UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Projet 1 : Programmation multi-threadée et évaluation de performances

LINFO1252

Projet réalisé par
Jonathan DE SALLE , Philippine DE SURAŸ

Projet réalisé dans le cadre du cours de
Systèmes informatiques

Professeur(s)
Étienne RIVIÈRE

Année académique 2020-2021

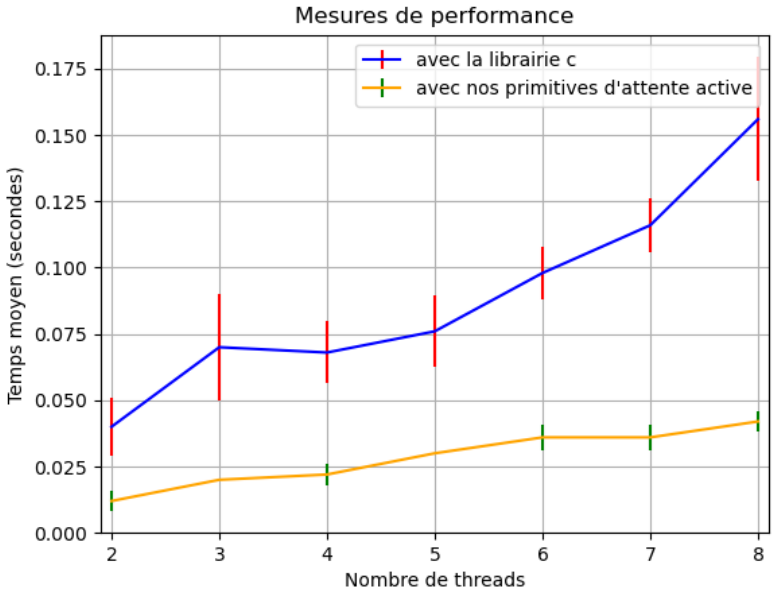
0.1 Résultats des évaluations

0.1.1 philosophe

Il est à noter que nous avons dû multiplier par 10 le nombre d'opérations pour réaliser ce graphe, le temps d'exécution étant trop court lorsque l'on faisait uniquement 10000 opérations, nous sommes donc passé a 100 000 afin de pouvoir mieux observer les différences sur le graphe.

Avant de faire tourner le programme philosophe, nous nous attendions à voir le temps d'exécution diminuer si l'on augmente le nombre de thread. Nous avons donc été surpris par l'allure du graphe car le temps augmente avec le nombre de threads. Nous supposons que le temps allouer et déverrouiller les locks est supérieur au temps des opérations, ils nous est donc plus coûteux de paralléliser celles-ci.

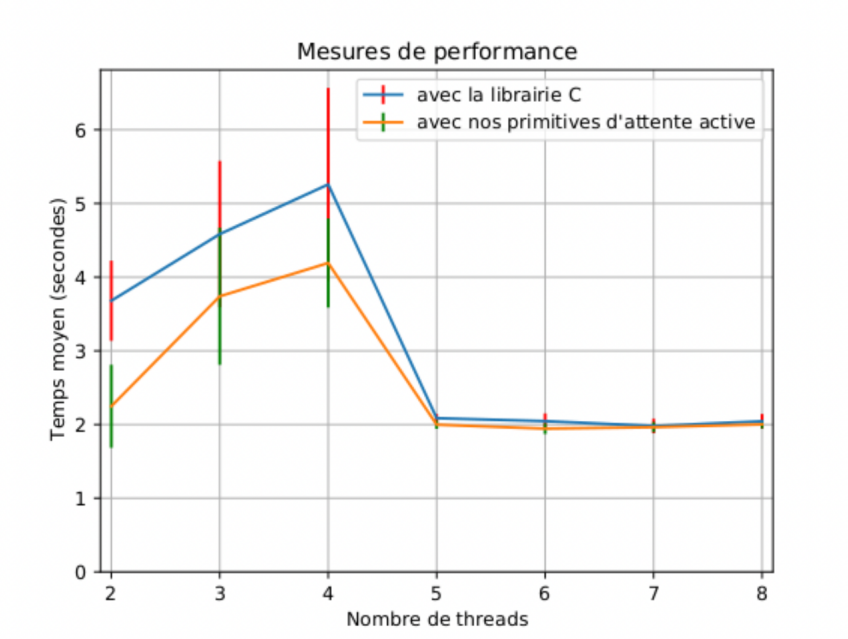
L'utilisation de nos spinlocks, à priori moins coûteux en temps pour des opérations courtes, semble effectivement réduire le temps total d'exécution de manière presque exponentielle.



0.1.2 reader/writer

On peut constater une augmentation de temps au fur et à mesure de l'augmentation des threads. Au début, cela nous a beaucoup surpris, nous pensions en effet que l'ajout de threads devrait accélérer le processus. Nous supposons que cela vient de l'utilisation des différent locks qui ne compensent pas forcément le gain de temps engendré par la parallélisation. Le temps d'éxécution diminue pourtant à partir de 5 threads et reste stable à partir de là.

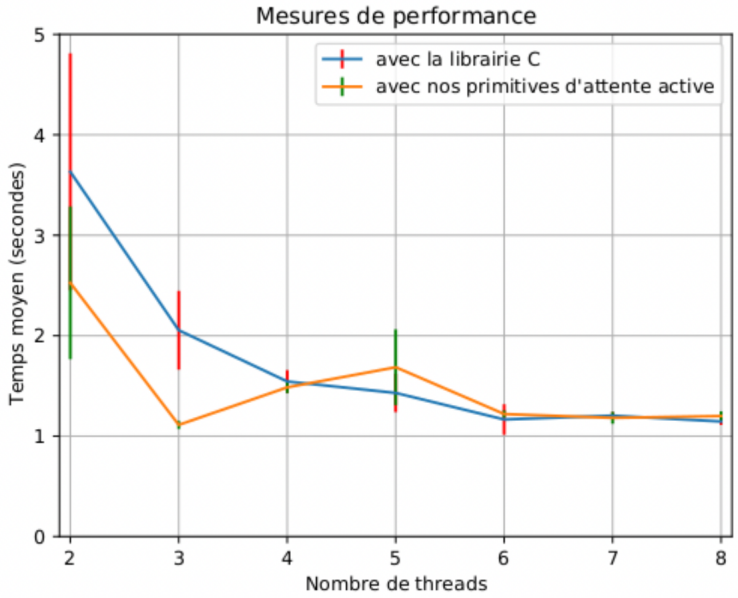
En comparant le graphe de la version POSIX de notre implémentation à l'implémentation employant nos spinlocks et notre version du sémaphore, nous constatons que la vitesse est plus élevée (jusqu'à 2 fois plus rapide dans le cas d'un seul thread d'exécution), ce qui correspond a ce que nous avions vu au cours : pour des opérations courtes en sections critique, le spinlock tend a être plus rapide.



0.1.3 producer/consumer

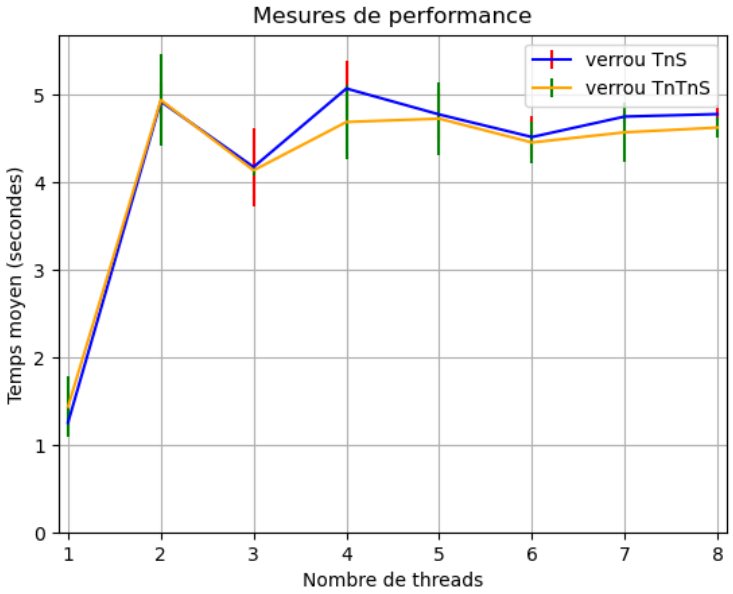
Dans le graphe du producteur/consommateur avec la libraire POSIX pour les primitives d'attente active, on peut constater une diminution du temps d'exécution lorsque le nombre de threads augmente. En effet, augmenter le nombre de threads permet de réduire le nombre de tours que chaque producteur/consommateur doit effectuer. On peut aussi constater que ce temps se stabilise après 4 threads car ceci est la limite de coeurs disponibles pour la machine et donc il n'est pas possible de faire plus d'opérations en parallèle que 4.

Concernant le graphe avec nos propres primitives d'attente active, on constate que la courbe est moins stable que précédemment mais le temps d'exécution pour 2 à 4 threads est légèrement inférieur. Notre courbe se stabilise à la même valeur que précédemment pour 6 à 8 threads, il y a par contre une augmentation pour 5 threads. On peut donc dire que dans l'ensemble, on a un comportement relativement similaire à celui avec la librairie C.



0.1.4 verrous

Au-delà de 1 thread, le temps d'exécution de nos verrous semble plutôt stable quel que soit le nombre de threads, avec des performances pratiquement similaires entre nos verrous test-and-set(TnS) et nos verrous test-and-test-and-set (TnTnS), avec un léger avantage pour les verrous implémentant l'algorithme TnTnS.



0.2 Conclusion

Nous avons été étonnés de voir que les graphes ne ressemblaient pas toujours à l'image que nous avions dans notre tête avant de lancer nos différents scripts.

Ce projet nous aura néanmoins permis de bien mieux comprendre l'utilisation des sémaphores et des mutexs qu'à la seule lecture du syllabus/cours théorique. Beaucoup de nos premières erreurs venaient d'une mauvaise utilisation ou d'une mauvaise syntaxe de ceux-ci, l'utilisation du manuel, ainsi que des exemples du syllabus nous a beaucoup aidé.

Nous avons également pu découvrir l'assembleur in-line et ses différents paramètres que nous n'avions jamais utilisé auparavant. Après pas mal de difficultés, principalement liées au fait de vouloir trop faire "de l'assembleur directement en c", nous sommes parvenus à implémenter les différents locks, ainsi que le sémaphore, nous permettant de mieux comprendre les différents algorithmes pouvant être utilisés dans l'implémentation d'un spinlock.

Enfin, nous avons pu améliorer nos connaissances quand à la création du script bash et du makefile, 2 outils que nous avions tout 2 déjà utilisé auparavant.