LINFO1252 ▷ Systèmes Informatiques
Rapport Projet 1

Bellière Arnaud | 62652000
Cheffert Théo | 25882000

UCLouvain

Professeur : E. Rivière

epl

1 Résultats des évaluations

1.1 Introduction

Tous nos résultats proviennent de l'exécution sur `studsrv`, une machine équipée de **16 cœurs**, afin d'évaluer la performance de nos algorithmes en fonction du nombre de threads alloués. Tous nos graphes proviennent de l'exécution consécutive de **5 mesures** afin de représenter la **moyenne** ainsi que l'**écart-type** pour chaque point. Cependant, certains résultats peuvent diverger de nos prévisions. En effet, une **machine réelle** effectue souvent plusieurs tâches en parallèle, ce qui peut augmenter le temps d'exécution des applications testées. Il pourrait aussi y avoir des processus de fond qui perturbent les mesures. Ces phénomènes expliquent en partie la présence de pics observés dans certains graphes dont nous parlerons plus tard.

Pour faciliter l'interprétation des résultats, nous avons choisi de ne pas superposer trop de courbes sur un même graphe. Ainsi, les courbes ont été réparties sur plusieurs graphes (2 ou 3 selon les cas). Le but est d'éviter les confusions dues à des courbes qui se confondraient trop, ce qui compliquerait l'analyse. Néanmoins, les graphes combinant plusieurs courbes sont disponibles en annexe[3] ou dans notre archive (en utilisant la commande `make plot` ou au format pdf dans le dossier "**plots**").

Par ailleurs, les graphiques issus des données obtenues en local (sur notre machine personnelle/VM) se trouvent également dans ce dossier. Ces graphes montrent rarement de gros pics de performance, ce qui conforte notre hypothèse selon laquelle les pics observés sur le serveur résultent de sa charge ou de l'environnement partagé.

Enfin, nous avons utilisé l'outil `Valgrind` sur tous nos codes afin de trouver de potentielles fuites de mémoire. Les résultats de ces tests se sont montrés concluant puisque ceux-ci n'affichaient aucune fuite de mémoire.

1.2 Algorithmes "Test-And-Set" & "Test-And-Test-And-Set"

1.2.1 Graphiques[3.0.1]

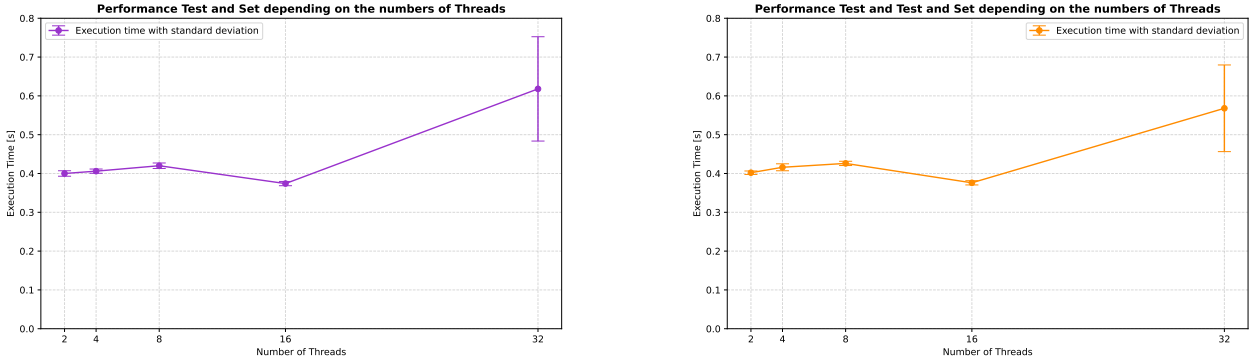

FIGURE 1.2.1 – Résultats des algorithmes `Test-And-Set` et `Test-And-Test-And-Set` sur `studsrv` en fonction du nombre de threads.

1.2.2 Attentes et observations

On s'attend à une courbe assez horizontale, constante, qui finit par exploser pour un nombre de threads élevés. En effet, cette explosion est due à de longs temps d'attente lors des verrouillages et déverrouillages des verrous. Ensuite, on remarque que `TATAS` présente une augmentation beaucoup plus modérée du temps d'exécution de **16 à 32 threads** par rapport à `TAS`, cela démontre une exécution plus rapide comparé à `TAS`. Enfin, on peut noter une **variance assez élevée** pour les mesures en 32 threads dans les deux cas. Cela est du à la fréquence d'accès concurrent qui augmente en 32 threads.

1.2.3 Interprétations des résultats observés

Les deux algorithmes ont un comportement très similaires avec une légèrement meilleur performance pour `TATAS`. On observe comme attendu une courbe avec une légère pente qui finit par augmenter pour un nombre élevé de threads.

`TAS` repose sur une boucle **d'attente active** dans laquelle chaque thread tente continuellement d'acquérir un verrou, provoquant une charge élevée sur le **bus mémoire** lorsque de nombreux threads accèdent en même temps à la variable partagée. `TATAS`, en revanche, réduit cette contention en testant d'abord localement dans le cache (lecture répétée sans écrire) avant de tenter de modifier la variable. Cela diminue les accès coûteux à la mémoire partagée et l'encombrement du **bus mémoire**.

Avec un faible nombre de threads, la différence entre `TAS` et `TATAS` est moins marquée car la contention reste limitée. Les différences d'implémentation des deux algorithmes ne deviennent significatives uniquement lorsque le nombre de threads et, par conséquent, la contention augmentent. L'algorithme `TATAS` minimise les conflits sur les accès mémoire, ce qui explique sa meilleure performance.

`TATAS` est un meilleur choix lorsque le nombre de threads augmente par rapport à `TAS`. Pour un nombre de threads peu élevé, les deux algorithmes se valent.

1.3 Problème des Philosophes

1.3.1 Graphiques[3.0.1]

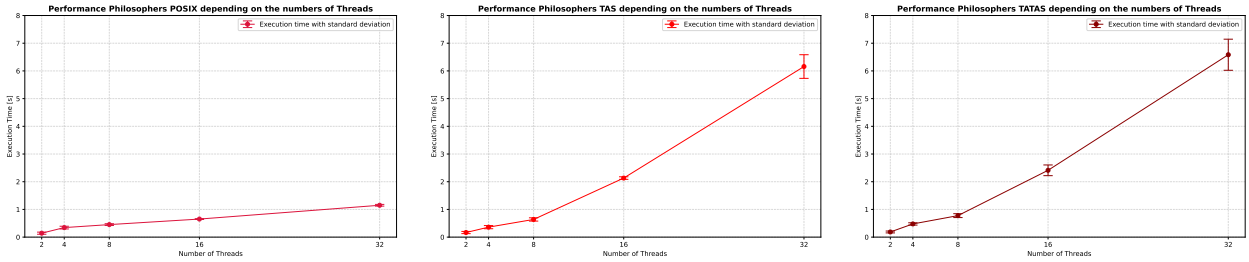

FIGURE 1.3.1 – Résultats du *problème des philosophes* sur `studsrv` en utilisant respectivement les primitives de synchronisation `POSIX`, l'algorithme `Test-And-Set` et `Test-And-Test-And-Set`.

1.3.2 Attentes et observations

On s'attend à une évolution **linéaire** du temps d'exécution. En effet, cela semble logique puisqu'on ne donne que le nombre **N** de philosophes par ligne de commande. Dès lors, plus il y a de philosophes, plus le programme prend du temps.

On observe que la pente du temps d'exécution en `POSIX` est relativement faible (voir Fig.**??** pour une meilleure vue générale). Cependant, lorsque nous utilisons les algorithmes `TAS` et `TATAS`, on remarque une très forte augmentation du temps d'exécution en fonction du nombre de threads. Les courbes de ces deux algorithmes sont assez similaires avec néanmoins une légère meilleure performance pour `TAS`.

Enfin, on peut noter une **variance globale faible** pour les 3 cas. Cela démontre une bonne stabilité des performances dans le cas `POSIX` et une stabilité un peu moins stable due à l'attente active dans les cas `TAS` et `TATAS` où la variance varie un peu plus.

1.3.3 Interprétations des résultats observés

Nous pouvons voir que l'évolution du temps de ce problème est **linéaire** ce qui correspond aux attentes énoncées précédemment. Enfin, on note également que les courbes des algorithmes `TAS` et `TATAS` sont très similaires mais que la performance est bien meilleure dans le cas `POSIX`.

Tout d'abord, les primitives `POSIX` sont optimisées pour la gestion des ressources partagées contrairement aux deux autres qui surchargent le bus mémoire à cause du **spinlock**. Ensuite, ces deux algorithmes sont beaucoup moins performants en raison de leur nature à tenter du **"brute-force"** au lieu de simplement **sleep**, comme ça peut être le cas dans `POSIX`. Enfin, l'implémentation en `POSIX` est mieux optimisée pour limiter les conflits entre threads par rapport à `TAS` et `TATAS` qui ont une contention plus élevée.

1.4 Problème des Producteurs-consommateurs

1.4.1 Graphiques[3.0.2]

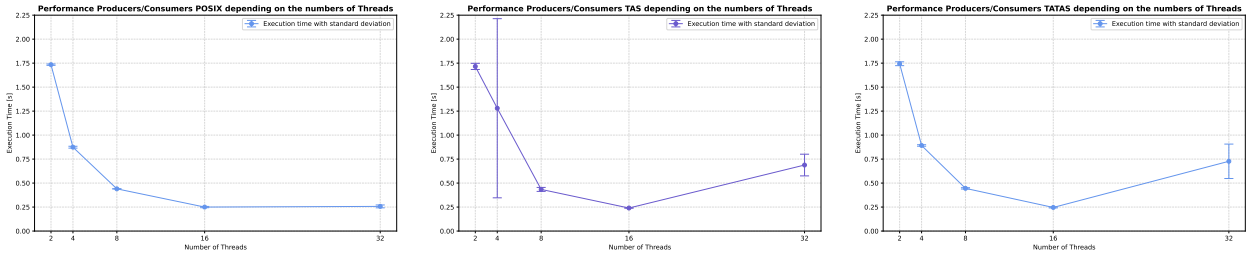

FIGURE 1.4.1 – Résultats du *problème des Producteurs-consommateurs* sur `studsrv` en utilisant respectivement les primitives de synchronisation `POSIX`, l'algorithme `test-and-set` et `test-and-test-and-set`.

1.4.2 Attentes et observations

Dans le cas des producteurs-consommateurs, nous nous attendons à observer une tendance décroissante dans un premier temps jusqu'à arriver à un point critique lorsque le nombre de threads ne compense plus le temps d'attente lié à la synchronisation. Cette tendance devient ainsi croissante. Nous observons comme attendu des courbes en **forme de "U"**. Dans les cas des algorithmes `test-and-set` et `test-and-test-and-set`, le trade-off nombre de thread/contention ne suffit plus pour réduire le temps d'exécution pour un nombre élevé de threads, là ou `POSIX` n'atteint pas le point critique pour les nombres de threads mesurés.

En ce qui concerne **la variance**, l'implémentation `POSIX` montre une variation très faible voire nulle de celle-ci. Cependant, on peut noter deux résultats intéressants pour `TAS` et `TATAS`. Tout d'abord, on a une variance considérable en 4 threads pour l'implémentation avec `TAS`, celle-ci vient de la présence d'une valeur aberrante lors des 5 prises d'échantillons. En effet, une valeur atteint **2.95 secondes** alors que les autres se situent toutes entre **0.85** et **0.88 secondes**. On peut donc en conclure que ce pic n'est sûrement pas du à notre implémentation mais à une valeur extrême qui s'est glissée dans nos données. Enfin, on peut aussi observer une variance modérée pour 32 threads avec `TAS` et `TATAS`. Cela coïncide avec la reprise de croissance observée à partir de 16 threads comme expliqué ci-dessus.

1.4.3 Interprétations des résultats observés

Les 3 algorithmes possèdent des comportements très similaires pour des nombres de threads faibles. Il est important de faire attention à ne pas utiliser un nombre trop élevé de threads lors de l'utilisation des algorithmes `test-and-set` et `test-and-test-and-set` puisque le temps d'exécution augmente lors d'une surutilisation de ceux-ci.

1.5 Problème des Lecteurs et écrivains

1.5.1 Graphiques[3.0.2]

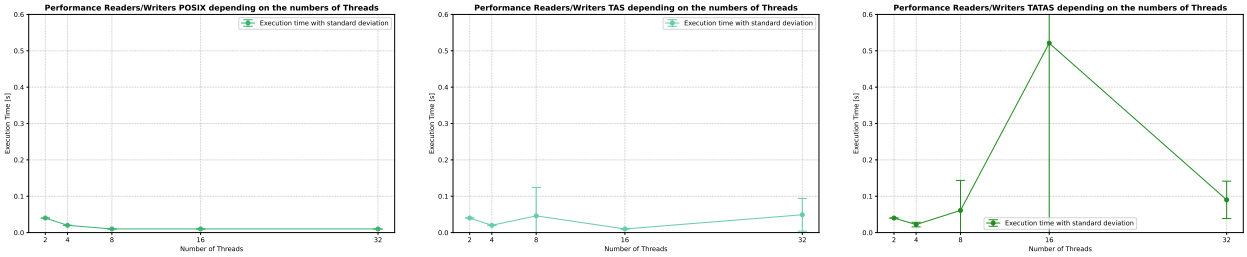

FIGURE 1.5.1 – Résultats du *problème des Lecteurs et écrivains* sur `studsrv` en utilisant respectivement les primitives de synchronisation `POSIX`, l'algorithme `test-and-set` et `test-and-test-and-set`.

1.5.2 Attentes et observations

La courbe attendue devrait être similaire au problème précédent, en **forme de "U"** en raison des similitudes que ces deux problèmes possèdent. Dans une premier temps, la pente est négative. Ensuite, celle-ci se stabilise. Et enfin, une augmentation du temps d'exécution apparait. Nous observons une **variation très élevées** des temps d'exécution pour un même nombre de threads lors de nos mesures pour ce problème qui, comme expliqué dans l'introduction, résultent probablement de la charge ou de l'environnement partagé sur le serveur `Studsrv`. Nous observons tout de même une tendance similaire pour les trois algorithmes avec tout d'abord une pente décroissante pour un petit nombre de threads, suivie d'une remontée. Les résultats sont ici plus flagrants, cependant, dans un graphe comportant les 3 courbes (disponible en annexe[3]), on peut mieux distinguer l'allure de celles-ci.

1.5.3 Interprétations des résultats observés

Nous observons dans le cas de `POSIX` une courbe de pente d'abord négative suivie d'une stabilisation comme attendu. La courbe est assez constante ce qui est cohérent avec nos attentes car `POSIX` peut bien coordonner les priorités sans surcharge significative. Il gère donc bien la montée en charge de manière assez stable.

On observe une attitude similaire au niveau de la stabilisation pour `TAS` et `TATAS`. En ce qui concerne le premier, on remarque que avec **8 threads** il y a un petit pic. Il pourrait être causé lorsque plusieurs threads lecteurs tentent de lire ou lorsqu'un écrivain monopolise le verrou. Cela crée, par conséquent, une augmentation de la contention. Les fréquents accès concurrentiels à la variable partagée créent une surcharge sur le **bus mémoire**. Ce pic pourrait donc être provoqué par la montée en charge sur le verrou avec attente active.

`Test-and-test-and-set` a, quant à lui, un pic beaucoup plus prononcé mais pour **16 threads**. Cependant, on remarque que le nombre de threads n'aident pas forcément à avoir de meilleurs temps pour un nombre de threads plus élevé. En effet, pour un nombre plus important, la vérification dans le cache local devient inefficace ce qui fait chuter notre performance (la charge est élevée à cause de la synchronisation de celui-ci).

Pour ces 2 pics, il est important de noter aussi que la **variance** est **assez grande**. Il ne faut donc pas exclure non plus des valeurs extrêmes qui se seraient glissées dans nos données. Notons aussi que ces mesures ont été prises sur un **serveur réel** qui peut être influencé par des applications extérieurs. Lors des mesures exécutées localement, nous n'observons pas de tels pics. De plus, l'algorithme `test-and-test-and-set` est moins performant que celui de `test-and-set` pour 32 threads. Cet algorithme n'est donc probablement pas adapté pour ce genre de problème.

2 Conclusion du projet

Au terme de ce projet, nous avons exploré les **mécanismes de synchronisation multi-threads** à travers l'étude de plusieurs algorithmes connus, tels que les philosophes, les lecteurs-écrivains, et les producteurs/consommateurs. En utilisant différentes approches, incluant des primitives `POSIX`, l'implémentation des algorithmes `Test and Set (TAS)` et `Test and Test and Set (TATAS)`, nous avons pu évaluer leur performance sur **une machine multi-cœurs** (16 coeurs) `studsrv`.

Nos résultats nous ont permis de voir les points forts et les points faibles des différentes approches.

`POSIX`, tout d'abord, s'est montré robuste et très stable, notamment grâce à des mécanismes optimisés réduisant la contention et favorisant une exécution fluide. Nous avons compris comment il fonctionnait et pourquoi il était aussi efficace par rapport aux autres. `TAS` et `TATAS`, quant à eux, ont souligné les défis posés par l'**attente active**, y compris sous forte charge. Ils ont également permis d'appréhender les concepts de **gestion des conflits** en multi-thread.

Ces expérimentations nous ont permis de mieux comprendre l'impact de la **contention** sur les performances et l'importance de choisir des algorithmes adaptés au problème. Ce projet nous a aussi permis de mieux comprendre la **gestion** d'un **programme multi-threads** afin de mieux protéger nos variables et mieux gérer les différents accès entre threads. Il nous a aussi montré que le comportement des algorithmes varie fortement en fonction du **nombre de threads**. Nous avons pu aussi mieux comprendre comment fonctionnaient les `locks, unlocks, mutexs` et `sémaphores`. Enfin, nous avons pu voir que bien choisir ses primitives de synchronisation avait un fort impact sur les performances de notre code.

3 Annexe

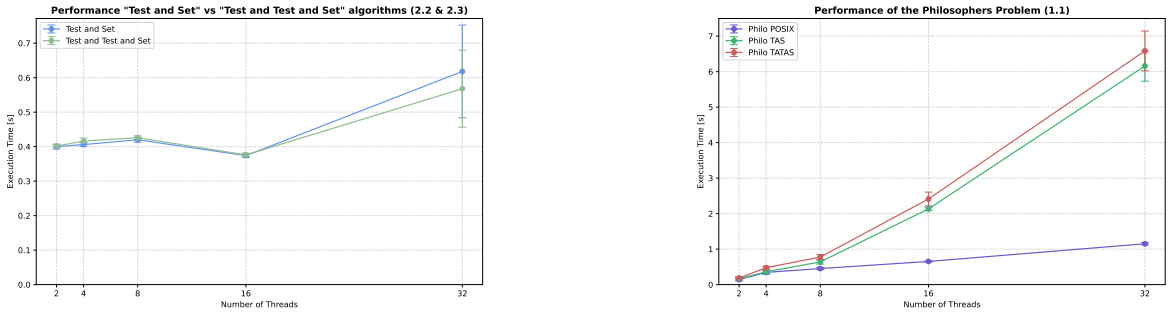

FIGURE 3.0.1 – Comparaison sur un graphe des algorithmes **Test-And-Set** & **Test-And-Test-And-Set** ainsi que du **Problème des Philosophes** sur `studsrv`.

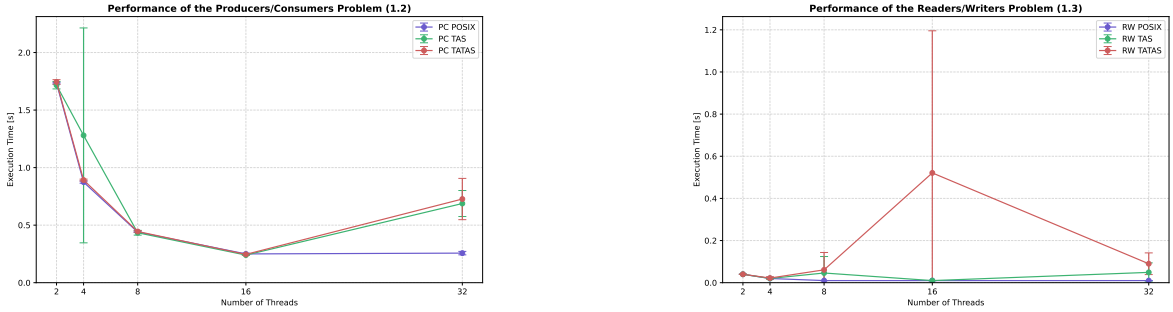

FIGURE 3.0.2 – Comparaison sur un graphe du problèmes des **Producteurs/Consommateurs** ainsi que du problème des **Lecteurs/Ecrivains** sur `studsrv`.