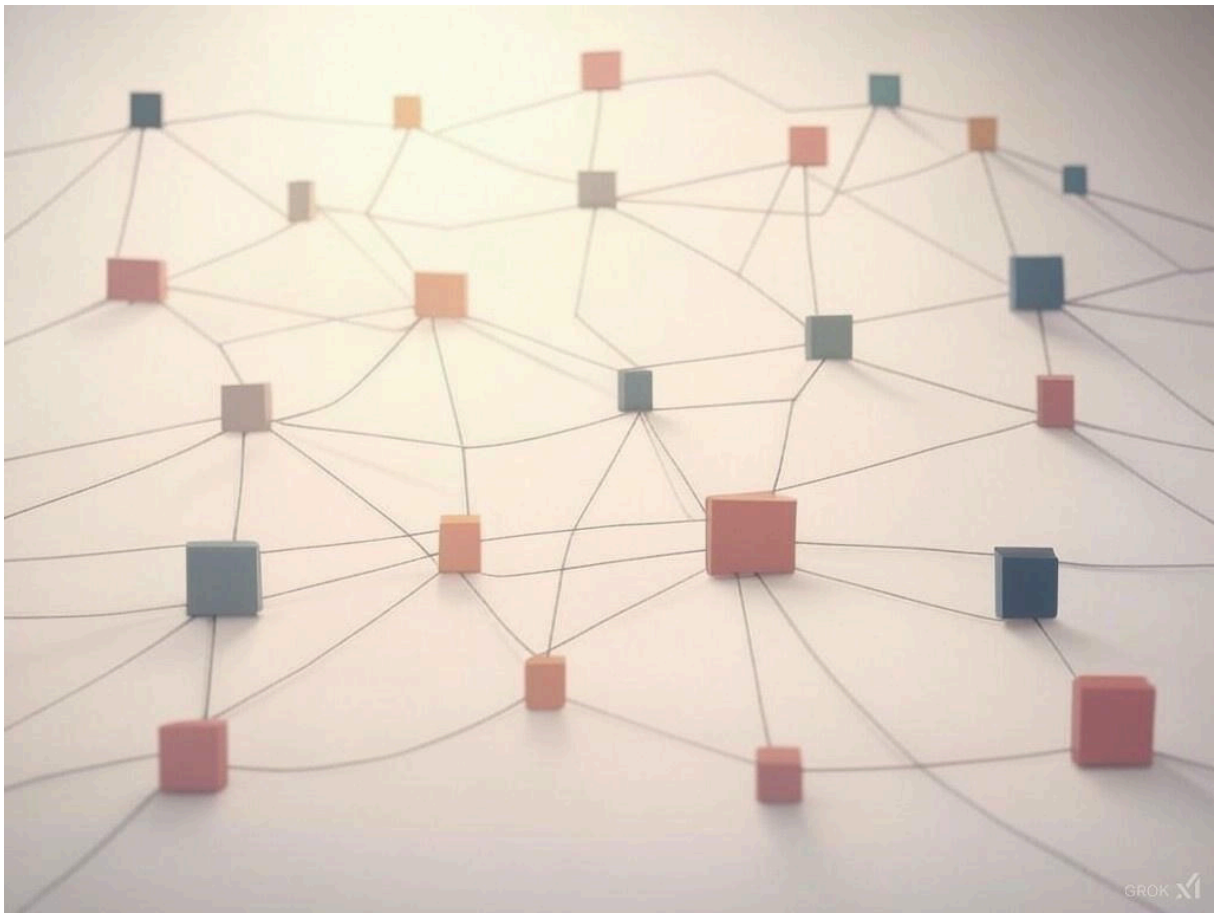

06/01/2025

Projet MA - PCM

Traveling Salesman Problem (TSP)

Albanesi Nicolas
HES-SO

Kandiah Abivarman
HES-SO

Stirnemann Jonas
HES-SO

Contents

1. Introduction

Le problème du voyageur de commerce **(TSP)** est un défi combinatoire où la complexité augmente de manière **factorielle** avec le nombre de villes, rendant sa résolution coûteuse en ressources. Pour **accélérer** le calcul, nous exploitons une **architecture multicœur** avec 256 cœurs afin de paralléliser l'exécution du programme.

Le projet vise à démontrer une **accélération** significative grâce à la **parallélisation**, en mettant l'accent sur l'efficacité du partage plutôt que l'exécution sur sur un seul coeur. Il nous faut en revanche un partage le plus **efficace** possible, des structures de données ***Lock free*** sont alors particulièrement adaptées.

Nous avons effectué plusieurs implémentations afin de les comparer. L'une avec une **FIFO** et une avec un **STACK**. L'implémentation commence par distribuer une tache par ville (choix arbitraire), ces taches vont alors être subdivisées et remises dans la **FIFO** ou le **STACK** jusqu'a une limite choisie, une fois la limite atteinte, un calcul séquentiel est effectué par chacun des coeurs. Cette limite a un impact important sur les performances, son choix est alors essentiel.

Le fonctionnement d'une **FIFO** pousse à explorer *l'arbre* sur un plan **horizontal**, c'est à dire que l'on traite tous les noeuds d'une certaine profondeur avant de passer à la suite. Ceci n'a pas l'air optimal car il pourrait être mieux d'explorer certaines branches en profondeur pour trouver plus vite des solutions intéressantes, ce qui permettrait de réduire le nombre de tâches à traiter et d'améliorer les performances globales.

Ce problème serait résolu par un **Stack** puisque le système est *Last in First Out* permet de traiter en priorité les tâches les plus récemment ajoutées. Cette approche favorise une distribution **verticale** des sous-problèmes, ce qui facilite l'exploration rapide des solutions prometteuses et amène à des *bounds* plus réguliers, réduisant alors le nombre de tâches total à effectuer. Cependant un **Stack** introduit un ***bottleneck*** car toutes les taches font des *Push* et *Pop* depuis le **même endroit** (Ce qui n'est pas le cas pour une **FIFO** où les *Push* et *Pop* se font à des **endroits différents**).

Nous avons réussi à accélérer le temps de calcul avec un partage des tâches sur plusieurs coeurs; par rapport à une implémentation purement séquentielle. Malgré son défaut, les performances de la **FIFO** sont supérieures à celles du **STACK**.

2. Implémentations

Après quelques tests, nous avons décidé de tester 4 principales implémentations pour les comparer.

Deux d'entre elles utilisent une **FIFO** mais ont une différence d'implémentation pour le stockage du chemin le plus court. Les deux autres utilisent un **Stack** et one également une différence pour le stockage des chemin les plus courts.

2.1. Stockage des chemins plus courts

Nous **stockons** les chemins les plus **courts** pour permettre de couper court à la recherche dans un chemin particulier. En effet, si l'on est en train de creuser un chemin qui est déjà **plus long** que le chemin qui a été **sauvegardé**, alors il n'est **plus utile** de continuer ce chemin.

2.1.1. Global shortest

Une des implémentation consiste à garder dans une variable globale le chemin le plus court, et chacun des threads va alors le comparer à son chemin courant. Cela permet de couper chacun des chemin avec l'actuel plus court trouvé sur l'ensemble des threads.

2.1.2. Local shortest

Cette deuxième implémentation garde dans chacun des threads un chemin le plus court, cela permet d'éviter de partager une variable globale à tous les threads et réduit dont un *bottleneck*, mais cela veut aussi dire que si un thread se retrouve avec que des chemins longs, il ne va pas couper court alors qu'un autre thread a trouvé un chemin bien plus court.

2.1.3. Hypothese du meilleur shortest

L'efficacité de ces deux méthodes est difficile à théoriser puisqu'elle repose sur la distribution des chemins aux différents threads. Or, nous n'avons pas un parfait contrôle de cette distribution, on dépend du scheduling, et de l'utilisation du processeur par l'OS.

On peut alors se retrouver avec des threads qui ont tous les chemins les plus longs tandis que les autres ont tous les chemins les plus courts. Ce qui veut dire qu'avec des *shortests* locaux, nous allons creuser des chemins inutiles.

On a alors décidé de tester ces différentes implémentations de manière empirique.

2.2. Mesures

Nous avons fait de multiple **mesures** avec différentes **limites** et différent nombre de **villes** et différent nombre de **threads**, afin de comparer et identifier les **tendances**. Pour des raisons de temps d'exécution et lisibilité, nous avons décidé de n'exposer que les mesures avec **16 villes** et pour des **limites** allant de **9 à 13**, le tout avec un nombre de **threads** allant de **6 à 256** par pas de 5 à 6 (c'est un *linespace*)

2.2.1. FIFO

Notre première implémentation utilise une ***Lock Free FIFO*** relativement classique, utilisant donc des CAS et ayant besoin de timestamping pour éviter les problèmes **ABA**. Comme décrit dans l'introduction, l'utilisation d'une **FIFO** n'est pas optimale pour notre problême puisque l'on ne creuse pas verticalement mais plutôt horizontalement, mais c'est tout de même la stucture qui nous donne les meilleurs performances.

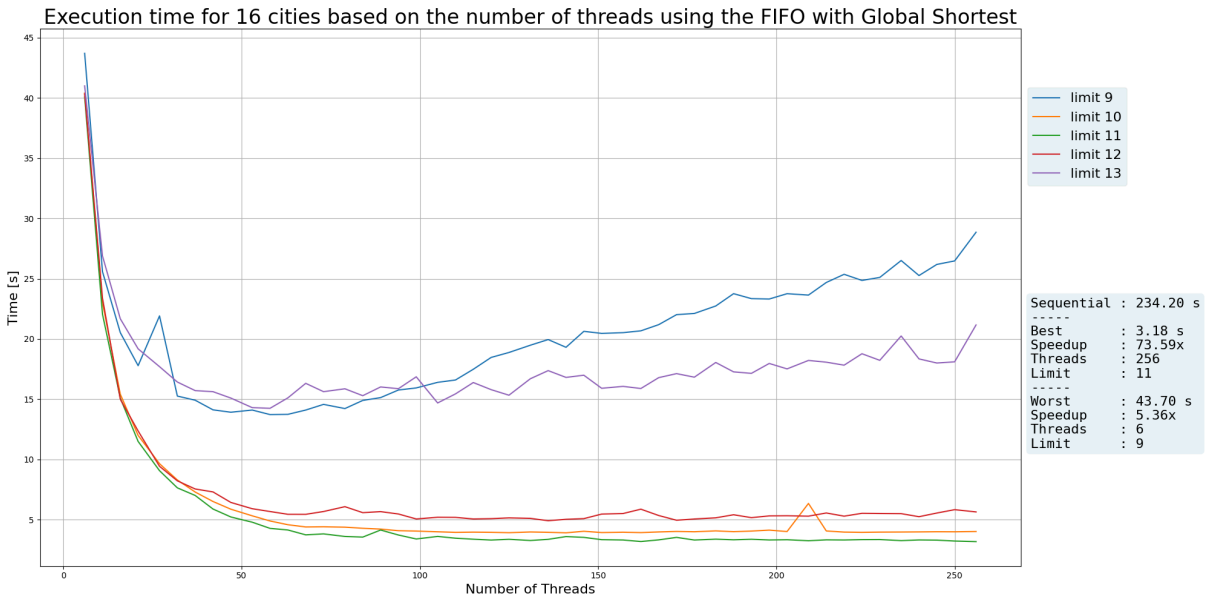

Figure 1: Graphique du temps d'execution en fonction du nombre de threads (FIFO Global Shortest, 16 villes)

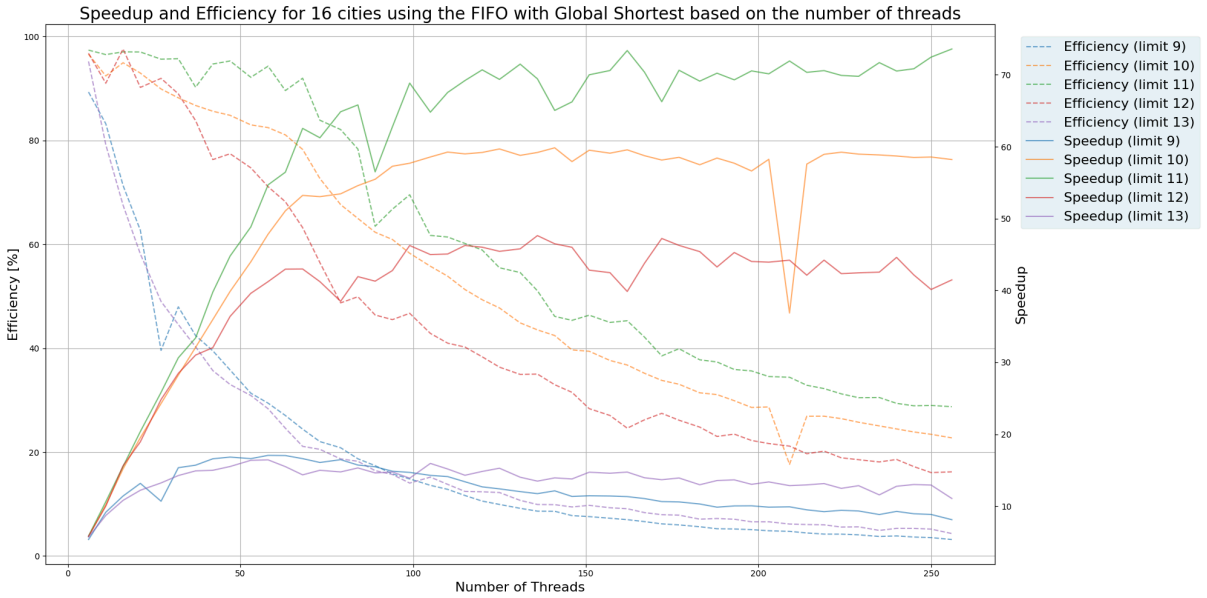

Figure 2: Graphique du speedup et de l'efficience en fonction du nombre de threads (FIFO Global Shortest, 16 villes)

On observe dans les Figure 1 et 2 une forte diminution du temps d'éxécution entre 6 et 75 coeurs puis les performances vont stagner, ce qui veut dire que nous améliorons les performances en augmentant le nombre de coeurs jusqu'à 75 coeurs, puis il n'est plus très pertinent d'en utiliser plus. On observe également que la limite optimale est de 11 puisqu'on obtient un Speedup de 73.59x. On peut présumer que le nombre optimal de thread se trouve au croisement entre Speedup et Efficiency, donc encore une fois aux alentours de 75 coeurs.

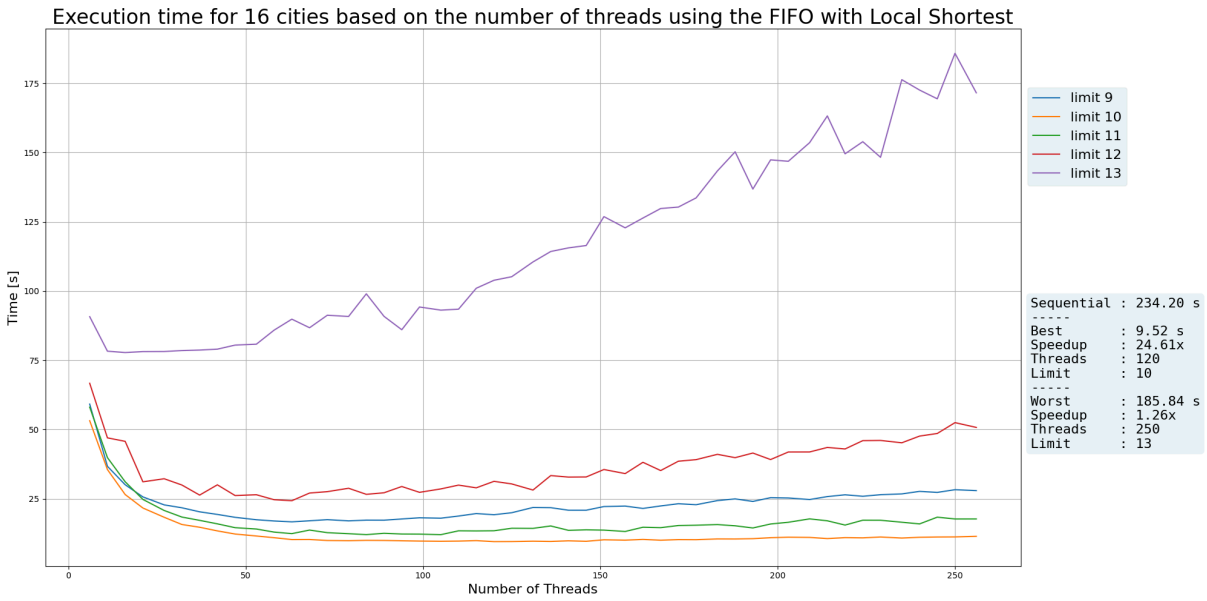

Figure 3: Graphique du temps d'execution en fonction du nombre de threads (FIFO Local Shortest, 16 villes)

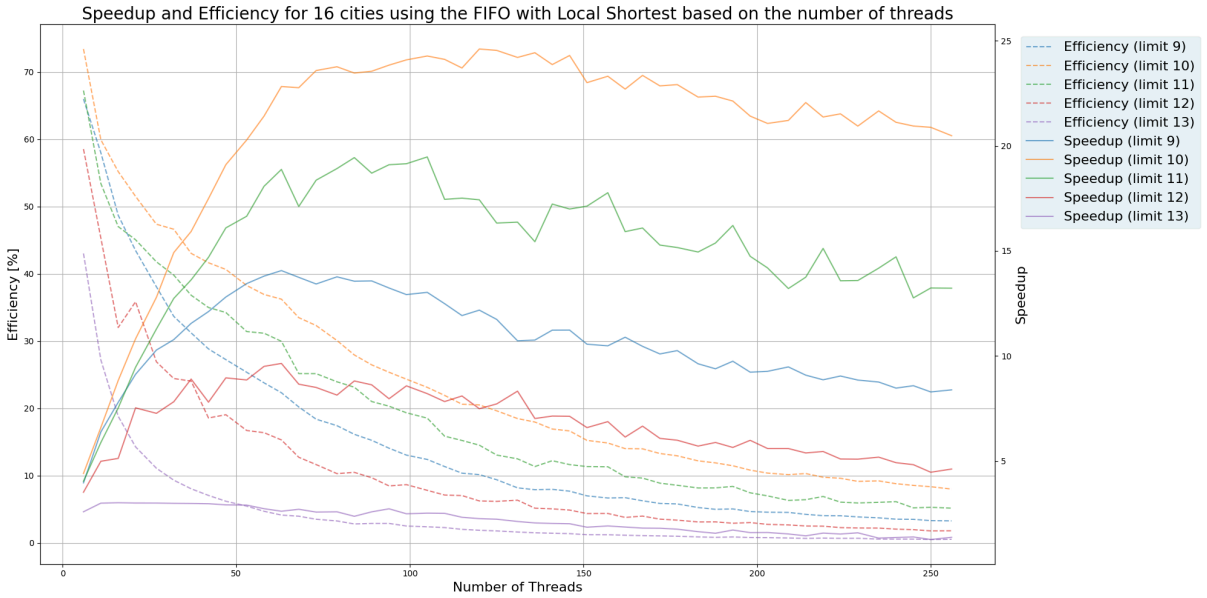

Figure 4: Graphique du speedup et de l'efficience en fonction du nombre de threads (FIFO Local Shortest, 16 villes)

On observe également dans les Figure 3 et 4 une forte diminution du temps d'éxécution entre 6 et 55 coeurs puis les performances vont stagner, ce qui veut dire que nous améliorons les performances en augmentant le nombre de coeurs jusqu'à 55 coeurs, puis il n'est plus très pertinent d'en utiliser plus. On observe également que la limite optimale est de 10 puisqu'on y obtient un Speedup de 24.61x. Le nombre optimal de threads est en revanche aux alentours des 35.

On voit que les performances générales sont moindre comparées à la globale, (73.59x vs 24.61x).

2.2.2. STACK

Le **STACK résout** le problème d'**horizontalité** discuté dans l'introduction mais **introduit** un ***bottleneck*** à cause de son implémentation. Toutes les opérations d'ajout et de récupération sont faites sur un même pointeur (La *head* et la *tail* sont identiques). Encore une fois tout est fait *Lock free* avec des CAS.

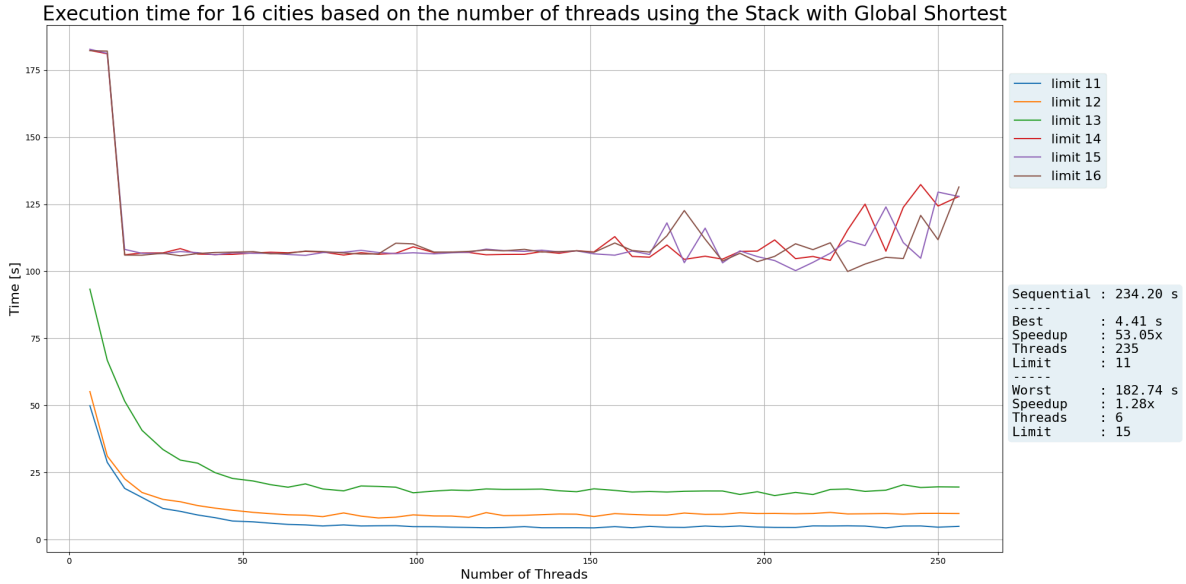

Figure 5: Graphique du temps d'execution en fonction du nombre de threads (Stack Global Shortest, 16 villes)

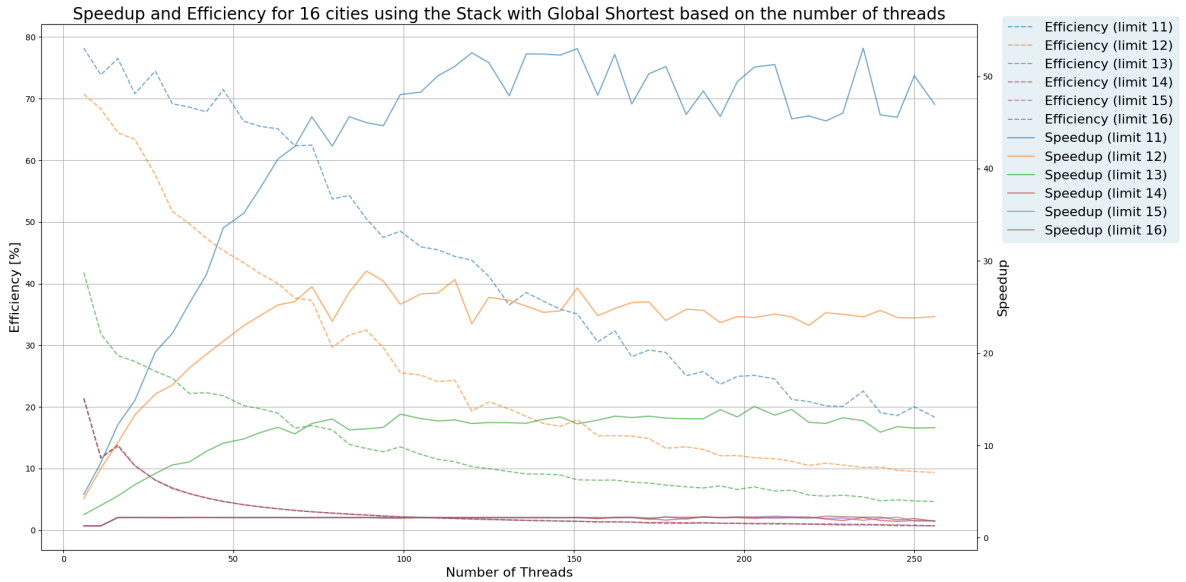

Figure 6: Graphique du speedup et de l'efficience en fonction du nombre de threads (Stack Global Shortest, 16 villes)

Les Figures 5 et 6 montrent une forte diminution du temps d'exécution entre 6 et 70 cœurs, indiquant une amélioration des performances à mesure que le nombre de cœurs augmente jusqu'à 70. Au-delà de cette valeur, les gains deviennent négligeables. Par ailleurs, la limite optimale semble se situer à 11, avec un Speedup maximal de 53,05x. Le nombre optimal de threads, quant à lui, se situe autour de 65 cœurs.

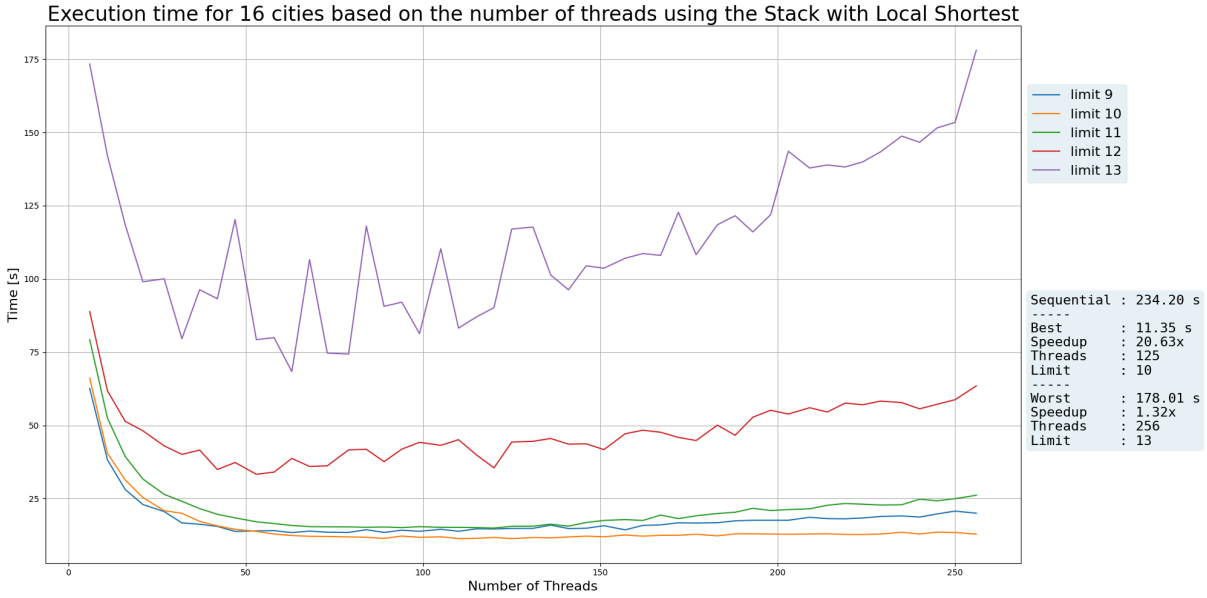

Figure 7: Graphique du temps d'execution en fonction du nombre de threads (Stack Local Shortest, 16 villes)

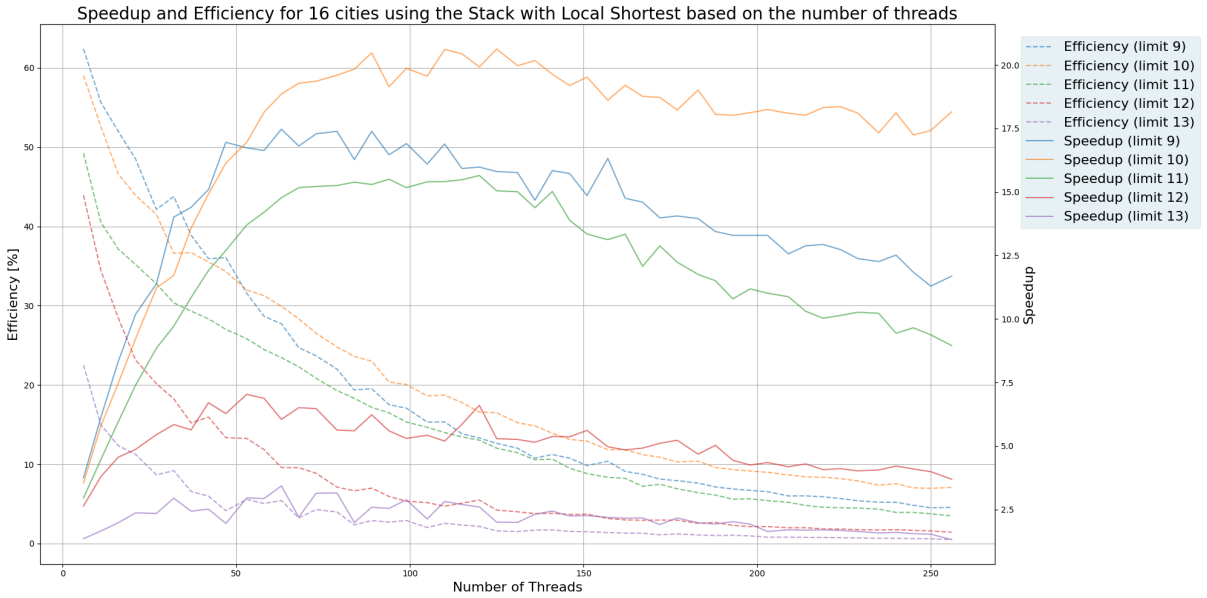

Figure 8: Graphique du speedup et de l'efficience en fonction du nombre de threads (Stack Local Shortest, 16 villes)

Les Figures 7 et 8 montrent une forte diminution du temps d'exécution entre 6 et 55 cœurs, indiquant une amélioration des performances à mesure que le nombre de cœurs augmente jusqu'à 55. Au-delà de cette valeur, les gains deviennent négligeables. Par ailleurs, la limite optimale semble se situer à 10, avec un Speedup maximal de 20.63x. Le nombre optimal de threads, quant à lui, se situe autour de 40 cœurs.

On voit que les performances générales sont moindre comparées à la globale, (53,05x vs 20.63x).

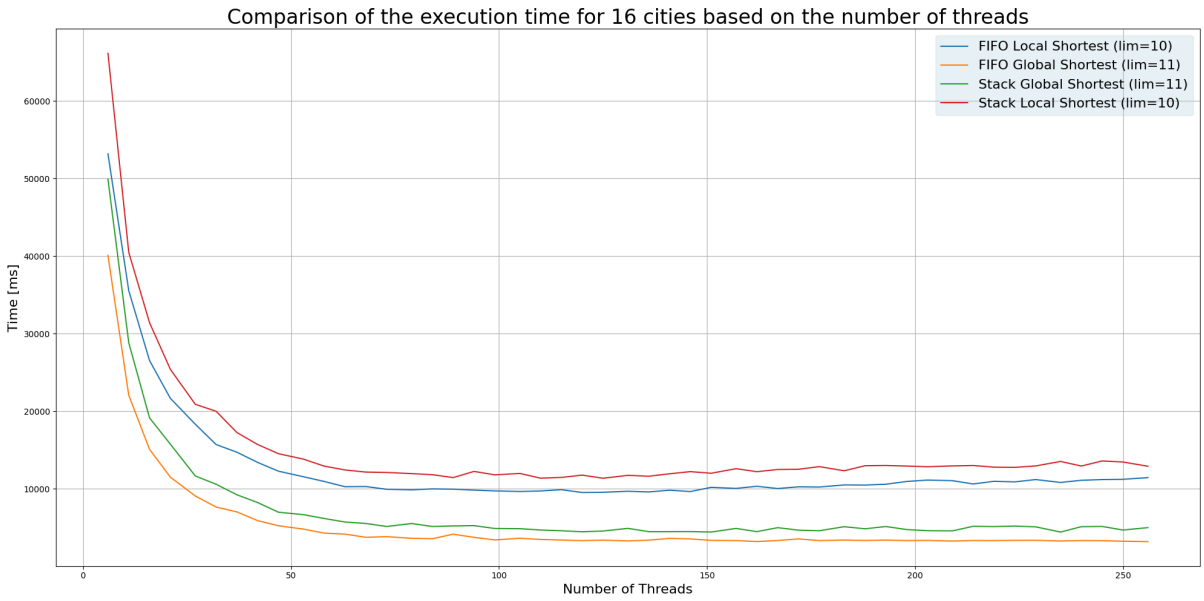

Figure 9: Graphique comparant le meilleur temps d'exécution de chaque algorithme (16 villes)

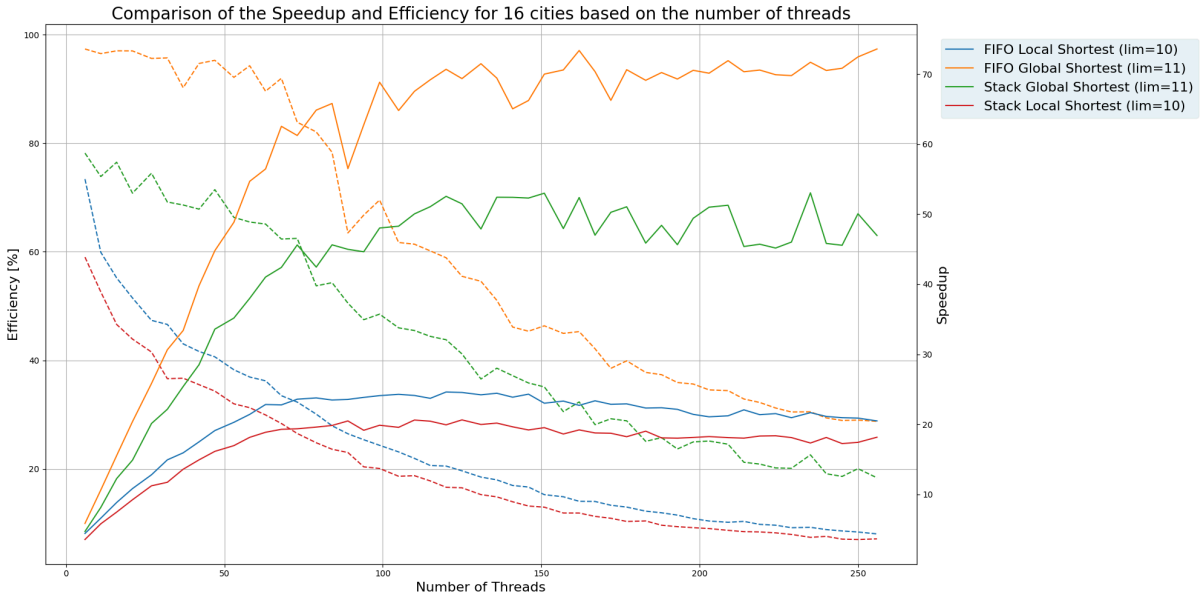

Figure 10: Graphique comparant le speedup et l'efficience des meilleurs temps de chaque algorithme (16 villes)

On peut voir sur la figure 9 que l'algorithme avec le temps d'execution le plus bas est la FIFO avec une variable globale pour le plus court chemin.

Toutefois, le STACK avec une variable globale reste plus rapide que la FIFO avec une variable locale.

2.3. Tableau récapitulatif

Voici un tableau récapitulatif des mesures sur **16 villes**

	Meilleur temps [s]	Meilleur Speedup	Meilleure limite	Seuil amélioration*	Nb threads optimal*
Séquentiel	234.20	X	X	X	X
FIFO Global	**3.18**	**73.59x**	11	75	75
FIFO Local	9.52	24.61x	10	55	35
STACK Global	4.41	53.05x	11	75	65
STACK Local	11.35	20.63x	10	55	40

Seuil amélioration : Seuil du nombre de threads, auquel il n'y plus vraiment d'améliorations du temps de calcul.

Nb threads optimal : Nombre de threads optimal en prenant le Speedup et L'efficiency en compte.

Note : Nous avons finalement effectué un test unique avec **20 villes** avec la **FIFO** et de manière séquentiel, obtenant alors **22 minutes 42** avec 256 coeurs et une limite de 14 et un peu plus de **36h** pour le séquentiel. Amenant alors une performance de presque **100 fois** plus rapide sur la version multicore.

3. Conclusion

Nous avons pu observer des **performances supérieurs** avec l'utilisation d'une **variable globale** pour le stockage du plus court chemin, avec la **FIFO** et le **STACK**.

Nos mesures nous ont aussi démontrées qu'au final la **FIFO** nous donne quand même des **meilleurs résultats** par rapport au **STACK**, malgré l'horizontalité de l'exploration de l'arbre.

Nous avons pu observer de significatifs **gains de performances** en **parralélisant**, allant au maximum **73.59 fois** les performances séquentielles. Ceci avec une **FIFO Global Shortest et une limite de 11 sur 16 villes**.

Une **amélioration** envisageable serait de mettre en place un système de **free pool** pour notre FIFO et notre STACK. Celui-ci nous permettrait d'éviter d'allouer et de désallouer dynamiquement les nœuds de manière répétée.

Il serait également possible d'implémenter de **multiples structures de partage** de tâches (plusieurs FIFO par exemple) pour **éviter** tout *Bottleneck* sur une **structure unique**.

Lien vers le code : https://github.com/Eleczoo/pcm_projet