École Polytechnique de Louvain

LINFO1252
Systèmes informatiques
Rapport du groupe 60

Programmation multi-threadée et évaluation de performances

Étudiants :
Sebastian KLINOVSKY 04392000
Alexandra ONCIUL 54212000

Professeur :
Etienne RIVIERE

13 mars 2025

Table des matières

1 Introduction

Dans le cadre du cours de Systèmes Informatiques, il nous a été demandé d'implementer plusieurs algorithmes (philosopher, producer-consumer, reader-writer). Nous avons du également réimplementer, en assembly, nos propres locks (primitives d'attente active) et sémaphores . Ce rapport présente les différents graphes obtenus lors des tests de performances. Pour avoir une idée du temps écoulé durant une exécution en fonction des threads, nous avons exécuter 5 fois chaque test avec 1, 2, 4, 8, 16, 32 et 64 threads. Nous avons calculé la moyenne (mean) et l'écart-type (standart deviation) de ces 5 exécutions et nous les avons placées dans un graphe. Chaque graphe de performance sera détaillé et expliqué par un paragraphe. Nous avons décider d'inclure également nos graphes obtenus en local (8 cores) étant donné que les graphes obtenus avec le serveur inginious (32 cores) nous semblent curieux.

2 Test-and-set and test-and-test-and-set

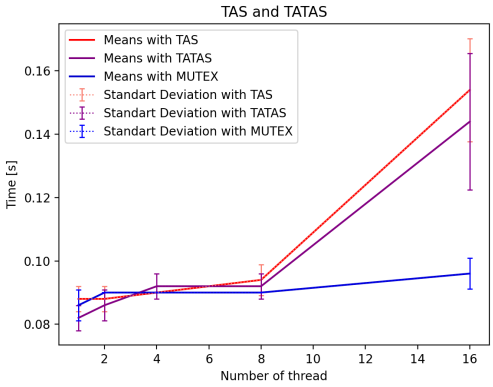

FIGURE 1 – Local performance

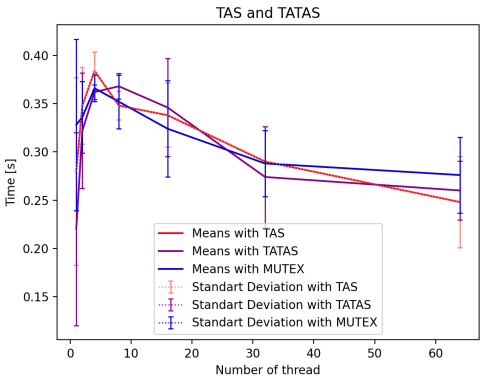

FIGURE 2 – Inginious performance

Dans le graphe en local, nous retrouvons les performances de l'algorithme test-and-set (TAS) ainsi que celles de test-and-test-and-set (TATAS). Nous avons effectué le test avec 10 000 cycles lock-unlock. Comme nous pouvons l'observer sur la figure 1, l'implémentation mutex de la librairie POSIX est plus rapide que nos implémentations, ce qui a clairement du sens, nous allons pas réinventer la roue mieux que des professionnels. Concernant notre algorithme TATAS, il est plus rapide que TAS. Tous les deux ont l'air de suivre une croissance similaire. Nous pouvons expliquer la meilleure performance de TATAS par le fait qu'il y a moins d'appels de la commande "xchg" qui ont un coût d'utilisation non négligeable pour son processeur mais aussi sur les autres prosseceurs car celui-ci bloque le controleur. En effet, TATAS vérifie d'abord si la valeur de la variable lock est égale à 1 avant de d'appeler "xchg" alors que TAS lui appelle constamment "xchg" afin de vérifier si la valeur de la variable lock a changé ou non.

Concernant les valeurs obtenues sur le serveur Inginious, nous ne comprenont pas pourquoi nos implémentations ont l'air plus rapides que celle de POSIX au long terme. Normalement plus on ajoute des threads, plus le temps avant de rentrer en section critique augmente, ce qui devraient formetment allonger le temps d'execution de nos performances avec nos locks et n'impacter d'aucune manière nos performances avec la librairie POSIX.

TATAS a l'air plus rapide que TAS la plupart du temps mais pas toujours, ce qui nous perturbe également. Nous pensons que cela puisse être du que nous faisons que 5 exécutions de tests et nos temps diffèrent trop d'une exécution à l'autre expliquant nos écarts types assez conséquents. Nous soupçonnons également un problème dans le script.

3 Algorithmes

3.1 Philosophers

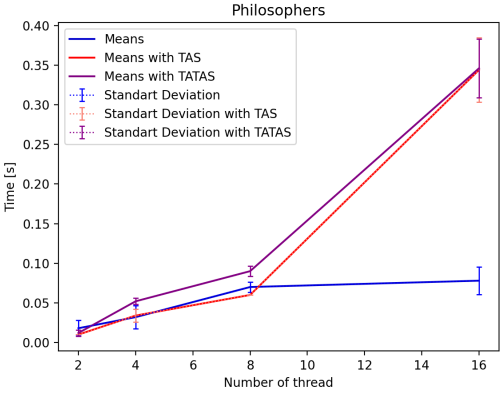

FIGURE 3 – Local performance

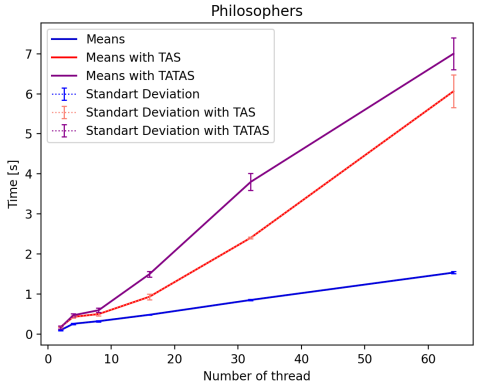

FIGURE 4 – Inginious performance

Dans ces graphes, nous comparons la performance de l'algorithme "Philosophers" avec les mutex de la librairie POSIX et avec nos implémentations de TAS et TATAS. Nous avons effectué le test avec 10 000 cycles dormir-penser.

Nous pouvons observer que la croissance est quasi-linéaire. Cela est expliqué par le fait qu'à chaque fois que nous rajoutons un thread, nous rajoutons une charge de travail d'un cycle. A chaque thread supplémentaire, on ajoute un mutex (baguette). Vu que chaque baguette est accedée uniquement par le philosophe à gauche ou à droite, les temps d'attentes augmentent considérablement à chaque nouveau thread. Nos implémentations de lock ne sont pas dotés du principe de "fairness" dont sont doté les mutex POSIX. Ces derniers choisissent le thread à exécuter en fonction de son temps d'attente et de sa priorité ("the highest priority thread that has been waiting the longest shall be unblocked"). Dans le cas de nos implémentations, il pourrait arriver qu'un thread ait un temps d'attente particulièrement long car il ne parvient pas à avoir le lock mais dans le cas des POSIX, le temps d'attente n'atteint pas la performance. Cela explique donc le fait que la courbe des mutex POSIX se stabilise alors que celle de nos implémentations sont croissantes.

Nous ne savons pas expliquer avec certitude pourquoi notre philosopher avec TATAS est plus lent que celle avec TAS. Cela est peut être du au fait que la vérification supplémentaire avec TATAS est inutile dans certains cas et que ce serait plus rapide de directement changer la valeur du lock.

3.2 Producer-Consumer

Dans ces graphes, nous comparons la performance de l'algorithme "Producer-Consumer" avec les mutex de la librairie POSIX et avec nos implémentations de TAS et TATAS. Nous

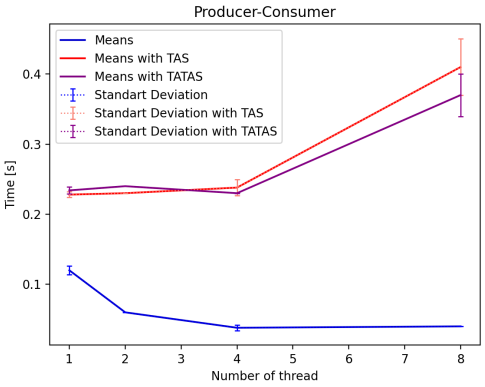

FIGURE 5 – Local performance

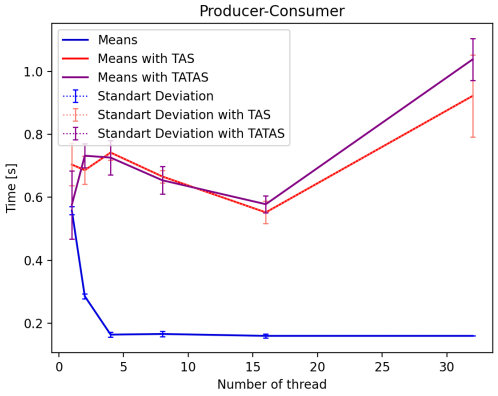

FIGURE 6 – Inginious performance

avons effectué des tests avec 8192 éléments produits et donc aussi consommés, pour ce faire nous avons divisé le nombre d'éléments par le nombre de threads présents afin d'avoir un certain nombre d'éléments par thread (nous avons fait un modulo pour le dernier thread afin de toujours tomber sur le compte juste). Sur les graphes, le nombre de threads de l'axe Y correspond au nombre de producer et consumer (1 correspond à 1 producer et 1 consumer donc 2 threads au total).

Nous remarquons, dans les performances en local, que TAS est plus efficace pour un petit nombre de threads mais qu'à partir d'un certain seuil (3 producers et 3 consumers), après c'est TATAS qui est plus efficace. Ce n'est cependant pas le cas pour les performances sur inginious qui ne suivent pas du tout la meme courbe, nous avons du mal à expliquer cela.

Les mutex de la librairie POSIX sont tout le temps plus efficace car en effet les mutex et sémaphores de POSIX sont bien plus efficaces lorsque le temps d'attente est long avant de rentrer en section critique, uniquement le blocage et déblocage impactent leurs temps d'exécution. Les algorithmes avec TAS et TATAS croient (en local) car lorsque on ajoute des thread, le temps d'attente moyen augmente du au fait que plus de threads vont attendre, et dans nos implémentations cela un effet direct sur notre temps d'execution. Par contre, les mutex/sémaphores POSIX ne sont absolument pas impactés par le temps d'attente avant la section critique, leur courbe se stabilise donc à partir de 4 threads. En local, notre algorithme avec TATAS est plus efficace comme attendu, ce n'est cependant pas le cas sur le serveur inginious.

3.3 Reader-Writer

Dans ce graphe, nous comparons la performance de l'algorithme "Reader-Writer" avec les mutex de la librairie POSIX et avec nos implémentations de TAS et TATAS. Sur les graphes, le nombre de threads de l'axe Y correspond au nombre de readers et writers (suivant la meme logique que producer-consumer). Nous avons effectué ce test avec 2560 readers et 640 writers, nous avons utilisé des variables globales pour tenir compte du nombre de writers et readers, elles-mêmes protégée par des locks. Nous pouvons observer qu'ici, cest notre implémentation qui est plus efficace, cela peut etre expliqué par le fait qu'il y a beaucoup de mutex à bloquer et débloquer et que le temps entre 2 accès à un lock est petit. Les temps de blocage et débloquage de nos locks sont très petits (il suffit

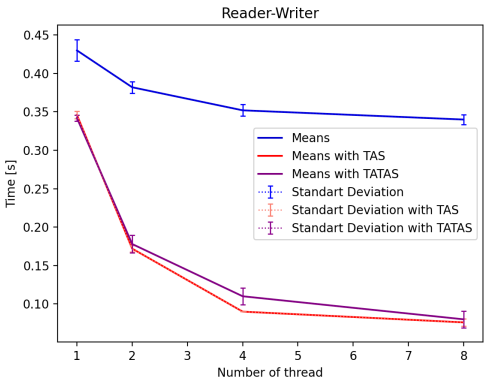

FIGURE 7 – Local performance

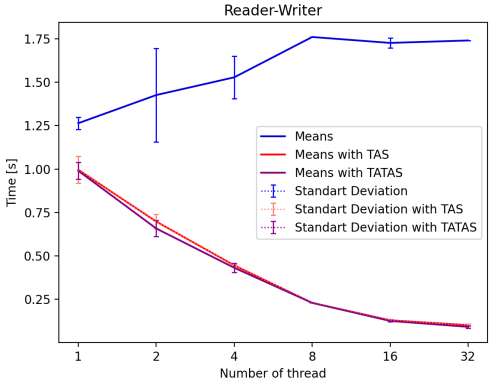

FIGURE 8 – Inginious performance

de modifier une variable). Le mutex de POSIX quant à lui prend plus de temps pour cela car il doit faire un syscall qui est bien plus conséquent. Pourtant lorsque le nombre de thread augmente, POSIX devraient prendre le dessus ce qui n'est pas le cas. Etant donné que, lorsque le writer écrit, les readers sont bloqués, le temps d'attente moyen (dans la boucle while) augmente ce qui influence le temps d'execution de nos locks. Nous sommes conscients qu'il y a ici un problème mais nous n'avons pas reussir à le résoudre. Nous ne savons cependant toujours pas expliquer pourquoi, en local, notre reader-writer avec TATAS est plus lent que celle avec TAS, nos hypothèses restent les mêmes. Sur le serveur Inginious, l'algorithme avec TATAS semble légèrement plus performant.

4 Conclusion

Pour conclure, nous avons obtenus des résultats concluants et d'autres moins concluants, cela doit être du à un problème d'implémentation, nous soupçonnons un problème dans le script, mais nous avons malheureusement pas eu le temps de régler cela. Nos résultats en local et sur inginious ne correspondent pas toujours car d'une exécution à une autre pleins de facteurs influencent le temps (CPU, les autres processus en cours d'exécution, la mémoire).

Nous voulions faire un résumé des principales différences entre les mutex/sémaphores de POSIX et nos implémentations test-and-set et test-and-test-and-set.

Premièrement, les mutex/sémaphores de POSIX fonctionnent avec un scheduler qui garantit la fairness. Concrètement, ils bloquent un thread par un appel à syscall qui les "endort" et laisse au scheduler au le noyau l'ordre de priorité (fairness). Alors que nos implémentations, le thread est bloqué dans une boucle while tant que le lock n'est pas remis à 0.

Deuxièmement, le temps ou le CPU est occupé est différent pour le blocage et déblocage. Les mutex/sémaphores de POSIX prennent plus de temps pour bloquer et débloquer car ils doivent effectuer un appel syscall qui est bien plus conséquent que la simple modification de la valeur de la variable lock.

Troisièmement, le temps ou le CPU est occupé est différent pour l'attente. S'il y a un long temps d'attente avant d'accéder à la section critique, les mutex/sémaphores de POSIX ne sont absolument pas impactés du point de vue de la performance vu que les threads sont mis en "pause" (sleep). Par contre, quant à l'attente dans nos implémenta-

tions, celle-ci impacte directement le temps d'exécution car lorsque un thread attend, le CPU sera occupé à faire tourner la boucle while.

Pour résumer, lorsque nous travaillons avec de nombreux threads et que notre section critique est bien remplie, les mutex/sémaphores de POSIX sont une solution plus optimisée. Par contre si vous cherchez à avoir des échanges rapides et que les temps d'attentes avant la section critique sont petits, les algorithme test-and-set et test-and-test-and-set sont une bonne solution.