Implémentation parallèle du calcul de centralité secondaire de Kermarrec et al.

Rapport projet final Visualisation

Théo GILBERT

Thomas CORREIA

Simon MERCIER

Sommaire

I) Introduction 2

II) Implémentation 2

1) Approche multi-threading suggérée par Kermarrec 2

2) Approche multi-threading retenue 3

3) Choix du critère d'arrêt 3

4) Le plugin 4

III) Tests 6

1) Graphe de Barbell 7

2) Graphes aléatoires 8

3) Aéroports 11

IV) Conclusion 14

1) Résultats 14

2) Difficultés et travail au cours du projet 14

I) Introduction

La centralité est une métrique calculable sur chaque nœud d'un graphe qui sert à déterminer l'importance de ce nœud pour ce graphe. Dans Tulip, il existe un plugin « betweenness centrality » qui itère sur tous les nœuds du graphe pour calculer cette centralité. L'inconvénient de cette méthode est que cela est long lorsque le graphe est conséquent.

L'article de Kermarrec présente une nouvelle métrique appelée « centralité de second ordre ». L'idée principale est d'effectuer une marche aléatoire sur un graphe et de calculer pour chaque nœud une liste de « temps de retour ». En effet, chaque nœud stocke dans une liste les intervalles de temps qui séparent les visites de cette marche aléatoire.

De plus, à partir 3 temps de retour stockés, chaque nœud calcule sa centralité à chaque fois qu'un nouveau temps de retour est enregistré. A noter qu'à l'inverse de betweeneness centrality, le nœud le plus central est celui ayant la métrique la plus faible et non la plus élevée.

L'objectif de notre projet est de reprendre ce calcul de la centralité de second ordre en utilisant le multi-threading afin de d'en améliorer les performances.

II) Implémentation

1) Approche multi-threading suggérée par Kermarrec

L'article est expliqué pour une seule marche aléatoire. Néanmoins, un paragraphe de l'article explique brièvement ce qu'il serait possible de faire. Il est suggéré que chaque nœud lance une marche aléatoire. Ainsi, chaque marche aléatoire serait propre à un seul nœud, et chaque nœud attendrait le retour de sa marche aléatoire pour avoir un nouveau temps de retour.

Nous pensons que cette approche est mauvaise pour 2 raisons :

- Lancer un trop grand nombre de threads est inutile puisque la plupart des machines sont limitées à 8 ou 16 threads, et même des machines très performantes ne peuvent pas atteindre plusieurs milliers de threads (nous parlons bien ici de threads s'exécutant réellement en parallèle, et non pas uniquement d'objets thread déclarés dans le code).
- La marche aléatoire qui est associée au nœud i marche inutilement tant qu'elle ne revient pas à son nœud de départ. Or, cette marche aléatoire pourrait calculer des temps de retour sur tous les autres nœuds sur lesquels elle passe, et ainsi faire économiser beaucoup de temps à l'algorithme.

C'est pourquoi nous avons décidé de lancer un nombre fixe de threads que l'utilisateur du plugin peut fixer dans les paramètres.

De plus chaque thread n'est pas associé à un nœud en particulier et peut calculer des temps de retours sur chaque nœud, ce qui permet un gain de temps conséquent.

2) Approche multi-threading retenue

Nous avons choisi d'implémenter notre plugin en C++ pour de meilleures performances mais aussi pour avoir accès à différentes solutions pour le multi-threading.

Nous avons d'abord essayé de paralléliser avec des « std ::async » qui sont des threads asynchrones ce qui correspondait à nos besoin.

Sauf que lorsqu'on lance notre algorithme, on se rend compte qu'il n'y a en fait qu'un seul thread qui travaille et que les marches aléatoires sont exécutées séquentiellement.

Après quelques recherches, nous avons découvert d'après plusieurs sources que Tulip ne supporterait pas la gestion de ce genre de thread. Nous ne sommes pas certain de ceci mais comme notre parallélisme avec « std ::async » fonctionnait sur un code C++ externe à Tulip, nous avons décidé de changer d'approche.

Nous avons donc essayé de paralléliser avec openMP. Cette fois-ci, nos marches aléatoires s'exécutent bien en parallèle.

Cette parallélisation s'effectue à l'aide de la commande suivante que l'on ajoute dessus de notre boucle for chargée de lancer les marches aléatoires :

#pragma omp parallel for

Il faut également fixer le nombre de threads désirés au lancement du plugin :

omp_set_num_threads(numberOfThreads);

Ceci permet donc une parallélisation simple et efficace de nos marches aléatoires.

3) Choix du critère d'arrêt

Dans l'article de Kermarrec, différentes options sont proposées pour choisir le critère d'arrêt, mais aucune solution n'a été retenue. Ce problème reste ouvert et les auteurs n'ont pas trouvé de solution vraiment meilleure que les autres. Une solution proposée est de considérer qu'à partir d'un certain nombre « k » de temps de retours enregistrés sur chaque nœud du graphe, l'algorithme a convergé. Nous avons choisi de nous inspirer de partir de cette proposition tout en l'améliorant :

Notre algorithme s'arrête lorsque 90% des nœuds ont enregistrés k temps de retour et lorsque 90% ont convergés.

Le nombre k de temps de retours désiré est fixé par l'utilisateur du plugin.

Pour le calcul de convergence, une fois qu'un nœud a au moins enregistré 4 temps de retour et qu'il en enregistre un nouveau, il calcule la différence entre sa nouvelle centralité et l'ancienne. Si la différence est inférieure à un certain pourcentage de la centralité, on considère que le nœud a convergé.

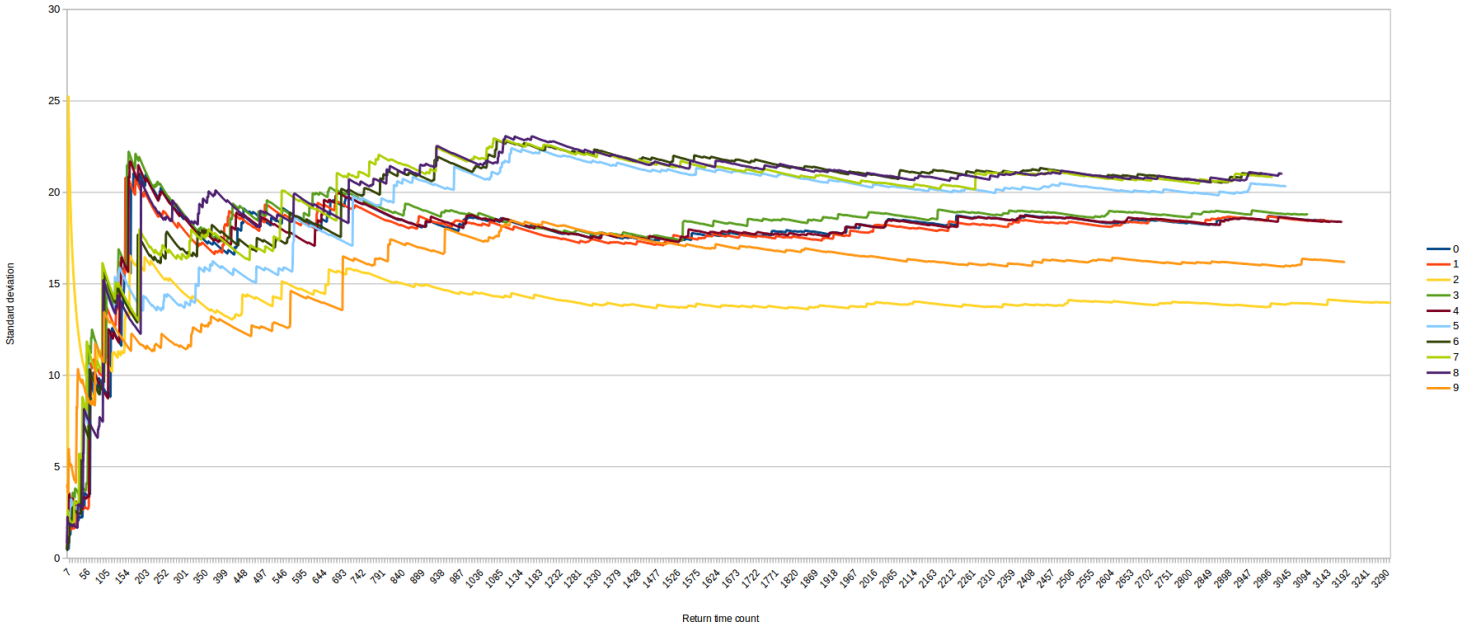
Ce pourcentage est également fixé par l'utilisateur du plugin. Bien que ce ne soit pas forcément utile : il est envisageable de mettre une valeur fixe dans une version future. La valeur par défaut est 1%.

Nous avons écrit un algorithme qui indique la déviation standard de chaque nœud (ordonnée) en fonction du nombre de temps de retour (abscisse).

Nous l'avons fait tourné sur le graphe de Barbell, petit graphe de 10 nœuds et 21 arêtes qui était présenté dans l'article. Nous l'avons fait tourné plus que nécessaire, en continuant même lorsque la convergence était atteinte.

Ainsi, chaque courbe correspond à un des 10 nœuds du graphe.

Nous pouvons observer qu'il y a toujours des variations de déviation standard et que ces valeurs ne vont jamais parfaitement converger.



4) Le plugin

Notre plugin est composé de deux classes principales : KermarreckAlgorithm et NodeWalkInfo. La première est la classe principale de l'algorithme, celle utilisée directement par Tulip, tandis que la seconde est une classe permettant de calculer la centralité de chaque nœud du graphe.

La classe KermarreckAlgorithm hérite de la classe tlp::DoubleAlgorithm, lui permettant ainsi de calculer et de modifier un attribut de type double pour chaque entité du graphe. Dans le cadre de notre plugin, la valeur de cet attribut correspond à la centralité secondaire des nœuds au sein du graphe. À l'exécution du plugin, la méthode check de cette classe est appelée. Celle-ci empêche au plugin de s'exécuter si le graphe n'est pas connexe. Lorsque le plugin s'exécute (au sein de la méthode run de cette classe), une instance de NodeWalkInfo est associée à chaque nœud, et un nombre *t* de marches aléatoires est lancée. Ce nombre *t* correspond au nombre de threads avec lequel exécuter l'algorithme (ce nombre étant réglable dans les paramètres du plugin). Ces marches aléatoires utilisent la classe NodeWalkInfo afin de mettre à jour la centralité secondaire d'un nœud lorsque celui-ci est atteint par une de ces marches. Celles-ci ne s'arrêtent alors que lorsqu'assez de nœuds ont convergés, ou lorsque chaque marche aléatoire a atteint un nombre d'étape assez important.

Une instance de la classe NodeWalkInfo, une fois associée à un nœud, permet de calculer la centralité secondaire de ce nœud lorsqu'il est atteint par une marche aléatoire. Cette classe permet également de déterminer si la centralité secondaire de ce nœud converge au cours des marches, permettant à celles-ci de s'arrêter lorsque nécessaire.

III) Tests

Nous avons réalisé nos tests sur différents graphes :

- Un graphe aléatoire connexe de 500 nœuds et 1000 arêtes
- Un graphe aléatoire connexe de 5000 nœuds et 10.000 arêtes
- Un graphe aléatoire connexe de 50.000 nœuds et 100.000 arêtes
- Le graphe sur les aéroports utilisé pour le TP2 qui comporte 1535 nœuds et 16.525 arêtes
- Le graphe de Barbell présenté dans l'article que comporte 10 nœuds et 21 arêtes

Pour chacun de ces graphes, nous avons effectué une série de tests sur un ordinateur portable ayant pour processeur un i7-6700HQ 2,6 GHz.

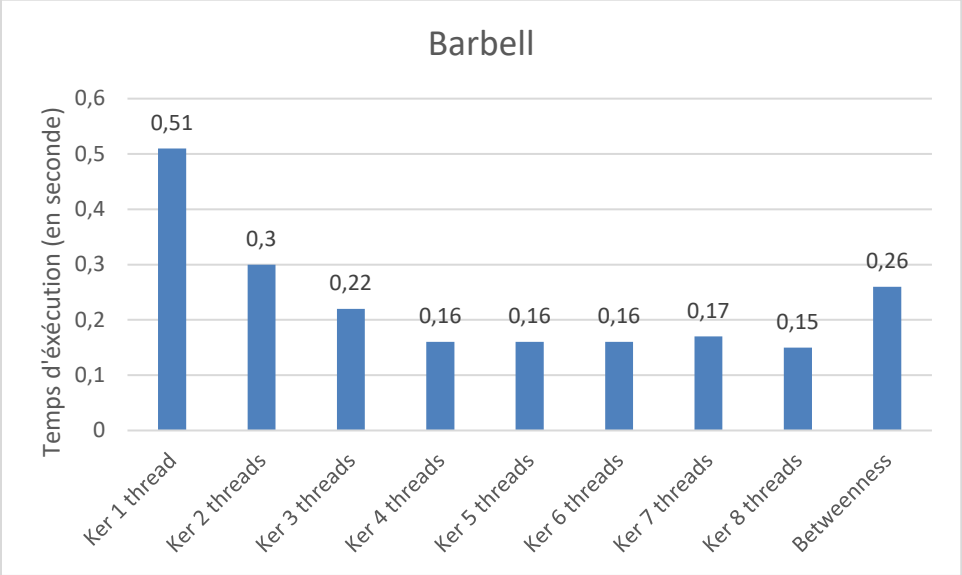
Nous avons :

- Calculé le temps d'exécution pour Betweenness ainsi que Kermarrec avec un nombre de threads allant de 1 à 8.
- Appliqué FM^3 pour mieux visualiser les nœuds centraux sur les graphes résultats de Betweenness et aussi de Kermarrec.
- Coloré les nœuds du noir vers le vert, plus le nœud étant vert clair plus celui-ci étant central (les échelles utilisées pour appliquer une couleur en fonction de la centralité ne sont pas linéaires et sont différentes pour les 2 algorithmes).

1) Graphe de Barbell

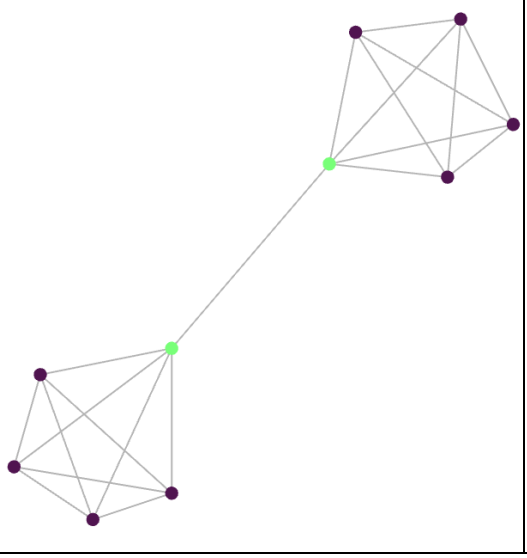
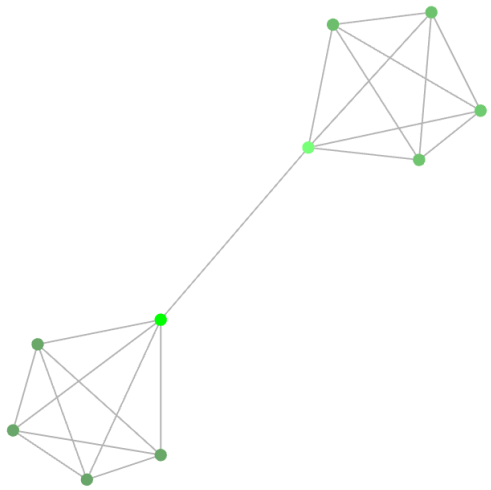
Ce graphe comporte 10 nœuds et 21 arêtes. C'est le graphe qui est utilisé dans l'article comme exemple.

Voici les résultats obtenus :



On peut observer une nette amélioration avec l'augmentation du nombre de threads de 1 à 4. Bien que la machine de test possède 4 cœurs et 8 threads, seulement 4 threads sont accessibles pour paralléliser, d'où l'absence de différence entre 4 et 8 threads.

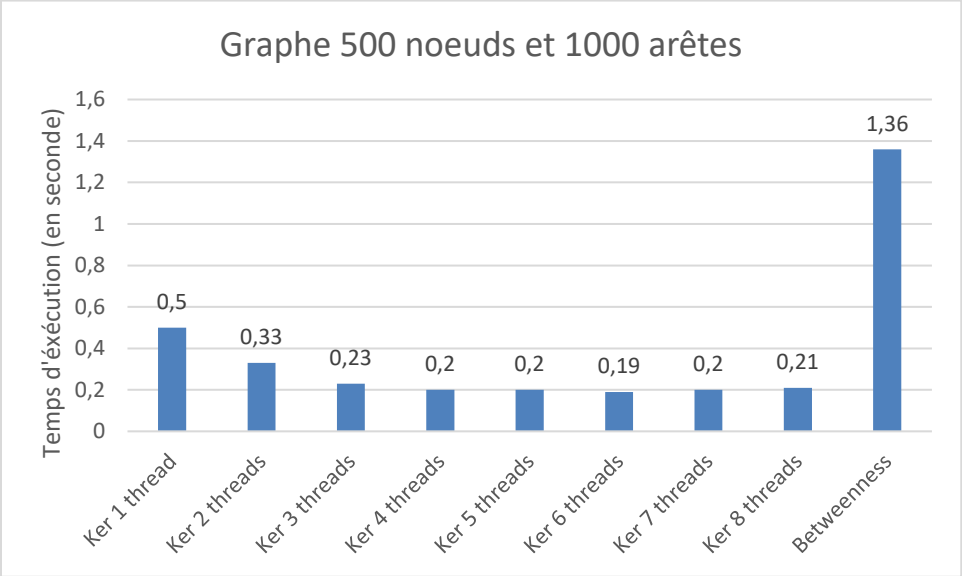
Ci-dessous les colorations :

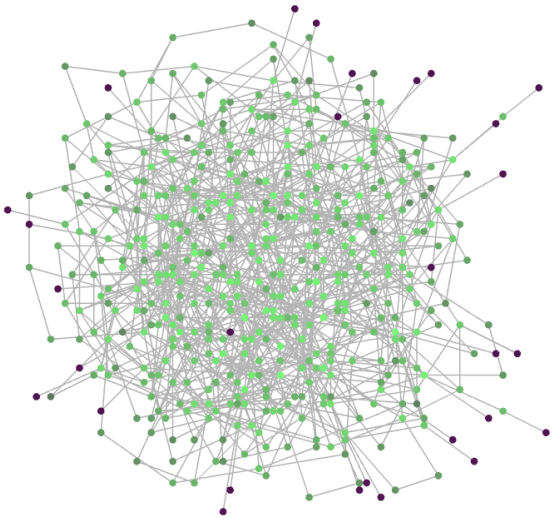
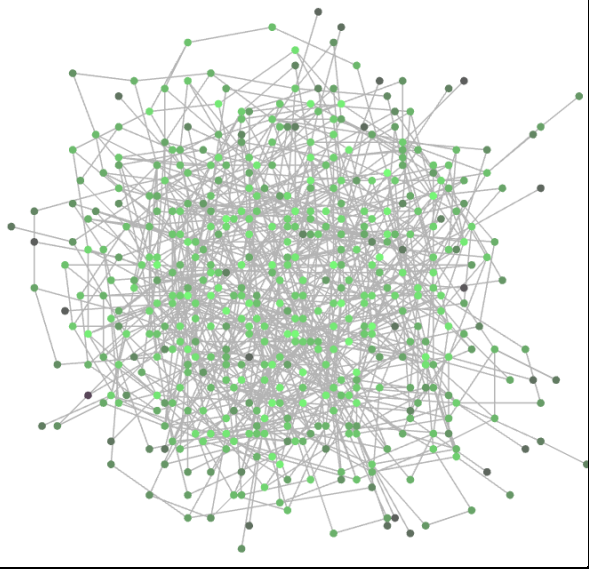
Betweenness	Kermarrec
	

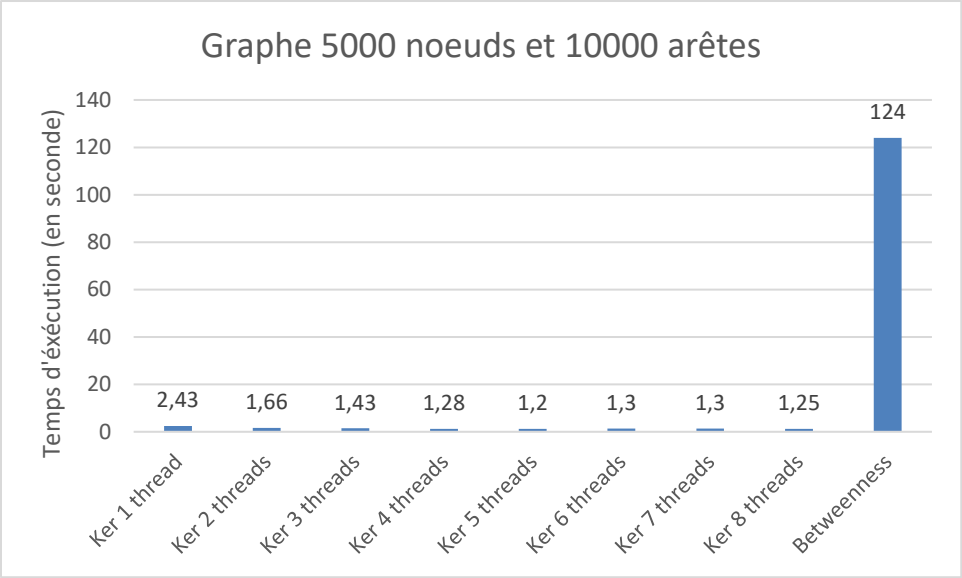
2) Graphes aléatoires

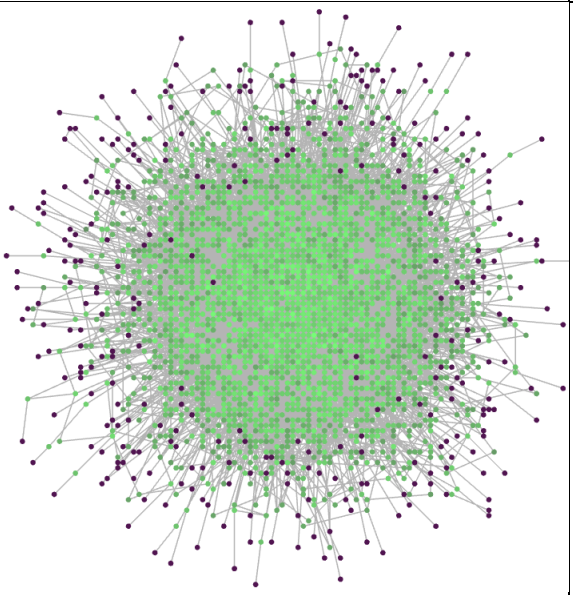
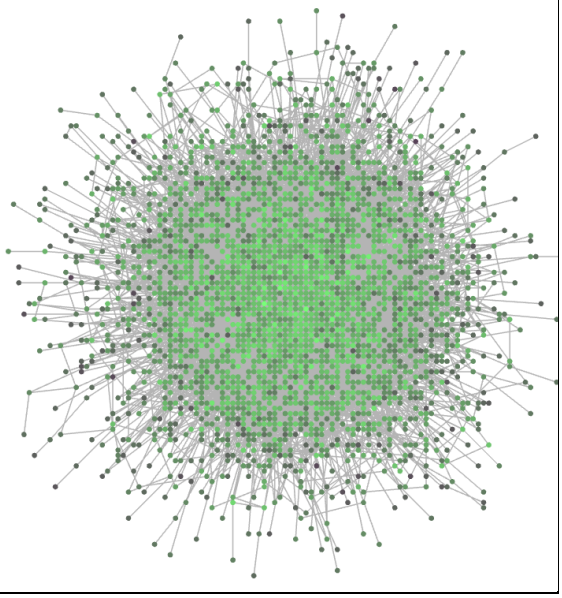
Nous avons ensuite testé notre algorithme sur des graphes aléatoires générés par Tulip.

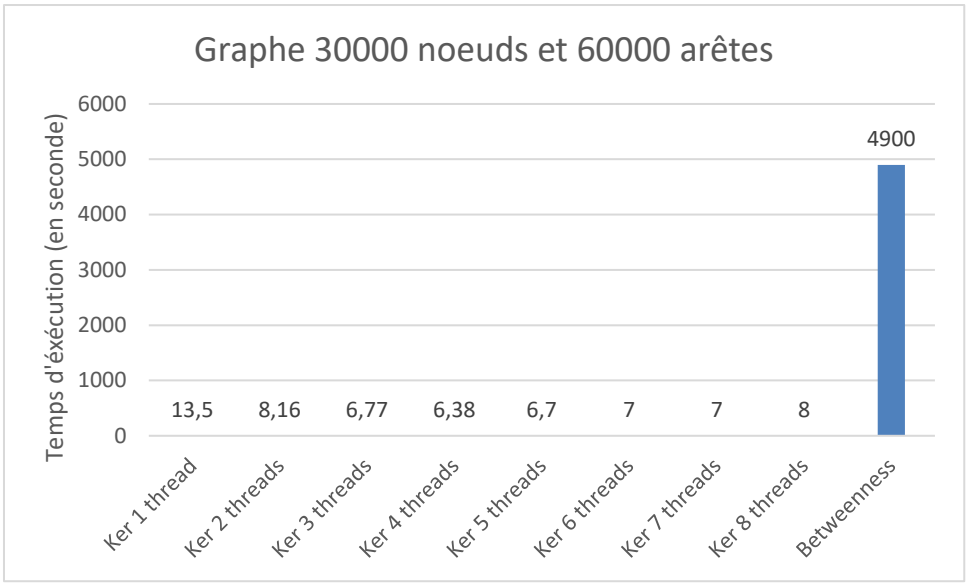
On peut observer sur les 3 graphiques qui suivent, que Kermarrec et bien plus rapide que Betweenness sur ces graphes aléatoires et que les résultats de centralité obtenus sont similaires.

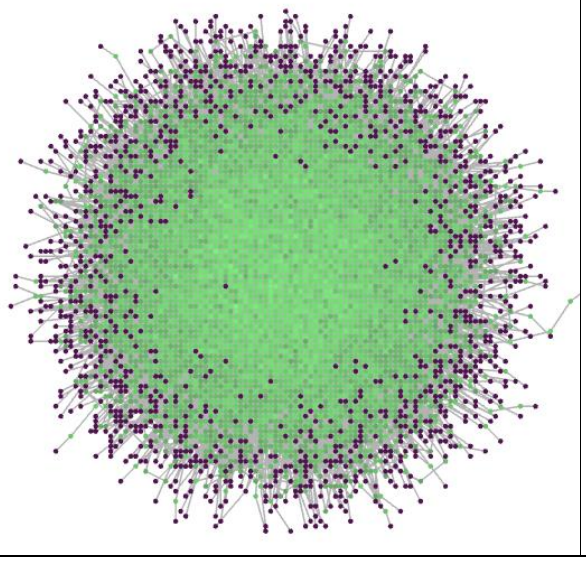
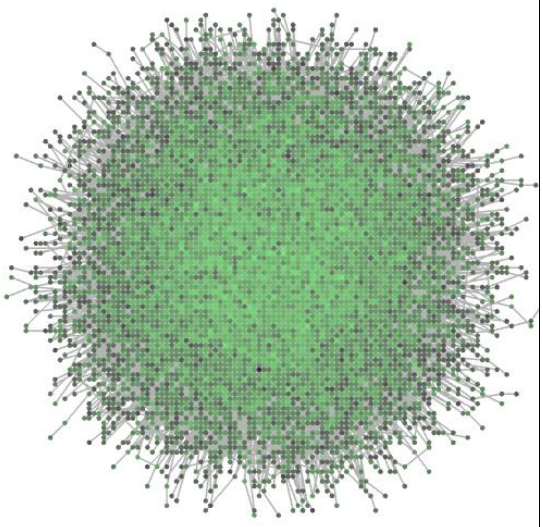


Betweenness	Kermarrec
	



Betweenness	Kermarrec
	

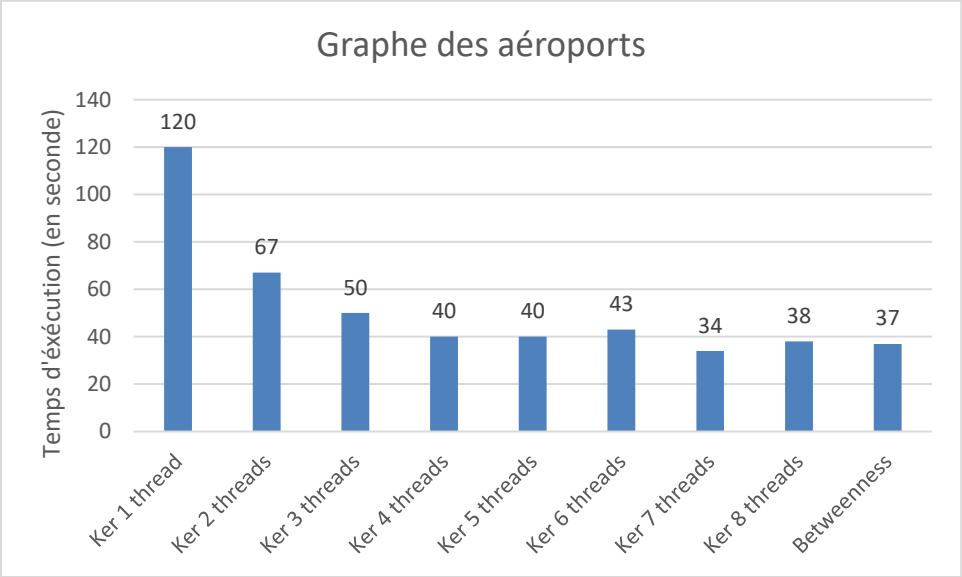


Betweenness	Kermarrec
	

3) Aéroports

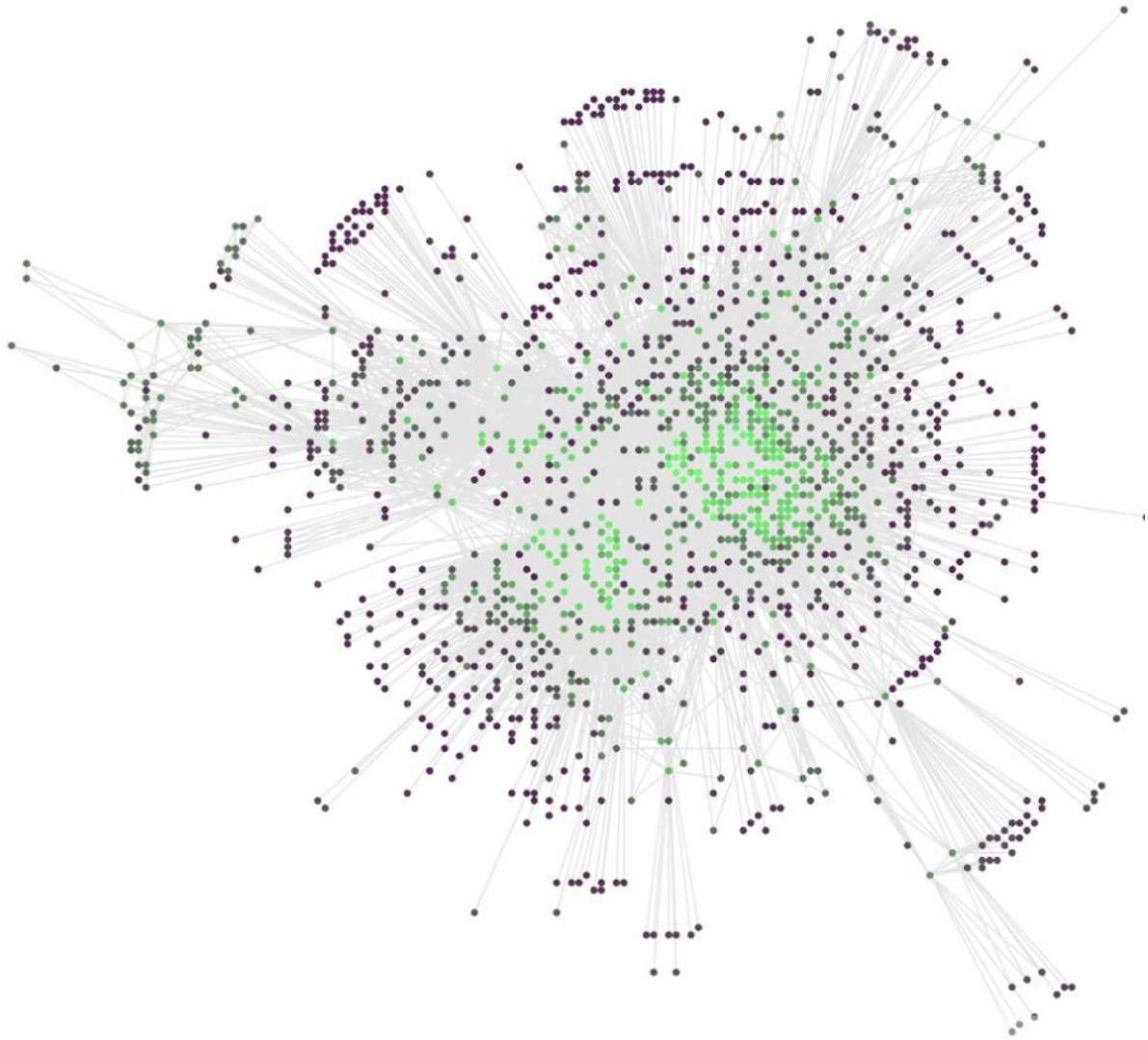
Ce graphe comporte 1535 nœuds et 16.525 arêtes mais est beaucoup plus complexe que les graphes aléatoires générés automatiquement par Tulip. En effet, nous obtenons des temps d'exécution plus long à la fois pour Kermarrec mais aussi pour betweenness centrality.

Et cette fois-ci, Kermarrec avec 4 threads n'est pas plus efficace que Betweenness.



Voici le résultat pour Betweenness :

Puis pour Kermarrec :



On observe répartition légèrement différente de la centralité. Les nœuds qui ont été placés le plus au centre par FM^3 paraissent favorisés par Kermarrec et moins par Betwenness.

IV) Conclusion

1) Résultats

Les graphes avec les nœuds colorés en vert nous montrent que nos résultats sont corrects, puisque nous pouvons voir visuellement que ce sont les nœuds les plus centraux qui sont le plus vert. De plus, la coloration est similaire aux graphes obtenus avec Betweenness.

En ce qui concerne la performance, nous obtenons pour les graphes aléatoires des résultats très largement meilleurs : pour le graphe de 30.000 nœuds nous passons de 4900 secondes à 8 secondes ! Plus le graphe est grand, plus le gain de temps est conséquent.

Néanmoins, pour le graphe de données réels, nous obtenons des résultats similaires.

Nous pouvons donc dire que notre algorithme est au moins aussi rapide que Betweenness Centrality, voire plus rapide ou beaucoup plus rapide.

2) Difficultés et travail au cours du projet

La première grosse difficulté de ce projet a été de mettre en place notre environnement de travail. En effet, nous avons passé beaucoup de temps pour parvenir à importer notre plugin c++ dans Tulip. Pour cela, nous avons dû compiler Tulip en y intégrant notre plugin.

Programmer une marche aléatoire n'a pas posé de problème.

La difficulté suivante a été de trouver un moyen de paralléliser nos marches aléatoires. Certaines méthodes fonctionnant dans un programme c++ classique mais ne fonctionnant pas dans Tulip, ajouté au fait que le debogage est impossible, il n'a pas été aisé de trouver une méthode fonctionnant sur Tulip. Nous avons donc dû chercher une autre solution efficace et apprendre à l'utiliser, openMP en l'occurrence.

Une fois notre marche aléatoire parallélisée, nous avons effectué des tests pour trouver un bon critère d'arrêt. Nous avons choisi de rendre cela paramétrable par l'utilisateur. Cela a pris du temps puisque nous avons découvert des bugs qui ont fait que notre algorithme ne convergeait pas.

Enfin, nous avons cherché des moyens de présenter l'efficacité de notre algorithme, via les résultats, graphiques et graphes présentés dans ce rapport.