UCLouvain

École polytechnique de Louvain

LINFO1252 – Systèmes Informatiques – Rapport

Projet 1 : Programmation multi-threadée et évaluation de performances

Professeur : RIVIÈRE Etienne

PAQUET Robin - 02192001
LIESENS César - 39161400

Décembre 2021

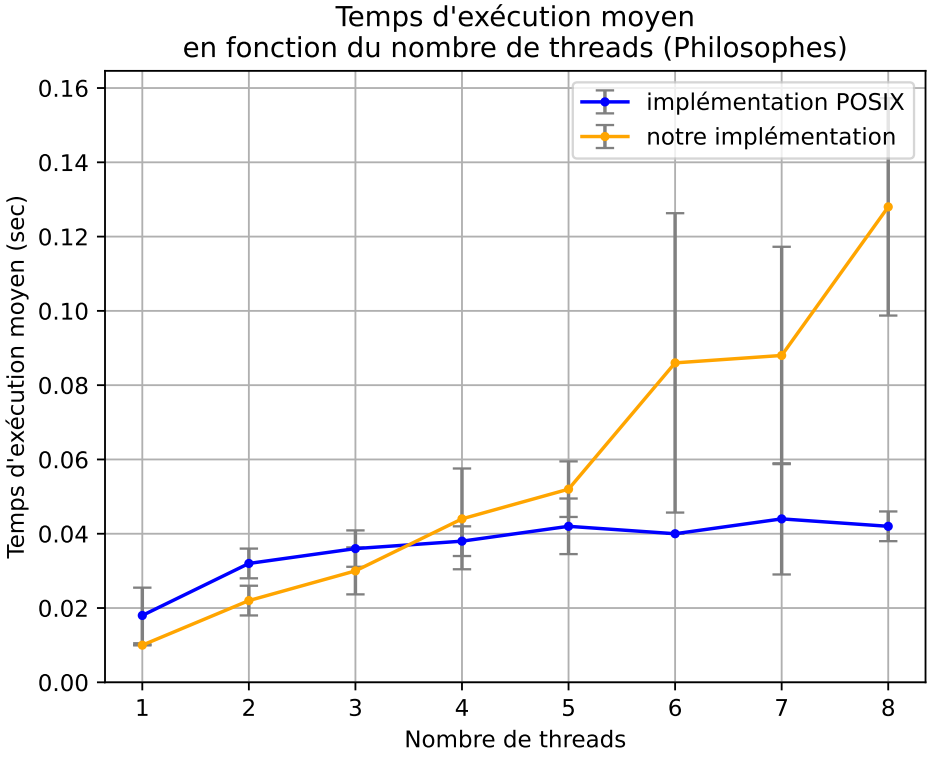
1 Introduction

Ce rapport a pour but d'évaluer et comparer les performances de différentes implémentations d'algorithmes de synchronisation de threads sur des problèmes typiques de ce domaine, et qui ont été étudiés lors du cours. Nous avons effectué nos mesures sur des machines virtuelles Ubuntu v21.10, configurées avec 4 coeurs et 8 GO de mémoire vive; celles-ci sont répétées cinq fois pour chaque configuration de nombre de threads, comme demandé.

Nous avons structuré le projet en plusieurs dossiers de manière à séparer le code, les données des prises de mesure ainsi que les graphiques produits pour chacune des tâches; à la racine du projet se trouve également un Makefile complet et fonctionnel, permettant de lancer le projet en une fois ou bien de réaliser chacune des tâches séparément (compilation, création des graphiques, mesures...).

2 Problème des Philosophes

Le programme implémentant le problème des philosophes prend un seul argument, qui correspond au nombre de philosophes (qui est donc aussi le nombre de threads). Chaque philosophe effectue 100.000 cycles *manger*/*penser* et il n'y a pas d'attente pour les phases *manger* et *penser*.

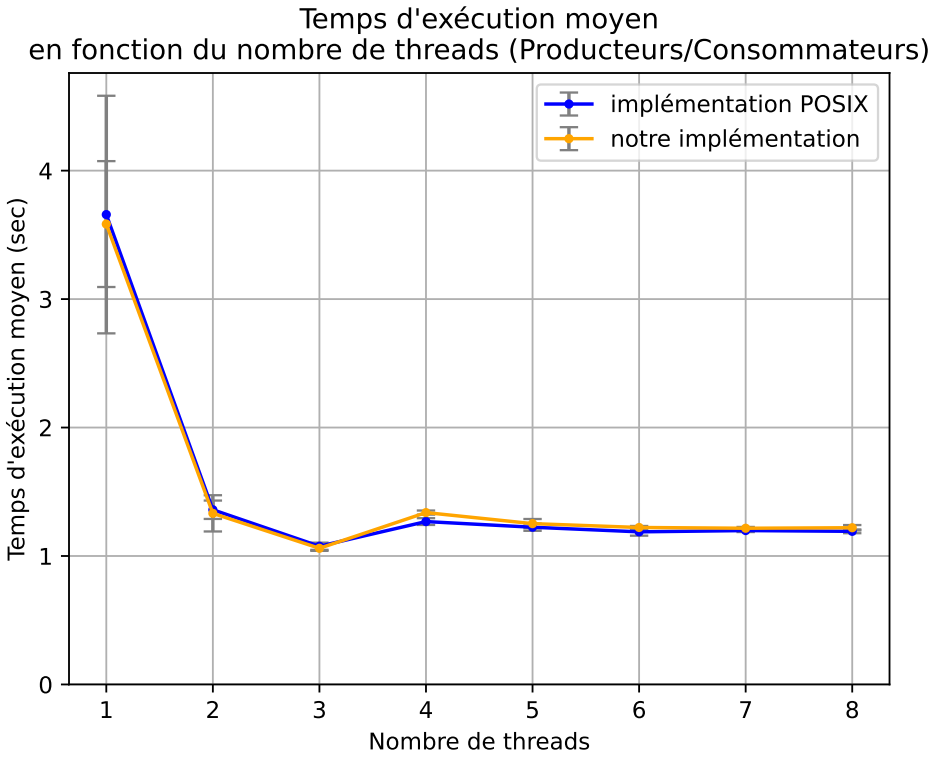


En regardant le graphique ci-dessus, on remarque que le temps moyen d'exécution des deux implémentations est assez proche jusqu'à quatre threads; l'implémentation POSIX semble suivre une très faible croissance voire rester quasi-constante, tandis que notre implémentation subit une montée très rapide à partir de quatre philosophes. Il est difficile de dire si celle-ci est linéaire, quadratique ou même exponentielle sans avoir plus de mesures – d'autant plus que les mesures de notre implémentation ont une variance fort élevée –, mais il est clair que le temps d'exécution croît en fonction du nombre de threads exécutés, ce qui pourrait s'expliquer par le fait que le nombre de cycles à effectuer par thread reste le même peu importe le nombre de philosophes, le temps total ne pouvant qu'augmenter. Répéter l'expérience sur une machine avec plus de coeurs pourrait permettre de déterminer si l'implémentation POSIX elle prend effectivement un temps constant peu importe le

nombre de threads (ce qui semble improbable), mais ne ferait que confirmer la différence d'efficacité avec notre implémentation.

3 Problème des Producteurs-Consommateurs

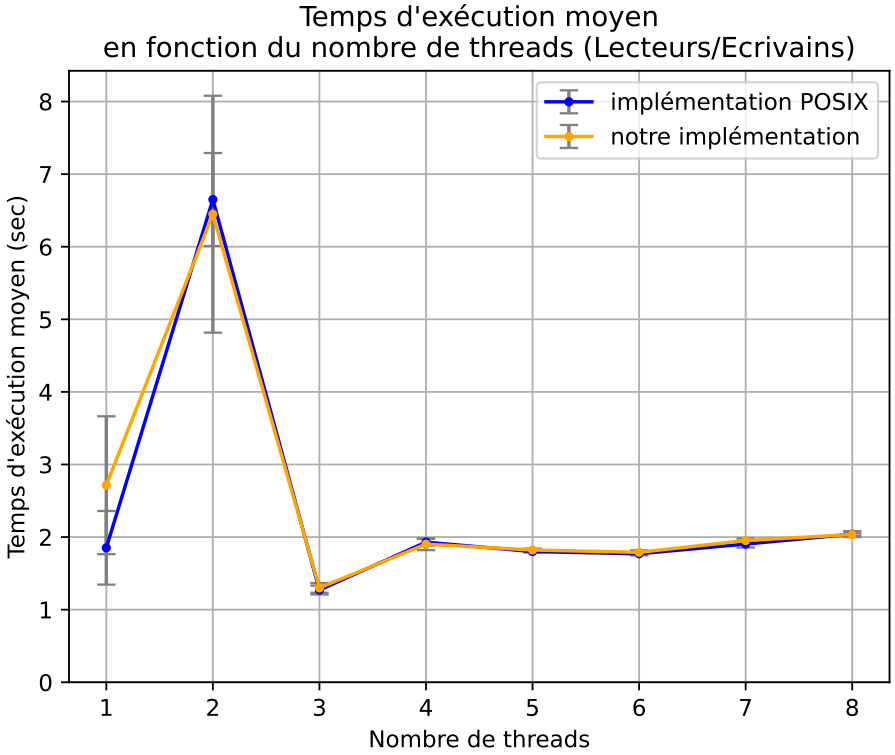
Pour l'implémentation du problème des producteurs-consommateurs, le nombre de consommateurs et de producteurs sont chacun un paramètre à donner au lancement du programme. Le nombre d'éléments produits et consommés est limité à 1024, comme demandé, et le buffer ne peut contenir que huit éléments à tout instant.



On observe tout de suite que les deux implémentations sont très similaires en terme de performance, et il est en effet difficile de les départager. Les deux commencent par un pic fortement marqué à un thread, puis diminuent fortement jusqu'à trois, et semblent se stabiliser après quatre threads. Pour ce qui est de la convergence du temps moyen d'exécution lorsque le nombre de threads augmente, cela pourrait s'expliquer par le fait que la majorité du temps passé à "calculer" (autrement dit, la fonction `work()`) est fait hors de la section critique des threads (qui est donc très courte, et ne contient qu'un appel à `rand()` et une incrémentation), entre chaque production ou consommation; on voit donc que la section critique prend de moins en moins de temps par rapport au reste de l'exécution du programme.

4 Problème des Lecteurs et Écrivains

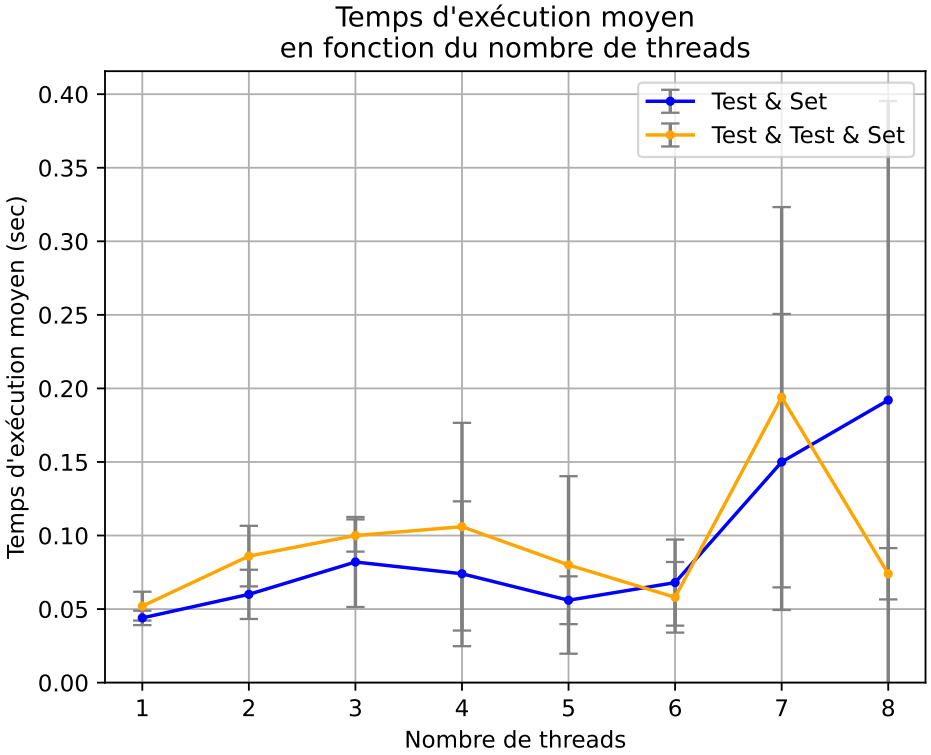
Concernant le problème des lecteurs/écrivains, comme pour celui des producteurs-consommateurs, les deux paramètres pour le nombre sont à donner lors de l'exécution du programme. Chaque écrivain effectue 640 écritures et chaque lecteur effectue 2560 lectures.



En observant le graphique, on remarque à nouveau directement une allure très similaire pour les deux courbes, ainsi qu'un pic à deux threads, aussi bien pour notre propre implémentation que pour celle de POSIX. Après avoir dépassé deux lecteurs & écrivains, on observe ensuite une diminution significative du temps d'exécution à trois lecteurs/écrivains, qui remonte ensuite légèrement puis semble rester constant ou faiblement croissant; cela pourrait s'expliquer par le fait que, de manière similaire au problème des producteurs/consommateurs, le nombre total d'actions à effectuer ne change pas quelque soit le nombre de threads, contrairement au problème des philosophes où rajouter des threads ne diminue pas la charge de chaque thread. Les deux courbes restent très proches tout au long du graphique, il est donc également difficile de départager la meilleure implémentation.

5 Verrous Test-and-Set & Test-and-Test-and-Set

Deux implémentations de verrous par attente active sont montrées et comparées dans cette section. Ces implémentations utilisent de l'assembleur *inline*, ainsi que l'opération `xchg`, qui permet d'échanger le contenu de ses deux opérandes de manière atomique.



Le fait que notre verrou Test and Set (TAS) soit légèrement plus performant que le verrou Test and Test and Set (TATAS) semblerait indiquer que notre implémentation du spinlock TATAS est erronée, puisque l'on s'attendrait évidemment à voir l'inverse. Les mesures du verrou TAS semblent également avoir un écart type très élevé, et il aurait probablement été préférable de faire plus de cinq mesures pour celui-ci afin de pouvoir faire une meilleure comparaison (mais cela est sans doute valable pour l'ensemble des programme et mesures faites pour ce projet). La montée croissante du temps d'exécution du verrou TAS est attendue, cependant les chutes à six et huit threads de celui-ci nous paraissent difficiles à expliquer; peut-être notre implémentation est-elle inexacte (mais elle semble fonctionner comme attendu, et est tirée directement du syllabus), ou les cinq mesures que nous avons prises étaient anormalement basses, ou encore peut-être que les ressources de notre machine étaient très peu utilisées à ce moment exact. Pour ce qui est du verrou TATAS, on remarque qu'il suit la même allure que l'implémentation TAS, ce qui semblerait confirmer que notre implémentation n'est hélas en réalité qu'une version moins efficace du verrou TAS.

6 Conclusion

Globalement, l'on constate que nos implémentations sont bien moins efficaces que leurs équivalents POSIX, ce qui n'est pas surprenant étant donné que celles-ci ont probablement été créées par des experts du domaine (ce que nous ne sommes hélas pas encore). Ce projet nous aura demandé de mettre en pratique les diverses connaissances et compétences acquises lors des cours magistraux ainsi que des travaux pratiques, afin d'implémenter diverses primitives de synchronisation telles que les sémaphores ou les verrous actifs. Cela nous a permis non seulement de consolider notre compréhension de la partie théorique du cours, mais aussi de renforcer nos compétences en C. Ce projet nous aura été finalement profitable et utile car il avait pour sujet des concepts complexes mais intéressants, malgré le temps et les efforts considérables qu'il aura requis.