ÉCOLE POLYTECHNIQUE DE LOUVAIN

LINFO1252 - SYSTÈMES INFORMATIQUES

Projet 1 - Rapport

Membres

Quentin PRIEELS - 57162000
Nathan ZEEP - 24622000

Groupe

Mercredi : 26

Année académique 2022-2023

***Résumé*—Ce document est le rapport du Projet 1 réalisé dans le cadre du cours *LEPL1252 - Système informatique*. Nous y présenterons notre analyse sur les différentes mesures des temps d'exécutions pour les algorithmes de synchronisations implémentés.**

I. Utilisation des primitives de synchronisation POSIX

A. Problème des philosophes

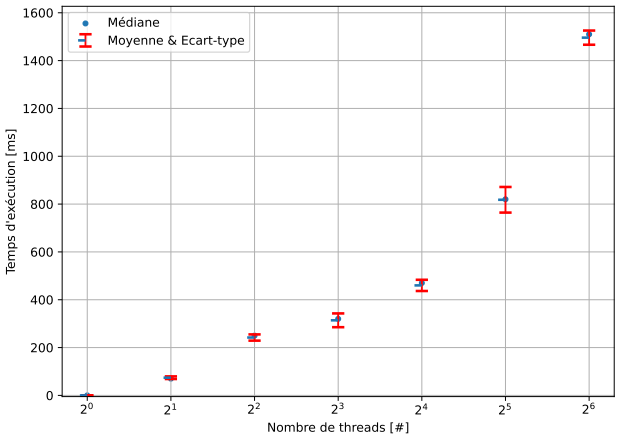

FIGURE 1. Temps d'exécution du problème des philosophes en fonction du nombre de threads

Pour plusieurs exécutions du problème des philosophes, la figure 1 nous indique que l'augmentation du nombre de threads fait croître linéairement [1] le temps d'exécution du programme. Cette croissance s'explique par le fait que chaque thread (c'est à dire chaque philosophe), lorsqu'il est en train d'exécuter sa section critique, bloque ses deux voisins.

Dans le meilleur cas, il y aura toujours un philosophe sur deux qui sera actif (et donc l'autre moitié des philosophes seront bloqués). Cela nous donne une borne inférieure, linéaire, qui permet d'expliquer l'augmentation du temps d'exécution. Tandis que dans le pire cas, il y aura toujours un philosophe sur trois qui sera actif, les deux autres tiers étant bloqués. Cela nous donne cette fois-ci une borne supérieure, linéaire également. La combinaison de ces deux bornes nous permet d'affirmer que, plus on augmente le nombre de threads, plus le temps d'exécution va augmenter, et ce linéairement avec une pente comprise entre 2 et 3.

Cependant, ce raisonnement ne prend pas en compte le temps de création des threads, qui est indépendant de l'exécution du problème considéré mais qui croit, lui aussi, lorsque le nombre de thread augmente.

1. Sur le graphe présenté à la figure 1, on remarque une croissance exponentielle dans un graphe logarithmique. Cela représente bien une croissance linéaire avec le nombre de threads.

B. Problème des producteurs-consommateurs

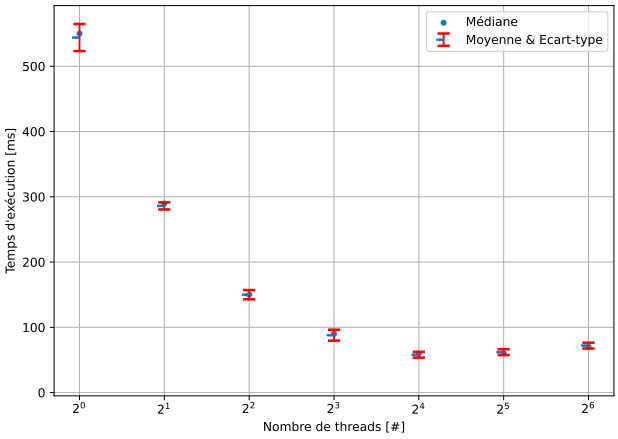

FIGURE 2. Temps d'exécution du problème des producteurs-consommateurs en fonction du nombre de threads

Comme le montre le graphique de la figure2, le temps d'exécution du programme implémentant le problème des producteurs-consommateurs décroît jusqu'à la valeur de 16 threads puis stagne/augmente légèrement si l'on en ajoute encore.

Cette valeur peut s'expliquer de deux façons :

— La première est la taille du buffer. Dans notre cas, le buffer pouvait contenir au plus 8 éléments. Utiliser un nombre de threads bien plus grand que le nombre d'éléments que peut contenir le buffer n'est pas utile puisque ceux-ci vont passer leur temps à attendre de pouvoir placer ou consommer un élément. De plus, on devra payer le coup de la création de ces threads supplémentaires.

— La seconde raison est la suivante : le nombre de coeurs de la machine impacte lui aussi la vitesse d'exécution. En effet, la machine sur laquelle s'est exécuté le programme possédait 32 coeurs. On peut donc supposer que chaque coeur était occupé par un thread ce qui permettait d'en avoir toujours un en attente pour placer ou consommer un élément dès que possible. [2]

2. En local, sur une machine 8 coeurs, le nombre optimal de threads est de 8. Il n'y a pas assez de placer pour que chaque thread soit toujours exécuté par un coeur et en utiliser d'avantage n'est pas utile à cause de la taille réduite du buffer.

C. *Problème des lecteurs et écrivains*

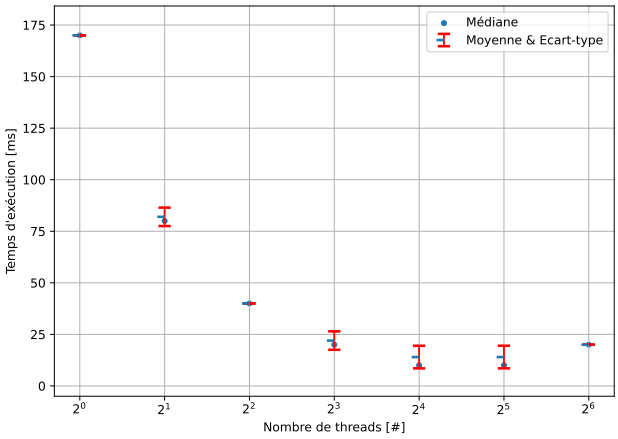

FIGURE 3. Temps d'exécution du problème des lecteurs et écrivains en fonction du nombre de threads

Le temps d'exécution du problème des lecteurs et écrivains suit une décroissance linéaire comme indiqué sur la figure 3, suivant la même logique d'interprétation qu'au point I-A.

La diminution observée le long du graphique provient du fait que le nombre de tâches initiales est prédéfini. De ce fait l'augmentation du nombre de threads permet sa distribution et accélère ainsi son exécution par parallélisation.

La légère augmentation lors des dernières mesures survient vraisemblablement du fait que le nombre de threads égalise ou dépasse le nombre de coeurs de la machine mise à notre disposition (32 dans ce cas-ci). De ce fait, l'initialisation de nouveaux threads prend plus de temps que ce que leur utilisation fait gagner.

II. MISE EN ŒUVRE DES PRIMITIVES DE SYNCHRONISATION PAR ATTENTE ACTIVE

A. *Verrou par attente active*

Les figures 4 et 5 représentent les temps d'exécutions des algorithmes test-and-set et test-and-test-and-set en fonction du nombre de threads, et ce pour différentes machines (8 et 32 coeurs). Sur chacune des machines, les temps d'exécution sont semblables, mais varient fortement en fonction du nombre de coeur disponible.

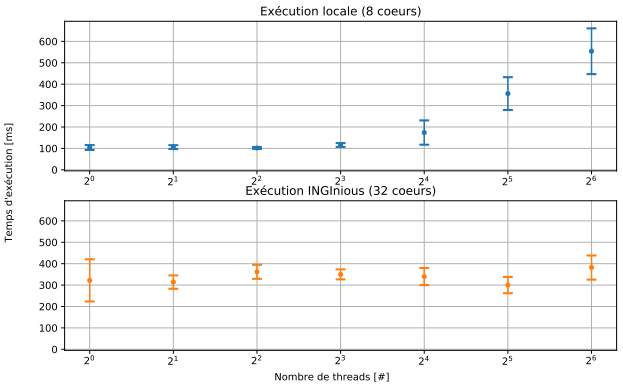

FIGURE 4. Temps d'exécution de l'algorithme test-and-set en fonction du nombre de threads

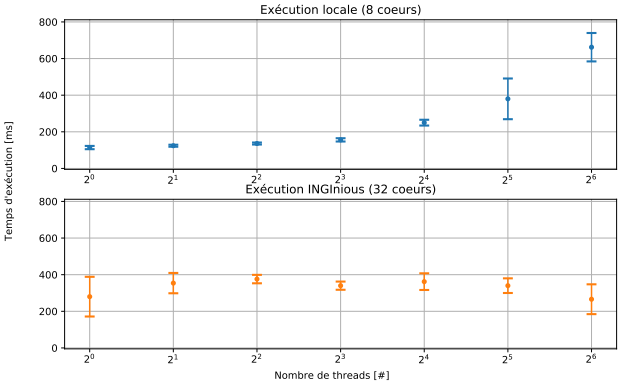

FIGURE 5. Temps d'exécution de l'algorithme test-and-test-and-set en fonction du nombre de threads

Ce que l'on remarque, c'est que le temps d'exécution est constant jusqu'à ce que le nombre de threads lancés dépasse le nombre de coeurs de la machine. Nous pensons que cela pourrait venir des changements de contexte que le processeur doit effectuer. Tant que le nombre de threads ne dépasse pas le nombre de coeur, chaque thread possède son coeur et le temps reste constant. Ce temps est d'ailleurs plus élevée au plus il y a de coeurs car, selon nous, la plus grande taille du processeur pourrait ralentir la communication et les blocages des bus réalisés par xchgl (voir section III).

Ensuite, le temps commence à augmenter. Nous expliquons ce phénomène par le fait que, certains threads sont en état *ready* et ne sont pas exécutés. Pour les exécuter, le processeur devra donc effectuer un changement de contexte et placer le nouveaux threads, ce qui diminue la performance.

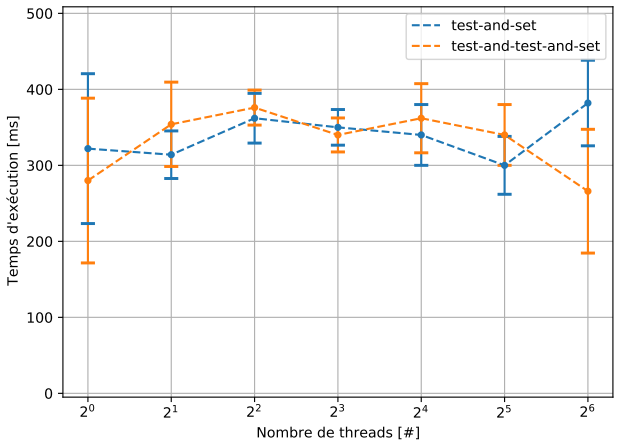

FIGURE 6. Comparaison du temps d'exécutions des algorithmes ts et tts en fonction du nombre de threads

La figure 6 montre la comparaison entre ces deux algorithmes sur une machine de 32 coeurs. Ce que l'on remarque, c'est que lorsque le nombre de coeurs dépasse celui de la machine, l'algorithme test-and-test-and-set semble plus rapide.

Cela s'explique par le fait que cette implémentation du verrou réalise moins d'opérations atomiques xchgl. En effet, l'opération n'est réalisée que si le verrou a lu la valeur 0 avant de tenter de réaliser l'échange exchl. Puisque cette dernière opération est coûteuse (voir section III), la réaliser un nombre plus petit de fois améliore la vitesse du programme.
Cependant, le gain reste marginal. La grande variance des mesures indique que ces résultats ne sont pas entièrement fiables et varient fortement d'une exécution à l'autre.

En effet, l'attente active peut être vue comme une course dans laquelle chaque thread essaye d'être le premier à obtenir le lock, le "gagnant" de cette course est non-déterminé et varie à chaque exécution. Les grands écarts-types des graphes de la section III illustrent bien cela.

III. Impact de l'attente active sur les algorithmes de départ

Il est facilement observable sur les figures 7 à 9 que l'utilisation de l'attente active a pour tendance d'augmenter le temps d'exécution des algorithme proportionnellement à l'augmentation du nombre de thread.

Cela est dû à l'utilisation d'une instruction atomique, ici nous utilisons xchgl. Cette opération, comme toutes celles qui utilisent une adresse mémoire, se doit en effet de bloquer tout accès mémoire aux autres processeurs pendant toute la durée de son opération afin d'éviter tout soucis de concurrence avec autre processeur car cela pourrait invalider l'opération.

A. *Problème des philosophes avec attente active*

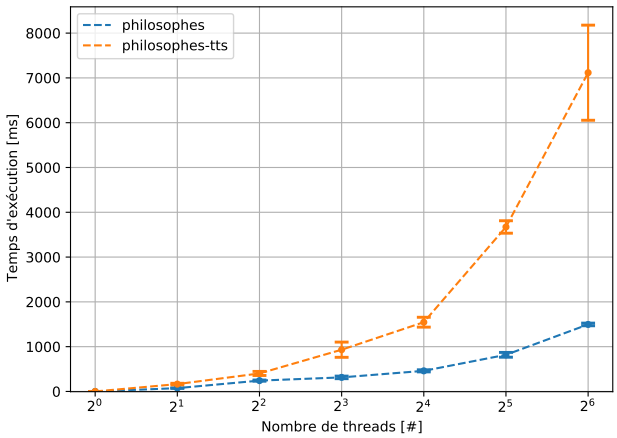

FIGURE 7. Comparaison du temps d'exécution des algorithmes philosophes avec et sans attente active en fonction du nombre de threads

Ce graphique illustre bien ce qui a été mentionné au pointI-A ainsi qu'au début de cette section. On a, comme attendu, que l'augmentation du temps d'exécution due à l'augmentation du nombre de threads et celle due à l'attente active s'additionnent, ce qui ne fait qu'augmenter la pente de l'augmentation linéaire subie par l'algorithme.

B. Problème des producteurs-consommateurs avec attente active

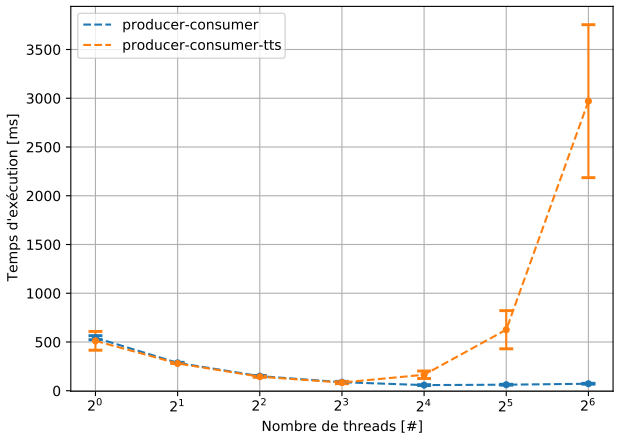

FIGURE 8. Comparaison du temps d'exécution des algorithmes producteurs-consommateurs avec et sans attente active en fonction du nombre de threads

Sur ce graphique on peut apercevoir que les bénéfices d'utiliser des threads sont anéantis par l'attente active qui augmentent le temps d'exécution d'un algorithme, surout lorsque le nombre de thread est plus grand que le nombre de places dans le buffer. La différence est d'autant plus flagrante à un grand nombre de threads car c'est dans les sémaphores que l'opération xchgl est beaucoup utilisée. En utilisant davantage de threads qu'il n'y a de coeurs dans la machine utilisée, la plupart des threads ne font qu'attendre. Cela résulte dans le fait qu'il enchaînent beaucoup plus d'échanges mémoires et c'est cela qui ralentit l'exécution de l'algorithme.

C. Problème des lecteurs et écrivains avec attente active

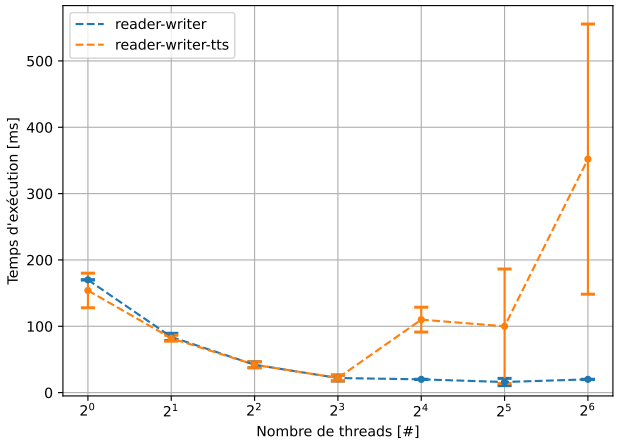

FIGURE 9. Comparaison du temps d'exécution des algorithmes lecteurs et écrivains avec et sans attente active en fonction du nombre de threads

Bien que son temps d'exécution moyen avec 1 thread semble légèrement inférieur, la valeur sans attente active se trouve dans l'intervalle de l'écart-type. Ce dernier graphe nous amène alors à la même conclusion que les précédents concernant l'attente active : ce procédé n'est jamais plus rapide qu'une synchronisation posix et est beaucoup moins fiable sur un faible nombre d'itérations.

IV. CONCLUSION

Les différents graphiques présentés dans ce rapport nous amènent à rester prudent lorsque l'on implémente des programmes concurrents.

Premièrement, nous constatons que les algorithmes d'attente active sont nettement plus lents que ceux qui utilisent des appels systèmes. En effet, la course entre chaque thread pour s'emparer du verrou, et les appels répétés à xchgl qui bloquent les accès mémoires aux autres threads, impactent directement la performance de ce genre de verrous.

Deuxièmement, le fait d'ajouter des threads n'est pas synonyme de gain de performance. En effet, le temps d'exécution en fonction du nombre de threads dépend fortement de l'application/l'algorithme implémenté ainsi que de la machine (et de ses ressources CPU notamment) sur laquelle il est exécuté.

Finalement, ce projet nous aura également appris qu'un même programme, tournant sur une même machine et avec le même nombre de threads ne s'exécute pas toujours en un temps donné. Le non-déterminisme peut avoir un impact important sur le temps d'exécution, même lorsque tout ces paramètres sont fixés par les choix de l'ordonnanceur, qui n'exécutera pas chaque thread dans le même ordre à chaque exécution.