UCLouvain

ÉCOLE POLYTECHNIQUE DE LOUVAIN

LINFO1252 - Systèmes Informatiques

Projet 1 : Programmation multi-threadée et évaluation de performances

Groupe 19

Décembre 2023

Abarkan Rayan (29712300)
rayan.abarkan@student.uclouvain.be
Niyikiza Cédric (64492300)
cedric.niyikiza.@student.uclouvain.be

Professeur : Riviere Etienne

1 Spécification de la machine utilisée pour l'évaluation de performance

Pour ce projet, les graphes qui ont été générés sont basés sur une exécution des programmes sur les machines de la salle Intel (*didac10* dans notre cas). Les spécifications de la machine utilisée :

— Processeur : Intel® Core™ i7-10700 CPU @ 2.90 GHz avec 4 cœurs

— RAM : 16 GB

2 Analyses des expérimentations

2.1 Philosophes

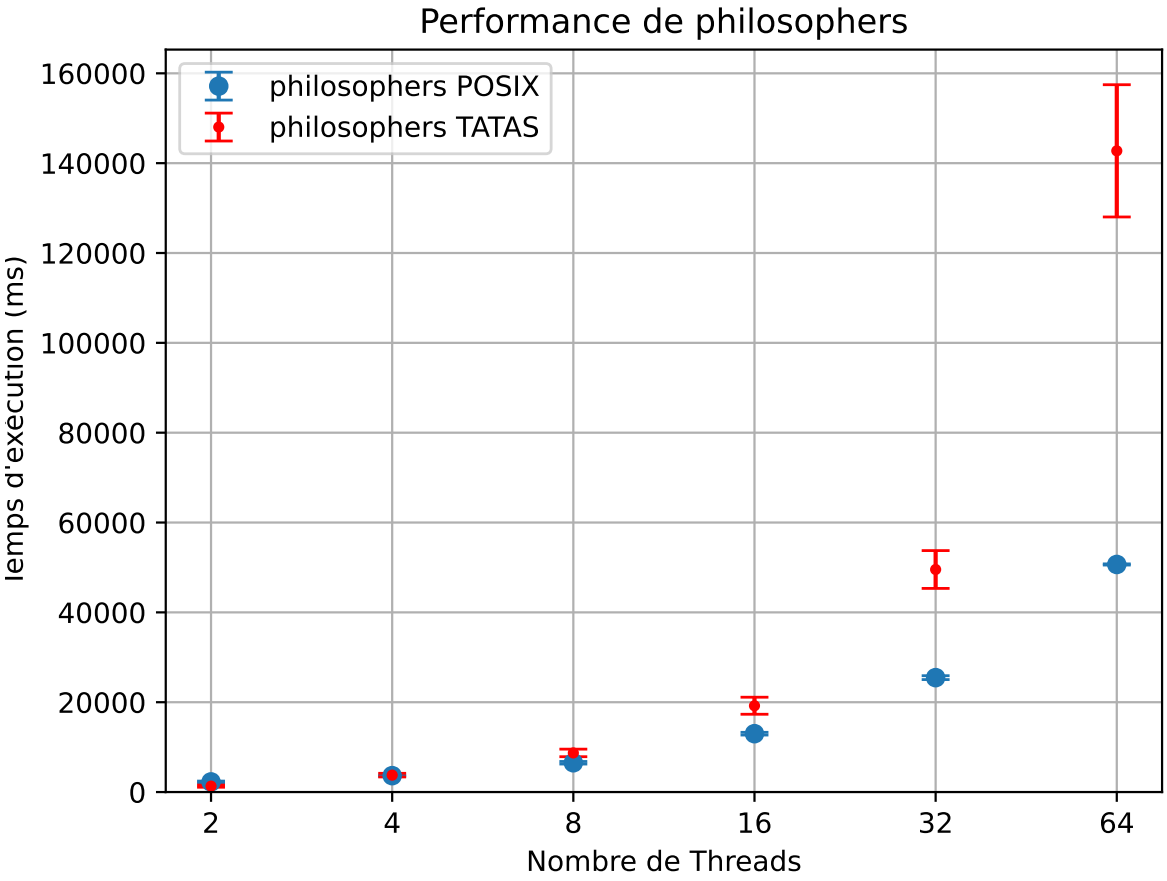

FIGURE 1 – Évolution du temps d'exécution en fonction du nombre de threads : Les points bleus représentent la version utilisant la librairie POSIX (mutex et/ou sémaphores), tandis que les points rouges représentent la version utilisant notre implémentation personnalisée basée sur des verrous avec attentes actives.

L'analyse comparative des mutex POSIX et de notre implémentation TATAS [1] pour le problème des philosophes a révélé des aspects intéressants. Les mutex POSIX, issus d'une bibliothèque standard, se montrent robustes et polyvalents, adaptés à divers contextes de multi-threading. D'un autre côté, notre approche TATAS excelle dans des situations à faible contention, avec son modèle d'attente active efficace pour des sections critiques brèves. Cependant, en cas de forte contention ou de sections critiques prolongées, les mutex

1. TATAS fait référence à l'implémentation de nos verrous avec l'algorithme Test-And-Test-And-Set.

POSIX semblent offrir de meilleures performances, soulignant ainsi l'importance du choix de la stratégie de synchronisation adaptée à chaque situation.

2.2 Producteurs-Consommateurs

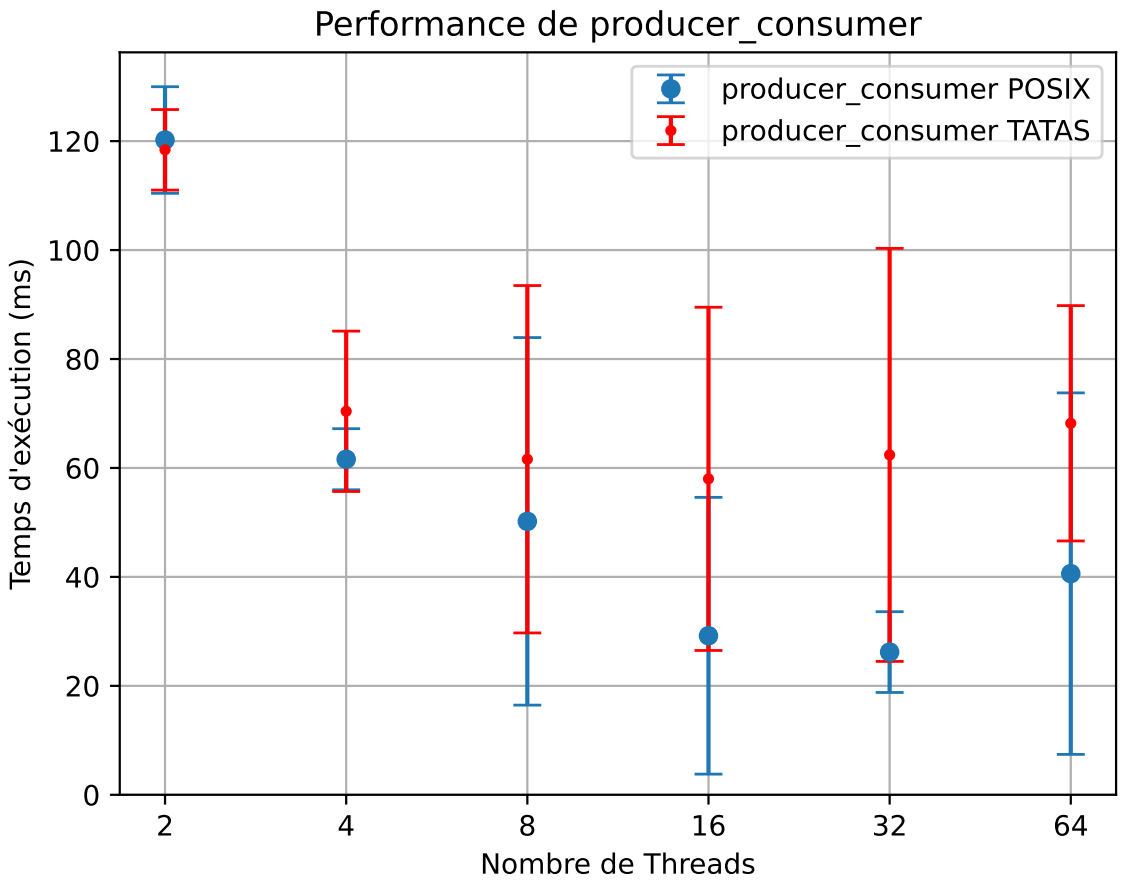

FIGURE 2 – Évolution du temps d'exécution en fonction du nombre de threads : Les points bleus représentent la version utilisant la librairie POSIX (mutex et/ou sémaphores), tandis que les points rouges représentent la version utilisant notre implémentation personnalisée basée sur des verrous avec attentes actives.

L'analyse des performances pour les programmes producteur-consommateur montre une augmentation de l'écart-type des temps d'exécution avec le nombre de threads. Cela indique que la fiabilité des performances diminue à mesure que le nombre de threads augmente, probablement en raison d'une contention plus élevée et d'une gestion plus complexe des ressources. Cette variabilité accrue avec l'augmentation des threads souligne l'importance de choisir le bon niveau de parallélisme pour équilibrer efficacement l'utilisation des ressources tout en maintenant une régularité dans les performances.

On peut également remarquer que peut importe la version utilisée, le temps d'exécution a tendance à décroitre et cela est propre au programme "*Producteurs-Consommateurs*". Plus il y a de threads, chaque thread aura moins de travail à fournir. C'est le seul graphe qui a cette tendance et on trouvait cela pertinent à relever.

2.3 Lecteurs et écrivains

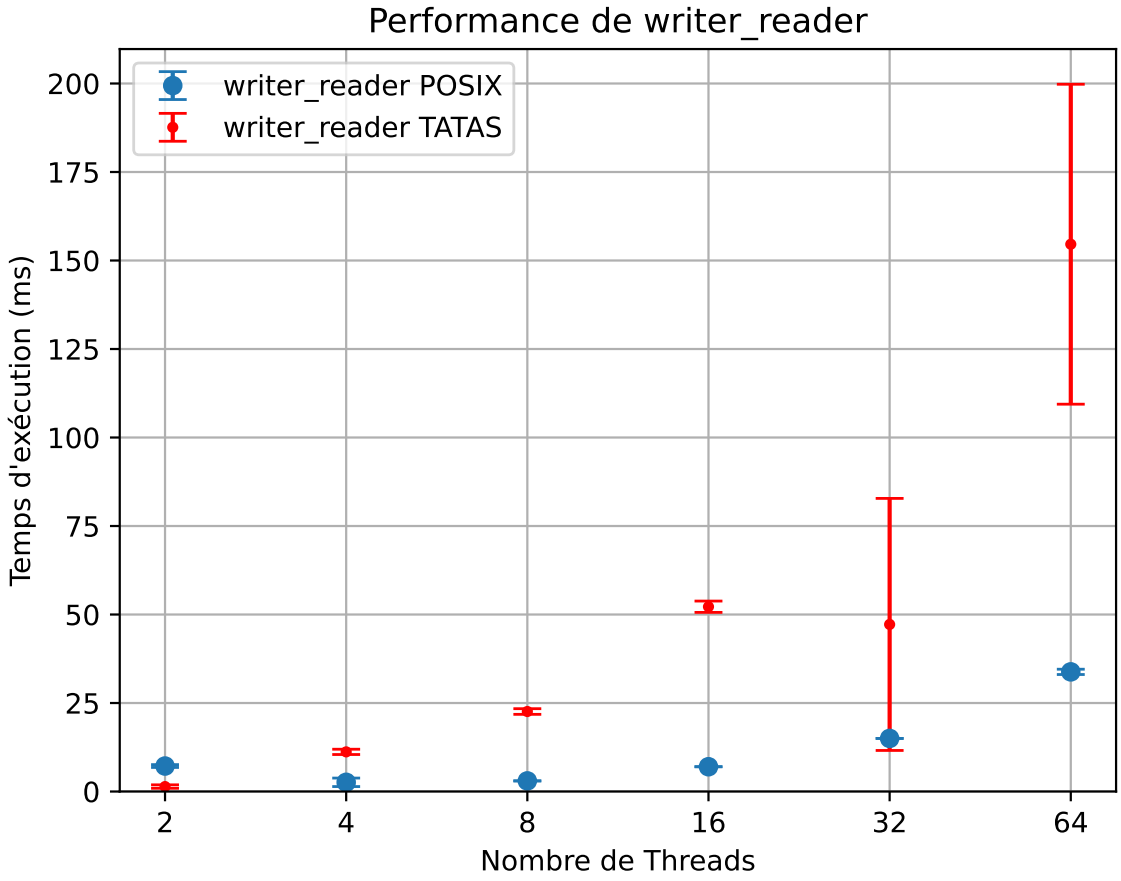

FIGURE 3 – Évolution du temps d'exécution en fonction du nombre de threads : Les points bleus représentent la version utilisant la librairie POSIX (mutex et/ou sémaphores), tandis que les points rouges représentent la version utilisant notre implémentation personnalisée basée sur des verrous avec attentes actives.

Ici, dans le cas du programme du lecteur et de l'écrivain, on remarque que la version utilisant la librairie POSIX a un temps qui augmente progressivement et l'écart type est très petit, ce qui prouve la stabilité et la fiabilité de l'implémentation des mutex/sémaphores POSIX. Quant à la version utilisant nos verrous, on constate que l'écart type devient important lorsque le nombre de thread est grand. Cela prouve que nos verrous ne sont pas optimisés pour gérer une synchronisation entre un très grand nombre de threads.

2.4 Test-And-Set & Test-And-Test-And-Set

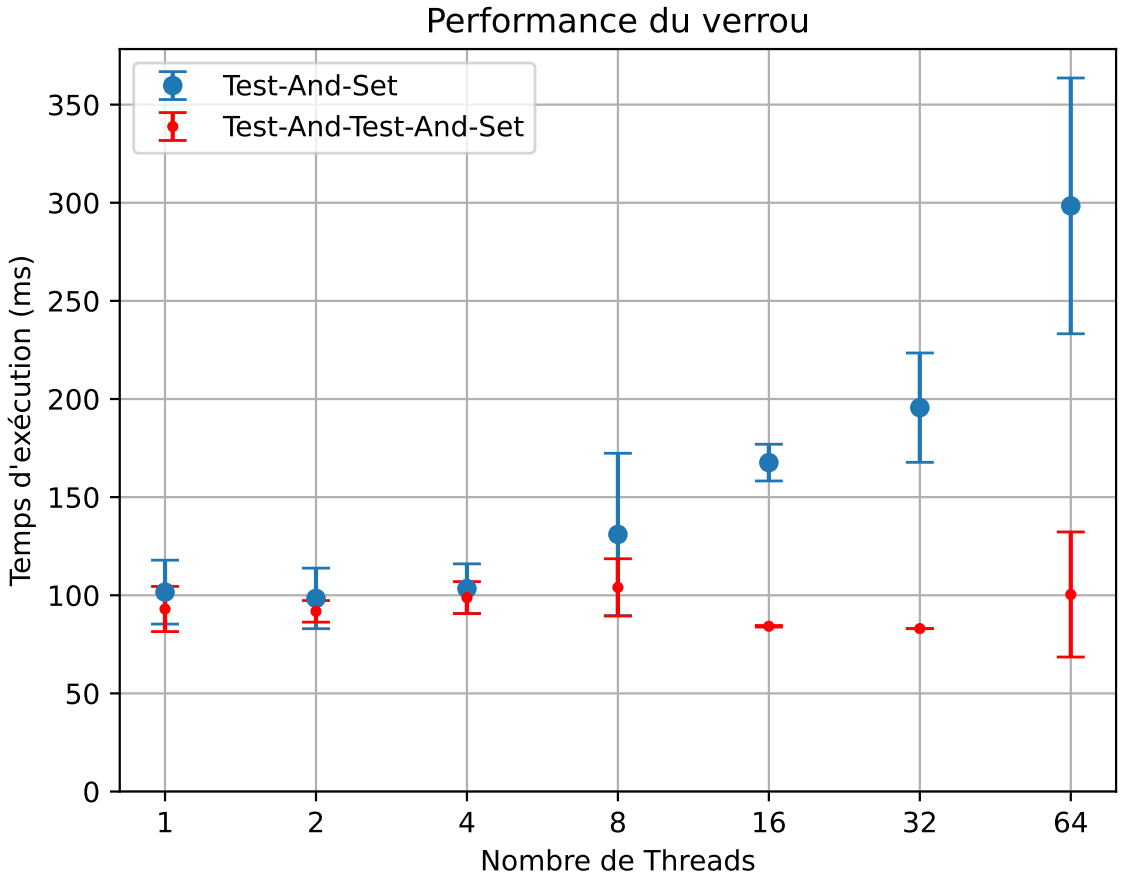

FIGURE 4 – Évolution du temps d'exécution en fonction du nombre de threads : Les points bleus pour la version utilisant l'algorithme ***Test-And-Set*** et les points rouges pour l'utilisation du ***Test-And-Test-And-Set***.

L'analyse qui peut être faite sur ce graphique est la différence entre les performances de notre verrou *Test-And-Set* (1) et le *Test-And-Test-And-Set*(2). Lors de l'utilisation du premier algorithme, on constate une instabilité lorsque le nombre de threads est élevé, cette instabilité est marquée par un écart-type plus important. Or, lorsque le deuxième algorithme est utilisé, même dans les cas où le nombre de thread augmente, l'écart-type lui ne va pas nécessairement augmenter.
Avant d'aborder ce que le second algorithme, *Test-And-Test-And-Set* optimise, il faut d'abord comprendre ce qui cause cette instabilité de l'algorithme *Test-And-Set*. Test-And-Set, un algorithme d'attente active, fait des opérations atomiques coûteuses en temps qui souvent ne sont pas nécessaires étant donné que le lock est déjà pris par un thread. La contention est élevée lors de l'utilisation de cet algoritme. C'est ce problème de contention que le *Test-And-Test-And-Set* va régler, il la réduit à l'aide d'une boucle while qui va être placée avant l'opération atomique, tant que le lock est déjà pris par un thread, il n'est pas nécessaire à un autre thread de faire une opération atomique pour s'accaparer le lock le permettant d'entrer dans une section critique.

3 Conclusion

Pour conclure l'analyse des performances de nos programmes, il y a plusieurs choses qui peuvent être relevés :

— Une différence observable entre les deux implémentations différentes (variance plus importante pour un très grand nombre de threads pour les programmes utilisant nos verrous, tandis que la variance n'est pas si élevée dans les programmes utilisant la librairie POSIX)

— Le Test-And-Set est un algorithme vraiment moins bon que le Test-And-Test-And-Set, TATAS permet de réduire le nombre d'opérations atomiques qui sont assez coûteuses ! Cependant, cela reste toujours moins bon que l'implémentation de la synchronisation faite dans la librairie POSIX. Ce qui est tout à fait logique, sinon quel serait l'intérêt d'utiliser cette libraire.