UCLouvain

epl

LINFO1252

Rapport Systèmes informatiques
Mesures de performances d'algorithmes multi-thread

Groupe 6 - 3

2 décembre 2020

ONCIUL Andru - 69721800
MOUNZER Amine - 69161800

Année 2020-2021

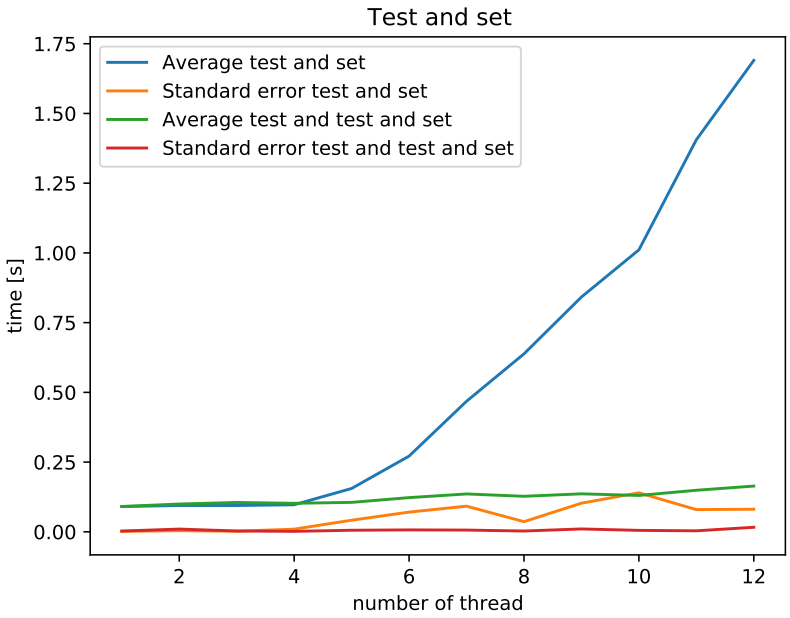
Professeur : Riviere Etienne

1 Introduction

Ce rapport présente les différents graphes obtenus lors des tests de performances effectués sur plusieurs algorithmes multi-thread. Chaque performance observée sera expliquée et détaillée dans un petit paragraphe approprié.
Dans les graphes de producer-consumer et reader-writer, nous effectuons, pour chaque itération, nos tests avec un nombre de thread (allant jusque 12) égal pour les producer/reader/consumer/writer excepté pour un nombre impair de thread. Dans ce cas la, on incrémente de 1 le nombre de thread reader/consumer.
Dans toutes nos explications pour désigner les sémaphores et mutex que nous avons nous-mêmes créer utilisant l'algorithme de test and test and set, nous utiliserons le terme *spinlock*. Quant aux implémentations existante de C présentent dans la librairie POSIX, nous les nommerons *POSIX*.

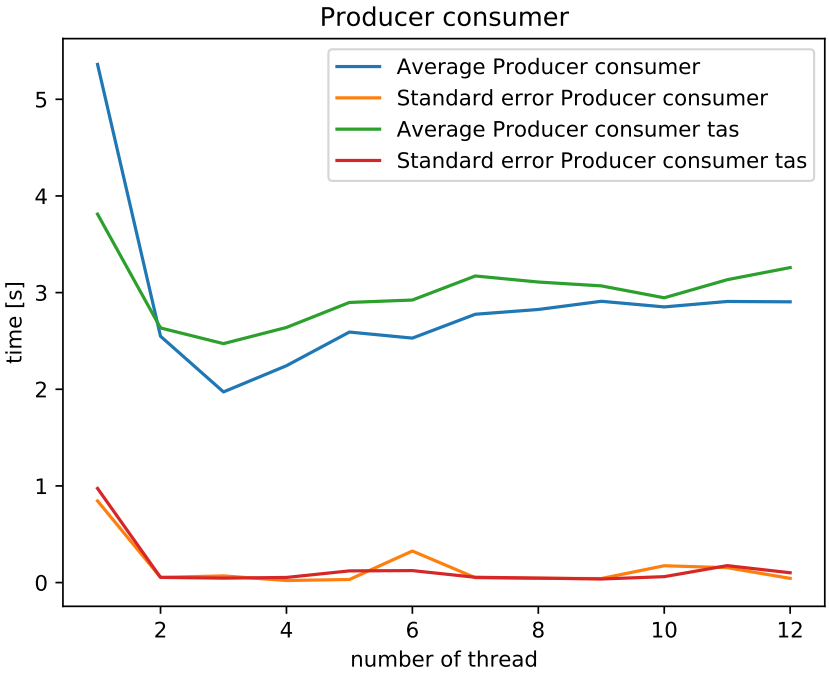
2 Algorithme

2.1 Test and set



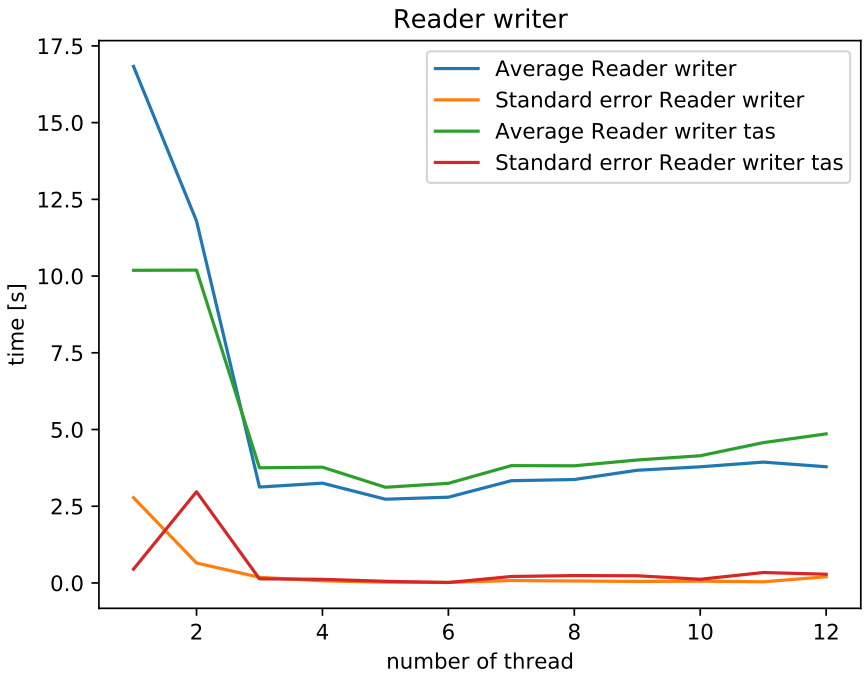
Dans ce graphe représentant les performances de l'algorithme *test and set* (et sa version améliorée), on peut aisément observer que les performances de la version classique croissent de manière exponentielle en fonction du nombre de thread utilisé, tandis que sa version améliorée obtient des performances constantes. Ce phénomène est explicable notamment grâce à la différence entre le nombre d'appels de la commande *xchg*. En effet, Dans la première version de l'algorithme, les différents thread qui veulent exécuter leur tâche effectuent constamment des appels de *xchg* afin de vérifier si la valeur de la variable **lock** a changé et donc de s'assurer que unlock a bien été appelé. Or les instructions atomiques comme *xchg* ont un coût d'utilisation non négligeable, notamment sur le processeur qui l'exécute mais également sur les autres processeurs car elles bloquent le contrôleur (le bus) qui permet le partage d'informations entre processeurs.
La solution pour pallier ce problème est la version améliorée du test and set, qui, avant d'appeler *xchg*, lit la valeur de la variable **lock**. Lorsque celle ci redevient nulle, les différents thread tentent alors d'appeler *xchg*, mais si la valeur de **lock** est encore égale à 1, alors les thread patientent sans effectuer d'appel qui pourrait impacter les performances.

2.2 Producer Consumer



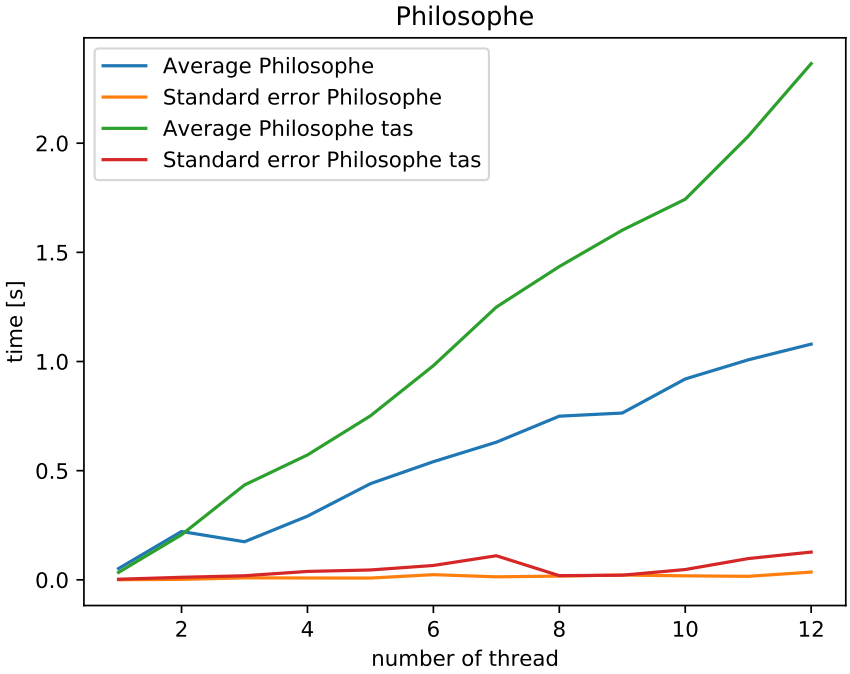
On peut voir dans ce graphe le temps moyen d'exécution de producer consumer pour un nombre de threads croissant. On remarque en premier qu'avec un nombre de threads croissant, le temps se stabilise et on a également un pic pour un petit nombre de threads. On peut voir un pic pour numberofthreads = 1 avec donc 1 producer et 2 consumer. Ce pic est expliqué par le fait que les consumer attendent que le producer crée une nouvelle valeur et se ruent tous les deux dessus et se bloquent entre eux. Ce pic a un impact différent pour le code avec les *spinlock* et pour celui avec les outils de synchronisation de *POSIX*.
En effet pour des échanges rapides et sans trop de thread, test and set améliorée est plus efficace car il permet des échanges plus rapides du thread qui s'exécute. Test and set amélioré devient moins efficace avec plus de thread car ceux-ci attendent en moyenne plus de temps avant d'avoir à nouveau le mutex. Cela signifie que les thread restent plus longtemps dans la boucle while du *spinlock*, ce qui ralentit fortement l'exécution totale.
POSIX devient plus intéressant à partir de numberofthreads = 3 (3 producer et 4 consumer). En effet, pour un code dans lequel les temps d'attente moyens sont grands, les mutex/sémaphores de *POSIX* sont le meilleur choix (Nous expliquerons les raisons de ce phénomène ultérieurement).

2.3 Reader Writer



La première chose que l'on remarque est que les performances avec les *POSIX* pour 1 seul thread (2 thread reader et 1 writer) sont bien supérieures aux performances obtenues par notre algorithme *spinlock*. On peut facilement expliquer cela par le fait que les échanges entre thread (le temps moyen passé à attendre d'entrer en section critique) est extrêmement rapide dans nos tests de performances. Les *spinlock* sont beaucoup plus intéressants (en terme de rapidité) dans cette situation là, mais dès que le nombre de thread augmente, on observe directement que les *POSIX* prennent le dessus.
Cette dernière observation est cohérente puisque dans reader writer lorsque le writer travaille, tous les autres thread sont bloqués, donc leur temps d'attente moyen pour entrer dans la section critique augmente. Ce temps d'attente moyen influence fortement le temps pris par test and test and set utilisé dans *spinlock*, alors qu'il n'influence absolument pas les *POSIX*. Ces derniers ne perdent du temps que lorsqu'ils doivent bloquer/débloquer un thread car ils sont obligés de les mettre en "pause" et de les "réactiver" par après.
De plus, un nombre de writer élevé découle sur un enchaînement d'occupation des writer (puisque notre algorithme donne la priorité au writer = reader starvation), donc avant que les reader ne recommencent à travailler, plusieurs writer se seront potentiellement enchaînés, ce qui augmente d'autant plus le temps d'attente en section critique pour les reader. Cependant les performances du *spinlock* n'explosent pas car lorsque les reader prennent le contrôle, leur efficacité rattrape fortement le temps perdu à attendre. Il est donc normal que les *POSIX* (sleep/wakeup mutex) soient plus efficaces lorsque le nombre de thread augmente.

2.4 Philosophe



Dans le graphe ci-dessus nous avons effectué les tests avec un million de cycle pour philosophe et non 10.000 comme demandé. Ce dernier était trop rapide et nous empêchait d'observer correctement les variations de temps entre les deux type de mutex et avec un nombre croissant de threads.
Ce qui saute au yeux dans ce graphe c'est la croissance presque linéaire des deux codes. En effet, en augmentant le nombre de philosophe on augmente également la charge de travail total car chaque philosophe doit faire tous ses cycles.
Les philosophes *spinlock* sont beaucoup plus lents car le temps d'attente moyen augmente avec le nombre de threads. Cet algorithme est différent des deux cités précédemment, ici le nombre de mutex est égal au nombre de thread mais, chaque baguette (= mutex) ne sera accédée que par le philosophe à sa droite ou à sa gauche. Ce temps d'attente qui augmente linéairement avec le nombre de thread est expliqué par deux facteurs.
Tout d'abord, on a des chaînes d'attente plus longues, c'est-à-dire que chaque philosophe attend la baguette à sa droite et lorsqu'une baguette de droite se débloque, ces derniers vont s'exécuter une fois que le philosophe à sa droite aura terminé.
Ensuite nos *spinlock* ne respectent pas le principe de fairness (équité). *POSIX* implémente bien une fairness se basant sur la priorité de chaque thread et leur temps d'attente. Ce qui signifie qu'avec beaucoup de thread et pour les *spinlock* cela arrivera souvent que un ou plusieurs thread aient un temps d'attente très long car il ne réussit plus a récupérer le **lock**.
Or nous avions bien vu qu'un temps d'attente plus long augmente le temps d'exécution des codes utilisant *spinlock*.

3 Conclusion

Pour conclure, nous trouvions intéressant de faire une comparaison générale entre les *POSIX* (sleep/wakeup mutex) et les mutex *spinlock* que nous avons implémenté durant ce projet.

Tout d'abord, commençons par bien définir les mutex utilisé par défaut en C de la librairie *POSIX*. Ces mutex sont appelés des mutex sleep/wakeup car la manière dont il bloque les thread lorsque que ces derniers entrent dans la section critique est la suivante : on effectue un syscall qui les mets en sleep et pour laisser le noyau gérer toute la partie synchronisation des threads. C'est donc le scheduler (appartenant au noyau) qui va choisir l'ordre de priorité et donc garantir l'équité (fairness).
Ce système est bien différent de notre implémentation des mutex avec test and set qui est basée sur un *spinlock* (busy waiting). Pour résumé, chaque thread est bloqué dans une boucle tant que la valeur d'une variable globale **lock** n'est pas remise à 0.

Ensuite, nous pouvons aisément donner quelques avantages et inconvénients des deux différentes méthodes :

Tout d'abord, définissons 3 parties dans l'exécution d'un *POSIX* ou *spinlock*.

- *attente* : lorsqu'un thread arrive a un appel lock et ce lock est occupé par un autre thread.
- *blocage* : le code qui bloque l'accès pour d'autres threads une fois que le thread trouve ce lock disponible.
- *déblocage* : le code qui débloque l'accès

Nous allons comparer les deux types de verrou par rapport au temps pendant lequel le CPU est occupé par ce verrou.

Tout d'abord le *spinlock* a un temps d'*attente* qui dépend du temps d'attente moyen entre les accès de chaque thread aux sections critiques. En effet puisque les threads sont coincés dans une boucle, si les temps d'attente moyens sont grands, le CPU sera occupé plus longtemps à faire tourner la boucle dans laquelle le thread est bloqué. Cependant le temps de *blocage* et de *déblocage* de ceux-ci sont très rapide puisqu'ils n'ont qu'à modifier la valeur de la variable **lock**.

Ensuite, les *POSIX*, quant à eux, ont un temps d'*attente* indépendant du temps moyen entre deux accès d'un thread à une partie critique. Ceci vient du fait que c'est le noyau qui met le thread en "pause" par un syscall. Cependant le temps de *blocage* et de *déblocage* est plus long car les syscall prennent plus longtemps. Un *POSIX* n'est donc pas une bonne idée si le temps entre deux accès à un lock est plus court que le syscall effectué par celui-ci.

Enfin, pour choisir entre les deux méthodes il faut surtout se fier sur ce dont vous avez besoin. Dans la plupart des cas les mutex/semaphores de *POSIX* seront intéressants car ils sont optimisés pour fonctionner avec de nombreux thread et pour effectuer des tâches abondantes dans la section critiques. Cependant le *spinlock* n'est pas une mauvaise option si le temps d'attente avant d'entrer en section critique est petit et que vous voulez des échanges rapides entre les thread.
Par ailleurs, beaucoup d'algorithmes multi-thread de nos jours utilisent des versions contenant un mélange de spinlock et de sleeping mutex. Par exemple si l'accès au **lock** échoue, on effectue un spinlock pour un court instant, en essayant de récupérer le **lock**. Si le spinlock échoue, alors on fait un syscall avec un sleep jusqu'à ce que le **lock** soit disponible.

4 Annexes : ressources

http ://www.alexonlinux.com/pthread-mutex-vs-pthread-*spinlock*
https ://probablydance.com/2019/12/30/measuring-mutexes-*spinlock*s-and-how-bad-the-linux-scheduler-really-is/
https ://matklad.github.io/2020/01/04/mutexes-are-faster-than-*spinlock*s.html
https ://stackoverflow.com/questions/22087146/*POSIX*-threads-and-fairness-semaphores22092652
https ://en.wikipedia.org/wiki/Test-and-set
https ://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html