UCLouvain
École polytechnique de Louvain

LINFO1252 – Systèmes informatiques

Projet 1 : Programmation multi-threadée et évaluation de performances

---

par
Martin Van Mollekot
Arthur Vandroogenbroek

Date : 2 décembre 2020
Professeur : Etienne Riviere

Table des matières

1 Introduction **1**

2 Évaluation de performances **1**
2.1 Les verrous par attente active 1
2.2 Le problème des philosophes 2
2.3 Le problème des producteurs-consommateurs 3
2.4 Le problème des lecteurs et écrivains 4

3 Pushing the envelope **4**

4 Conclusion **5**

1 Introduction

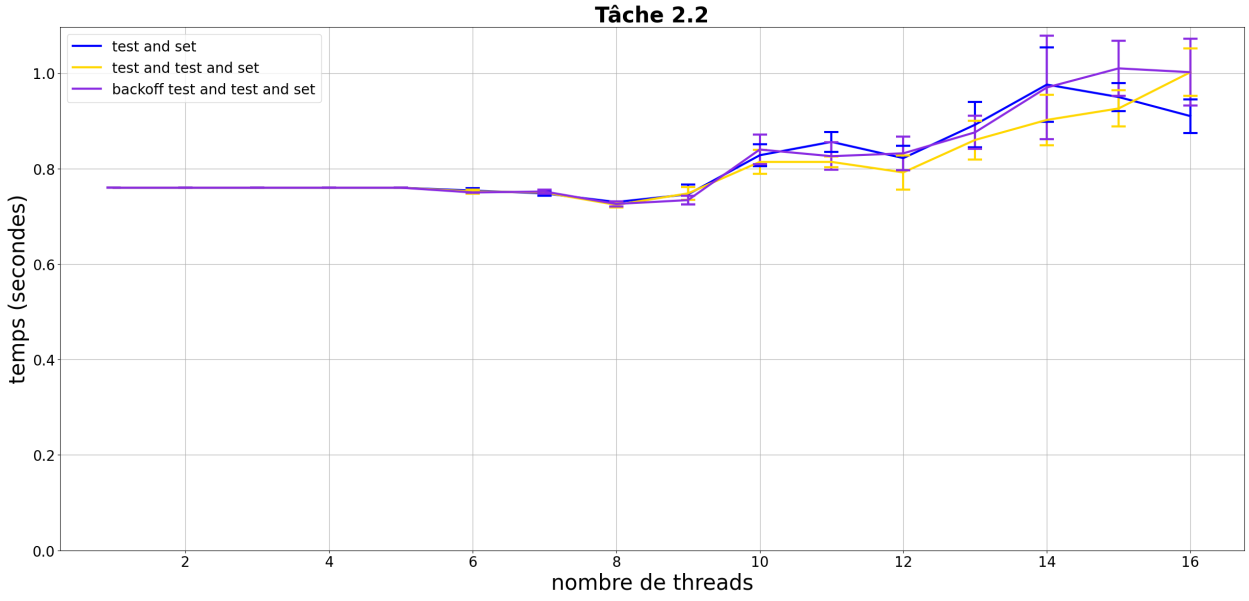
Dans le cadre du cours de Systèmes Informatiques *LINFO1252*, nous avons évalué les performances de 3 problèmes implémentés en C : celui des philosophes, des producteurs-consommateurs et des lecteurs-écrivains. Nous avons d'abord utilisé les mutex et les sémaphores de la librairie *lpthread* et avons ensuite implémenté la synchronisation par attente active à l'aide d'instructions atomiques de 3 manières :*"test and set"*, *"test and test and set"* et *"backoff test and test and set"*.

2 Évaluation de performances

Nous avons mesuré l'évolution des performances de chaque problème en fonction du nombre de threads alloués. Chaque mesure a été répétée 5 fois pour plus de précision. Les graphes de ce rapport représentent l'évolution des moyennes des mesures, avec à chaque fois une barre d'erreur correspondant aux déviations standards.
De plus, par crainte d'avoir des résultats perturbés par les autres tâches en cours sur nos ordinateurs, nous avons réalisés nos tests sur une machine virtuelle à 8 cœurs d'Amazon Web Sevices (instance c5a.2xlarge du service EC2). Voici les résultats que nous avons obtenus.

2.1 Les verrous par attente active

Afin de mesurer les performances de nos différents verrous, nous avons réalisés 6400 cycles de lock-opération-unlock au total, répartis de façon égale entre plusieurs threads. L'opération en question a une durée aléatoire mais une espérance de durée définie (grâce à une boucle *while(rand() < RAND_MAX)*). Le scénario parfait étant que le temps ne varie pas en fonction du nombre de threads, on constate que ce dernier est en légère hausse après que le nombre de threads ait dépassé le nombre de cœurs de la machine due de l'imperfection du verrou.

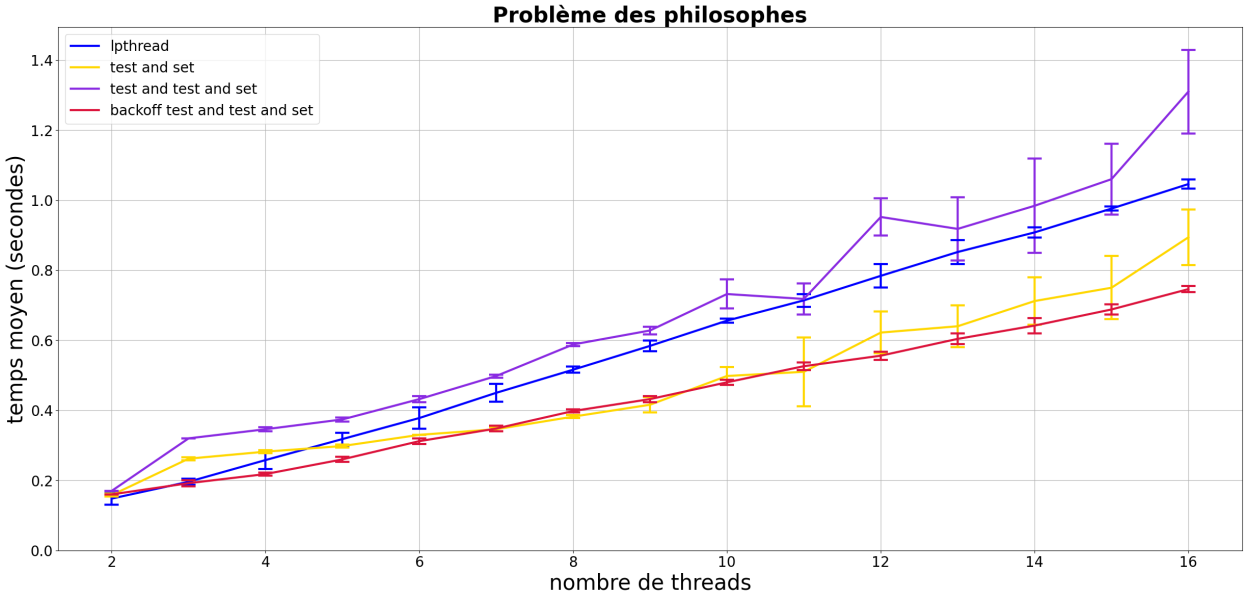


Étant donné que la section critique prend du temps, on devrait s'attendre à avoir de très mauvais résultats avec la logique *test and set* où on se contente d'utiliser la commande *xchgl* - qui bloque le bus lors de son exécution - en boucle jusqu'à avoir la priorité sur le verrou. Des résultats qui devrait s'améliorer avec la logique *test and test and set* qui utilise *xchgl* seulement quand on voit que le verrou est libre et qui attend le reste du temps. Et on devrait encore s'améliorer avec la logique *backoff test and test and set* qui, en plus d'attendre le changement en mémoire avant de tenter l'opération atomique, attend un temps aléatoire et de plus en plus long (qu'on a réalisé avec une boucle *while(rand() < RAND_MAX/a)* avec un a de plus en plus grand jusqu'à une certaine limite). Cela permet de ne pas avoir tous les threads qui attendent le verrou qui utilise en même temps *xchgl* quand il se libère.

En pratique, on n'obtient pas exactement cela : les temps mesurés restent relativement constants jusqu'à 8 threads, quelque soit la logique utilisé, puis augmentent légèrement en même temps que l'écart type des mesures. Le temps et l'écart type qui augmentent quand le nombre de threads dépasse le nombre de cœurs s'expliquent par le fait qu'on ait aucune garantie que le thread qui récupère le verrou ne soit pas retiré de l'état *Running* par le scheduler (une incertitude et donc un plus grand écart type). Le temps constant et l'absence d'amélioration entre les différentes implémentations du verrou est un peu plus difficile à expliquer. L'explication la plus plausible que l'on ait trouvée est que le programme n'ait pas besoin d'utiliser le bus pour faire sa boucle while dans la section critique. En effet, à l'aide de différents tests, nous avons remarqué qu'il y avait bien une légère diminution de l'utilisation de la commande *xchgl* mais ça ne semble pas avoir d'impact sur le temps d'exécution.

2.2 Le problème des philosophes

Le problème des philosophes consiste à avoir N philosophes, représentés par N threads, qui peuvent soit penser soit manger. Pour manger ils ont besoin d'avoir accès aux deux couverts situés de part et d'autre de leur assiette (un couvert ne pouvant être utilisé que par un philosophe à la fois). Ici nous avons choisi que les actions de penser et de manger étaient instantanées (sans vouloir offenser pour nos amis philosophes).



Librairie lpthread Nous avons mesuré que le temps pris par le programme dédié au problème des philosophes dépend linéairement du nombre de threads. Cela est dû au fait que la partie du programme qui prend le plus de temps est l'attente des couverts. En effet, pour avoir la priorité sur un verrou, le thread va attendre que le thread voisin l'ait unlock, thread voisin qui attend lui-même que son voisin unlock un autre verrou, etc.

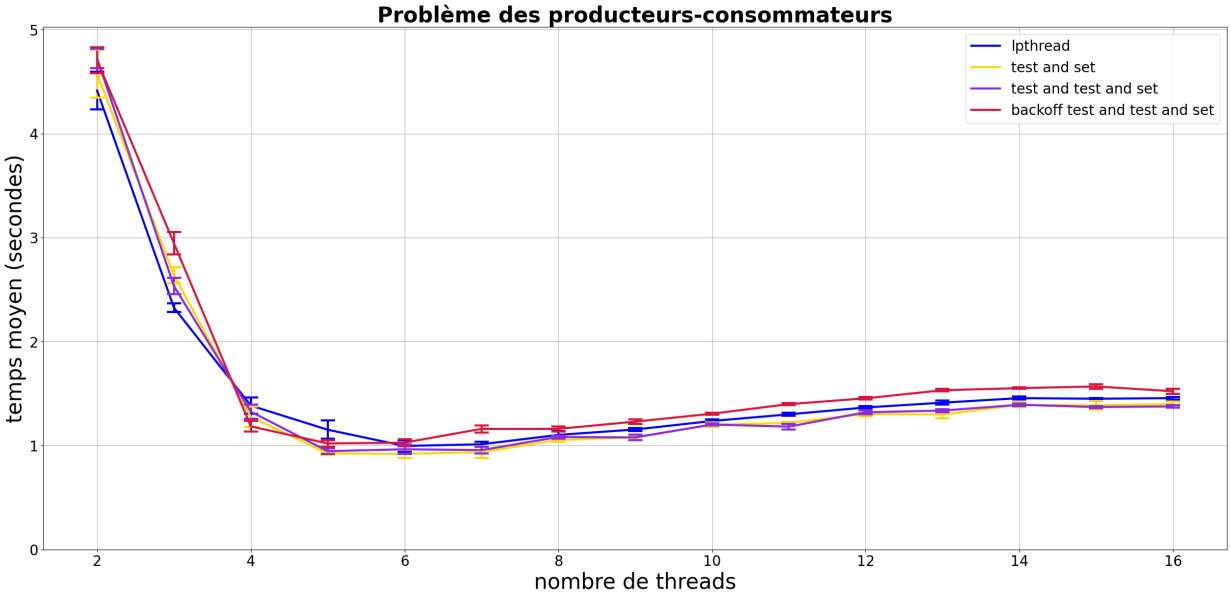
Nos verrous On constate de légères différences [1] au niveau des performances, ce qui nous indique que la méthode de test des verrous n'était pas forcément la plus appropriée pour faire ressortir leurs différences de performances. On remarque que les verrous avec backoff sont globalement les plus performants. Cela est dû au fait qu'avec un backoff, un thread qui vient de passer prioritaire sur le verrou de son premier couvert a beaucoup plus de chance d'avoir

1. Encore plus impressionnantes à plus grande échelle. cf. la section 3

accès rapidement à son deuxième couvert. En effet, il doit attendre moins de temps avant de constater si le verrou de son autre couvert est libre par rapport à un autre thread qui attend depuis plus longtemps. Sans backoff, ce thread pourrait (ou non d'où l'incertitude) attendre assez longtemps pour être retiré de l'état *Running* par le scheduler. On aurait donc un verrou bloqué par un thread qui est en attente, ce qui fait baisser les performances.

2.3 Le problème des producteurs-consommateurs

Le problème des producteurs-consommateurs consiste à avoir un certain nombre de producteurs de ressources qui transmettent ces dernières à des consommateurs via un buffer. Pour simuler une production ou une consommation d'une ressource, nous faisons une boucle *while(rand() < RAND_MAX/10000)* en dehors de la zone critique. En augmentant le nombre de threads, on augmente le nombre de consommateurs et de producteurs (on a le même nombre de consommateurs et producteurs sauf si on a un nombre impair de threads alors on a un consommateur de plus) mais pas le nombre de données produites et consommées qu'on a fixées à 1024.



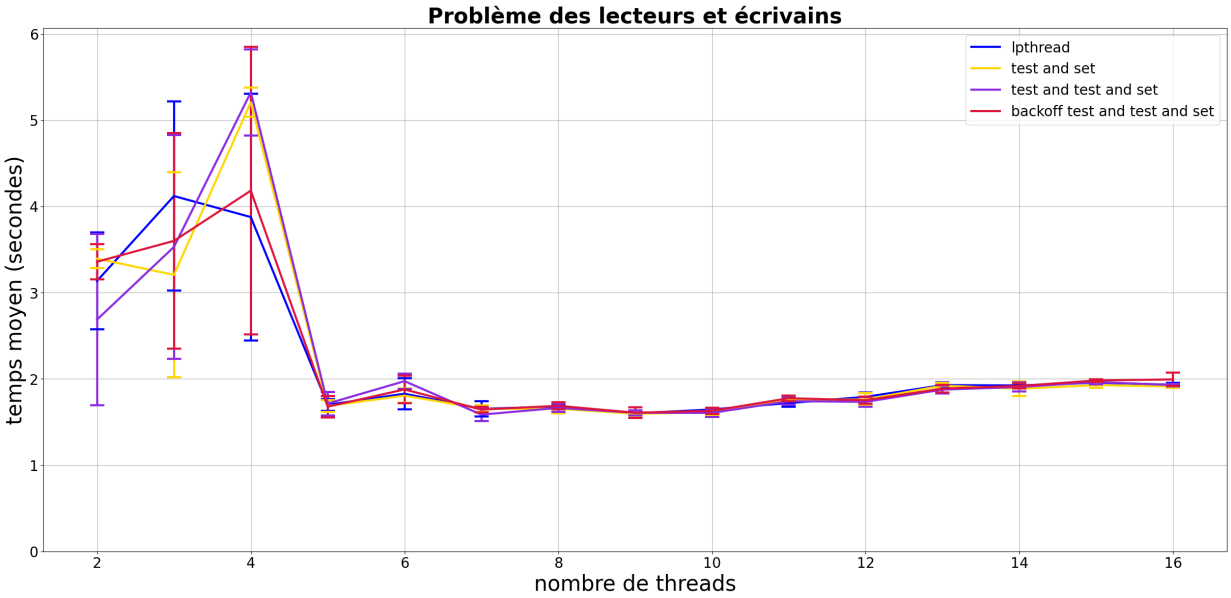
Librairie lpthread Théoriquement, on s'attend à obtenir des temps qui diminuent de manière inversement proportionnelle par rapport au nombre de threads jusqu'à atteindre le nombre de cœurs de la machine. En effet, lorsqu'on a un nombre pair de threads, on a un travail total constant à accomplir mais il est réalisé par de plus en plus de threads. Si on regarde de plus près le cas des nombres de threads impairs, on pourrait se dire que comme les producteurs et les consommateurs ont la même vitesse en moyenne, avoir un consommateur en plus est inutile. Cependant, comme cette vitesse est aléatoire, il est possible que les consommateurs doivent attendre les producteurs pendant quelques instants. Voilà pourquoi on peut quand même s'attendre à des améliorations pour les nombres impairs de threads (par rapport au nombre pair qui le précède). Étant donné que la section critique est relativement petite par rapport aux actions (produire/consommer), on peut négliger les pertes de temps liées au fait qu'un thread en attende un autre à cause d'un sémaphore.

En pratique, on remarque que nos prévisions sont plus ou moins correctes exceptions faites de l'amélioration entre 2 et 3 threads qui est plus impressionnante que prévu. Mais on a remarqué à l'aide de tests supplémentaires qu'il y avait beaucoup plus de moments où l'unique consommateur devait attendre le producteur par rapport à ce que l'on pensait. De plus, les performances arrêtent de s'améliorer à 7 threads plutôt que 8. C'est sans doute dû au fait que notre programme ne peut pas utiliser tous les cœurs de la machine. On voit aussi qu'au lieu d'avoir un temps constant au-dessus de 8 threads, on prend légèrement plus de temps, probablement pour les mêmes raisons que pour le problème des philosophes.

Nos verrous On ne voit pas de différences notables entre nos verrous. Cela s'explique par le fait que le temps d'attente ici est en moyenne bien plus grand que pour les autres problèmes, ce qui annule les effets positifs de nos implémentations comme le backoff (qui fonctionne pour des temps d'attente d'un ordre de grandeur bien plus faible qu'ici).

2.4 Le problème des lecteurs et écrivains

Ce problème consiste à avoir un certain nombre de lecteurs et d'écrivains qui veulent lire ou écrire sur un document. Si les lecteurs peuvent lire à plusieurs en même temps, les écrivains doivent être seul pour écrire (pas de lecteur ni d'autre écrivain). Dans notre implémentation les écrivains sont prioritaires. Pour symboliser une action d'écriture, un thread fait une boucle *while(rand() < RAND_MAX/10000)* puis lit instantanément sur le document. Idem pour la lecture avec les étapes inversées. Nos écrivains (resp. lecteurs) , tous ensemble, font au total 640 (resp. 2560) écritures (resp. lectures).



Librairie lpthread Étant donné que la section critique est très courte par rapport aux actions lire/écrire, on devrait avoir un temps qui décroît de manière inversement proportionnelle au nombre de threads pour les mêmes raisons qu'au problème précédent. En pratique, on a pas du tout ça [2] : on a un temps qui augmente de manière assez aléatoire quand on passe à 2 lecteurs puis on obtient quelque chose de constant entre 5 et 16 threads.

Malgré nos recherches, nous ne sommes toujours pas certains de la raison de ces étranges résultats. Premièrement, nous avons séparé le problème en 2 : nous avons d'abord une période où les threads lecteurs et écrivains travaillent en même temps puis il n'y a plus que des lecteurs. On a alors remarqué que le temps de la première période évoluait comme on le souhaitait (±constante/x puis constant à partir de 8 threads). La partie que nous ne comprenons pas vient de la deuxième période : on a un temps beaucoup plus grand avec 2 lecteurs qu'avec un seul puis ça redevient constant avec 3 lecteurs. A priori ça ne devrait pas arriver vu que les lecteurs ne sont pas censés se bloquer les uns les autres et qu'il n'y a plus d'écrivains. On s'est alors dit qu'il y avait peut-être quelque chose dans notre manière d'utiliser les verrous qui faisait que ce n'était pas du tout optimisé pour et uniquement pour 2 lecteurs seuls. Avec des tests et grâce à nos propres verrous, on a vu que ce n'était pas le cas car l'utilisation des verrous reste sensiblement la même entre 1 et 8 threads lecteurs seuls.

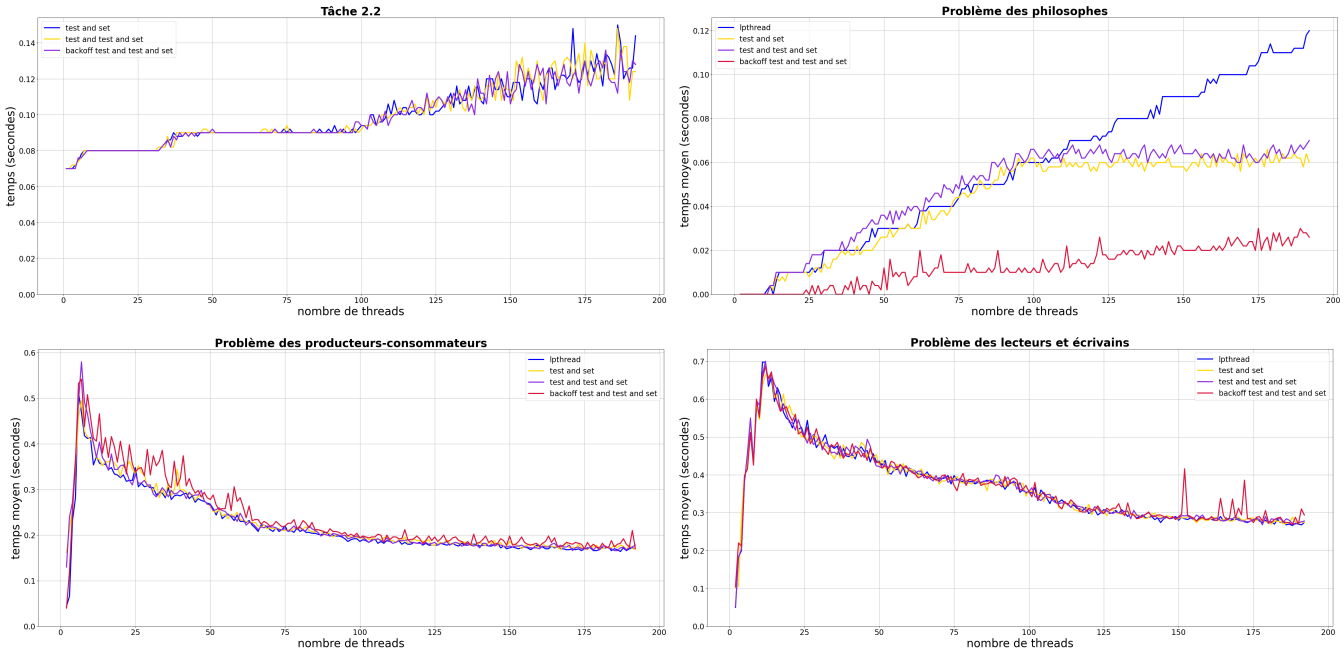
Nos verrous De la même manière que pour le problèmes des producteurs-consommateurs, nos améliorations de verrous produisent des diminutions de temps quasi nulles voire contre-productive à cause des longs temps d'attente et du faible nombre de lock/unlock.

3 Pushing the envelope

Le réveil du Titan En temps d'aventuriers, nous nous promenions il y a quelques jours dans une grotte Irlandaise et nous sommes passés très près d'une bête endormie... Trop près ! Nous avons réveillé le monstre aux 96 cœurs [3] !

2. Il faut attendre une plus grande échelle pour effectivement observer ce comportement
3. instance c5a.24xlarge du service EC2 d'AWS

Après un combat acharné, nous avons réussi à le maitriser et un marché fût conclus : on lui rendit sa liberté et en échange il nous prêta sa puissance pour notre projet.



On remarque qu'à une si grande échelle, nos prédictions théoriques sont mieux vérifiées. Mais on pourrait faire un second rapport entier sur ces résultats tellement intéressants, sur les paliers de temps lpthread dans le problème des philosophes (probablement témoins d'une grande stabilité), sur l'épatante efficacité du verrou backoff pour les philosophes, sur le fléchissement au niveau du nombre de cœurs pour le problèmes des lecteurs-écrivains,...
Notons tout de même que quelques changements ont été réalisés (principalement sur les RAND_MAX et sur la taille du buffer des producteurs consommateurs) dans les codes afin de pouvoir faire tourner notre projet en un temps raisonnable (moins d'une heure).

4 Conclusion

De nombreux enseignements peuvent être tirés de ce projet. Nous avons été frappé par le caractère changeant de nos prédictions théoriques, parfois très précises (e.g. les petites oscillations entre les threads pairs et impairs des producteurs-consommateurs) parfois complètement inexactes (e.g. l'inexactitude complète dans les prédictions générales des lecteurs-écrivains). Nous nous attendions à devoir rédiger un rapport très théorique, mais nous avons dû nous adapter au vu des résultats obtenus en essayant de trouver des explications rationnelles à nos mesures.
Bien que nous sommes globalement satisfaits des résultats obtenus, il est dommage que nous n'ayons pas trouvé d'explication satisfaisante au comportement des lecteurs-écrivains.
De plus, nos mesures nous ont poussées à réaliser de nombreux tests jusqu'à arriver à une meilleure compréhension de nos résultats. Nous avons ensuite lancé notre projet sur une machine bien plus puissante, pour retomber sur nos résultats théoriques. À plus grande échelle, les cas particuliers et les détails des cas limites s'estompent pour donner une vision avec plus de recul sur les problèmes analysés.
On pourrait dire que la théorie nous permet de comprendre un phénomène dans sa globalité mais que seule l'expérience et la pratique révèlent les détails inattendus et les cas particuliers de certains problèmes.