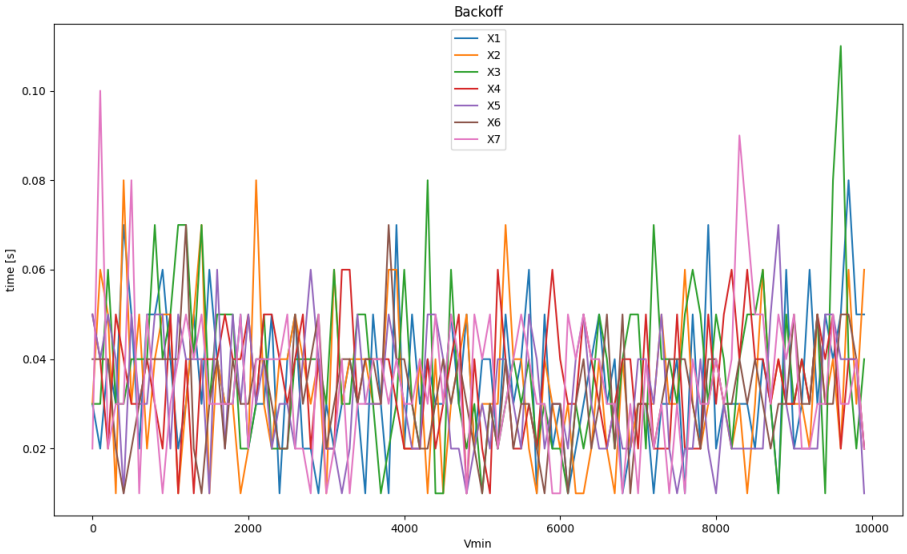
École Polytechnique de Louvain

Rapport de projet

Projet 1: LINFO1252
Programmation multi-threadée et évaluation de performances



Tuteur groupe 4:
Tom Rousseaux

Membres du groupe :
Brieuc Pierre 55481800,
Damien Sprockeels 68641400

November 30, 2020

1 Utilisation du makefile

- ▷ DEFAULT: Le default est main ici
- ▷ ALL: Une combinaison de tous les perf
- ▷ MAIN: Compile l'entièreté des codes en C
- ▷ A.OUT: Lance le debugger gdb
- ▷ PERF1: Mesure et affiche les performances de la tâche 1.5
- ▷ PERF2: Mesure et affiche les performances de la tâche 2.2
- ▷ PERF1-2: Mesure et affiche les performances de la tâche 2.5
- ▷ PERFB: Mesure et affiche les performances de la tâche 2.6
- ▷ CLEAN: Nettoie les fichiers crées par les autres commandes

2 Utilisation des primitives de synchronisation POSIX

2.1 Philosophes

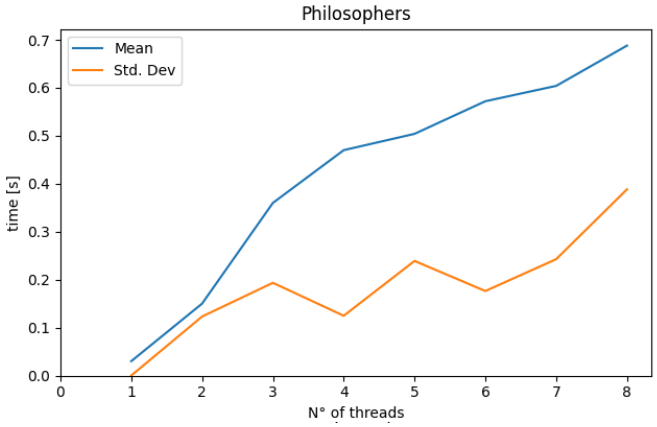


FIGURE 1: Évolution du temps (moyen & écart type) pour 10^6 cycles penser/manger avec un nombre de threads $\in [1, 8]$

Observations: (1) Le temps moyen et l'écart type augmentent globalement linéairement avec le nombre de philosophes (=threads). (2) L'ajout d'un thread, passant d'un nombre impair à un nombre pair de philosophes, produit une augmentation constante de temps. (3) Tandis que dans le cas opposé, le temps moyen agmente de façon irrégulière. L'écart type est représentatif de la stabilité de l'algorithme. (4) Plus le nombre de threads est important, plus l'ordre des cycles des philosophes est aléatoire, plus la performance de l'algorithme est instable. (5) Si l'on regarde plus en profondeur, on peut voir qu'un nombre de threads pair est plus stable qu'un nombre de threads impair.

Interprétations:

1. Le nombre total de cycles à effectuer augmente linéairement avec le nombre de threads.
2. La concurrence est telle qu'on peut considérer que le nouveau thread fait ses cycles quand les autres ont fini.
3. La concurrence est plus forte car les possibilités sont plus nombreuses et donc plus aléatoire.
4. La concurrence est proportionelle avec le nombre de threads lancés.
5. Le nombre de mutex étant pair, la concurrence est plus fluide si un thread sur 2 est actif simultanément.

2.2 Producteurs-consommateurs

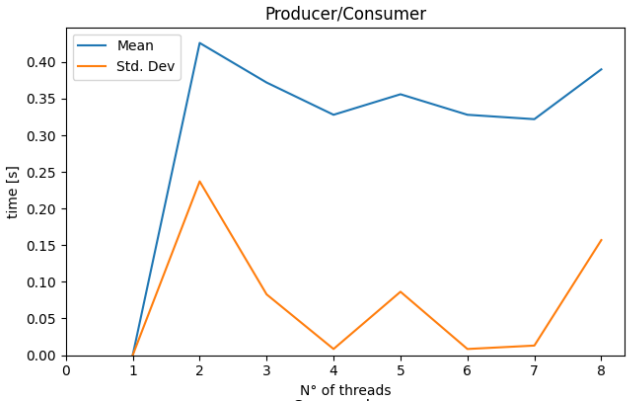


FIGURE 2: Évolution du temps (moyen & écart type) pour 1024 éléments produits avec un nombre de threads $\in [1, 8]$

Observations: (1) Nous avons un pic pour le temps moyen avec 2 threads (1 consommateur et 1 producteur). (2) Plus le nombre de threads est important, plus le programme est performant. (3) Lorsque le nombre de threads arrive à 8, le temps moyen augmente. (4) L'écart type reste faible peu importe le nombre de threads.

Interprétations:

1. Le producteur doit faire les 1024 cycles seul.
2. Les 1024 cycles sont partagés entre plusieurs producteurs, ce qui augmente la performance.
3. Les sémaphores bloquent les threads plus fréquemment du à la petite taille du buffer (facteur limitant).
4. Les sémaphores rendent le code plus stable et la concurrence plus fluide.

2.3 Lecteurs-écrivains

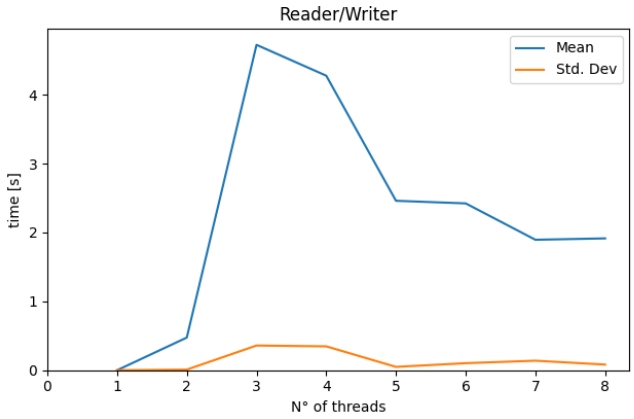


FIGURE 3: Évolution du temps (moyen & écart type) pour 640 écritures et 2560 lectures avec un nombre de threads $\in [1, 8]$

Observations: (1) Pour 2 threads (1 écrivain et 1 lecteur), le temps nécessaire est très faible. Un temps comparable au problème des philosophes avec autant de threads: seul un des deux threads a accès à la "base de données" à la fois. Le temps reste plus grand car une boucle while simule une action dans cet algorithme.
(2) Au dela de 3 threads, il y a concurrence donc la sémaphore commence à agir. (3) Plus le nombre de threads est important, plus le temps raccourcit.
(4) L'écart type reste faible peu importe le nombre de threads.

Interprétations:

1. Les sémaphores n'interviennent pas dans ce cas.
2. Ce comportement revient à 1 producteur et plusieurs consommateurs.
3. Plus le nombre de consommateurs est important, plus les cycles finissent rapidement.
4. Les sémaphores rendent le code plus stable et la concurrence plus fluide.

3 Mise en œuvre des primitives de synchronisation

3.1 Implémentations de nos primitives d'un mutex

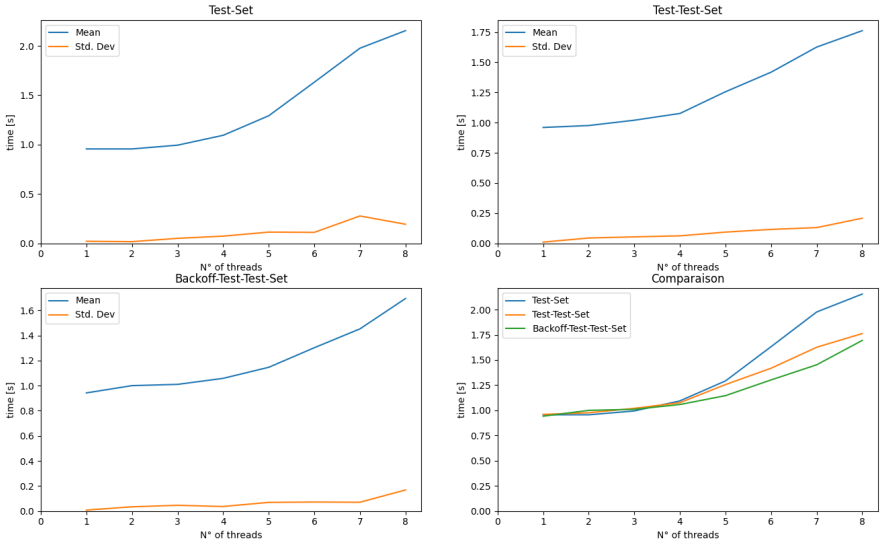


FIGURE 4: Évolution du temps (moyen & écart type) pour 6400 cycles avec un nombre de threads $\in [1, 8]$

Observation & Interprétation: (1) Les trois implémentations ont le même comportement pour un nombre de threads augmentant. Seul la pente diffère. D'abord, le verrou Test-and-Set à attente active a la pire pente car la contention est sans cesse forte. Ensuite, le verrou Test-and-Test-and-Set est plus performant car la contention n'est forte que periodiquement. Enfin, le verrou Backoff-Test-and-Test-and-Set est le plus performant car la contention reste relativement faible tout le temps. (2) Les écarts type sont faibles dans les trois cas. Les verrous sont thread safe et leur performances sont stables.

3.2 Comparaison de nos primitives d'un mutex

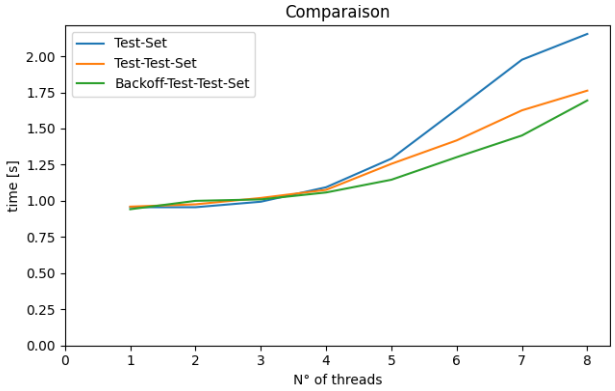


FIGURE 5: Évolution du temps moyen pour 6400 cycles avec un nombre de threads $\in [1, 8]$ par nos implémentations de mutex

Observations: (1) Pour 1 thread il n'y a pas de différence entre les 3 algorithmes. (2) Pour 2-4 threads, l'attente active est plus rapide que la deuxième implémentation. (3) A partir de 4 threads , l'écart entre les 3 implémentations grandit. (4) L'écart entre Backoff-Test-and-Test-and-Set et Test-and-Test-and-Set est plus petit que celui entre Test-and-Test-and-Set et Test-and-Set.

Interprétations:

1. Il n'y a jamais d'attente pour une thread. Les trois verrous sont les mêmes dans ce cas.
2. L'attente est quasi inexistante pour si peu de threads, donc les attentes passives diminue la performance de leur verrou. Nous avons l'ordre inverse que pour une forte contention.
3. Plus la contention est forte, plus les facteurs limitants se répercutent dans la performance des verrous.
4. La contention est plus partielle dans Test-and-Test-and-Set donc plus efficace. La contention est encore plus partielle pour le verrou Backoff donc l'implémentation est encore plus efficace.

3.3 Philosophes

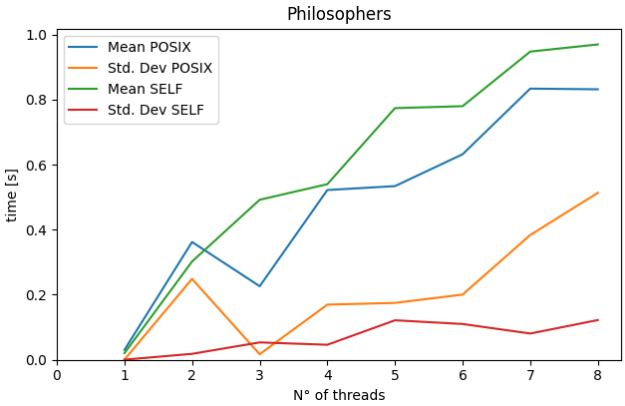


FIGURE 6: Comparaison entre l'expétience de la Figure 1 et leurs mutexs revisités par attente active

Observations: (1) Nos primitives sont moins performantes que les primitives POSIX. (2) Les 2 implémentations ont relativement la même évolution. (3) L'écart type de notre implémentation est plus faible que celui de l'implémentation POSIX.

Interprétations:

1. Nos primitives sont par attente active donc chaque thread augmente la contention sur le verrou. Plus il y a de mutexs présents, plus la différence de temps est marquée.
2. Les **Interprétations** faites au problème des philosophes avec des mutexs POSIX sont aussi valides pour nos implémentations.
3. Une nouvelle fois, il s'agit de l'attente active. Elle rend la performance des mutex assez stable.

3.4 Producteurs-Consommateurs

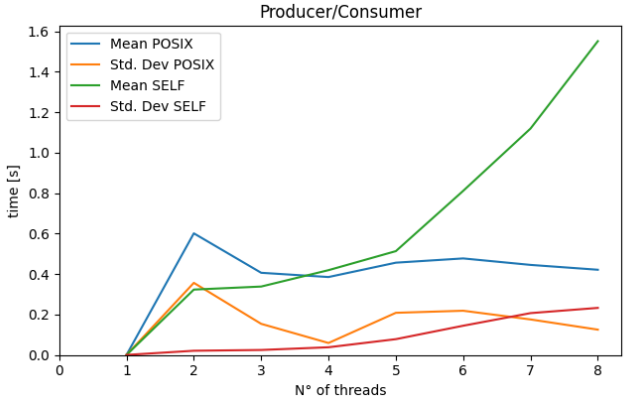


FIGURE 7: Comparaison entre l'expérience de la Figure 2 et leurs mutexs et sémaphores revisités par attente active

Observations: (1) Pour 3-5 threads, l'efficacité de nos primitives est semblable à celle des primitives POSIX. (2) Au delà de 5 thread, il y a une baisse significative de l'efficacité de nos primitives. (3) L'écart type de notre implémentation est plus faible en général. (4) Pour 2 threads, on observe une pointe dans le graphe des primitives POSIX.

Interprétations:

1. Notre mutex est d'une performance presque similaire au mutex POSIX pour peu de threads.
2. Nos sémaphores ne sont pas aussi efficaces que celles POSIX. De plus, la contention de notre verrou diminue l'efficacité de celui-ci ainsi que celle de ceux présents dans nos sémaphores.
3. De nouveau, il s'agit de la contention causée par l'attente active de nos mutex.
4. L'écart type pour 2 threads pour les primitives POSIX est nettement plus élevé que pour d'autres nombres de threads, on peut donc supposer que c'est la raison pour laquelle on a une pointe à ce moment-là.

3.5 Lecteurs-Écrivains

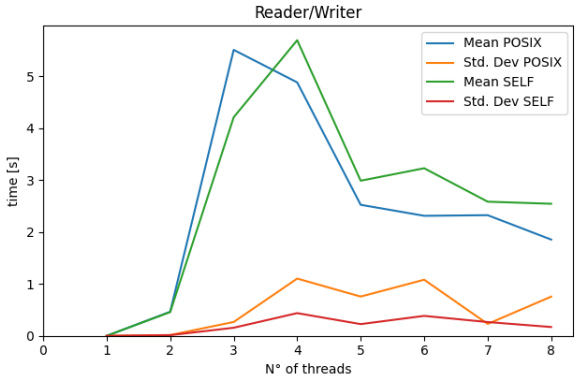


FIGURE 8: Comparaison entre l'expétience de la Figure 3 et leurs mutexs et sémaphores revisités par attente active

Observations & Interprétations: (1) On observe que le programme exécuté avec nos primitives a un comportement semblable à celui du programme exécuté avec les primitives POSIX, avec un temps nécéssaire légèrement plus grand qui s'explique par le fait que nos primitives sont un peu moins performantes que celles POSIX. (2) L'écart type de notre implémentation est plus faible en général, pour la même raison que citée pour les autres comparaisons.

3.6 backoff test-and-test-and-set

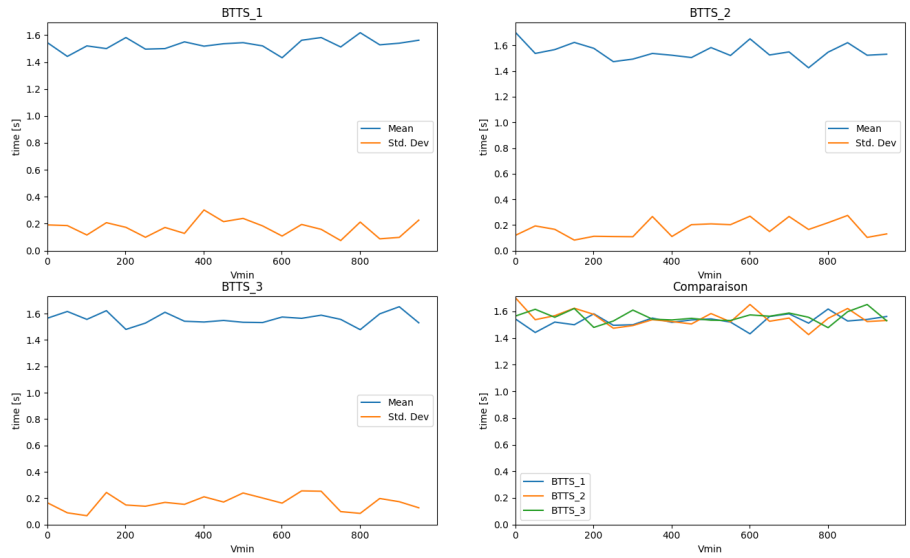


FIGURE 9: Test de performances de notre verrou Backoff-Test-and-Test-and-Set

Observations & Interprétations: (1) La performance ne change pas de manière considérable avec Vmin (=temps d'attente minimum). Le graphe est bien plat. (2) La performance ne change pas de manière considérable avec Vmax (=temps d'attente maximum). Les trois graphes sont bien identiques.
(3) Les écarts types sont faibles.
(4) A chaque fois qu'on relance les tests de performances, les pics se trouvent à différents endroits donc il ne s'agit pas de la performance du verrou mais bien le reflet de celle des coeurs (qui effectuent d'autres opérations à coté du programme.)

4 Conclusion

- ▷ La performance de chaque algorithme et verrou dépend de son nombre de threads par rapport aux ressources accessibles à ceux-ci.
- ▷ Le verrou Backoff-Test-and-Test-and-Set sera préférable par rapport aux 2 autres implémentations étudiées lors de ce rapport.
- ▷ L'algorithme des philosophes est bien pour les programmes ayant besoin de faire de petites opérations sur un petit nombre de ressources partagées.
- ▷ L'algorithme des Producer/Consumer est bien si l'on veut avoir plus de ressources partagées et en effectuant des opérations plus coûteuses en temps.
- ▷ L'algorithme des Reader/Writer est bien pour les programmes ayant des acteurs qui ne modifient pas les ressources partagées.