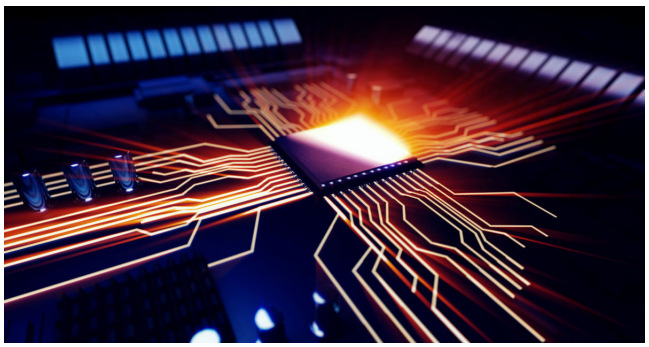
UCLouvain

École polytechnique de Louvain

LINFO1252 – Systèmes informatiques

Programmation multi-threadée et évaluation de performances



Année académique :
2022-2023

Auteurs :	**NOMA :**
Sanchez-Rivas Xavier	1617-18-00
Pihet Henri	6615-19-00

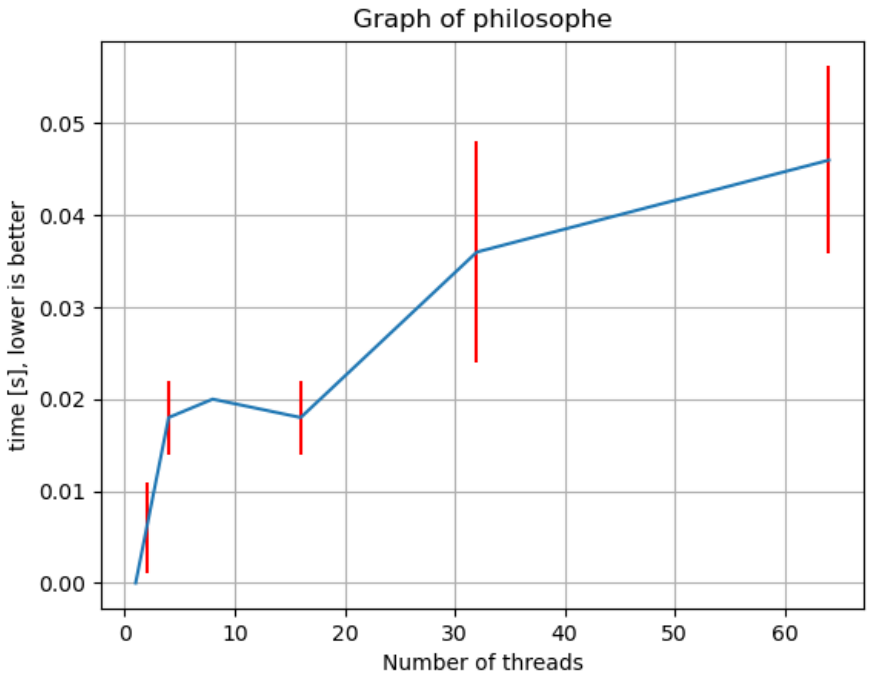
7 décembre 2022

1 Introduction du projet

Ce projet a pour but d'apprendre la programmation multi-threadé et effectuer une évaluation des performances. Dans un premier temps, il nous a été demandé d'implémenter différents problèmes : problème des philosophes, des producteurs-consommateurs, des lecteurs-écrivains. Dans un deuxième temps, il nous est demandé de mettre en place des primitives de synchronisation "POSIX" par attente successives et par attente active.

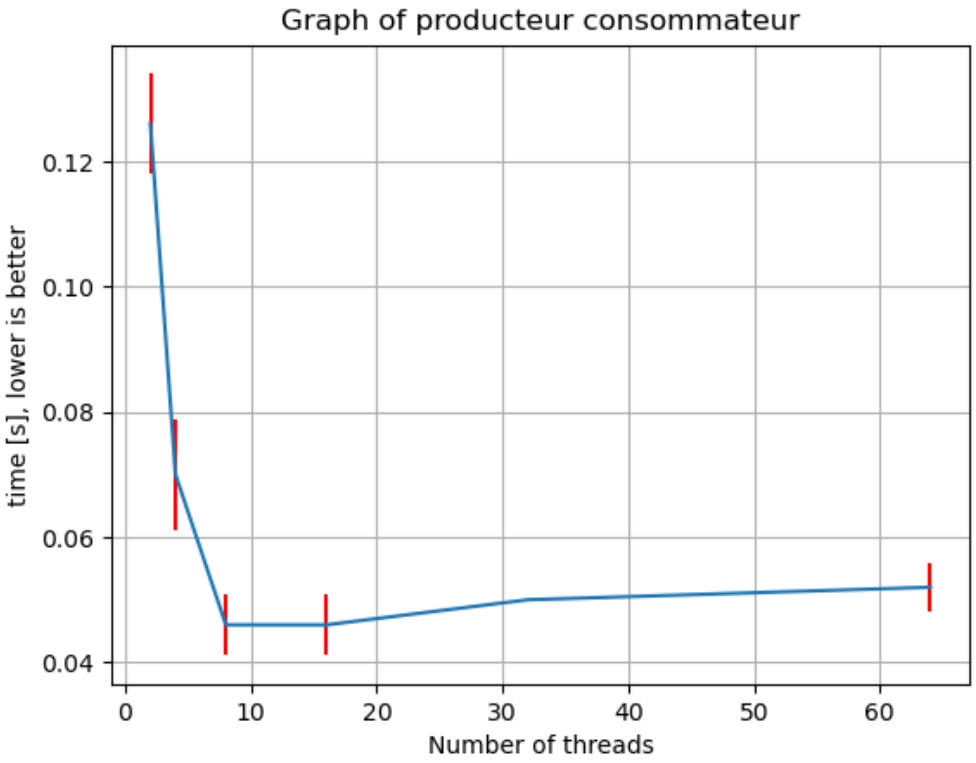
2 Mesure de performances

2.1 Problème des philosophes



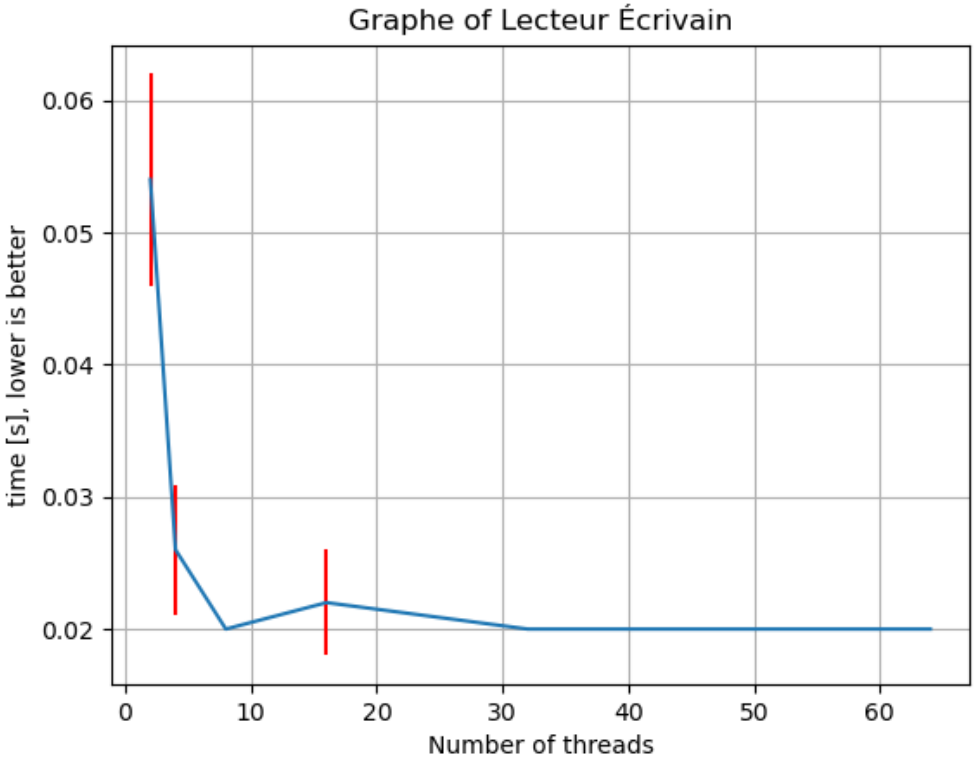
Pour ce cas-ci, de manière générale, on voit que quand le nombre de threads simultanés augmente, le temps d'exécution augmente aussi. Ces résultats peuvent s'expliquer par le problème lui-même. Comme un philosophe a besoin de 2 baguettes pour pouvoir manger, même si on augmente le nombre de threads cela ne va diminuer le temps d'exécution vu que si le philosophe tient les 2 baguettes d'autres philosophes n'auront pas de baguettes pour pouvoir manger et donc les autres vont devoir quand même attendre.

2.2 Problème des producteurs et consommateurs



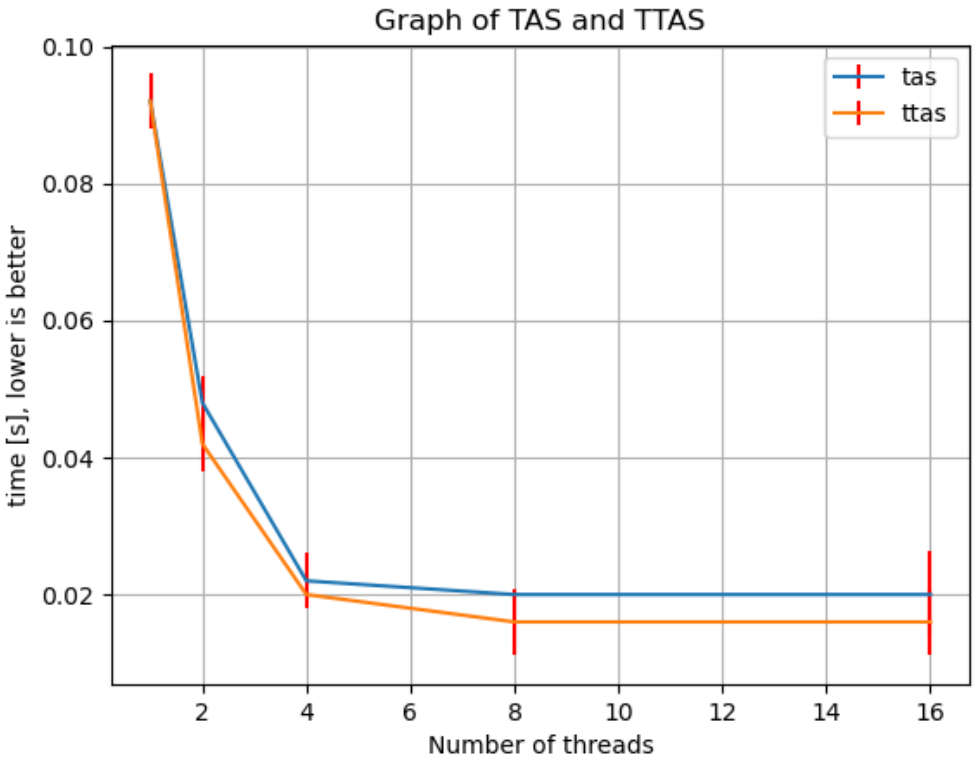
Dans le cadre du problème des producteurs et consommateurs, on remarque que lors des premiers ajouts de threads dans le programme, le temps d'exécution diminue drastiquement jusqu'à 8 threads simultanés. Après ce palier, l'ajout de threads ne va faire qu'augmenter le temps d'exécution. Si jamais les données sont vides, les consommateurs doivent attendre que le buffer se remplisse et sont mis en attente. De l'autre coté, nous avons les producteurs qui sont en attente d'avoir de la place pour mettre leurs données.

2.3 Problème des lecteurs et écrivains



Pour ce problème, il faut que si un écrivain écrit, les lecteurs ne peuvent venir lire le fichier qu'écrit l'écrivain sinon on risque de faire buguer le système. Mais on peut avoir plusieurs lecteurs simultanément car ils ne se gênent pas.

2.4 Test-and-Set et Test-and-Test-and-Set



On remarque qu'ici, plus nous augmentons le nombre de threads meilleur sera le temps d'exécution mais il n'y a plus d'évolution à partir du 8ème threads. L'algorithme Test-and-Set garantit d'avoir l'exclusion mutuelle. Cette différence de temps pourrait s'expliquer par le fait qu'avec les mutex POSIX il y a une latence entre l'appel "lock()" et l'exécution de la section critique.

Nous avons aussi l'algorithme Test and Test and Set, qui par rapport à l'algorithme Test-and-Set, effectue en permanence des appels à avec instruction xchg.

3 Conclusion

En résumé, ce projet nous a appris à utiliser en profondeur les threads et leur coordination entre eux, ainsi que, l'utilisation de primitives de synchronisation (POSIX) "semaphore" et "mutex".

Avec toutes les données récoltées lors de ce projet, nous pouvons en tirer quelques conclusions. Avec l'attente active on remarque que le temps est meilleur qu'avec les primitives de synchronisation POSIX.

Nous n'avons malheureusement pas réussi a implémenter tous les algorithmes demandés à temps. Ce retard est justifié par une mauvaise gestion du temps, beaucoup

de temps a été passé sur la compréhension des threads/lock/unlock et semaphores. Même si le projet n'est pas fini à temps nous avons pu améliorer nos capacités de problème solving et nos connaissances dans les outils tels que git/gcc/gdb.