UCLouvain

École Polytechnique de Louvain

# Rapport LINFO1253 - Projet 1

## Programmation multi-threadée et évaluation de performances


**Etudiants:**

Groupe 13

Gustin Théo - 42052000
Adrien Notte - 32832000

**Titulaire:**

Etienne Riviere


Année Académique 2022-2023

7 décembre 2022

# 1 Introduction

Ce projet a pour but de nous familiariser avec le concept de programmation multi-threadée et évaluation de performances. Pour ce faire nous devions implémenter 3 algorithmes multi-threadés et évaluer leurs performances sur une machine de 32 cœurs. Nous avons implémenté : *Dining Philosophers*, *WriterReader* et *ProducerConsumer*. Pour cela, nous avons, dans un premier temps, utilisé des mutex et sémaphores POSIX standards. Après cela, nous avons créé nos propres verrous à attente active avec les algorithmes *test-and-set* et *test-test-and-set* en utilisant de l'*inline assembly*. Nous avons, par la suite, utilisé ces verrous nouvellement implémentés pour créer nos propres sémaphores. Pour finir, nous avons créé de nouvelles versions de nos 3 premiers algorithmes en utilisant nos propres verrous et sémaphores.

# 2 Analyse de performances

## 2.1 *Dining Philosophers*

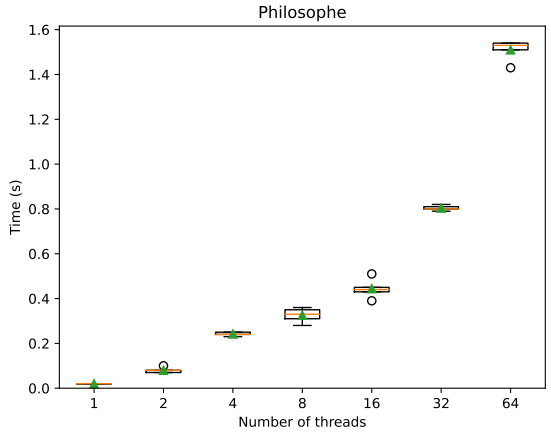



**Figure 1** – *Dining Philosophers* with POSIX

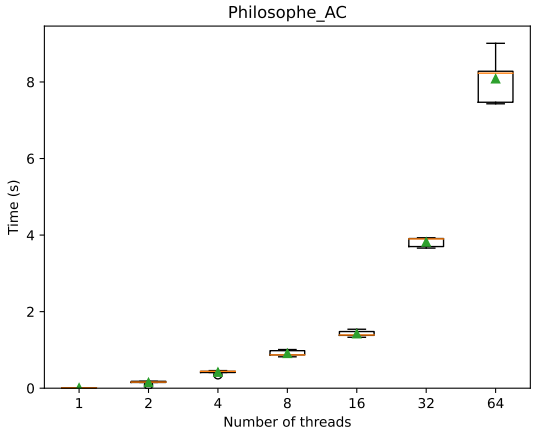



**Figure 2** – *Dining Philosophers* with spinlock

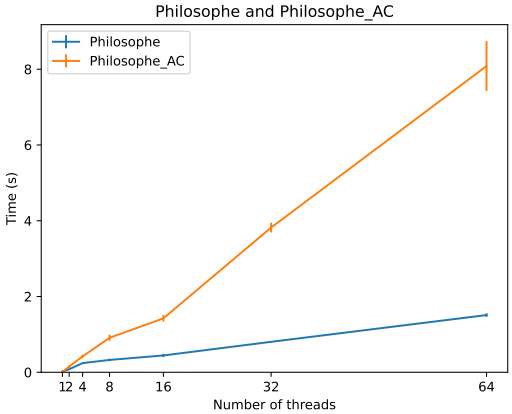



**Figure 3** – Comparaison Philosophe et Philosophe_AC

Ces figures représentent la moyenne, la médiane et la déviation standard du temps d'exécution par rapport aux nombres de threads impliqués de l'algorithme *Dining Philosophers* avec POSIX (Philosophe) et avec les verrous à attente actifs que nous avons implémenté (Philosophe_AC) .
Lorsqu'on compare les figures 1 et 2, on voit que l'implémentation avec POSIX et avec les spinlock ont un comportement semblable lorsqu'on augmente le nombre de threads. En effet, on voit que, comme attendu, le temps d'exécution augmente avec le nombre de threads. Cependant, si on regarde l'échelle temporelle, on voit que la version à attente active explose beaucoup plus vite.

Bien que l'on utilise des verrous à attente active utilisant l'algorithme *test-and-test-and-set*, ils induisent quand même une augmentation du temps d'exécution.

## 2.2 *WriterReader*

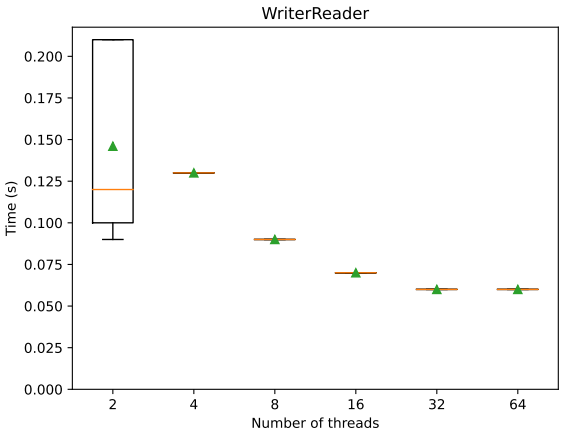



**Figure 4** – *WriterReader* with POSIX

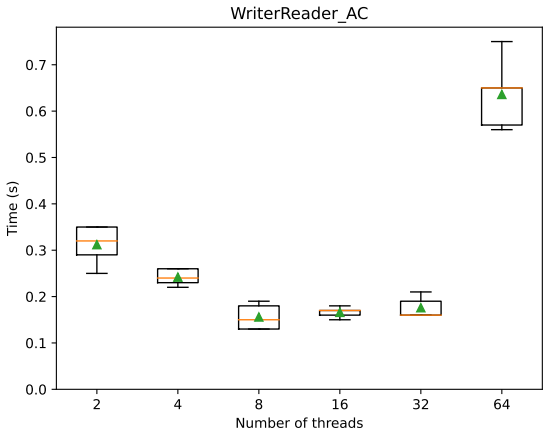



**Figure 5** – *WriterReader* with spinlock

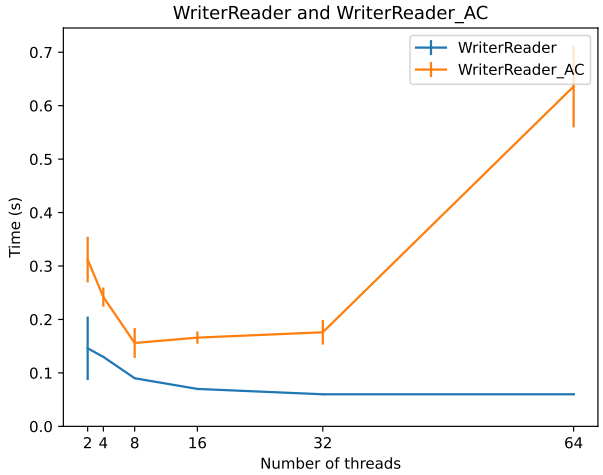



**Figure 6** – Comparaison WriterReader et WriterReader_AC

Ces figures représentent la moyenne, la médiane et la déviation standard du temps d'exécution par rapport aux nombres de threads impliqués de l'algorithme *WriterReader* avec POSIX (WriterReader) et avec les verrou à attente actifs que nous avons implémenté (WriterReader_AC).
On observe sur la figure 4, qu'avec POSIX nous avons une variation des temps entre les tests avec 2 threads. Cette variation survient avec 64 threads pour la figure 5. En comparant les figures 4 et 5 et avec la figure 6, on observe que le temps d'exécution de l'implémentation spinlock est environ deux fois plus grand que celui de celle en POSIX. Seules les mesures avec 64 threads faites avec la version spinlock dérogent à la règle. Effectivement, le temps d'exécution de cette implémentation explose lorsqu'il est testé avec 64 threads sur cette machine atteignant presque dix fois le temps nécessaire à l'algorithme POSIX. Cette explosion est due à l'utilisation d'un verrou d'algorithme *test-and-test-and-set* qui gère mal les grands nombres de threads qui font de petites tâches.

## 2.3 *ProducerConsumer*

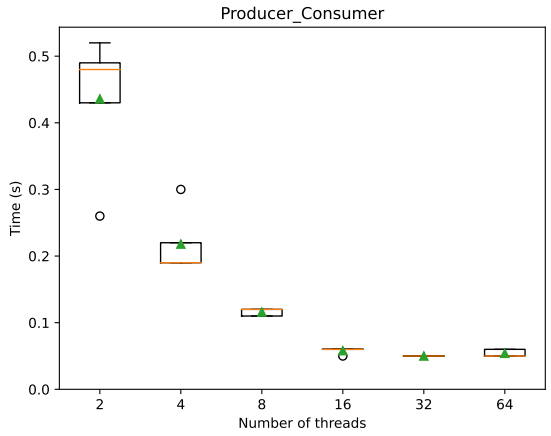



**Figure 7** – *Producer_ Consumer* with POSIX

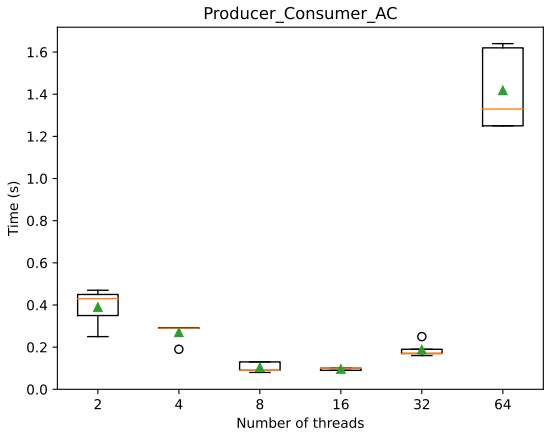



**Figure 8** – *Producer_ Consumer* with spinlock

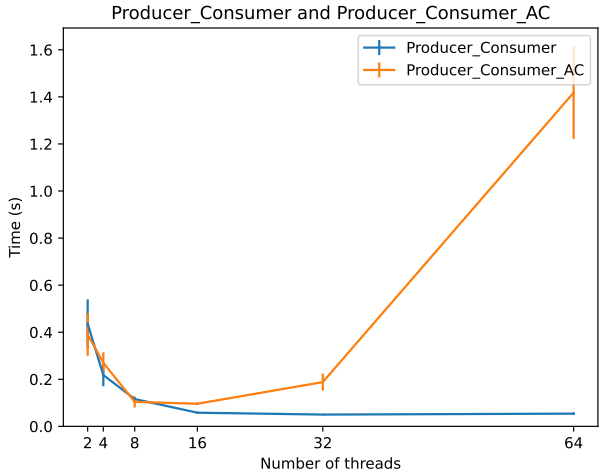



**Figure 9** – Comparaison Producer_Consumer et Producer_Consumer_AC

Ces figures représentent la moyenne, la médiane et la déviation standard du temps d'exécution par rapport aux nombres de threads impliqués de l'algorithme *ProducerConsumer* avec POSIX (ProducerConsumer) et avec les verrous à attente actifs que nous avons implémenté (ProducerConsumer_AC).
On peut sortir des figures 7 et 8 que le temps d'exécution de *ProducerConsumer* décroît avec le nombre de threads. Néanmoins, comme pour *WriterReader*, nous observons que nos tests avec 64 threads pour la version utilisant notre spinlock explose (figure 8). En comparant nos deux graphes a la figure 9, nous pouvons voir que le temps d'exécution est assez similaire jusque 8 threads. Passer cela, la performance de notre solution en attente active croît alors que la performance de POSIX reste assez constante. Cette différence est encore une fois liée à l'inefficacité de l'algorithme *test-and-test-and-set* dans la gestion d'un grand nombre de threads.

## 2.4 *Test-and-set et test-and-test-and-set*

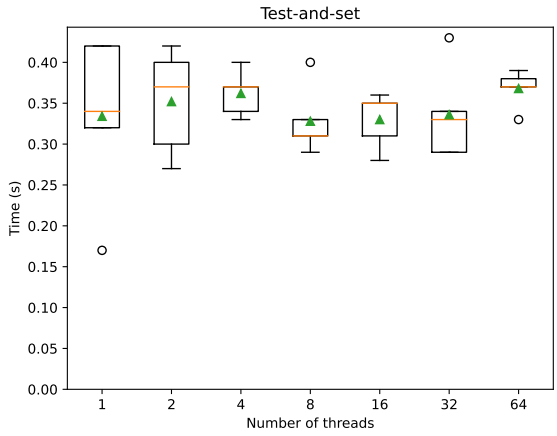



**Figure 10** – *Test-and-set*

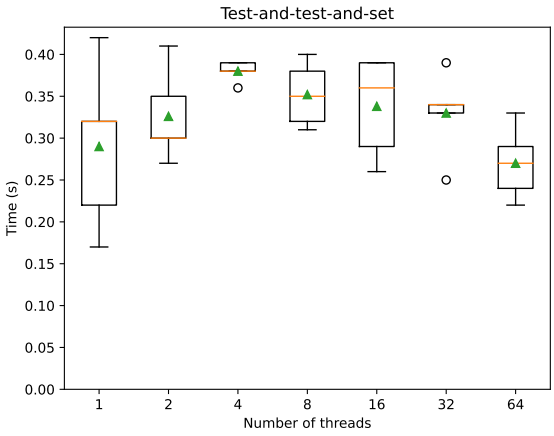
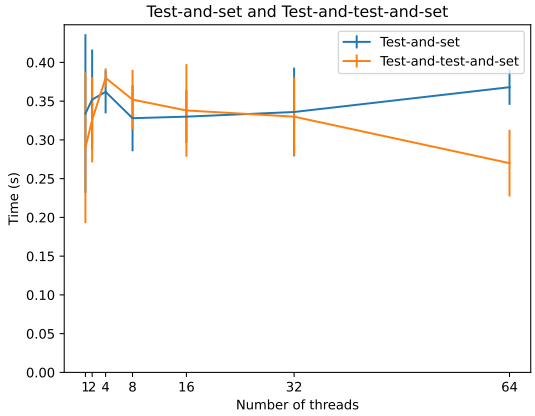



**Figure 11** – *Test-and-test-and-set*




Sur nos figures 10 et 11, nous observons que nos algorithmes ont des performances en temps approximativement constantes comme cela est prévu par la théorie. Note figure 12 nous permet d'affirmer que test-and-test-and-set est meilleur que test-and-set avec beaucoup de threads. Nous avons pu le constater aux points précédents, nos solutions des programmes d'attente active explosent à 64 threads. Cela est sûrement dû au fait que test-and-test-and-set est d'autant moins performant que le nombre de threads augmente.

**Figure 12** – Comparaison test-and-set et test-and-test-and-set

Cette augmentation de temps est simplement due au fait que quand le verrou passera à 0, tous les threads qui bouclaient vont exécuter l'opération atomique cela induit donc un pic d'occupation du bus. Ceci peut conduire à une réduction de la performance générale du système.

## 2.5 Conclusion

Pour conclure ce rapport, on peut constater que nos résultats sont en accord avec la théorie vue au cours bien que nous étions soumis à la variance d'un programme multi-threadé sur une machine réelle. Les tâches d'implémentation de nos propres formes de verrous à attente active ainsi que de nos propres sémaphores nous ont permis de nous approprier chacun de ces concepts. Nous avons donc à la suite de ce projet une compréhension approfondie de l'architecture d'un programme multi-threadé.