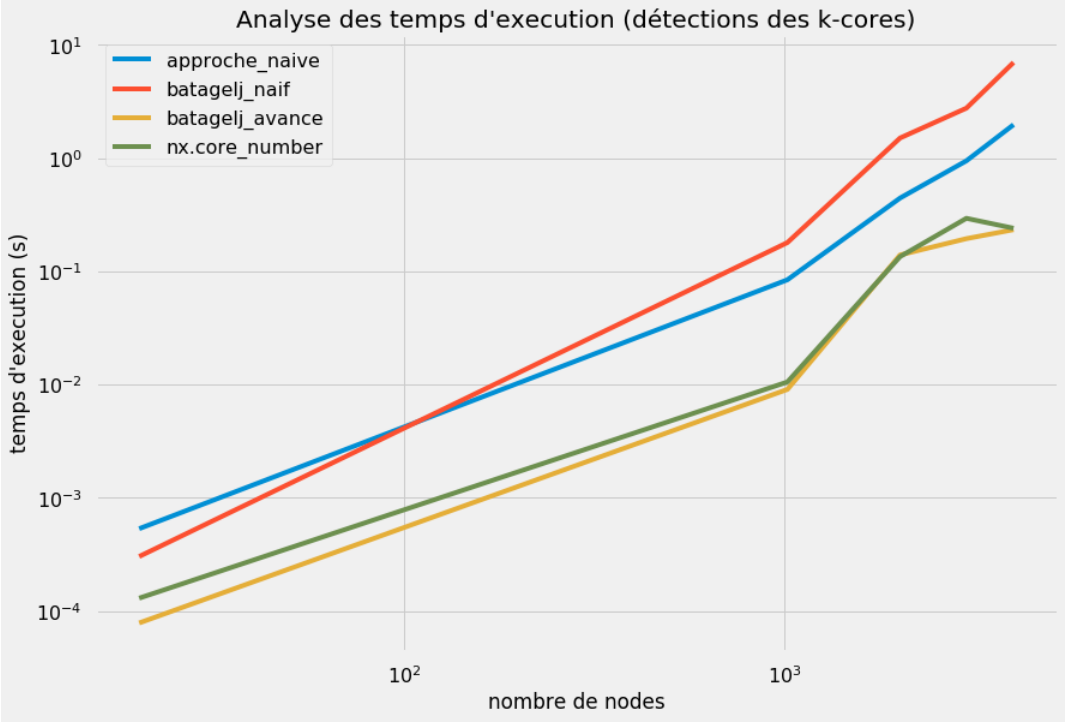

Figure 3: *temps d'exécution selon le nombre de nodes, en échelle logarithmique*

Nous voyons que, sans surprise, l'approche avancée de Batagelj et la fonction nx.core_number ont quasiment le même comportement. Par contre, il est surprenant de voir que avec plus de 100 noeuds l'approche naïve devient plus efficace que celle de Batagelj *basique*. Ceci montrer bien l'intérêt de lire en détail les articles présentant les algorithmes.

2.2 Parallélisation

L'optimisation algorithmique étant par nature limitée, nous nous sommes ensuite tournés vers des moyens de distribuer les calculs afin d'améliorer les performances de la décomposition en k-cores. Le revue de littérature présentée dans [2] décrit différentes approches parallélisées et distribuées à cette fin. N'ayant pas à notre disposition d'architecture distribuée qui nous aurait permis de vraiment évaluer les performances d'un algorithme distribué, nous avons choisi de nous intéresser aux algorithmes parallélisés.

2.2.1 Essais en CUDA

Dans un premier temps, nous avons souhaité réaliser une implémentation en GPU afin d'exploiter la parallélisation massive permise par ces derniers. L'objectif était de coder l'algorithme à l'aide du framework CUDA. Le seul article que nous avons trouvé proposant un algorithme adéquat était [5]. La procédure consiste en deux algorithmes parallélisables : le premier (algorithme 2 dans l'article) permet de calculer le k-core maximal dun graphe, et le deuxième (algorithme 4 dans l'article) exploite cette information pour élaguer itérativement les cores inférieurs tout en conservant de manière astucieuse les identifiants des noeuds supprimés. Nous n'avons eu aucun problème à calculer le k-core maximal, mais nous n'avons pas réussi à obtenir une décomposition correcte en output du second algorithme.

Après plusieurs jours de travail nous avons demandé conseil à l'un des auteurs de la revue de littérature [2] – qui est également notre professeur en cours d'apprentissage avancé sur les graphes – qui nous a confirmé que, selon lui, la méthode proposée ne pouvait fonctionner car elle supprimait itérativement les noeuds en partant du core le plus élevé (contrairement aux autres algorithmes qui partent du core le plus faible), détruisant ainsi une information nécessaire dans certains à la détections du numéro de core des noeuds restants. Nous avons donc choisi de ne pas continuer à investiguer cette méthode et nous avons choisi à la place de travailler sur algorithme de parallélisation basé sur du multithreading.

2.2.2 Parallélisation en Python

Nous avons alors choisi d'implémenter l'algorithme parallélisé présenté dans [3]. Le pseudo-code associé est présenté ci-dessous (algorithme 3). La parallélisation est opérée en découpant le graphe en blocs de noeuds

que l'on envoie aux différents threads au fur et à mesure qu'ils terminent leurs traitements respectifs.

- Dans la première partie (lignes 2 à 4), l'array *deg* est scanné en parallèle par chaque thread, et le thread qui traite le noeud courant l'ajoute à son *buffer* local si son coeur actuel est égal au niveau courant (*level*).
- La deuxième partie (lignes 5 à 13) correspond au traitement par chaque thread des noeuds contenus dans son *buffer* local suite à la première étape. Un noeud du buffer est traité en traitant séquentiellement ses voisins : si leur coeur est supérieur au niveau courant, on décrémente cette valeur d'une unité ; si cela les place dans le coeur correspondant au niveau courant, ils sont alors ajoutés à la fin du buffer. Le traitement du buffer se termine lorsqu'il ne reste plus de noeuds à traiter, et l'algorithme se termine lorsque tous les noeuds du graphe ont été traités.

Algorithm 3 Parallel K-core decomposition (PKC)

Input: un array contenant les degrés de chaque noeud du graphe
Initialisation:
Variable globale : $i \leftarrow 0$
Variables locales aux threads : $l \leftarrow 0$, $s \leftarrow 0$, $e \leftarrow 0$, un array *buff* de taille $n/n\_threads$
Output: un array contenant la décomposition en k-cores

```
1:  while i < n do
2:      for v = 0 to n − 1 do
3:          if deg[v] = l then
4:              buff[e] ← v; e ← e + 1
5:      while s < e do
6:          v ← buff[s]; s ← s + 1
7:          for u ∈ Adj(v) do
8:              if deg[u] > l then
9:                  a ← atomicSub(deg[u], 1)
10:                 if a = l + 1 then
11:                     buff[e] ← u; e ← e + 1
12:                 if a ≤ l then
13:                     atomicAdd(deg[u], 1)
14:     Barrier synchronization
15:     atomicAdd(i, e)
16:     s ← 0; e ← 0; l ← l + 1
```

Dans un premier temps, afin de bien comprendre la manière dont l'algorithme est parallélisé, nous proposons une implémentation basée sur la librairie standard *threading* de Python. L'algorithme étant basé sur la lecture et l'écriture de variables partagées, des situations de concurrence (*race conditions*) peuvent survenir et rendre les résultats incorrects. Pour protéger ces variables, nous les encapsulons dans une classe et nous implémentons des *locks* avant leur lecture et leur écriture. Ainsi, nous nous assurons que les threads se synchronisent avant d'accéder aux ressources partagées, au prix d'un certain coût de latence du fait de ces synchronisations préalables.

Au-delà de son intérêt pédagogique afin de comprendre comment est parallélisé l'agorithme – ce qui s'est avéré très utile pour écrire ensuite la version Cython – elle n'a a priori pas beaucoup d'intérêt du point de vue des performances. En effet, l'existence du GIL (Global Interpreter Lock) en Python empêche l'exécution simultanée de plusieurs threads. Son objectif est d'empêcher l'existence de situations de concurrence entre les threads, ce qui apporte une grande stabilité, mais au prix d'une impossibilité de réaliser une véritable parallélisation sur les threads. En conséquence, nous proposons également une implémentation de l'algorithme basé sur la librairie standard *multiprocessing*. Cette dernière permet réellement de paralléliser les calculs dans la mesure où plusieurs processus Python sont créés et exécutés en parallèle. Dans ces conditions, le GIL ne pose alors pas de problème car aucun *multithreading* n'est réalisé sur ces processus. Là encore, nous utilisons des *locks* – via les objets *thread safe* de classes *Array* et *Value* proposés par *multiprocessing* – afin d'empêcher les situations de concurrence.

2.2.3 Parallélisation en Cython + OpenMP

Comme vous l'avons vu, les deux implémentations précédentes ont d'importantes limitations. Le *multithreading* est de fait très limitée par l'existence du GIL. Le *multiprocessing* va quand lui être limité dans le sens où cette méthode est particulièrement efficace pour des problèmes indépendants et CPU-bound (i.e. dont la

limite est la puissance de calcul du/des CPU). Dans notre cas, la demande CPU est probablement limitée, et surtout la parallélisation n'est pas totale car les processus vont devoir lire/écrire des ressources partagées, ce qui entraîne beaucoup d'*overheads*. Face à ces limites, nous avons voulu implémenter une parallélisation en Cython, basée sur l'API OpenMP. Cette approche a le double avantage de permettre de libérer le GIL, et présente une structure en threads native qui permet de partager des ressources efficacement entre les threads.

Le code associé à cette implémentation est visible dans le notebook associé. Nous définissons les variables locales via des pointeurs dans un premier temps, puis nous les instancions de manière locale sur chaque thread dans l'environnement défini par *with nogil, parallel():*. La boucle for parallélisable est réalisée grâce à la commande *prange*, qui permet une parallélisation thread safe basée sur OpenMP. L'accès en lecture/écriture aux variables partagées est réalisé grâce à des opérations atomiques, appelées via la librairie de C. Enfin, conformément à la présentation de l'algorithme, nous devons poser un verrou à la fin de la deuxième boucle while pour incrémenter le niveau, ce que nous faisons en rappelant le GIL, assurant une synchronisation préalable des threads.

Le code de notre implémentation est correctement compilé. En revanche, il semble provoquer une fuite de mémoire qui entraîne souvent le crash de la console. Nous n'avons pas malheureusement pas réussi à trouver sa source, et le peu de documentation et d'examples traitant de notre cas spécifique (manipulation d'un buffer local avec appel/écriture à des ressources globales) ne nous a pas permis de conclure cette implémentation avec succès.

2.3 Résultats

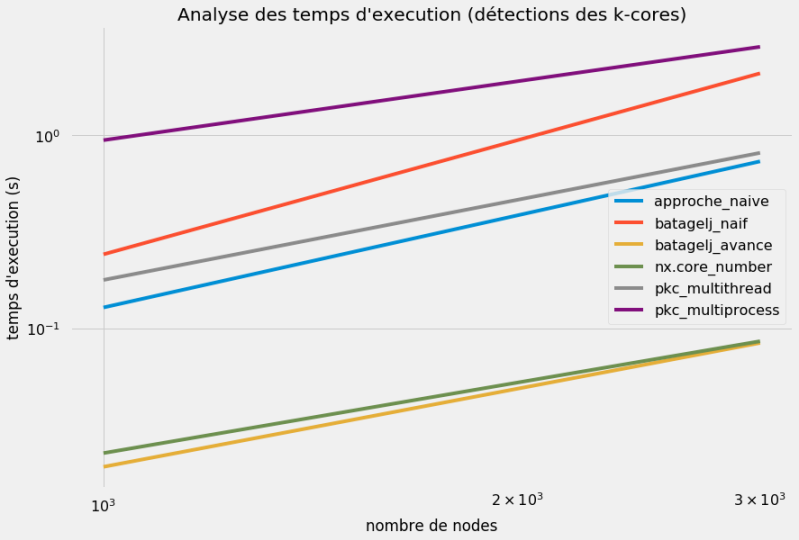

Figure 4: *Temps d'exécution selon le nombre de noeuds du graphe (moyenne sur plusieurs itérations), en échelle logarithmique*

La figure 4 permet de comparer le temps d'exécution de chacun des algorithmes que nous avons implémentés en fonction du nombre de noeuds du graphe. Voici les observations que nous pouvons effectuer :

- les approches "state of the art" utilisées dans les principales libraires d'analyse de graphes se détachent nettement des autres approches grâce à une importante optimisation algorithmique.
- comparé aux approches naives, sur le nombre de nodes testé, les approches basées sur la distribution des calculs en Python n'apportent pas un gain significatif voire détériorent les résultats. Au vu des pentes des courbes, il est cependant possible que ces dernières passent mieux à l'échelle pour des graphes de très grande taille (par exemple si nous avions un graphe social de plusieurs millions de personnes - comme ont certaines entreprises aujourd'hui).
- le multiprocessing affiche de moins bons résultats que le multithreading, ce qui semble étonnant au vu des éléments relatifs au GIL mentionnés plus haut. Il est possible que la tâche que nous réalisons soit dans une large mesure I/O bound car les différents processus doivent sans cesse attendre l'accès à une ressource partagée, de sorte que le coût de création des processus devient rédhibitoire.

- Du fait des limites évoquées précédemment, nous n'avons pas pu comparer les performances de notre implémentation parallélisée en Cython. La fonction que nous analyserons ensuite, plus simple, permettra d'étudier plus en détail les avantages de cette idée, et leur impact sur le temps d'exécution.

3 Algorithme de détections du *coude*

Le code pour reproduire les résultats ci-dessous se trouve dans le notebook `2_critere_du_coude.ipynb`.
Une fois le travail effectué sur la fonction précédente, nous nous sommes penchés sur l'optimisation de la partie seconde de l'article, l'extraction des mots clefs une fois les k-cores extraits.
S'il est possible de paralléliser comme précédemment, il nous a semblé que la simplicité relative de la tâche permettait un *deep dive* dans le codage en Cython et ses conséquences sur la rapidité d'algorithmes simples. Nous avons ensuite effectué une parallélisation simple sous Cython.

3.1 Cythonisation

La fonction que nous avons optimisé est celle intervenant après les densités calculées, donnant le seuil selon le critère du coude.

Ce choix a été fait du fait de la simplicité de la fonction et de l'apparence première qu'il est difficile de faire plus efficace que numpy dans un tel cas, sans paralléliser. Nous montrerons le contraire !

L'algorithme est le suivant :

Algorithm 4 Détermination du coude

Input: liste des densités (classées)
Output: "coude" (point seuil pour la suite, dans l'algorithme général)

1: Calculer la distance entre chaque point et la ligne allant du premier au dernier
2: **Output** le point dont la distance est maximale. Si toutes les distances sont nulles, retourner premier =0

3.1.1 Cython naïf

Une première approche est de simplement rajouter `%% Cython` au début de la cellule (sans changer autre chose).

3.1.2 Réecriture de fonctions en C

A partir du code existant, nous avons crée des fonctions `cdef`, permettant d'exécuter du code en C. Ceci a été un défi, étant donné qu'aucun de nous deux n'avait travaillé sur du C auparavant. Nous avons aussi déclaré les types de variables, et replacé les fonctions Python (`def`) en fonction mixtes `cpdef`, qui permettent de fonctionner en C lors que l'on utilise les types fondamentaux de C, et passer en Python sinon.

3.1.3 Ajout des memoryviews

Étant donné que nous devons travailler sur des array, un élément qui diminue la performance est l'utilisation de pour accéder aux données. Afin d'éviter cela, `Cython` propose les Memorviews, qui sont des strcutres C qui pointent vers les arrays Numpy.
Nous pouvons les voir dans notre fonction par exemple dans des déclarations comme `cdef double[:] first_point`.

3.1.4 Désactivation des vérifications

Enfin, la documentation de `Cython` précise que lors d'un `lookup` sont effectué deux checks (au moins):

- Vérification qu'il n'y ait pas d'indices négatifs (et si c'est le cas ils sont traités de façon appropriée ensuite).
- Contrôle des limites (*bounds checking*).

Éliminer ces deux checks est rapide via des décorateurs, mais pour cela il a fallu auparavant modifier légèrement le code pour ne plus nécessiter d'indices négatifs (ce qui était le cas et a été changé).

3.1.5 Déclaration des `numpy arrays` comme contigus

Nous pouvons déclarer les arrays comme contigus afin d'accéder plus rapidement, en les déclarant par exemple `c_elbow_continous(double[:, ::1])` dans la déclaration de la fonction (noter le `::1`). Si cette méthode marche, étant le type de manipulation d'arrays effectuée ici il est peu probable que la performance s'en trouve grandement améliorée. Cette intuition sera par la suite confirmée.

3.2 Calcul parallèle

Dans la section précédente, nous avons vu comment il pouvait être complexe de mettre en place la parallélisation pour un algorithme comme la détection de k-cores dans un graphe.

Dans le cas qui nous occupe à présent, nous allons privilégier la simplicité. Ainsi, dans la boucle qui calcule la distance d'un point à la ligne (premier point, dernier point), nous enlevons le GIL *Global Interpreter Lock* qui empêcherait la parallélisation. Ceci a des conséquences concrètes sur notre code, étant donné que nous ne pouvons plus utiliser de variable ou fonction Python à l'intérieur du `nogil`. Une fois le GIL desactivé nous pouvons effectuer les calculs en parallèle.

Ce procédé est à priori simple à l'aide de `Cython` et `OpenMP` (même si le bebugging demeure parfois complexe et chronophage). Nous avons simplement utilisé deux fonctionnalités issues de `cython.parallel : with nogil:` avant la boucle, et `prange` à l'intérieur de celle-ci. Remarquons qu'utiliser `with nogil, parallel():` et un `range` "normal" semble donner aussi des résultats convenables.

D'un point de vue technique, cette modification de notre boucle permet à `OpenMp` d'ouvrir un *Thread Pool* et distribuer le calcul.

3.3 Résultats

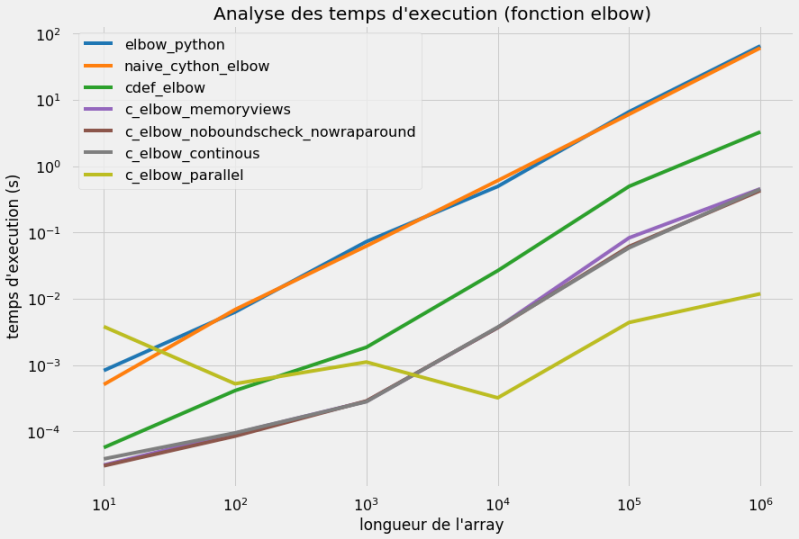

Figure 5: *Temps d'exécution selon la longueur de l'array dont il faut trouver le coude (moyenne sur plusieurs itérations), en échelle logarithmique*

- la fonction Cython naïve (ajout simple de `%% Cython` sur le code Python) ne semble pas avoir d'effet au delà des petits arrays
- on remarque que la première amélioration conséquente de la performance a lieu avec l'ajout des fonctions `cdef` (la fonction devenant plus de dix fois plus rapide), et une deuxième fois avec les `MemoryViews`. Les modifications suivantes n'ont pas changé de façon significative la performance de l'algorithme.
- dès que les arrays sont assez longs pour compenser la coût ajouté de la méthode, la parallélisation devient très rentable. Pour un array de taille 10^6, notre algorithme en parallèle est 37 fois plus rapide que la version `Cython` la plus efficace, et au total plus de 5000 fois plus rapide que l'algorithme naïf. Nous remarquons cependant que a partir d'un certain seuil le temps d'exécution de l'algorithme parallélisé commencé à augmenter, et nous pouvons émettre l'hypothèse que après un seuil les ratio entre les temps d'exécution des algorithmes demeurent relativement stage.

4 Conclusion

Nous avons vu que sur un sujet complexe, à la croisée de plusieurs disciplines, les éléments vus en cours permettent d'optimiser notre code de façon très significative, aussi bien pour des fonctions complexes qui sont le sujet de papiers de recherche (trouver les k-cores), que de algorithmes plus simples (trouver le "coude"). Ceci dit, et comme nous l'avons évoqué et vécu, ces changements peuvent être relativements complexes, surtout lorsque la parallélisation nécessite de grandes précautions pour éviter les fuites de mémoire.

Nous garderons en souvenir ces enseignements lors de nos projets à venir, académiques ou professionnels.

References

[1] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks, 2003. cite arxiv:cs/0310049.

[2] Apostolos Papadopoulos Fragkiskos Malliaros, Christos Giatsidis and Michalis Vazirgiannis. The core decomposition of networks: Theory, algorithms and applications. In Springer, editor, *The VLDB Journal*, 2019.

[3] Humayun Kabir and Kamesh Madduri. Parallel k–core decomposition on multicore platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1482–1491. IEEE, 2017.

[4] Antoine J.-P. Tixier, Fragkiskos D. Malliaros, and Michalis Vazirgiannis. A graph degeneracy-based approach to keyword extraction. In Jian Su, Xavier Carreras, and Kevin Duh, editors, *EMNLP*, pages 1860–1870. The Association for Computational Linguistics, 2016.

[5] Alok Tripathy, Fred Hohman, Duen Horng Chau, and Oded Green. Scalable k-core decomposition for static graphs using a dynamic graph data structure. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1134–1141. IEEE, 2018.