Ecole polytechnique de Louvain

LINFO1252 - Système informatiques

Project 1 - Multi-thread

Academic year 2022-2023

Submission date : November 7^{th} 2022

Group: 43

Laurent Deleu (6040-13-00)

Maxime Devillet (0866-19-00)

Professor and assistants:

Etienne Rivière

François de Keersmacker

Maxime Piraux

Tom Rousseaux

Nikita Tyunyayev

UCLouvain

Introduction

Dans ce rapport, nous allons vous présenter notre projet réalisé dans le cadre du cours LINFO1252. Nous commencerons d'abord par les différents problèmes multi-threads que nous avons codés ainsi que les résultats des tests de performance pour différents types de primitive. Dans un second-temps, nous comparerons directement nos deux algorithmes de primitives d'attente active. Ce rapport et ce code sont fortement inspiré par le travail réalisé par Maxime Devillet et Pierre Accou durant l'année 2021-2021. Etant moi-même le réalisateur de ce travail, je me suis permis de réutiliser et de m'insiprer de mon travail.

Note préalable : La métrique principale est ici le temps d'exécution des différents codes pour des nombre de threads différents. Par souci de lisibilité et de compacité, nous avons décidé de présenter les graphes de la façon suivante. Cependant, le nombre de thread/coeurs étant une variables discrète, nous ne devrions normalement pas tracer de ligne continu entre les mesures (boxplot uniquement).

1 Le problème des philosophes

Les résultats suivants sont issus de mesures faites sur les machines de la salle Intel. En effet nous avons reçu des erreurs "time out" en voulant l'exécuter sur INGInious. Les dernières valeurs (sur 64 threads) sont donc à prendre avec méfiance.

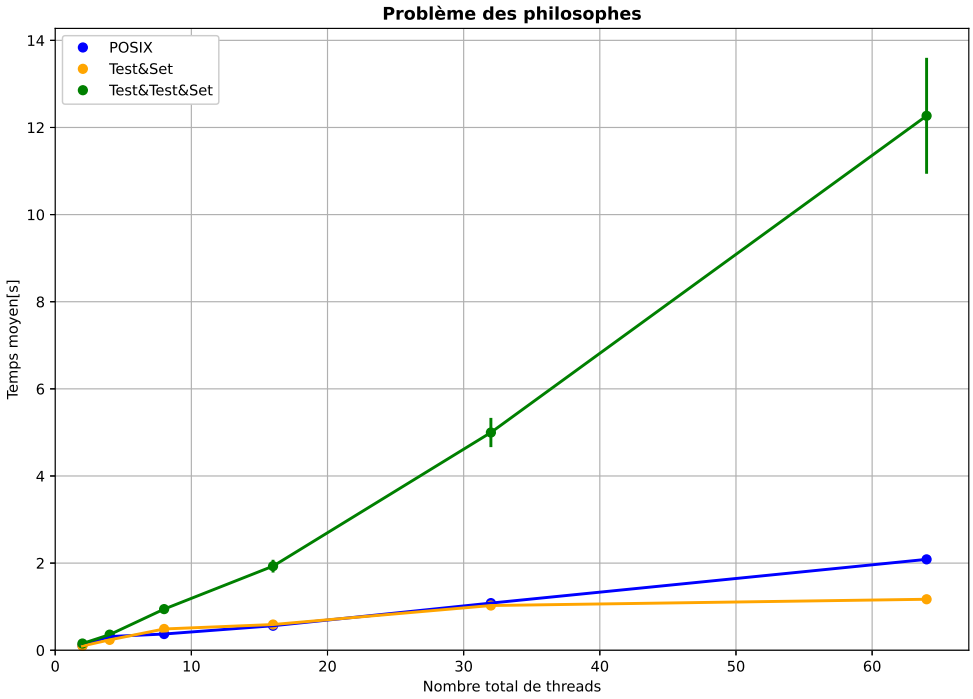

Figure 1: Comparaison de 3 types de primitives sur l'algorithme multi-threads des philosophes depuis ordinateur de salle Intel (16 coeurs)

Posix: Quand on exécute le programme destiné à résoudre le problème du philosophe, on peut se rendre le temps d'exécution augmente quasi linéairement avec le nombre de threads. Cela peut s'expliquer par le fait que le nombre de philosophes (threads) en attente pour que des baguettes se libèrent, augmente, tout autant que le nombre de tests correspondant. On peut donc en conclure que pour les problèmes à attente active, le nombre de threads n'est pas un avantage.

Test and set: Test and set, est quand à lui plus lent que posix pour un nombre de threads faible mais plus celui-ci augmente, plus ctest and set devient efficace.

Test and test and set: Ici le graph nous montre que test an test and set à tendance à suivre posix en diminuant au fil du temps mais une fois les 32 threads atteints, on constate une augemetation du temps d'exécution

2 Le problème des producteurs-consommateurs

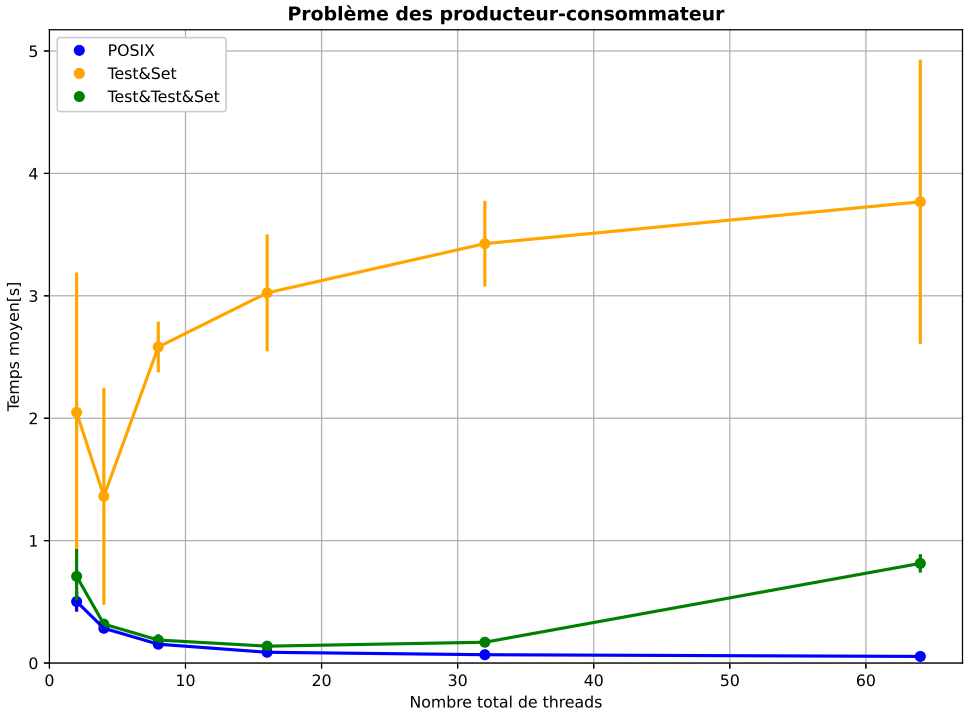

Figure 2: Comparaison de 3 types de primitives sur l'algorithme multi-threads des producteurs-consommateurs depuis INGInious (machine 32 coeurs)

Posix On peut remarquer ici que plus le nombre de threads augementent, plus le temps d'exécution diminue. Celui-ci diminue de moins en moins vite avec un ecart-type qui lui aussi diminue avec l'augementation du nombre de threads. Etant donné que nos résutats proviennent d'une machine

virtuelle 32 coeurs/64 threads nous ne pouvons pas observer ce qui se passe une fois le nombre de threads maximum atteint.

test and set Test and set lui, ne tend pas à diminuer mais augemente au fur et à mesure que le nombre de threads augementent, ce qui pourrait etre expliqué par les appels fréquents à xchgl infructeux qui augemente avec le nombre de threads exécutés simultanément

test and test and set

3 Le problème des lecteurs et écrivains

Nous n'avons ici pu effectuer que des mesures sur notre implémentation POSIX et Test-and-Test-and-Set, Test-and-Set ne fonctionnant pas pour des raisons inconnues.

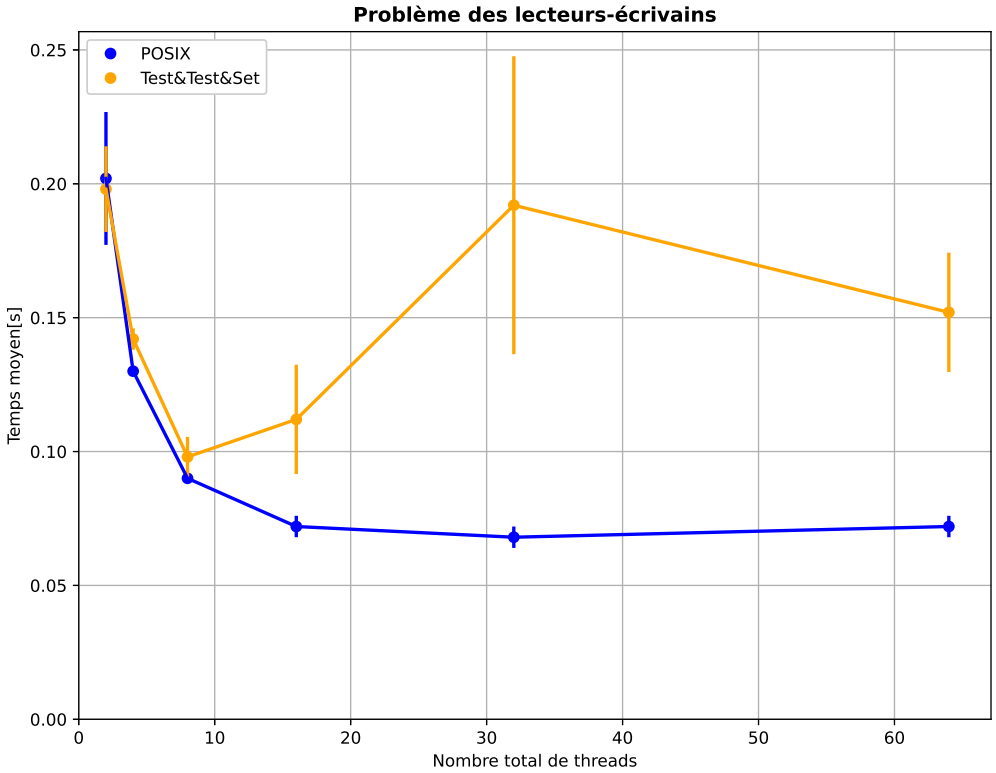

Figure 3: Comparaison de 3 types de primitives sur l'algorithme multi-threads des lecteurs-écrivain depuis INGInious (machine 32 coeurs)

Posix: Ici le temps d'exécution décroît fortement avec l'augmentation des threads avant de stabiliser (forte ressemblance avec la loi d'Amdahl) proche d'une valeur asymptotique. En effet, il n'y a ici presque pas de contrainte sur le nombre de lecteurs (ou écrivains) ce qui laisse le champs libre à des

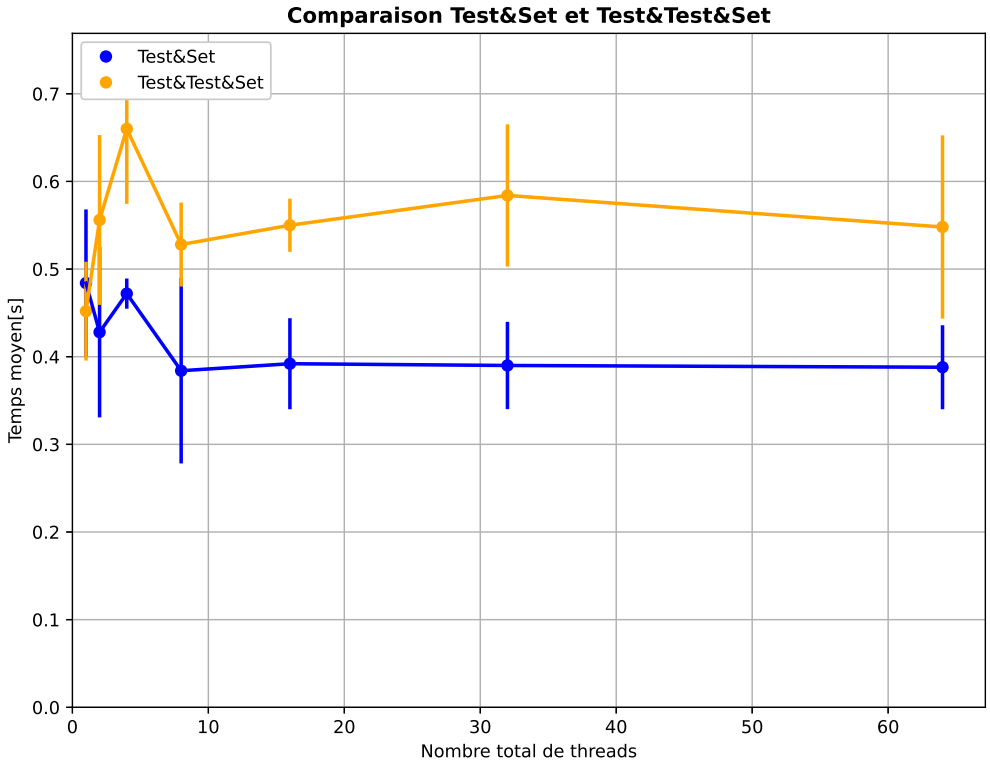

Figure 4: Comparaison de 2 types de primitives d'attente actives un algorithme multi-threads de base depuis INGInious (machine 32 coeurs)

processus avec peu d'interruptions et donc de meilleures performance. L'augmentation du nombre de threads prend quand même le pas à au final comme décrit dans la loi citée plus haut.

test and test and set: La courbe semble suivre la même tendance dans un premiers temps mais semble avoir un maximum local dans les alentours de 32 threads. Dans tout les cas, l'implémentation POSIX reste plus performante.

4 Test&Set et Test&Test&Set

ici on peut remarquer que Test and set est globalement moins couteux que test and test and set. Ce qui nous semble peu probable, en effet, la grosse différence de coût entre les deux fonctions se distingue par la fonction atomique xchgl qui à répétition entraine une dégradation des performance. La fonction test and test and set permet donc de tester si il est nécessaire d'utiliser cette opération et attendra jusqu'a ce que la ressource à utiliser soit disponible, et donc à terme, faire gagner du temps.

5 Conclusion

Pour conclure, nous pouvons dire que l'on peut coder un problème selon plusieurs stratégies. Ses stratégies ont pour chacune leurs avantages et leurs inconvénients. Dans notre cas, l'implémentation de notre interface de sémaphore ne fut pas réellement un problème mais lorsque nous comparons nos résultats, nous en avons des biens différents entre les 2 stratégies. Pour certains algorithmes, il est plus simple d'utiliser le package semaphore.h afin de résoudre notre problème alors que dans d'autres cas, notre interface de sémaphore rend la performance bien meilleure. On a aussi pu constater que entre test and set et test and test and set il y avait une différence de stratégie. Test and set est plus gégé dans son opération car ne demande pas d'attendre avant de tenter d'effectuer un xchgl ce qui le rend très efficace avec peu de threads lorsque les ressources sont accessible et libre, tandis que test and test and set demande d'attendre et de vérifier que la ressource soit libre avant d'effectuer xchgl ce qui lui fait perdre du temps lorsque celui-ci est souvent libre mais le rend particulièrement efficace avec beaucoup de threads lorsqu'il faut attendre longtemps avant de pouvoir utiliser la ressource.