PROJET

Programmation système

Samy Metadjer
Mohammed Ansari

Master informatique 2017-2018

SOMMAIRE

CONCEPTION
EXPLICATION

Le but du projet est de concevoir un scheduler qui permettrait de découper un processus en plusieurs sous-tâches, stockées dans une structure de type LIFO ou Deque. Ces sous-tâches seront ensuite partagées entre différents threads, ce qui réduira le temps d'exécution.

Le processus utilisé dans notre projet est le quicksort. Fondé sur la méthode de conception « diviser pour régner », ce tri donne lieu à des sous-tâches lors de chaque partition.

Pour concevoir ce projet, la première étape fut d'implémenter la pile et la deque qui vont accueillir les taches et leurs arguments.

PILE ET WORK SHARING

L'idée de « work sharing » consiste à stocker les différentes sous-tâches dans une pile, et créer des threads qui vont piocher dedans, récupérer une tâche, puis l'exécuter.

Si cette exécution donne lieu à une sous-tâche, elle sera elle aussi stockée dans la pile, et sera exécutée à son tour par un thread.

L'implémentation de la pile est très simple, c'est une liste chainée de sous-structures contenant une tâche et ses arguments associés. On a une méthode permettant d'allouer la pile, une pour push dedans, et une pour pop un élément de cette dernière.

Les threads s'exécutant en concurrence, l'accès à cette pile doit être sécurisé. En effet, il faut faire attention à ce que plusieurs threads n'essayent pas d'y push ou pop un élément au même moment. Pour parer à ce problème, nous avons dû utiliser des « mutex ».

« MUTEX »

Deux mutex ont été utilisés dans cette implémentation, un pour l'accès à une variable permettant de savoir si des threads sont toujours en exécution, mais nous reviendrons à cette variable plus tard. Et un pour l'accès à la pile : il faut lock le mutex pour accéder à la pile, ce qui bloquera les autres threads qui n'ont pas le mutex, et nous pourrons ainsi agir sur la pile sans danger.

La variable permettant de savoir si des threads sont en exécution permet de créer une condition d'arrêt de chaque thread :

- *Si un thread est en exécution, il est toujours susceptible de créer une nouvelle tâche.*

Même si la pile de tâche est vide, un thread ne doit pas quitter. Il doit attendre que tous les threads finissent l'exécution d'une tâche en cours. Les conditions d'arrêts d'un thread sont donc 0 tâche dans la pile et 0 exécution de taches par d'autres threads. Lorsque tous les threads ont quittés, on peut conclure que l'exécution du processus est fini, et que le tableau de base est trié.

DEQUE ET WORK STEALING

L'idée de « work stealing », quant à elle, consiste à stocker les différentes sous-taches dans une deque. Cette fois ci, les threads auront leur propre deque et chaque nouvelle sous-tache ainsi créer par un thread est stocké dans sa propre deque.

Cette deque est implémentée telle une liste doublement chainée. On peut pop et push des deux côtés, en haut et en bas.

Lorsqu'un thread a fini d'exécuter toutes ses tâches, il essaye de « voler » les tâches d'un autre thread.

« VOLER »

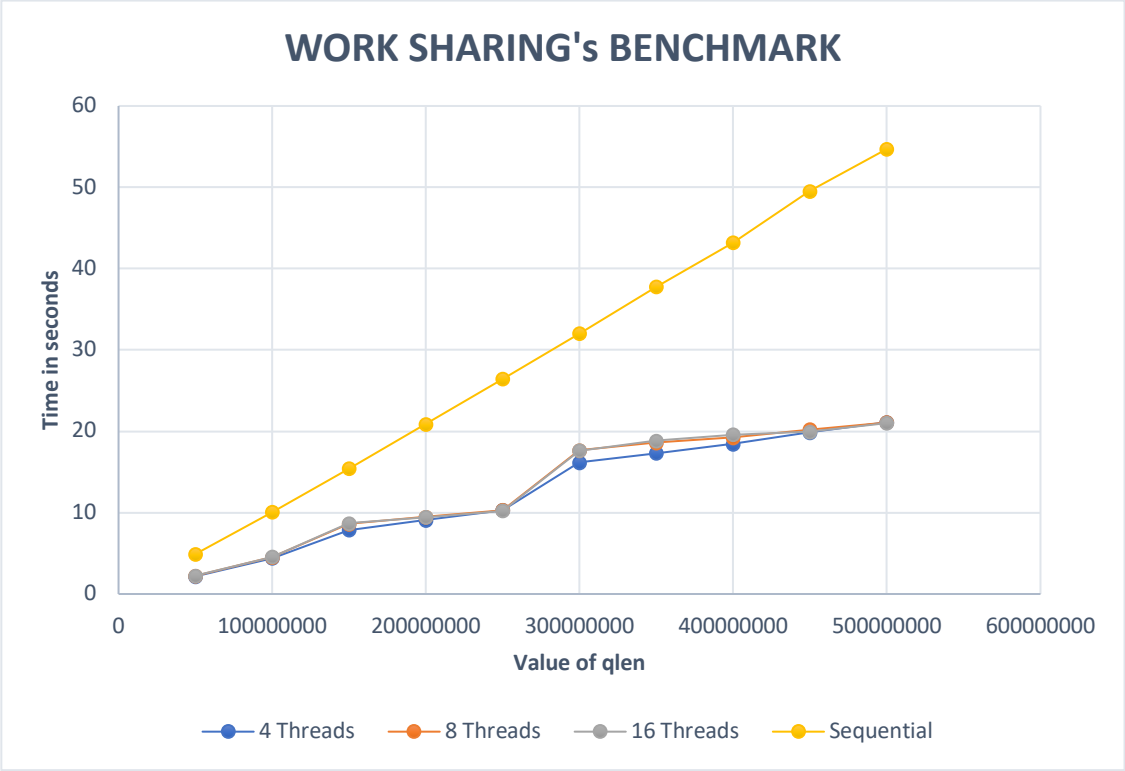
Lorsqu'un thread a fini l'exécution de toutes ses tâches, il va essayer de voler une tâches à un autre thread en essayant d'accéder à sa deque par le bas. Encore une fois, il faut faire attention à la sécurité des deques, et donc ne pas permettre à deux threads de modifier une deque en même temps. Ce pourquoi, chaque deque a un mutex qui lui est propre. Le thread propriétaire fonctionne dans la même idée que la pile, et les threads libres vont eux essayer venir voler une tâche à ce thread. Si le vol est fait dans les règles de l'art, il va exécuter la tâche, sinon il va attendre une milliseconde et va repartir à l'attaque d'une autre deque.

Nous avons fait des tests pour des tableaux de différentes tailles et avec différentes quantités de threads crées pour voir l'efficacité des différentes techniques d'exécutions.

BENCHMARK

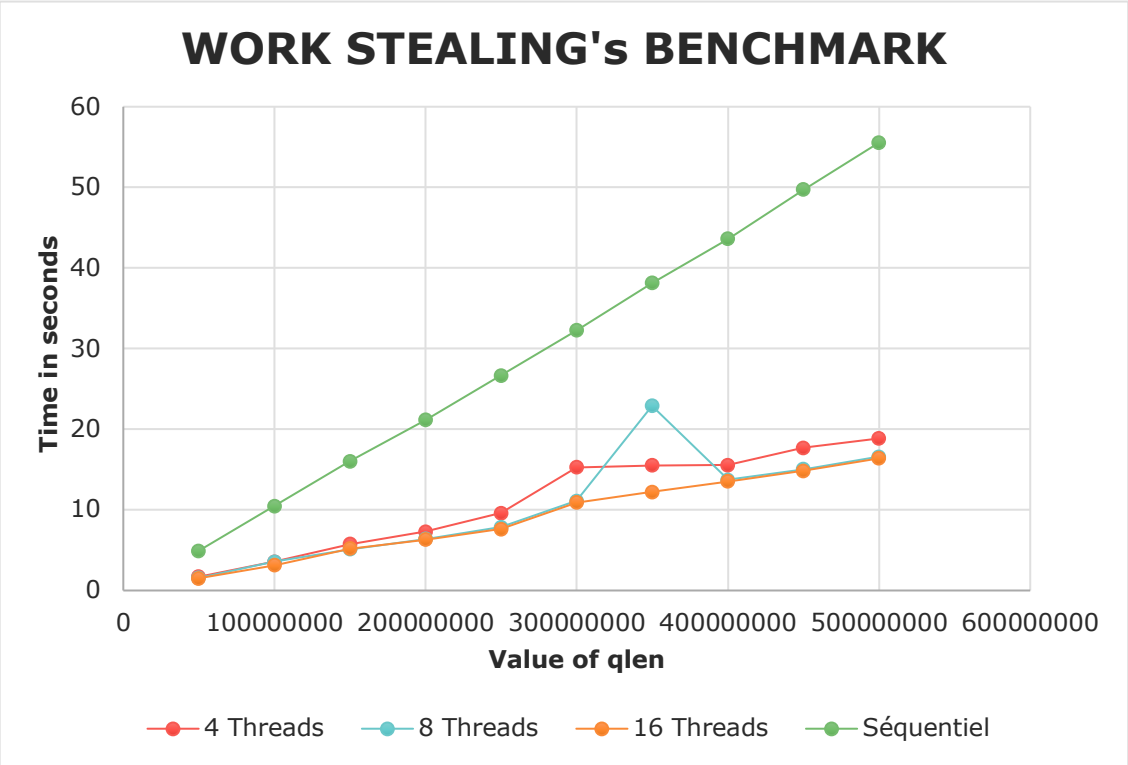
Value of qlen	16 Threads	8 Threads	4 Threads	Sequential
50000000	2,255379	2,230759	2,167087	4,906778
100000000	4,575139	4,5413	4,394616	10,076726
150000000	8,681908	8,632705	7,865779	15,425082
200000000	9,445715	9,484737	9,084463	20,872657
250000000	10,240912	10,297153	10,308225	26,424098
300000000	17,640994	17,684048	16,161677	32,02824
350000000	18,855986	18,600104	17,304653	37,759642
400000000	19,587931	19,223812	18,464272	43,19994
450000000	19,956273	20,211633	19,872616	49,546871
500000000	21,005385	21,079543	21,11698	54,709522

Work Sharing



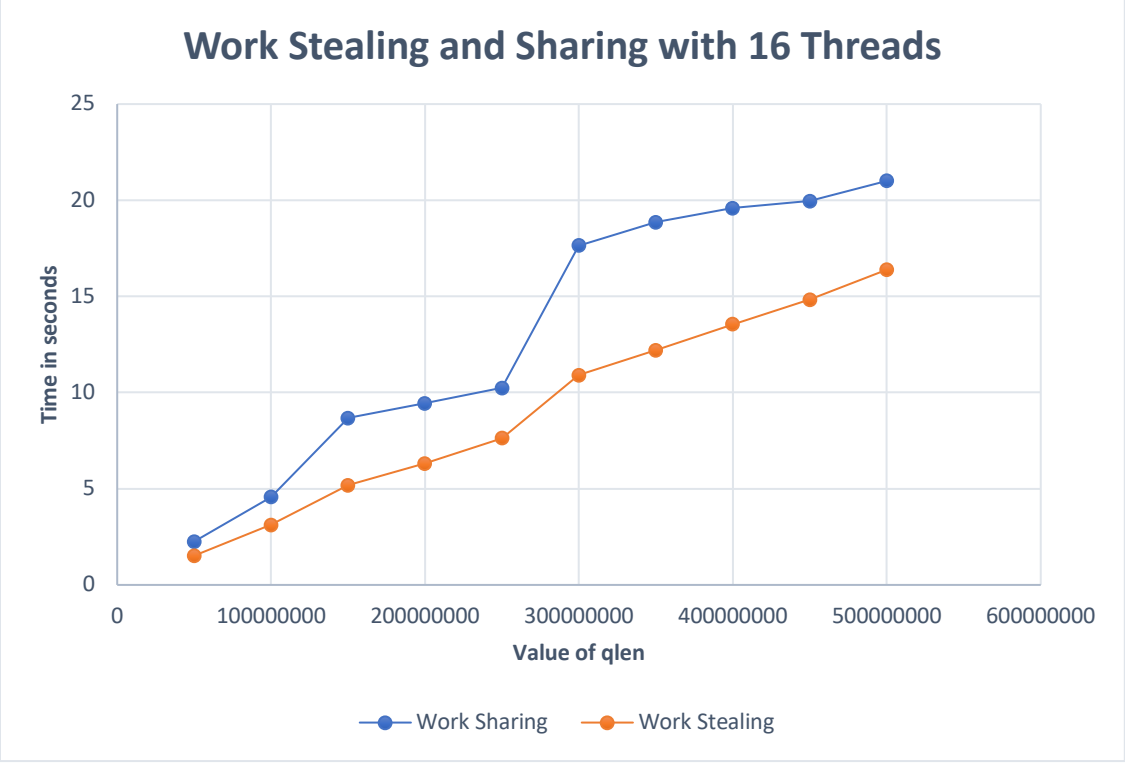
Value of qlen	4 Threads	8 Threads	16 Threads	Sequential
50000000	1,704184	1,562881	1,519636	4,935509
100000000	3,584241	3,567231	3,11488	10,470009
150000000	5,769258	5,122411	5,186517	16,02367
200000000	7,320783	6,402961	6,309251	21,153895
250000000	9,589489	7,83811	7,622674	26,675101
300000000	15,286413	11,112552	10,903765	32,275549
350000000	15,51958	22,893519	12,20956	38,136616
400000000	15,545933	13,731171	13,547363	43,626961
450000000	17,71376	15,026512	14,834174	49,712723
500000000	18,859127	16,592349	16,388741	55,552409

Work Stealing



Value of qlen	Work Stealing	Work Sharing
50000000	1,519636	2,255379
100000000	3,11488	4,575139
150000000	5,186517	8,681908
200000000	6,309251	9,445715
250000000	7,622674	10,240912
300000000	10,903765	17,640994
350000000	12,20956	18,855986
400000000	13,547363	19,587931
450000000	14,834174	19,956273
500000000	16,388741	21,005385

Comparaison entre Work stealing et Work Sharing



CONCLUSION

Les benchmarks parlent d’eux même. Comparé à une exécution séquentielle, les gains en performances sont énormes. Et dans une suite logique, on peut voir que le work stealing est bien plus performant que le work sharing.