Auteur : WOLFGER Killian Date : 19/11/2019

Compte-rendu du TP C : Accélération du tri rapide

Auto-évaluation

8/10 : Je pense avoir bien réussi l'exercice et les différents problèmes que j'ai rencontré m'ont amené à chercher des solution desquelles j'ai pu beaucoup apprendre.

Pédagogie

Pour moi, le but de l'exercice était de nous apprendre à rendre multi-thread un programme de calcul fonctionnant de manière séquentiel pour l'optimiser sans en perturber le résultat. Cette exercice soulève de nombreux problèmes, comment bien optimiser le temps de calcul, faire attention à l'intégrité des données et à la sortie finale du programme, ou encore quelle est la meilleur manière de gérer un ensemble de thread pour un problème de cet ordre. J'ai tout d'abord eu un peu de mal avec la lecture du sujet, et à me représenter comment l'algorithme originel procédait au tri. Une fois cela fait je me suis mis à chercher dans les exemples fournis une manière de multi-threader correctement le programme. Après avoir implémenté une thread pool j'ai rencontré des problèmes de calculs, en effet le tableau sortit par l'algorithme parallèle différait de l'algorithme séquentiel, je suis resté bloqué un long moment sur cet erreur mais il s'agissait tout simplement d'une tâche que j'avais oublié de comptabiliser (voir conception plus bas). Même sans la trouver, j'avais une idée assez précise d'où se situait l'erreur ou comment elle surgissait grâce aux nombreuses informations que j'affichais dans le terminal, notamment le temps d'exécution des deux méthodes, le nombre de tâches effectués par les threads, ou encore le nombre d'éléments différents entre les deux tableaux.

Choix de conception

Pour déterminer que le tri parallèle du tableau global est terminé j'utilise un AtomicInteger me permettant de comptabiliser le nombre de tâches soumises à l'Executor Completion Service. A chaque fois que ma fonction **_paralTrierRapidement(int[] t, int début, int fin)** est appelée je vérifie si c'est la version séquentielle ou parallèle qui doit être utilisé (en fonction de la taille du sous tableau). Si c'est la version séquentielle je n'incrémente pas mon compteur et je lance la fonction **seqTrierRapidement(int[] t, int début, int fin)** deux fois avec le sous tableau relatif pour chaque appel. Si c'est la version parallèle qui doit être utilisé je rappelle **_paralTrierRapidement(int[] t, int début, int fin)** de la même manière et j'incrémente mon compteur de 2. C'est dans ma fonction principal qui est **paralTrierRapidement(int[] t, int début, int fin)** et qui lance pour la première fois **_paralTrierRapidemen(...)** que je demande à mon Executor Completion Service de take() une tâche tant qu'il en reste au moins une grâce à une boucle while prenant en condition la valeur de mon compteur. Une fois le compteur à 0, c'est à dire quand il n'y a plus aucune tâche à effectuer, le thread "principal" sort de cette boucle, ferme l'executor grâce à la fonction **shutdown()** et assigne à l'Executor Completion Service une valeur null. La fonction se termine alors et le tableau est trié de manière correcte.

Conditions d'observation du gain

Taille utilisée : 95_000_000

Sur 10 tests réalisé, la version parallèle est environ 3.37 fois plus rapide

Spécificités matérielles de la machine :

-Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz

-4 processeurs physiques

-4 processeurs logiques

Choix optimal des paramètres

Pour que le tri des deux morceaux du tableau soit réalisé en parallèle uniquement si leur taille est supérieur à P% du tableau global j'ai rajouté à la fonction **_paralTrierRapidemen(...)** et **paralTrierRapidemen(...)** une variable contenant la valeur de P. Dans le main j'ai effectué deux boucles, l'une procédant à l'itération de P de 0% à 100% par tranches de 10, et dans la seconde à plusieurs lancement de l'algorithme parallèle sur la même valeur de P pour avoir un résultat plus précis. Pour chaque valeur de P j'ai donc effectué une moyenne sur une dizaine de lancement avec le programme parallèle. On peut voir dans le tableau joint en annexe que 10% de la taille maximale est la meilleur valeur en pas de 10 en 10. Grâce à des lancement supplémentaires et plus précis j'ai pu remarquer que l'on atteignait le meilleur résultat aux alentours de P = 5%. Le tableau en annexe ne renseigne cependant que les résultats obtenus par tranches de 10%.

Mesures effectuées afin de déterminer la valeur de P

P	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Durée	3515	3184	3442	3912	4433	4465	4452	4513	7885	8662	10380
Ratio	2.95	3.26	3.01	2.65	2.34	2.32	2.33	2.3	1.31	1.2	1

Ratio = temps d'exécution séquentielle / temps d'exécution parallèle

Durée = temps d'exécution exprimé en millisecondes

P étant plus précisément la valeur dans la condition suivante :

if(partSize > 1000 && ***partSize > taille*P)****)*

{

ecs.submit(() -> _paralTrierRapidement(t, début, p-1, P), null);

ecs.submit(() -> _paralTrierRapidement(t, p+1, fin, P), null);

nbTasks.addAndGet(2);

}