Systèmes Informatiques

Hadrien ALLEGAERT - 07991800 Hugo DE GEORGI - 12311800

7 décembre 2021

1 Partie 1

1.1 Introduction

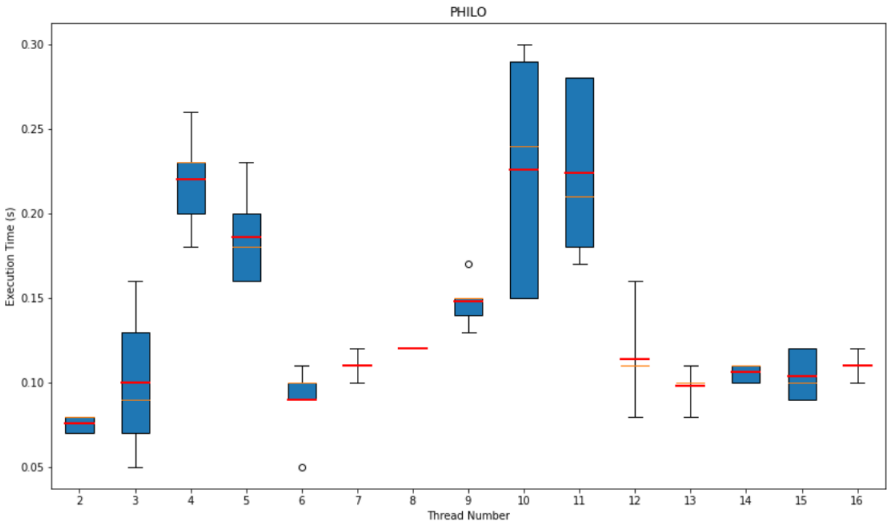
Le but de la première partie du projet était de mettre en oeuvre les synchronisations vues en cours, au travers des différents problèmes tels que le problème des philosophes, des producteurs-consommateurs et des lecteurs et écrivains. Le but de cette mise en oeuvre étant par la suite de comparer les performances de ceux-ci, c'est-à-dire leurs temps d'exécution en fonction du nombres de threads utilisés.

1.2 Analyse graphique des résultats

Une fois les synchronisations implémentées, place à l'analyse des résultats du temps de exécution.

Problème des Philosophes

Pour rappel, le problèmes des philosophes peut être vu comme des philosophes, étant assis à une table, avec entre chaque philosophe une "baguette", le nombre de philosophe et de baguette étant donc les mêmes. Les philosophes peuvent faire deux actions, penser ou manger, et pour qu'un philosophe mange il faut qu'il dispose de deux baguettes, c'est à dire de la baguette à sa droite et la baguette à sa gauche. Chaque philosophe est donc représenté par un thread, et chaque baguette (chopstick) par un mutex. Pour éviter tous problèmes de blocage, il faut une certaine discipline entre les philosophes sinon un deadlock peut se produire assez rapidement.

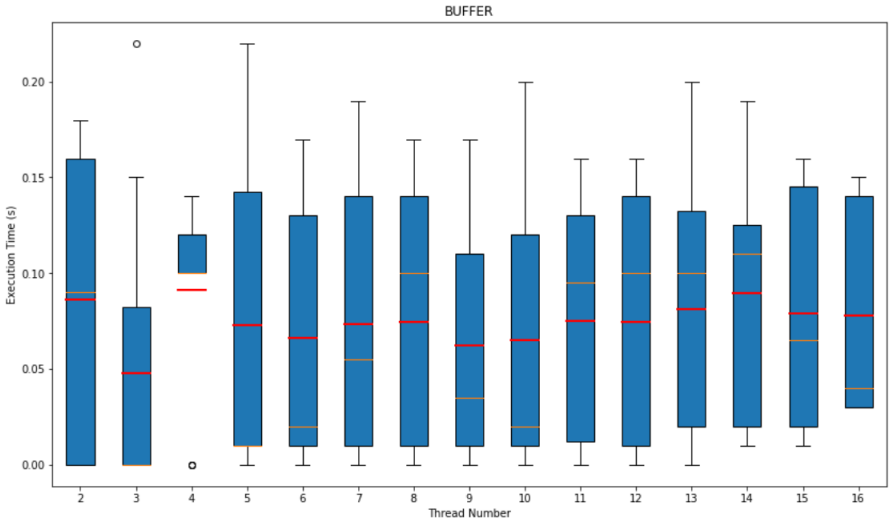


Problèmes des Philosophes

Maintenant, regardons figure 1. Nous avons décidé d'analyser nos données à l'aide d'un boxplot (ou boîte à moustaches en français). Cette représentation permet d'avoir un nombre de données assez important directement visuellement. On peut retrouver avec la ligne continue rouge la moyenne, avec la ligne discontinue cyan la médianne, la boite bleu représente les données entre le 1^{er} et le 3^{e} quartile. Les traits ("moustaches") dépassants la boite bleu représentent le reste des données, et lorsque que l'on voit des petits cercles en dehors de toutes ces données, c'est que cette donnée ne devrait pas exister et est abbérante tellement elle est différente des autres.
Passons maintenant à l'analyse de ces données.

Problème des Producteurs-Consommateurs

Intéressons-nous maintenant au problème des Producteurs-Consommateurs, qui consiste à avoir un ou plusieurs producteurs ainsi qu'un ou plusieurs consommateurs. Les producteurs vont remplir un buffer d'objets tant que celui-ci n'est pas rempli, et les consommateurs vont utiliser venir prendre des objets tant que le buffer n'est pas vide, le buffer étant protégé par un mutex. Le but est de faire en sorte que les producteurs ne bloquent pas les consommateurs, et inversément.



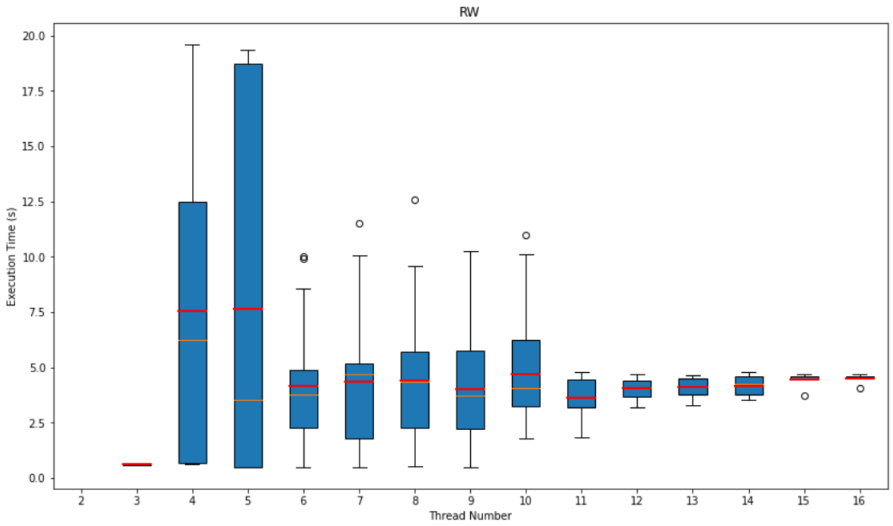
Producteurs-Consommateurs

Cette fois-ci, comme on peut le voir sur la figure 2, le temps de exécution en fonction du nombre de threads est assez constant et la moyenne reste entre 0.05 et 0.10 seconde. Encore une fois, on peut donc se demander si augmenter le nombre de thread est donc nécessaire en vue d'une très légère amélioration des performances (par exemple pour 3 threads, la moyenne est d'environ 0.05 et est la plus basse). On pourrait donc ici utiliser l'écart type pour choisir le nombre de thread dans lequel on a le moins de chance d'avoir une valeur beaucoup plus importante que la moyenne attendue. Une facteur aléatoire (fonction rand()) entre quand même en jeu car sur plusieurs tests, les moyennes sont parfois un peu plus élevées et les box plus ou moins étendues.

Problèmes des Lecteurs-Ecrivains

Le problème des lecteurs-écrivains est assez similaire au problème des producteurs-consommateurs, mais cette fois-ci, le buffer, qui est en l'occurence ici plus une base de données où l'on écrit et lit des données, peut être accédé par plusieurs lecteurs à la fois, c'est à dire accédé en lecture mais pas en modification. Il ne peut tout de même pas être accédé en même temps en lecture et en écriture pour des raisons évidentes de modifications simultanées.
Plusieurs lecteurs vont donc pouvoir lire les données, et lorsqu'un écrivains va vouloir modifier la base de données, il doit attendre que tous les lecteurs aient fini et dévérouille le mutex. Lorsqu'un écrivain

dispose du mutex, plus personne ne peut utiliser les données, y compris les écrivains, chacun doit attendre.



Lecteurs-Ecrivains

Le graphique de la figure 3 est encore différent des deux premiers. On remarque en premier que dans la gauche de celui-ci, les valeurs sont plus élevées. La box est plus étendue pour 4 et 5 threads, ce qui veut dire que les valeurs sont reparties sur une grande plage entre 1 seconde et 18 secondes environ, et la moyenne est aux alentours de 8. Pour les nombres de threads plus grand, on a tout d'abord une diminution de la hauteur de la box, ce qui signifie que les valeurs se ressemblent plus et sont plus concentrées, cela surtout à partir de 11 threads où les moustaches disparaissent presque complètement. Ensuite la moyenne a elle aussi diminuée, ce qui signifie qu'un nombre de thread élevé améliore les performances. On retrouve parfois des valeurs "errantes" en dehors des moustaches.

2 Partie 2

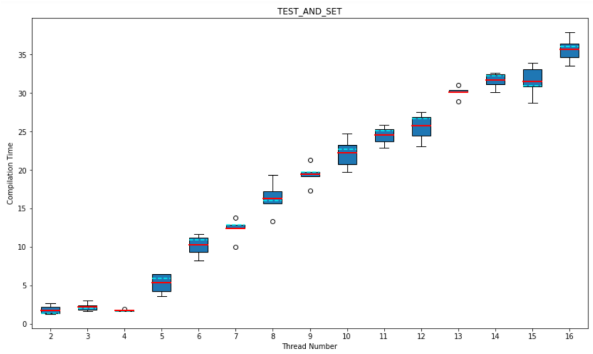
2.1 Performance du verrou test-and-set

Après l'implémentation du verrou test-and-set, nous pouvons passer à l'analyse graphique de celui-ci. Comme pour dans la tâche partie 1, nous avons fait ces tests avec 2 fois le nombres de coeurs physiques de la machine.
Le résultat que nous obtenons est assez satisfaisant, la moyenne augmentant proportionnellement au nombre de threads utilisés. On pourrait donc faire une régression linéaire sans trop d'erreur. Avec 2, 3 ou 4 threads, on peut voir que le temps de exécution est plutôt faible (2/3 secondes) par rapport au reste du graphique. A partir de 5 threads, l'augmentation du temps commence pour passer de 5 secondes pour 5 threads, à environ 35 secondes pour 16 threads.

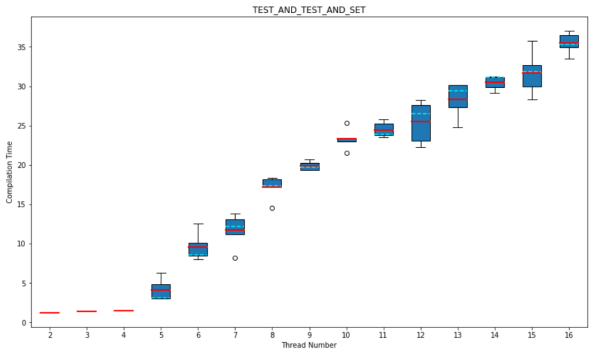
Une rapide inspection visuelle du graphe permet, sans trop de doute, d'émettre l'hypothèse que la complexité temporelle de la gestion de l'entrelacement des threads est d'ordre linéaire, tout en ne pouvant pas écarter l'hypothèse d'une complexité pire que celle-ci.
Le test and set étant basé sur un code assembleur x64-86 relativement basique, ne faisant qu'un échange ou un set de variable à 0 ou 1, celà ne semble pas dérisoir de conclure sur une complexité linéaire.

Cette méthode, bien qu'effectuant un test supplémentaire si le premier lock n'est pas passé, semble avoir une complexité linéaire lui aussi.
Cette méthode est basée sur le test_and_set précédent, sauf qu'on effectue une boucle d'attente si le mutex est déjà utilisé par un autre thread concurrent. Cette boucle peut semblé prendre du temps mais en voyant les plot finaux on se rend compte de l'équivalence de complexité, tout en ayant une

sécurité supplémentaire sur le lock du mutex.



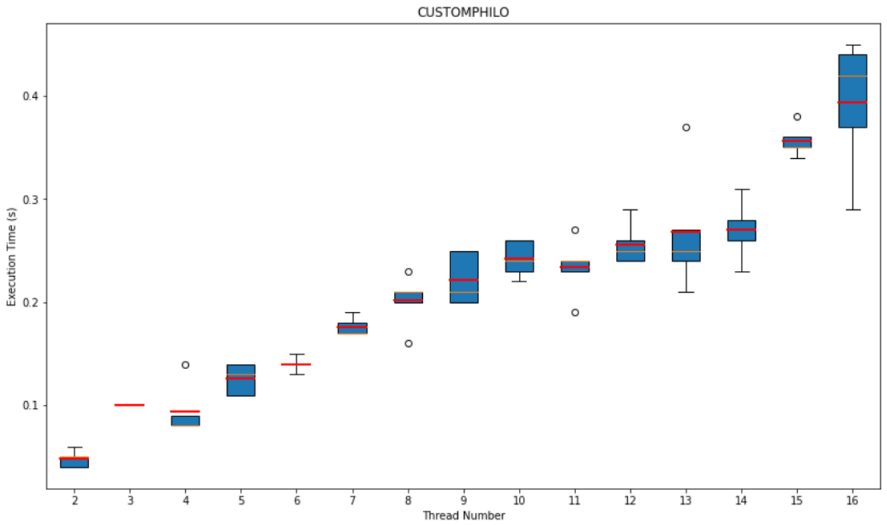
Test-and-Set



Test-and-Test-and-Set

2.2 Nouveaux plots

Une fois que nous avions nos code de la partie 1, il était relativement simple de les adapter à nos mutex et semaphore customs. Il suffisait de remplacer les lignes de déclaration et d'appel aux librairies classiques par nos types de données abstraits lock_t et sem_t2 et leurs appels correspondant.

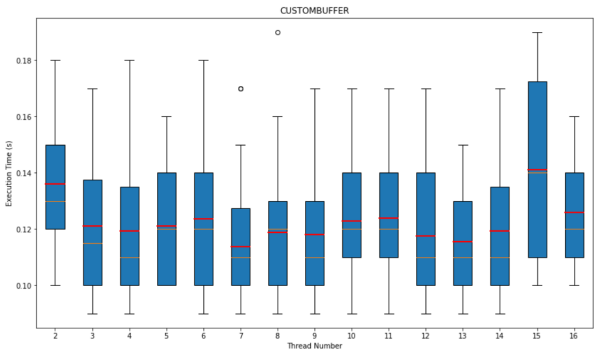


Problème des Philosophes updated

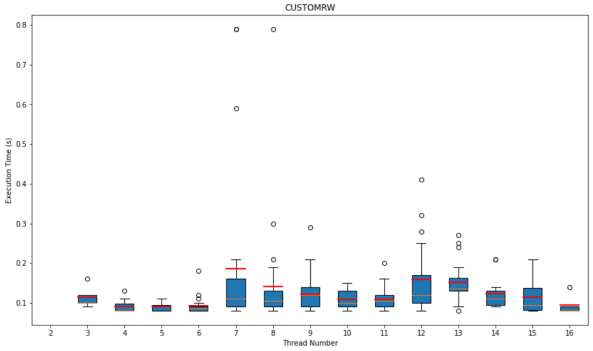
On voit nettement la tendance linéaire se dégager et on voit aussi que pour un même problème, dont le nombre d'itération est de 100000 fois le nombre de threads, ce qui veut bien dire que dans certain cas, ajouter des threads n'augmentera pas la vitesse d'un programme, parfois les performances seront mêmes moins bonnes, dû au coût de gestion de l'entralecement des threads.

Dans le buffer, on peut nettement voir une tendance constante se dégager avec des données plus éparce que dans la version avec les mutex et semaphores classiques.
On voit aussi que nos mutex et semaphores custom sont moins performants que leurs cousins optimiser des librairies standard (moyenne de 0.13s pour le custom contre 0.06s pour le standard).

L'on peut aussi voir ce phénomene de constance dans le reader-writer qui est realtivement serré autour d'une asymptote en 0.15s, ceci est surement du au fait que els sections critiques des threads dans le buffer et le read-writer sont accessibles une a une et donc les threads ne aggne en performance



Producteurs-Consommateurs updated



Lecteurs-Ecrivains updated

que sur le travaile post lecture ou pre ecriture de la donné dans le buffer (ou le heap pour RW) le nombre de threads ne fait que mettre en evidence le cout de la gestion de ceux ci, et vu la disparité des donnée du buffer vers 15 threads on peut considérer que ces couts peuvent etre conséquents.

2.3 Conclusion

Nous pouvons voir que la gestion de l'entralecement des threads amène avec lui un coût d'opération qui peut être considérables, selon le problèmes que l'on essaye de résoudre ce coût peut être absorbé dans le gains de performance que le programme peut faire, mais une fois que l'on arrive à une certaine saturation dans le nombre de threads, ou que le programme ne s'y prète pas, l'ajout de thread supplémentaire engendre une perte de performances.