LINFO1252 : Système Informatique

Nathan Tihon

November 30, 2021

1 Introduction

Le but de ce projet est d'implémenter et évaluer les performances de plusieurs algorithmes de synchronisation. L'évaluation de performances ayant été effectuée s'articule autour de 2 paramètres, le nombre de *thread* et le type de primitive de synchronisation utilisées. Le nombre de thread sera un paramètre entier comprit entre 1 et $2n$ où n est le nombre de coeurs du processeur utilisé tandis que les primitives utilisées appartiendront quant à elle à 2 catégories.

- Les mutex et sémaphores POSIX provenant de `<pthread.h>` et de `<semaphore.h>`.
- Les verrous à attente active (spinlock) ainsi que les sémaphores construites sur ces verrous.

Les figures analysées dans ce rapport seront composées de 2 courbes représentant le temps moyen en fonction du nombre de threads, accompagnées de "barres d'erreur" représentant l'écart-type des mesures. Les couleurs représentent la catégorie de primitives utilisées (paramètre catégoriel), l'axe des abscisses montre le nombre de thread (paramètre entier) et l'axe des orconnées représente le temps d'exécution en fonction de ces paramètres.

Il est à noter qu'à partir de la section 3 les figures présentent toutes une valeur nulle pour un thread unique. Cela est dû au fait que nous analysons des algorithmes de *synchronisation*. Il nous faut donc au minimum 2 threads pour pouvoir analyser le comportement de ces algorithmes.

2 Verrous à attente active

Commençons tout d'abord par analyser la performance des spinlocks, car nous utiliseront ceux-ci dans la suite de nos analyses. Il est ici proposé de comparer 2 algorithmes d'implémentation de spinlocks : l'algorithme *test-and-set*, et l'algorithme *test-and-test-and-set* [1], j'ai prit la liberté d'également afficher les performances des mutex POSIX afin d'avoir un témoin.

La figure ci-dessous représente les performances des verrous à attente active :

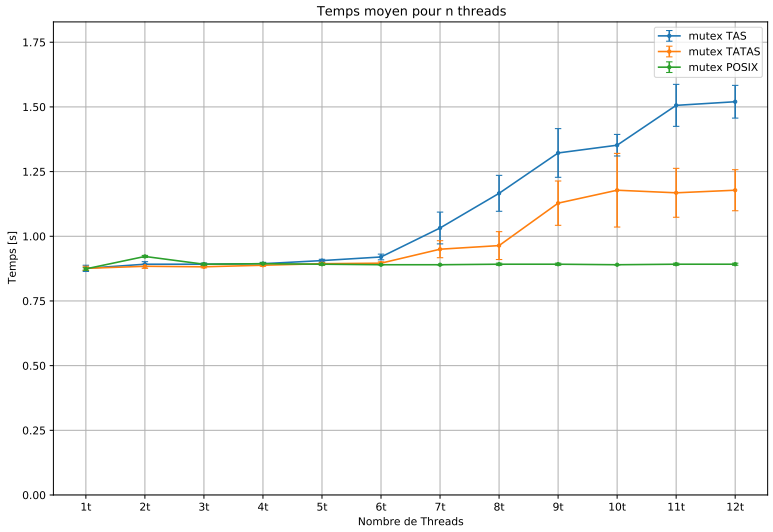

Figure 1: Temps d'exécution d'un verrou à attente active dans différentes conditions

[1] Ces algorithmes sont nommées TAS et TATAS pour la suite de ce rapport

Nous pouvons observer que les performances des 3 types de mutex sont assez semblables jusqu'au seuil de 6 threads, à partir duquel les performances divergent. On observe que les performances des mutex POSIX sont extrêmement stable peu importe la quantité de threads, ce qui en fait un parfait témoin. On peut ensuite observer que les performances de l'algorithme TAS se dégradent plus vite que celle de l'algorithme TATAS lorsque l'on augmente le nombre de threads. Pour être plus précis, on observe que la différence de temps d'exécution entre les mutex TAS et POSIX est 2 fois plus grande que la différence entre les mutex TATAS et POSIX.

Cela peut être expliqué par le fait que l'algorithme TAS effectue des blocages répétés du bus de cache (à cause de l'instruction `xchg` s'exécutant répétitivement jusqu'à acquisition du spinlock) ce qui ralentit considérablement la vitesse à laquelle les différents coeurs synchronisent leur cache (et donc les variables partagées).
Au vu des meilleures performances des mutex TATAS, j'ai choisi d'implémenter les sémaphores sur base de ceux-ci.

3 Problème des philosophes

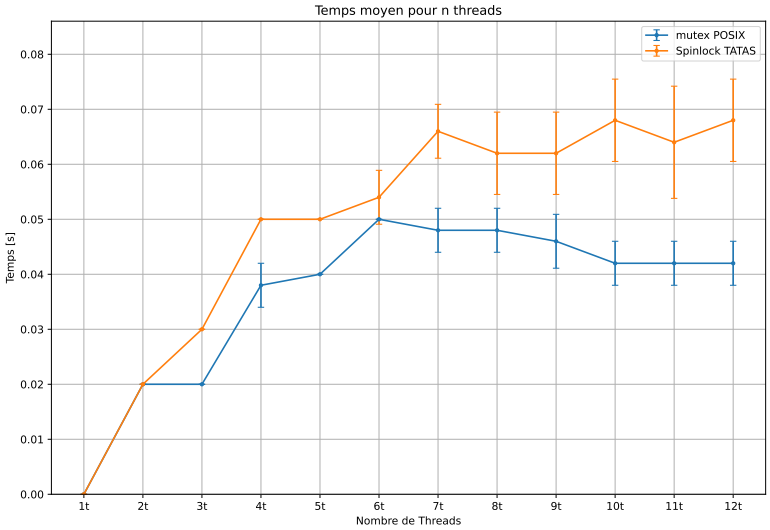

Figure 2: Temps d'exécution du problème des philosophes dans différentes conditions

On observe que les deux courbes commencent par une phase de croissance pour se terminer par une phase de stabilisation. On observe également que les mutex POSIX sont généralement plus stable que les spinlock TATAS. En effet on voit clairement que l'écart-type est bien inférieur.

Nous savons que théoriquement, le temps d'exécution est censé être constant. Suivant la parité du nombre de philosophes et en considérant la disposition des baguettes nous avons que si le nombre de philosophes est pair, la moitié des philosophes peuvent effectuer leurs opération pour ensuite laisser place à la seconde moitié, donnant un temps d'exécution constant. Dans le cas où le nombre de philosophes est impair, $\frac{n-1}{2}$ philosophes peuvent effectuer leurs opérations en même temps, résultant également en un temps constant (théoriquement plus long que pour un nombre pair).

Mais nous n'observont pas du tout un temps d'exécution constant. Il y a donc une perte de performance dûe à la synchronisation. De plus, les spinlocks sont moins performant que les mutex POSIX pour cette

application. Cela est cohérent avec ce que l'on a vu au cours : l'attente active est moins efficace lorsque le temps d'exécution d'une section critique (SC) est du même ordre de grandeur que le temps entre 2 accès à la SC.

4 Problème des Producteurs-Consommateurs

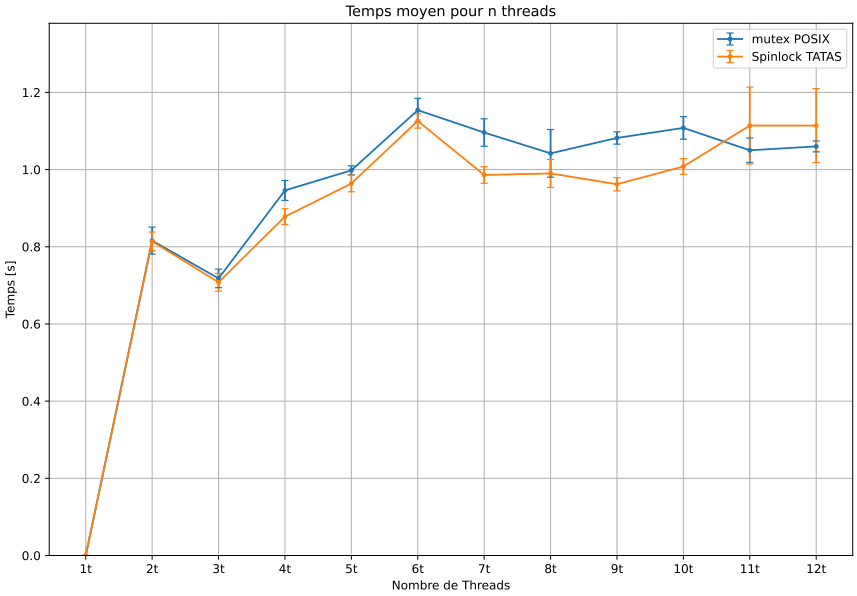

Figure 3: Temps d'exécution du problème des producteurs-consommateurs dans différentes conditions

On peut remarquer ici que la différence de temps d'exécution entre les deux algorithmes est moins marquée que pour le problème des philosophes. Cela est dû au fait que le temps d'exécution des SC est ici négligeable par rapport au temps d'accès entre 2 SC. On remarque également que pour un travail équivalent le temps d'exécution augmente, il y a donc également une perte de performance dûe à la synchronisation.

5 Problème des lecteurs-écrivains

On observe que la tendance est généralement croissante pour les deux courbes, à l'exception d'une baisse au niveau des threads 5 et 6. On remarque également une divergence des deux courbes pour un nombre de threads supérieur à 7. La divergence peut être expliquée de la même manière que celle dans le cas du problème des philosophes, à savoir que le temps d'accès entre 2 SC est négligeable par rapport au temps d'exécution des SC.

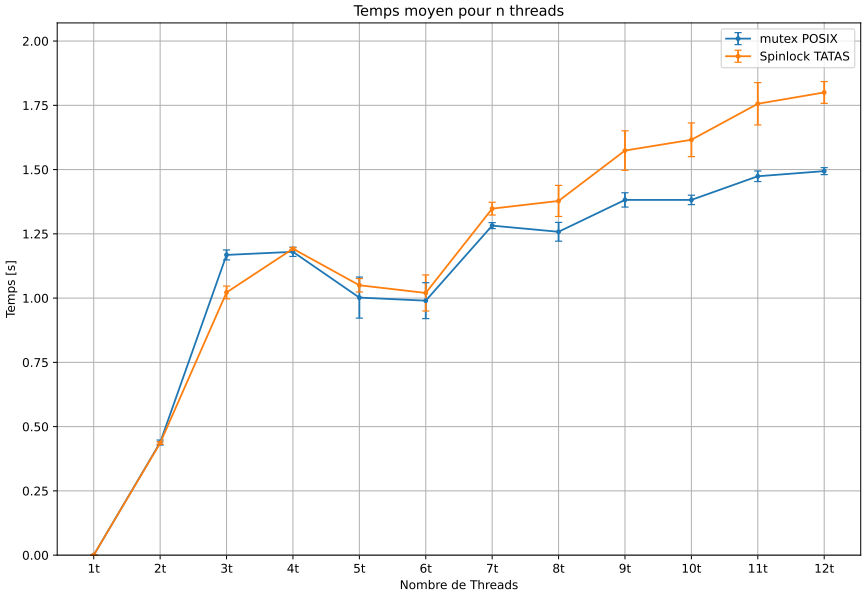

Figure 4: Temps d'exécution du problème des lecteurs-écrivains dans différentes conditions

6 Conclusion

D'un point de vue théorique, j'ai pu apprendre lors de ce projet qu'ajouter des threads n'est pas synonyme d'amélioration de performances. En effet, j'ai pu voir qu'il est nécessaire de prendre en compte le temps d'exécution des sections critiques ainsi que le temps entre 2 accès à celles-ci. J'ai également pu apprendre à effectuer des instructions atomiques en assembleur, tout en prenant conscience de l'impact de ces instructions sur les performances *générales* de l'ordinateur. L'écriture d'opération en assembleur m'a également permis d'en apprendre davantage sur la gestion des registres des processeurs et les interactions de ces derniers avec le bus de cache.

D'un point de vue pratique, je me suis forcé à écrire du code propre et *générique*. Ce qui m'a permit de découvrir l'utilité des unions afin d'écrire des interfaces facilement utilisables avec des types de données différentes.

En conclusion, ce projet m'a permit d'ouvrir les yeux sur le point de vue *hardware* du multithreading tout en améliorant mes capacités d'écriture et d'analyse de code nécéssaire à tout programmeur.