MP :X2I3010

Programmation multi-coeurs

Projet

Mémoire transactionnelle logicielle

Auteurs :
Raphaël Pagé
Glenn Plouhinec

Professeur :
Matthieu Perrin

31 mars 2019

UNIVERSITÉ DE NANTES

1 Introduction

Ce projet du cours de Programmation Multi-coeurs a pour objectif de nous familiariser avec l'algorithme *Transactional Locking 2*. Pour ce faire, nous avons implémenté cet algorithme en Java, puis nous avons conçu une application utilisant cette implémentation. L'application que nous proposons dans ce projet permet d'approximer la valeur de PI en utilisant la méthode de Monte Carlo.

Notre rapport est composé de deux parties. Dans la première partie, nous présentons notre implémentation de TL2 et discutons les différents choix que nous avons pris. Dans une deuxième partie nous présentons comment nous utilisons la méthode de Monte Carlo afin d'approximer PI et comment la mémoire transactionnelle peut être utilisée dans ce cas de figure.

L'ensemble de la programmation est réalisé en Java 8 sous Intellij. La définition de la mémoire transactionnelle logicielle et le fonctionnement de TL2 n'est pas redonné dans ce rapport. Merci de bien vouloir vous référer au sujet de ce projet pour de plus amples informations à ce sujet. Ce rapport est accompagné d'un projet maven (nommé *memoireTransactionnelle*) composé de trois packages. Le package *TL2* contient notre implémentation de TL2. Il est lui-même composé de trois packages, *except* (contenant des exceptions utilisées par les lectures et les commits), *inter* (contenant les interfaces des registres et des transactions) et *impl* (contenant l'implémentation des registres et des transactions).
Le package *applicationtest* contient un code exécutable que nous utilisons afin de tester notre implémentation. Dans celui-ci, un certain nombre de threads tentent de lire et d'additioner le contenu de deux registres partagés, puis de mettre à jour la valeur de ceux-ci.
Le package *montecarlo* contient le code executable de notre approximation de PI par la méthode de Monte Carlo.
Notre projet est accessible sur GitHub [1].

1. `https://github.com/RaphaelPage0110/memoireTransactionnelle`

2 Implémentation de TL2

Nous ne détaillerons que très peu notre implémentation de *TL2*, celle-ci étant assez bien commentée et compréhensible. Nous préférons alors expliquer les plus gros problèmes rencontrés dans l'avancement de ce projet, et les solutions qui en ont découlé. De manière générale, notre implémentation reprends les grandes idées du pseudo-code qui nous était fourni. Un registre contient une valeur, ainsi qu'une date. Il possède également une copie locale de ces deux variables (de type ThreadLocal). Le registre possède notamment les méthodes *read()* et *write(value)*. Une transaction contient la liste des registres sur lesquels a été effectué une opération de lecture, et une autre liste de registres pour les opérations d'écritures ; elle contient également une horloge globale partagée par toutes les transactions du système, et "une date de naissance", initialisée au début de la transaction, lors de l'appel à la méthode *begin()*. La transaction possède alors les méthodes *begin()* et *try_ to_ commit()*.

Lors de notre première implémentation de l'algorithme, nous nous sommes confrontés à un problème d'inter-blocage des registres, amené par la méthode *try_ to_ commit()*. En effet, le thread T_1 qui faisait appel à cette méthode posait un verrou sur tous les registres, dans un ordre quelconque (l'ordre dans lequel il a écrit ou lu dans un registre). Seulement, il suffisait qu'un autre thread T_2 fasse appel à cette même méthode au même moment pour poser un verrou sur un registre que T_1 n'avait pas encore eu le temps de verrouiller, pour bloquer T_1. Ce problème a été résolu en s'inspirant du problème des philosophes vu en TP, nous avons alors ajouté un ordre sur les registres pour les verrouiller et déverrouiller dans l'ordre de leur numéro identifiant.

Le dernier problème en date a été rencontré lorsque nous exécutions un grand nombre de transactions en parallèle sur un registre : des valeurs incohérentes apparaissaient lors de l'essai de la méthode *increment()* proposée dans le sujet. Simplement, notre méthode *begin()* ne réinitialisait pas **toutes** les variables locales, nous faisions uniquement appel à la méthode *clear()* pour vider l'ensemble des registres lu/écrits, mais nous ne réinitialisions pas les valeurs locales de ces registres avant cela. Ce qui avait pour effet de laisser des "résidus" de valeurs qui étaient validées, mais paraissaient alors incohérentes.

3 Approximation de PI par la méthode de Monte Carlo

3.1 La méthode de Monte Carlo

La méthode de Monte Carlo consiste à répéter des expériences à résultat aléatoire un nombre important de fois. Il est ensuite possible d'approximer la probabilité de succès d'une expérience en comptant le nombre d'expériences réussies et en le divisant par le nombre total de répétitions de l'expérience.

Il est possible d'approximer la valeur de PI en utilisant cette méthode pendant l'expérience suivante[2]. Imaginons le jeux de fléchettes illustré par la figure 1.

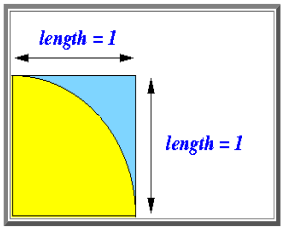

FIGURE 1 – Jeux de fléchette.

Ses dimensions sont de 1×1 et un quart de cercle de rayon 1 est tracé sur celui-ci. L'expérience consiste à lancer une fléchette sur ce jeu. Nous considérons l'expérience comme étant un succès si la fléchette est à l'intérieur du quart de cercle. Puisque l'aire du carré est de 1, la probabilité qu'une fléchette atteigne le quart de cercle est égale à l'aire du quart de cercle, qui est lui-même égal à $\pi/4$.
La méthode de Monte Carlo nous permet d'estimer le taux de succès de cette expérience, et donc d'estimer $\pi/4$.

3.2 Implémentation de l'expérience

Afin de simuler cette expérience, nous allons affecter à un certain nombre de threads (choisi par les utilisateurs) la tâche de lancer une fléchette et de déterminer si celle-ci est atterrie dans le quart de cercle. Si c'est le cas, ce thread incrémentera la valeur d'un registre partagé (initialisé à 0) en utilisant notre implémentation de TL2.
Un lancer de fléchette est simulé en affectant une valeur aléatoire (comprise entre 0 et 1) à deux variables X et Y qui représentent les coordonnées à laquelle la fléchette est lancée. Il ne reste plus qu'à déterminer si ces coor-

2. Cette expérience et les images l'accompagnant sont tirées de ce site : `https://bit.ly/2UmJu1e`

données respecte l'inégalité $x^2 + y^2 < 1$ pour déterminer si la fléchette est lancée dans le cercle.

4 Discussion et Conclusion

L'algorithme TL2 garantit des propriétés de sûreté et de vivacité sur l'ensemble des transactions pour garantir la cohérence de la valeur des registres. Pour rappel, la propriété de *sûreté* nous assure que "rien de mal ne se produit", et la *vivacité* assure que "quelque chose de bien finira par arriver".

La *sûreté* d'une transaction est assurée par le fait que, si un autre thread a modifié la valeur du registre depuis la dernière lecture, la transaction est annulée. Ainsi, il est impossible de lire une valeur invalide dans un registre.

Dans *TL2*, la *vivacité* d'une transaction quand à elle, est assurée par le fait qu'un thread qui effectue des opérations de lecture et écriture sur un registre, arrivera toujours à faire appel à la méthode *try_ to_ commit()*, peu importe que celle-ci se solde par un échec ou une réussite. De manière générale, toutes les fonctions de l'algorithme sont *starvation-free*, mais l'algorithme en lui même n'est pas totalement starvation-free dans le sens où, si l'on souhaite faire une transaction, on n'est pas certain que celle-ci sera validée.

Par ailleurs, une transaction T_1 se solde par un échec uniquement si une valeur lue a été modifiée par une transaction T_2 avant que T_1 ne fasse son commit.

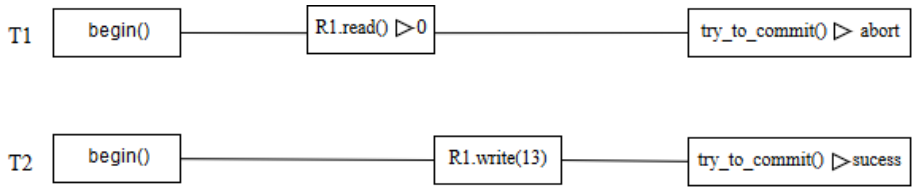

FIGURE 2 – Échec d'une transaction T_1 sur un registre.

Dans les autres cas, les transactions se soldent toujours par une réussite.

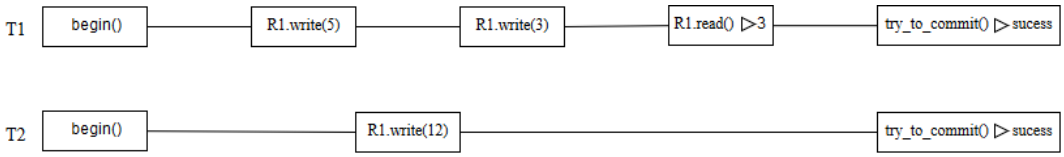

FIGURE 3 – Linéarisabilité des transactions sur un registre, en écriture puis lecture.

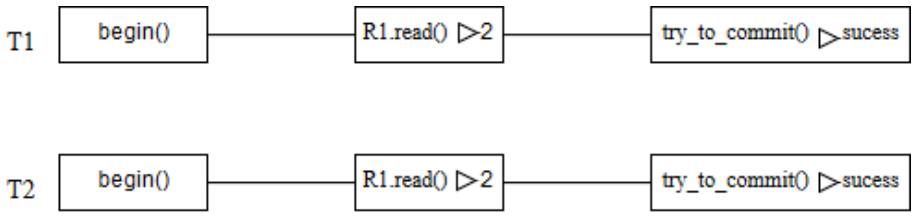

FIGURE 4 – Linéarisabilité des transactions sur un registre, avec des lectures concurrentes.

Finalement, *TL2* nous apporte les propriétés d'atomicité, de cohérence et d'isolation sur les transactions.