Compte rendu TP7 Java

Fabien Simonet, ZZ3 F5, promotion 2021

J'ai utilisé un générateur de données appelé Javafaker. L'instanciation et l'utilisation d'un Faker est relativement longue ce qui permet d'allonger le temps d'exécution de la génération des étudiants afin d'avoir une comparaison plus intéressante entre le temps d'exécution des étudiants sans thread et avec threads.

Une fois la génération d'étudiants implémentées, les premiers tests ont donné les résultats ci-dessous :

```
Generation of 150000 students 15 times without multiple threads.
Execution time of the generation 1: 14.9049674s
Execution time of the generation 2: 15.2988824s
Execution time of the generation 3: 12.958169s
Execution time of the generation 4: 13.6423437s
Execution time of the generation 5: 13.580662s
Execution time of the generation 6: 13.2208292s
Execution time of the generation 7: 13.3429568s
Execution time of the generation 8: 13.2905469s
Execution time of the generation 9: 13.118414s
Execution time of the generation 10: 13.3358739s
Execution time of the generation 11: 12.3985611s
Execution time of the generation 12: 13.0976101s
Execution time of the generation 13: 12.9001618s
Execution time of the generation 14: 12.8690121s
Execution time of the generation 15: 12.8175502s
Execution time average : 13.385102706666665s
Generation of 150000 students 15 times with 8 threads.
Execution time of the threaded generation 1: 2.8833517s
Execution time of the threaded generation 2: 2.9810744s
Execution time of the threaded generation 3: 2.8490522s
Execution time of the threaded generation 4: 2.8518452s
Execution time of the threaded generation 5: 3.0129163s
Execution time of the threaded generation 6: 2.8392522s
Execution time of the threaded generation 7: 2.9331861s
Execution time of the threaded generation 8: 3.8583435s
Execution time of the threaded generation 9: 3.6694168s
Execution time of the threaded generation 10: 3.7266803s
Execution time of the threaded generation 11: 3.6515884s
Execution time of the threaded generation 12: 3.9917093s
Execution time of the threaded generation 13: 3.9339978s
Execution time of the threaded generation 14: 3.6094783s
Execution time of the threaded generation 15: 3.6416114s
Threaded execution time average : 3.3622335933333334s

Process finished with exit code 0
```

On peut voir que la génération d'étudiants avec 8 threads est environ 4 fois plus rapide que la génération non threadée. C'est moitié moins que ce à quoi on pourrait théoriquement s'attendre mais cela me parait tout de même cohérent.

J’ai ensuite ajouté une écriture en base de données, le résultat obtenu est le suivant :

```
Generation of 150000 students 15 times without multiple threads.
Execution time of the generation 1: 648.0461517s
Execution time of the generation 2: 676.3349474s
```

Le temps d’exécution est beaucoup trop long (11 minutes environ) pour la génération non threadée. Je n’ai pas testé la génération threadée car ces résultats me semblaient incohérents. L’écriture en base de données prend du temps mais pas au point de multiplier le temps d’exécution par 50.

En regardant mon code, je me suis rendu compte que je faisais une boucle for sur chacun de mes étudiants et que je les ajoutais en base un à un. J’ai fait quelques recherches et je me suis rendu compte que la connexion Java a une propriété autoCommit qui permet d’activer ou de désactiver le commit automatique. Par défaut, l’autoCommit est activé et se fait donc à chaque ajout d’étudiants. J’ai donc essayé de le désactiver avant d’exécuter mes requêtes d’ajout de chacun des étudiants, de faire le commit manuellement, puis de réactiver l’autoCommit une fois tout ça terminé. Cela permet de ne faire qu’un seul commit plutôt que 150 000 !

Le code ainsi modifié est le suivant :

```java
public void addStudentList(List<Student> studentList) throws Exception {
    connection.setAutoCommit(false);
    PreparedStatement ps = connection.prepareStatement( sql: "insert into student values(?, ?, ?, ?, ?, ?)");

    for (Student student: studentList) {
        addStudent(student);
    }

    connection.commit();
    connection.setAutoCommit(true);
}
```

Les résultats obtenus sont les suivants :

```
Generation of 150000 students 15 times without multiple threads.
Execution time of the generation 1: 12.2090766s
Execution time of the generation 2: 12.210034s
Execution time of the generation 3: 12.0759203s
Execution time of the generation 4: 12.3989099s
Execution time of the generation 5: 12.5627569s
Execution time of the generation 6: 11.868679301s
Execution time of the generation 7: 12.4957881s
Execution time of the generation 8: 11.901961s
Execution time of the generation 9: 12.677375s
Execution time of the generation 10: 12.1539009s
Execution time of the generation 11: 13.6651954s
Execution time of the generation 12: 12.1694806s
Execution time of the generation 13: 13.3471864s
Execution time of the generation 14: 11.9440223s
Execution time of the generation 15: 56.5318434s
Execution time average : 15.3474753400666664s
Generation of 150000 students 15 times with 8 threads.
Execution time of the threaded generation 1: 64.5518967s
Execution time of the threaded generation 2: 63.0981502s
Execution time of the threaded generation 3: 60.106134s
Execution time of the threaded generation 4: 58.1081664s
Execution time of the threaded generation 5: 64.3999941s
Execution time of the threaded generation 6: 58.1401807s
Execution time of the threaded generation 7: 56.2428428s
Execution time of the threaded generation 8: 57.0316206s
Execution time of the threaded generation 9: 61.6846893s
Execution time of the threaded generation 10: 61.7044577s
Execution time of the threaded generation 11: 63.0427051s
Execution time of the threaded generation 12: 63.717547s
Execution time of the threaded generation 13: 59.3066292s
Execution time of the threaded generation 14: 58.4638335s
Execution time of the threaded generation 15: 57.999023701s
Threaded execution time average : 60.50652473339999s
```

On constate que désormais, l'ajout des étudiants en base de données est quasiment instantané, mise à part la dernière itération, mais que la génération threadée prend 20 fois plus de temps qu'auparavant, alors que je ne fait pas l'ajout en base de données sur la génération non threadée.

En commentant la génération non threadée et n'exécutant que la génération threadée (avec persistence), on obtient le résultat suivant :

```
Generation of 150000 students 15 times with 8 threads.
Execution time of the threaded generation 1: 5.4719498s
Execution time of the threaded generation 2: 5.0332251s
Execution time of the threaded generation 3: 4.8957742s
Execution time of the threaded generation 4: 5.545841s
Execution time of the threaded generation 5: 5.4097413s
Execution time of the threaded generation 6: 5.7463535s
Execution time of the threaded generation 7: 4.637998s
Execution time of the threaded generation 8: 5.5168662s
Execution time of the threaded generation 9: 4.9030737s
Execution time of the threaded generation 10: 5.8818536s
Execution time of the threaded generation 11: 4.7485761s
Execution time of the threaded generation 12: 5.8084275s
Execution time of the threaded generation 13: 4.799097s
Execution time of the threaded generation 14: 4.5480047s
Execution time of the threaded generation 15: 53.2175554s
Threaded execution time average : 8.410955806666667s
```

Il semble donc se passer quelque chose sur la dernière itération. Par manque de temps, je ne suis pas allé plus loin dans mes recherches. Mais j'ai appris pas mal d'enseignements de ce tp. Le premier est qu'il y a des différences entre la théorie et la pratique concernant les threads et ce n'est pas parce qu'on à n threads que l'on divise forcément le temps par n. Je me suis aussi rendu compte que la parallélisation des tâches ne règle pas tous les problèmes et qu'il est parfois judicieux d'optimiser d'autres parties du codes plutôt que de paralléliser, comme par exemple les accès à la base de données. Enfin, je n'ai pas pris le temps d'essayer de paralléliser les accès à la base de données, mais je suis curieux de savoir s'il est possible de le faire, sachant que cela risque de poser des problèmes d'accès concurrent à la base en pagaille.

Dans le code que je vous envoie, j'ai volontairement réduit le nombre de générations d'étudiants (5 fois 150 000 au lieu de 15 fois 150 000) et commenté le code relatif aux écritures en base de données. Si vous souhaitez tester l'utilisation de la base, il suffit de décommenter le code.