Java Performance - TP4

November 23, 2018

Thomas COUCHOUD
thomas.couchoud@etu.univ-tours.fr
Victor COLEAU
victor.coleau@etu.univ-tours.fr

Chapter 1

Introduction

Dans ce TP nous allons nous attaquer à de petits exercices dont le code est très peu performant afin d'étudier les possibilités d'amélioration.

Ce document présente les différentes actions mises en place ainsi que leur résultats.

Chapter 2

Exercices

2.1 Exercice 1

On remarque que beaucoup de types utilisés des classes (Float, Integer, ...), il serait donc peut-être plus efficace d'utiliser des types primitifs directement afin d'éviter d'inutiles opérations d'unboxing/boxing.

De plus nous avons changé les "k = k + 1" en "k++". Cependant cela n'aura surement aucun effet sur les performances car le nombre d'opérations est le même. Il est à noter que le compilateur le change en ++k, évitant ainsi de stocker une valeur intermédiaire pour la renvoyer par la suite.

Après des mini benchmarks nous obtenons la sortie suivante:

Listing 2.1 – benchmark.txt

```
Benchmark         (i)   Mode  Cnt       Score       Error  Units
Exercice1.test      0  thrpt   10   70006.601 += 1883.562  ops/s
Exercice1.test      1  thrpt   10   71369.058 += 1905.356  ops/s
Exercice1.testA     0  thrpt   10  457780.134 += 6152.443  ops/s
Exercice1.testA     1  thrpt   10  467063.858 += 7418.504  ops/s
```

Avec nos améliorations on peut constater que le score est multiplié par environ 5. Cela est probablement du aux changement des types plus qu'à la transformation des incrémentations.

2.2 Exercice 2

La première remarque est que dans le calcul d'une valeur de fibonacci, le calcul des deux précédentes est nécessaire. Or dans l'implémentation donné fibonacci(i-1) et fibonacci(i-2) sont indépendantes alors que fibonacci(i-1) utilise et donc recalcule lui-même fibonacci(i-2). De ce fait, le calcul de fibonnaci(n) demande le calcul de 2^n fois fibonnacci(1) + fibonacci(2). Une grande partie de ces calculs sont inutiles et peuvent être très long sur des grandes valeurs.

Nous allons donc mettre en place un système de cache des valeurs au travers d'une map (fibonaciA). Celle-ci aura pour but de sauvegarder les valeurs une fois calculées et de les renvoyer à la place de faire le calcul les fois suivantes.

De plus, étant donné que l'on demande la valeur de fibonacci(43), nous pouvons aussi renvoyer la valeur directement en l'ayant calculée auparavant (fibonacciB).

Listing 2.2 – benchmark.txt

```
Benchmark         (i)   Mode  Cnt         Score          Error  Units
Exercice2.test     43  thrpt   10           ~= 0                ops/s
Exercice2.testA    43  thrpt   10  24448866.918 += 2929078.246  ops/s
Exercice2.testB   N/A  thrpt   10  73504959.848 +=  747624.575  ops/s
```

L'amélioration est ici flagrante avec le cache, et encore plus avec la valeur renvoyée directement. Avec le cache cela s'explique car une très grosse majorité des calculs sont évités. Dans le cas de la valeur directe, il ne s'agit en réalité que d'un appel de getter.

2.3 Exercice 3

Dans les deux exercices suivants, nous n'avons pas pu comparer avec les méthodes de base car ces dernières font crasher la VM. Nous pouvons cependant déduire les potentielles améliorations grâce au temps d'exécutions des tests unitaires. En effet les méthodes de base durent environ 1m30 alors que les versions améliorées sont aux alentours de 30s (pour 3a) et 1m (pour 3b).

2.3.1 Exercice 3a

Dans une première méthode "A" nous avons:

1. Retiré le synchronized de la méthode. En effet on a déjà un mutex à l'intérieur de la fonction et il n'est donc pas nécessaire d'en avoir deux. De plus le synchronized sur la méthode bloque toutes les autres méthodes synchronized ce qui n'est pas optimal car ralenti l'ensemble des appels à l'objet.

2. Changé les Integer en int afin d'éviter les boxing/unboxing inutiles.

3. Ajout direct du future dans la liste au lieu de la création d'un objet intermédiaire.

4. Shutdown de l'ExecutorService car il n'est plus nécessaire d'ajouter de nouveaux jobs par la suite.

Une deuxième méthode "B" reprend les modifications de la méthode "A" mais utilise cette fois-ci un AtomicInteger afin de retirer le mutex dans la fonction d'incrémentation, la gestion de la concurrence étant directement faite au sein de la méthode incrementAndGet.

Les résultats sont les suivants:

Listing 2.3 – benchmark.txt

```
Benchmark          (i)    (j)   Mode  Cnt    Score     Error  Units
Exercice3a.testA    10     10  thrpt   10  299.473 += 104.322  ops/s
Exercice3a.testA    10  10000  thrpt   10  271.266 +=  58.819  ops/s
Exercice3a.testA   100     10  thrpt   10   29.685 +=   3.044  ops/s
Exercice3a.testA   100  10000  thrpt   10   27.533 +=   0.929  ops/s
Exercice3a.testB    10     10  thrpt   10  361.475 +=  12.941  ops/s
Exercice3a.testB    10  10000  thrpt   10  200.724 +=  24.818  ops/s
Exercice3a.testB   100     10  thrpt   10   30.502 +=   2.522  ops/s
Exercice3a.testB   100  10000  thrpt   10   14.917 +=   3.028  ops/s
```

On voit que dans le cas de très petites instances (10 threads, 10 incrémentations), le AtomicInteger est légèrement plus efficace que le mutex. Cependant dès que la charge de travail par thread augmente un petit peu (10 threads, 10000 incrémentations), le mutex reprend l'avantage.

Dans le cas de très grosses instances (100 threads, 10000 incrémentations), le mutex domine (jusqu'à deux fois plus efficace).

2.3.2 Exercice 3b

Nous avons ici repris les mêmes idées que dans l'exercice 3a cependant nous avons du changer le incrementAndGet du AtomicInteger en accumulateAndGet afin d'y ajouter le calcul du modulo.

Les résultats sont les suivants:

Listing 2.4 – benchmark.txt

```
Benchmark          (i)    (j)    (k)   Mode  Cnt    Score     Error  Units
```

```
Exercice3b.testA   10     10     10  thrpt   10  365.719 += 14.945  ops/s
Exercice3b.testA   10     10  10000  thrpt   10  372.611 +=  5.261  ops/s
Exercice3b.testA   10  10000     10  thrpt   10  118.434 +=  3.041  ops/s
Exercice3b.testA   10  10000  10000  thrpt   10   34.882 +=  0.916  ops/s
Exercice3b.testA  100     10     10  thrpt   10   31.480 +=  1.349  ops/s
Exercice3b.testA  100     10  10000  thrpt   10   31.489 +=  1.144  ops/s
Exercice3b.testA  100  10000     10  thrpt   10   11.943 +=  0.485  ops/s
Exercice3b.testA  100  10000  10000  thrpt   10    3.262 +=  0.122  ops/s
Exercice3b.testB   10     10     10  thrpt   10  367.267 +=  8.472  ops/s
Exercice3b.testB   10     10  10000  thrpt   10  368.095 += 13.389  ops/s
Exercice3b.testB   10  10000     10  thrpt   10   45.170 +=  4.755  ops/s
Exercice3b.testB   10  10000  10000  thrpt   10   43.152 +=  3.459  ops/s
Exercice3b.testB  100     10     10  thrpt   10   31.768 +=  1.219  ops/s
Exercice3b.testB  100     10  10000  thrpt   10   31.432 +=  1.360  ops/s
Exercice3b.testB  100  10000     10  thrpt   10    4.452 +=  0.385  ops/s
Exercice3b.testB  100  10000  10000  thrpt   10    4.042 +=  0.291  ops/s
```

La première chose que l'on remarque est que la méthode avec l'AtomicInteger n'est que peu sensible à la variation du modulo (k). Par exemple dans le cas de i=10/j=10000, que le modulo soit à 10 ou 10000 le score est d'environ 43. En revanche la méthode avec le mutex y est bien plus sensible: passage de 118 à 34 ops/s.

Concernant la comparaison entre A et B, on remarque que les valeurs se ressemblent fortement à l'exception du cas 10/10000/10 où la méthode A est bien meilleure.

2.4 Exercice 4

La première chose que nous remarquons est que l'on fait une boucle for pour ajouter tous les éléments d'un tableau.

La première méthode consistant à utiliser le .addAll de la liste (exercice4A). Cependant cela nous a forcé à convertir pleins de types ce qui ne va probablement pas être optimal.

Une deuxième méthode a été d'utiliser directement les streams avec un flatMap (exercice4C), cependant encore une fois cela implique de faire des conversions de type.

Notre dernière méthode consiste à d'abord calculer la taille du tableau final, puis le remplir directement (exercice4B). Cela évite d'ajouter une fois dans une liste, puis d'ajouter dans un tableau.

Listing 2.5 – benchmark.txt

```
Benchmark                                      (i)   Mode  Cnt        Score
   Error  Units
Exercice4.test                  0123456789abcdef  thrpt   10  1261932.848 +=
   62680.505  ops/s
Exercice4.test   00112233445566778899AABBCCDDEEFF  thrpt   10   711723.644 +=
   9534.811  ops/s
Exercice4.testA                 0123456789abcdef  thrpt   10   616737.894 +=
   17372.461  ops/s
Exercice4.testA  00112233445566778899AABBCCDDEEFF  thrpt   10   384871.426 +=
   10044.636  ops/s
Exercice4.testB                 0123456789abcdef  thrpt   10  2476346.319 +=
   1193288.087  ops/s
Exercice4.testB  00112233445566778899AABBCCDDEEFF  thrpt   10  1852733.463 +=
   854509.849  ops/s
Exercice4.testC                 0123456789abcdef  thrpt   10   847892.612 +=
   40829.721  ops/s
Exercice4.testC  00112233445566778899AABBCCDDEEFF  thrpt   10   689494.593 +=
```

```
   18839.536  ops/s
```

Le A est environ deux fois plus lent que l'original. Ce qui prouve que notre conversion de types n'est pas efficace du tout et qu'il vaut mieux travailler directement des bytes.

Le C est environ 1,5 fois plus rapide. En effet comparé à la méthode A, on change toujours le type mais on s'affranchit de la liste intermédiaire et construisons directement le tableau final.

Enfin la méthode B est la meilleure en étant 2 à 3 fois plus rapide que l'originale. Cela est du au fait que l'on créé directement le tableau final mais cette fois sans aucune conversion de type.

2.5 Exercice 5

Ici la réfléxivité est utilisée sur un objet dont on connait déjà le type et qui contient déjà la méthode à appeler. Tout cet enchainement est inutile car nous pouvons appeler le .getName() directement sur notre objet. Cela va nous permettre de s'affranchir de la recherche de la méthode et invokation de cette dernière.

Listing 2.6 – benchmark.txt

```
1 Benchmark             (i)   Mode  Cnt         Score          Error  Units
2 Exercice5.test     Thomas  thrpt   10   1257489.509 +-   68643.635  ops/s
3 Exercice5.test    Clement  thrpt   10   1261393.467 +-   55132.028  ops/s
4 Exercice5.test      Louis  thrpt   10   1220180.478 +-  185204.978  ops/s
5 Exercice5.testA    Thomas  thrpt   10  57787733.041 +-  329682.747  ops/s
6 Exercice5.testA   Clement  thrpt   10  57331680.503 +- 1011760.686  ops/s
7 Exercice5.testA     Louis  thrpt   10  57808907.426 +-  332110.507  ops/s
```

Encore une fois on remarque que ces changements ont grandement augmenté les performances.

Chapter 3

Conclusion

Ce TP nous a permis de découvrir certaines pratiques heurtant les performances Java. On peut notamment citer le boxing/unboxing qui, à grande échelle, affecte grandement l'efficacité d'un programme et n'est donc pas à sous-estimer.

Grâce à ce TP nous avons pu prendre conscience d'une partie de l'étendue des possibilités d'amélioration, même les plus inattendues, tout en estimant l'impact qu'elles peuvent avoir.