UCLouvain

École polytechnique de Louvain

SIN1252

Evaluation de performances d'algorithmes de synchronisation

Charleroi *Groupe 1 :*
Simon Halliez
Benita Tia

Professeur :
Etienne Riviere

Année académique 2022-2023

Date de soumission : 7 décembre 2022

1 Introduction

Dans ce rapport nous analysons les performances de nos implémentations pour la résolution des problèmes du philosophes, du producteur/consommateur et du lecteur/écrivain.
Dans un premier temps, nous avons travailler avec les mutex et sémaphores provenant de la librairies POSIX. Puis, il nous a fallu créer nos propres spinlocks (verrous par attente active) en appliquant l'algorithmes de `test-and-set` (TAS) et celui de `test-and-test-and-set` (TTAS).

Chaque mesure résulte du temps moyen d'exécution de chacune de nos fonctions sur une machine qui possède 32 coeurs. Chaque moyenne est calculée sur base de cinq valeurs. Pour chacune de nos fonctions, nous avons effectué des mesures avec 1, 2, 4, 8, 16, 32, 64 fils d'exécution.

D'ailleurs, vous pourrez remarque nous avons commencé nos mesures avec deux threads pour les problèmes des producteurs/consommateurs et celui des lecteurs/écrivains. Cela nous semblait logique de ne pas commencer à un car un thread n'est pas suffisant pour exécuter leurs taches simultanément

2 Première partie : POSIX

2.1 Philosophe

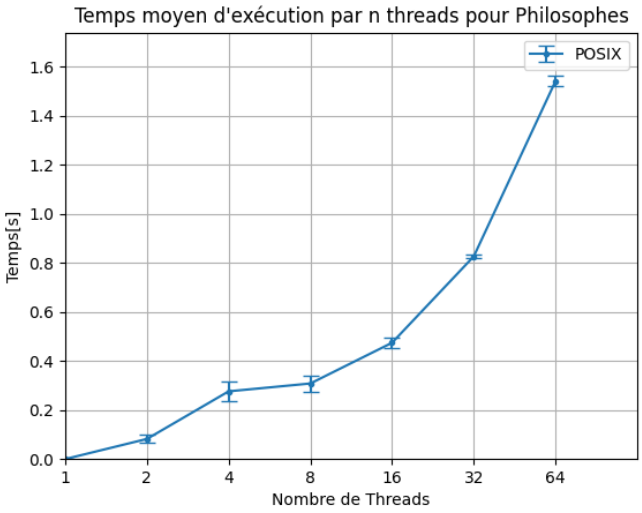

FIGURE 1 – Le graphe des performances de philosophes avec POSIX

Nous pouvons observer une croissance presque exponentielle. Le nombre de philosophe augmente de manière exponentielle et chacun doit faire un nombre d'action par cycles qui ne cesse d'augmenter. Cela semble logique que temps d'exécution augmente aussi.

Au début, notre graphe a un temps d'exécution rapide car il n'y a pas de soucis de concurrence.
En effet, il y a peu de philosophes, tout les mutexs sont libres et peuvent être exécuter simultanément. Mais comme le montre le graphe cela ne dure pas longtemps et les philosophes doivent attendre.
Il y a donc une perte de performances à cause de la synchronisation.

2.2 Producteurs-Consommateurs

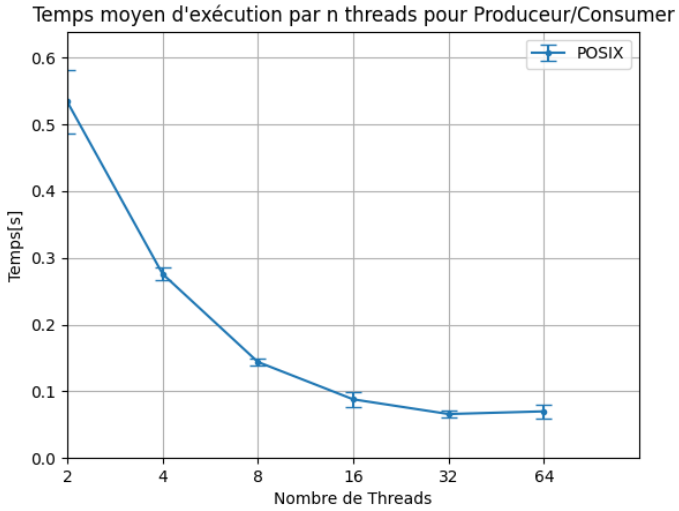

FIGURE 2 – Le graphe des performances de Produceur/Consummer avec POSIX

Nous pouvons observer sur le graphe que le temps d'exécution diminue au fur et à mesure que le nombre de thread double. En effet, plus nous augmentons le nombre de threads, plus nous profitons des 32 cœurs de la machine de test. Cette diminution s'atténue car la création des différents fil d'exécutions prend une place de plus en plus importante.

De plus, nous remarquons que la décroissance du temps d'exécution est presque nul au-delà de 32 cœurs. A ce niveau, la création ne permet pas d'exploiter plus de cœur et donc de gagner du temps. Le temps passé dans la section critique est le même pour chacune des exécutions. Il s'agit du temps de 8192 get et 8192 put.

2.3 Lecteurs et écrivains

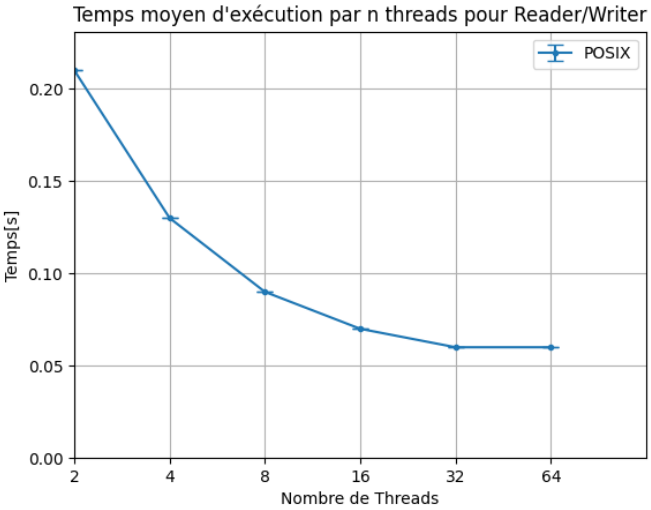

FIGURE 3 – Le graphe des performances de Les Lecteurs/Ecrivains avec POSIX

Pour l'exécutable du Reader Writer, nous avons opté comme pour le Producteurs-Consommateurs, d'utiliser qu'un seul argument pour le nombre de thread car nous devions systématiquement avoir le même nombre de chaque types de fil d'exécution
Nous remarquons une décroissance exponentielles du temps d'exécution moyen. En effet, lorsqu'on augmente le nombre de threads, il y a de plus en plus de reader et writer qui peuvent effectuer leurs tache.
En outre, comme pour le producteurs-Consommateur, il y a une stabilisation a partir de 32 threads.

3 Deuxième partie : Test-and-set et Test-and-test-and-set

3.1 TAS et TTAS

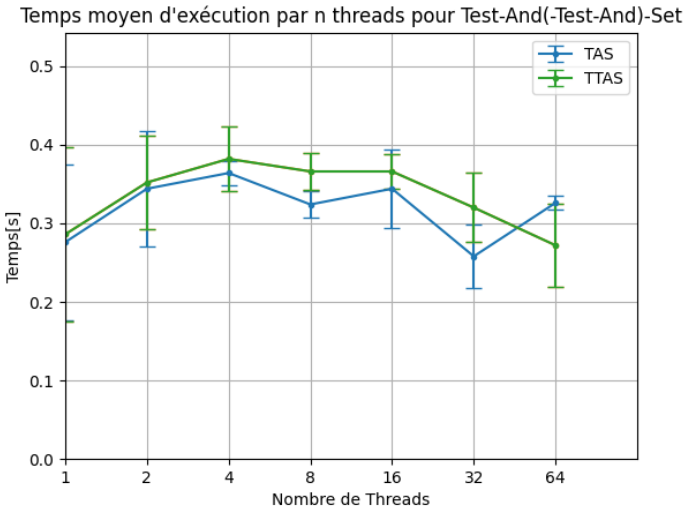

FIGURE 4 – Le graphe des performances de TAS et TTAS

Nous pouvons observer sur le graphe que les performances de notre mutex test-and-set et celle de notre mutex test-and-test-and-set sont relativement similaires. La seul différence enter ces deux algorithmes est l'insertion d'une boucle d'attente active dans l'implémentation du double testeur.
Cette boucle a la particularité de ne pas effectuer d'appel à l'instruction atomique mais de vérifier directement la valeur dans notre mutex dans le cache local. Elle permet d'éviter de saturer le bus avec des appels à l'instructions xchgl qui est une instruction qui bloque le bus.

Notre hypothèse est que le bus n'est pas saturé lors de l'exécution du programme. Nous ne tirons donc pas davantage à utiliser l'algorithme test-and-test-and-set.

3.2 Les Philosophes

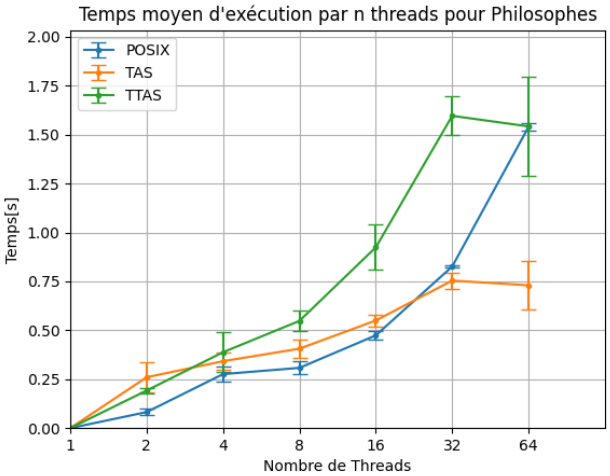

FIGURE 5 – Le graphe des performances de Philosophes avec POSIX, TAS, TTAS

Lorsque nous comparons les performances des nos mutex et sémaphores contre celles de POSIX, nous pouvons voir que les fonctions utilisant la librairie POSIX sont plus rapides jusqu'à 16 thread et elles sont plus plus stables que les autres tout le long. [1].

1. Voir les écarts types.

Pourtant, nous remarquons qu'à partir de 32 threads, les TAS et TTAS se stabilisent ce qui ne semble pas être le cas pour le POSIX qui semble continuer a croitre de manière exponentielle.

Nos spinlocks fonctionnent plus lentement que ceux de POSIX. Le temps d'attente de nos threads peut être plus long car ils prennent plus de temps à rentrer dans la section critique. En effet, comme vu en cours, le fait de faire constamment appel à « xchg » sature le bus ce que l'on peut remarquer dans notre cas.
Cependant, nous pouvons aussi observer que le TTAS est plus lent que le TAS ce qui théoriquement ne devrait pas être le cas.

3.3 Les Producteurs et Consommateurs

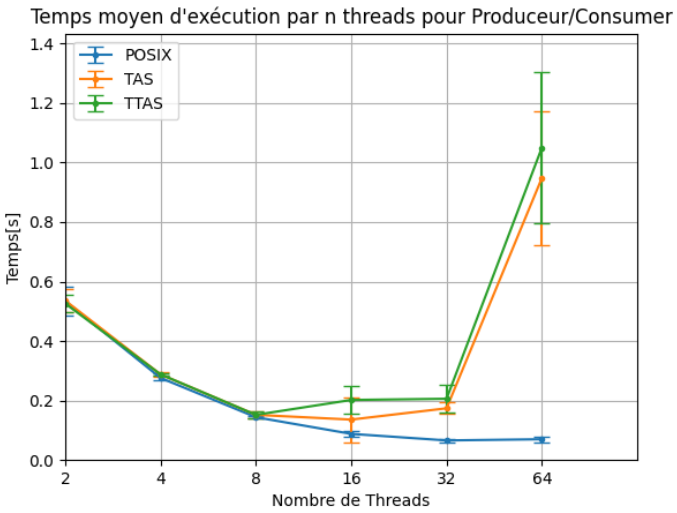

FIGURE 6 – Le graphe des performances de Producteurs/Consommateurs avec POSIX, TAS, TTAS

Le temps d'exécution est similaire pour nos trois modèles de deux à huit threads. La synchronisation ne prend donc pas de place importante dans cette tranche. La majorité du temps est consacrée à la production et la consommation des données.
Ente 8 et 32 threads, le temps est relativement constant pour nos deux spinlocks alors celle de la librairie POSIX continue de légèrement diminuer. L'efficacité des différents verrous doit être la cause de ces différences. Au-delà de 32 threads, la durée d'exécution explose pour nos algorithmes.

L'explication la plus plausible est que le bus est saturé par des appels à xchgl. La production et la consommation sont donc fortement ralenties.
Le modèle au double tests devrait logiquement saturer moins vite. Il est probable que le passage de trente-deux à soixante-quatre fil d'exécution soit trop brutal pour observer une réelle différence. La librairie POSIX ne sature pas le bus et continue donc de diminuer son temps d'exécution. Cette décroissance s'atténue car la création de threads prend une place de plus en plus importante et les différents cœurs sont tous utiliser.

3.4 Les Lecteurs et Ecrivains

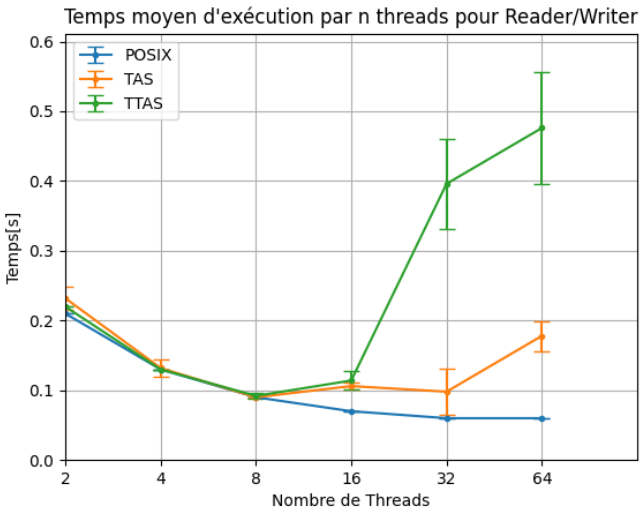

FIGURE 7 – Le graphe des performances de Lecteurs/Ecrivains avec POSIX, TAS, TTAS

Au début du graphe, nous pouvons observer que les temps d'exécution sont le mêmes tout types de spinlock confondu mais à partir du 8ème thread cela change. En effet, dès que le nombre de thread augmente, le temps d'exécution des POSIX de cesse de diminuer tandis que celui des autres augmente.
Nous supposons que leurs bus sont saturés et immobilisent les lecteurs et écrivains qui doivent attendre pour obtenir l'accès aux sections critiques.

Les POSIX sont plus efficaces car ils ne font pas d'attente dans une bloucle, ce qui ne sature pas le bus d'appels à xchlg. Cela permet un gain de performance.

4 Conclusion

Nous avons appris que nos attentes ne reflètent pas la réalité. Les résultats des performances de TAS et TTAS nous ont surpris car nous pensions que nous aurions le contraire point de vue performances.
De plus, nous avons constaté que l'utilisation de plus threads n'est pas forcément synonyme de meilleurs résultats pour des algorithmes de spilock à cause de la saturation du bus.

Enfin, la création de nos propres spinlocks était un défi que nous avons accepté de relevé au mieux mais la librairie POSIX est très bien faites. Elle est beaucoup plus stable et efficace que nos créations dans la gestion d'un grand nombre de threads.