UCLouvain
École polytechnique de Louvain

LINFO1252 - Systèmes Informatiques / 2024-2025

Rapport Projet 1
Programmation multi-threadée et évaluation de performances
Groupe 26

Auteurs :
Ephraim Tshiebue - 2334 2100
Arthur Louette - 5108 2100

Professeur :
Étienne Rivière

8 décembre 2024

1 Introduction

Dans ce projet nous avions comme tâche réaliser dans un premier temps des programmes de synchronisations et de concurrence tel que le problème du philosophe,des producteurs consommateurs et des lecteurs/écrivains. Dans un premier temps, nous les avons réalisés avec les sémaphores vu au cours et que nous offre le langage. C Cette partie représente la partie 1. Pour la partie 2, nous nous sommes essayés à la synchronisation par attente active en utilisant des instructions atomiques. La finalité est de comparer les temps d'éxécutions des primitives de synchronisations POSIX et les primitives basées sur l'attente active.

2 Méthodologie et contraintes

2.1 Problèmes étudiés

Dans le cadre de ce projet, 3 problèmes classiques ont été abordées. Ce sont tout 3 des codes mettant en place une situation de programmation concurrente où les différents acteurs représentent en réalité des threads ayant des tâches à effectuer et qui doivent accéder à des ressources communes. Cela demande donc une forme de synchronisation.

1. Philosophes : Ce problème se compose de N philosophes qui alternent entre deux états qui sont "penser" et "manger. Un mutex POSIX assure l'accès aux ressources partagées. 2. Producteurs/Consommateurs : Dans le cadre de celui-ci, les threads producteurs insèrent des données dans un buffer partagé. Les consommateurs, de leur coté, retirent ces données. Des sémaphores POSIX assurent la synchronisation et le buffer est limité à 8 cases. 3. Lecteurs/Ecrivains : Les threads lecteurs peuvent accéder à plusieurs à la ressource partagée tandis qu'un thread écrivain a un accès qui lui est exclusif. Pour ce problème, la synchronisation est aussi assurée par des sémaphores POSIX.

2.2 Primitives

Comme dit précédemment, l'ajout de primitives de synchronisation est essentielle car pour permettre à différents agents de réaliser leurs tâches respectives, il est primordiale qu'une forme de synchronisation soit en place. Pour la partie 1, nous avions utilisé la librairie POSIX tandis que pour la partie 2, nous avons codé un "Test and Set" et "Test and test and set" qui assurent l'attente active entre thread. Cette opération a été possible grâce à des instructions atomiques via de l'assembleur inline que nous avons vu en cours.

2.3 Mesures

Ce projet avait une portée intéressante car il nous a permis de mettre en commun plusieurs concepts et plusieurs programmes. Après la programmation de ces différents problèmes, nous devions réaliser différents script bash afin d'évaluer correctement leur performance. Ces scripts bash éxecutent les programmes 1 nombre de fois donné et pour un nombre de thread distincts. A la fin, il crée un fichier CSV avec une colonne pour le nom du programme, une pour le nombre de thread et une dernière avec le temps moyen des éxécutions. Sur base de ces valeurs,il était donc possible de réaliser des graphiques avec python afin de pouvoir constater les différentes courbes et peut-être tirer des conclusions par rapport à celles-ci.

3 Analyse des performances

Dans cette partie, nous allons visualiser les graphiques et tenter de les décrire et expliquer au mieux. Toutes les éxécutions ont été réalisées sur Macbook Air avec puce M1 et nous avons donc adapté nos éxécutions à celui-ci. N'étant pas encore des experts en la matière, nous avons par moment demandé à l'IA des axes de compréhension et informations supplémentaires pour justifier correctement ces valeurs.

3.1 Graphiques Partie 1

Voici les graphiques de la partie 1, réalisés à l'aide de sémaphore POSIX :

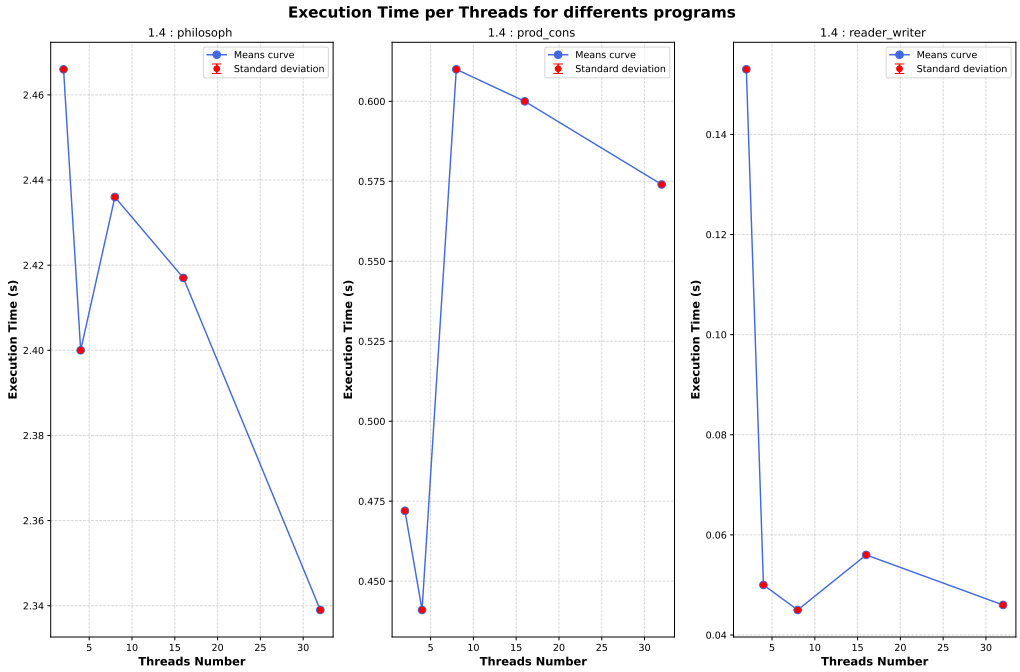

FIGURE 1 – Le graphique montre le temps d'exécution en fonction du nombre de threads pour trois programmes différents, synchronisés à l'aide de primitives POSIX. Chaque subplot correspond à un programme et affiche la courbe des moyennes avec des points rouges représentant l'écart-type. On observe que pour Philosoph, les temps d'éxécution diminuent en fonction du nombre de thread. L'inverse est généralement plus probable mais cela peut s'expliquer par le fait que l'ajout de threads permet l'amélioration du parallélisme.

3.1.0.1 Analyse :

- Philosophes : Hormis le passage de 4 à 16 threads, les temps d'éxécution diminuent en fonction du nombre de thread. L'inverse est généralement plus probable mais cela peut s'expliquer par le fait que l'ajout de threads permet l'amélioration du parallélisme et une gestion efficace des sémaphores. Après 8 threads, les temps d'éxécutions sont de moins en moins long ce qui prouve une meilleure synchronisation entre les philosophes mangeurs et penseurs. Plus on ajoute de philosophes, plus il y a de mangeurs donc les tâches se font plus rapidement.
- Producteurs-consommateurs : Le temps d'exécution augmente légèrement avec le nombre de threads, avant de connaitre une légère chute avant le passage à 16 threads. Cela s'explique par le fait que jusqu'à 8 threads la contention pour le buffer suit son cours normalement car celui-ci a 8 cases. Par la suite, afin de pouvoir accepté plus de threads, des aménagements ont été fait. Ces aménagements permettent donc d'atteindre une forme d'équilibre et une contention modérée.
- Lecteurs-écrivains : Pour celui-ci, les temps sont généralement décroissant même si nous pouvons observer une légère augmentation entre 8 à 16 threads. Nous pouvons donc dire que le parallèlisme permet une meilleure utlisation des sémaphore et que les intéractions entre lecteurs et écrivains sont globalement bien synchronisées. La différence de temps avec Producteurs-Consommateurs peut s'expliquer par le fait qu'ici les opérations se font dans la zone critique.

3.2 Graphiques Partie 2

3.2.1 Test and set & Test and test and set

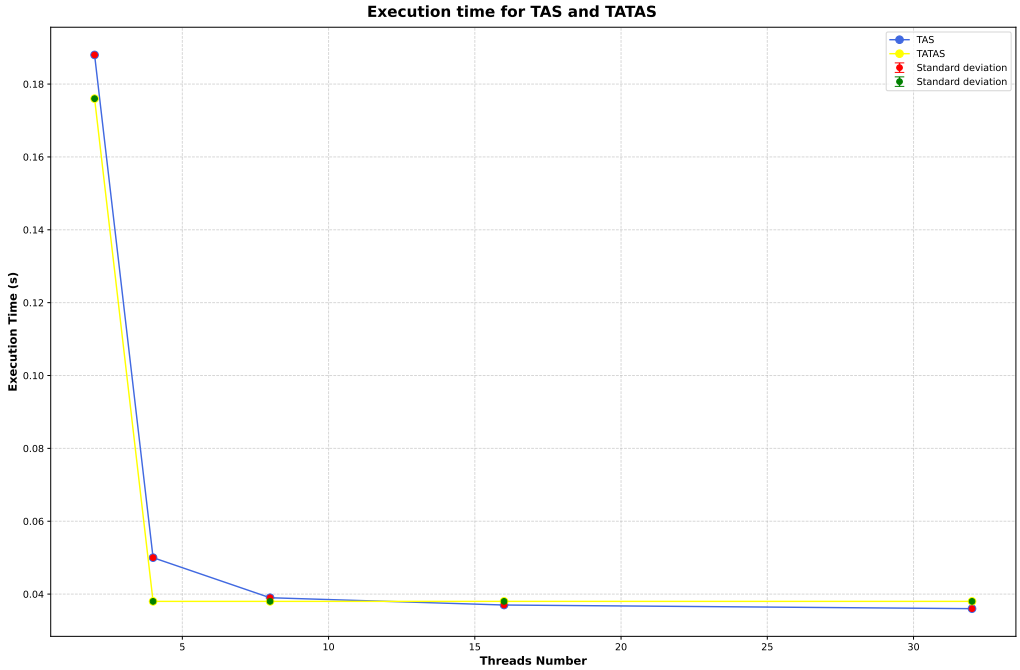

FIGURE 2 – Temps d'exécution en fonction du nombre de threads pour TAS = "Test and set" et TATAS = "Test and test and set".

3.2.1.1 Analyse :

— Ces 2 programmes affichent des temps d'éxécution relativement similaires. Même si nous constatons une descente légèrement plus vertigineuse pour TATAS qui se stabilise mieux par la suite. Cette différence s'explique bien sur par la différence d'implémentation entre ces 2 programmes et la manière dont il gère les sections critiques.

3.2.2 Primitives POSIX & Primitives d'attente active

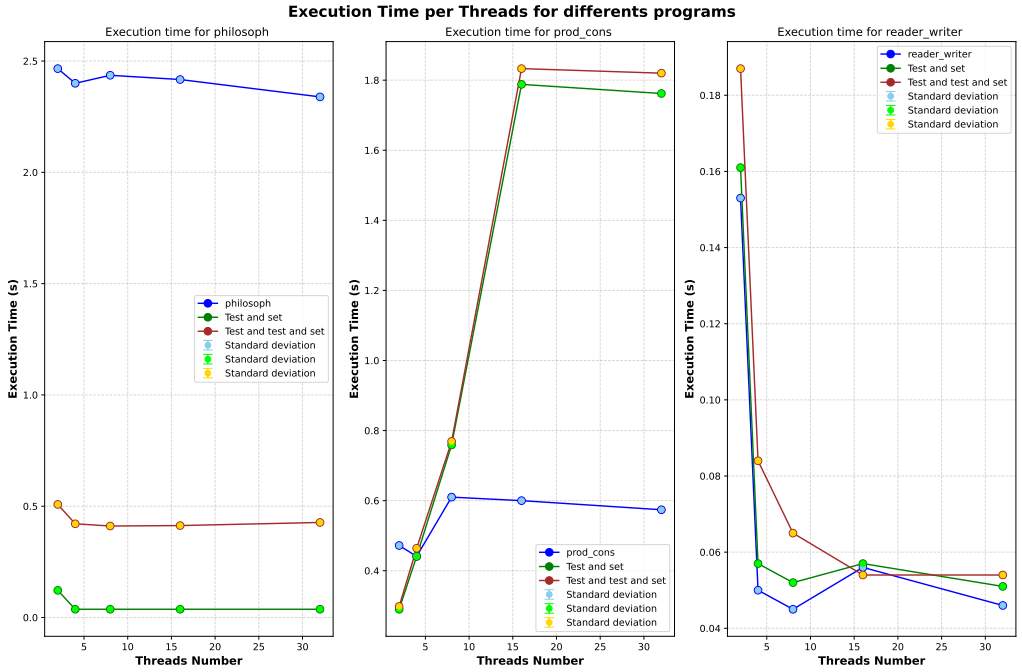

FIGURE 3 – Temps d'exécution en fonction du nombre de threads pour des programmes réalisées avec les primitives POSIX et d'attente active.

3.2.2.1 Analyse :

- Philosophes : Nous pouvons constater que les temps d'éxécutions des différentes implémentations sont assez différentes. En premier lieu, pour chaque nombre de threads distincts, l'éxécution utilisant POSIX a été de minimum 2 secondes supérieur aux autres. Cela veut dire que dans ce cas-ci, nos primitives d'attentes actives ont permis une meilleure gestion des différents agents. En effet, les section critiques sont plus courtes et les actions "manger" et "penser" se se font donc plus rapidement. De plus, l'ordre des threads est asymétrique, ce qui peut aussi fortement contribue. Comme vu au cours, l'ordre des threads a un impact considérable. Nous pouvons aussi dire que pour ces nouvelles implémentations, le parallelisme arrive à un stade d'équilibre à partir de 4 threads
- Producteurs-consommateurs : Pour les 3 courbes nous constatons que les temps d'éxécution sont croissants. Cependant, pour les 2 nouvelles courbes, il y a une ascension de 2 à 16 threads avant de connaitre une stabiilisation jusqu'à 32 threads. Le temps d'exécution augmente légèrement avec le nombre de threads, avant de connaitre une légère chute avant le passage à 16 threads. Cela s'explique par le fait que la contention pour le buffer augemente représente un soucis car chaque agent doit attendre le verrou pour réaliser ses actions. Cela peut entrainer une surcharge et donc plus de temps d'adaptation.
- Lecteurs-écrivains : Pour celui-ci, les temps sont généralement décroissant et sont mêmes meilleurs pour les valeurs de la partie 1.4. Ca s'explique par le fait que ces sémaphores POSIX utilisent un système d'attente passif. La courbe de Test and set suit la même forme que la courbe des sémaphores POSIX tandis que celle de Test and Set est bien différente. En effet, nous observons un descente jusqu'à 16 threads avant de venir se stabiliser. Nous pouvons donc dire que le parallèlisme permet une meilleure utlisation des sémaphore et que les intéractions entre lecteurs et écrivains sont globalement bien synchronisées.

4 Conclusion

En conclusion, ce projet fut enrichissant car étant segmenté en plusieurs partie interdépendantes, il nous a demandé une bonne organisation. En effet, nous avions plusieurs fois du revenir sur certaines portions du code car l'implémentation ne correspondait pas avec certaines parties du projet. De plus, ces résultats nous permettent de nous rendre compte que notre manière d'implémenter influence directement l'éxécution de notre travail et qu'il est important d'aborder différentes techniques de travail pour trouver la meilleure. De plus, ce projet nous a permis de concilier plusieurs concepts de programmation et plusieurs langages, ce qui est une bel introduction à ce qui est effectué dans la vie en général. Pour finir, nous espérons que notre travail comprenant les différents codes, nos graphiques et même ce rapport ont permis de satisfaire la question de recherche initiale.