Programmation Multi-cœurs
Le patron de conception « Thread Pool »

Le but de cet exercice est d'introduire un patron de conception adapté spécifiquement au parallélisme : les thread pools. À la fin de ce travail, vous serez en mesure de :

— choisir et utiliser un service d'exécution adapté pour votre programme ;
— identifier le nombre optimal de threads à lancer en parallèle selon votre configuration ;
— implémenter un service d'exécution en Java.

Le temps nécessaire pour effectuer ce travail est estimé entre deux et quatre heures.

N'hésitez pas à partager votre expérience entre vous, et de poser des questions pendant les heures de présentiel ou par e-mail. Le travail (code Java et rapport d'environ une à deux pages) doit être rendu sur Madoc avant la troisième séance de TP.

1 Introduction

L'interface historique de Java pousse à identifier fortement les tâches à exécuter (implémentant l'interface `Runnable`) et les threads qui les exécutent. Quand le nombre de tâches à exécuter est grand, lancer un thread par tâche peut mener à des surcoûts importants, ce qui risque de dégrader les performances de votre programme. Comme un espace de mémoire est alloué spécifiquement à la pile de chaque thread, lancer et exécuter un thread n'est pas anodin. De plus, le temps de commutation de contexte augmente quand le nombre de threads dépasse le nombre de cœurs disponibles.

Une meilleurs solution est de lancer un nombre fixe de threads au début de l'exécution de votre programme, gérés par un thread pool. Dès qu'un thread est inactif, le thread pool lui affecte une nouvelle tâche à exécuter. Les threads inactifs sont mis en attente jusqu'à ce que de nouvelles tâches soient disponibles.

Les thread pools sont fréquement utilisés dans les serveurs multi-threadés. Chaque connection arrivant sur le serveur est traîtée comme une tâche et transmise à un thread pool. Les threads traitent les requêtes de connection de manière concurrente. Un avantage important des thread pool avec un nombre de thread fixe est que ses performances se dégradent gracieusement : si un nouveau thread était créé à chaque requête HTTP, l'application s'arrêterait de répondre soudainement à toutes les requêtes dès que le nombre de threads dépasse la capacité du système. Avec une limite sur le nombre maximal de threads, le système ne servira pas les requêtes aussi vite qu'elles arrivent, mais il les servira aussi vite qu'il le peut.

Le but de ce distanciel est de paralléliser un programme affichant une représentation graphique de l'ensemble de Mandelbrot, en utilisant un thread pool. Une grande portion du code a déjà été faite pour vous, et est disponible sur Madoc. Comme le projet utilise la bibliothèque `javax.swing`, il est nécessaire de choisir un JRE adapté (par exemple JavaSE-1.7) à la création de votre projet.

2 Description des thread pools

2.1 L'interface `ExecutorService`

Le paquetage `java.util.concurrent` de la bibliothèque standard de Java fournit les classes et interfaces nécessaires à la création de thread pools, ainsi que plusieurs modèles de thread pools. Ces derniers

implémentent l'interface `ExecutorService`. Parmi les principales méthodes de cette interface, la plus importante pour nous est `submit`, qui permet de passer une tâche à un thread pool et est disponible avec trois signatures paramétriques (`T` peut être instancié par n'importe quel type d'objet) :

1. `Future<T> submit(Callable<T>)`
2. `Future<?> submit(Runnable)`
3. `Future<T> submit(Runnable, T)`

Nous avons déjà rencontré l'interface `Runnable` en cours. Elle possède une méthode `run` qui peut être exécutée par un thread. L'interface `Callable<T>` est similaire, à ceci près que la méthode à implémenter a pour signature `T call()`. Autrement dit, l'exécution d'une tâche peut retourner un résultat. Lors de son invocation, `submit()` retourne immédiatement une valeur de type `Future<T>` qui représente l'exécution asynchrone de `call()`. En particulier, le résultat de la tâche sera accessible par la méthode `Future.get()` à la fin de son exécution.

Plusieurs implémentations de `ExecutorService` sont accessibles par les méthodes statiques de la classe `Executors` (Factory). Dans l'exemple ci-dessous, un thread pool de taille fixe (5 threads) est utilisé pour exécuter 50 tâches simples. La JavaDoc de `Executors`, dont la lecture est obligatoire, décrit les autres possibilités.

```
ExecutorService threadPool = Executors.newFixedThreadPool(5);
for(int i = 0; i < 50; ++i){
  threadPool.submit(new MyCallable(i));
}
```

2.2 L'interface `Future<T>`

Un *futur* (parfois appelé *promesse*), est un moniteur qui représente le résultat attendu d'une tâche qui n'a pas encore forcément été exécutée. En Java, un futur implémente l'interface `Future<T>`. La méthode booléenne `isDone()` permet de savoir si l'exécution de la tâche est terminée, auquel cas le résultat peut être récupéré par la méthode `T get()`. Il s'agit d'une méthode bloquante qui met le thread appelant en pause jusqu'à ce que le résultat soit prêt ou qu'une interruption ait lieu. Reportez-vous à la Javadoc pour voir toutes les possibilités ofertes par les futurs.

Pour maximiser le parallélisme, il est conseillé de toujours vérifier si le résultat est prêt avant d'y accéder, comme dans l'exemple ci-dessous.

```
Future<T> = executorService.submit(callable);
while(more_work_to_do()){
  do_more_work();
  // If we can't use the value yet, we do not.
  if (future.isDone()){
    T t = future.get();
    use(t);
  }
}
// Now we have nothing else to do, let's wait...
T t = future.get();
use(t);
```

3 Dessin de l'ensemble de Mandelbrot

3.1 Présentation mathématique

L'ensemble de Mandelbrot est défini comme l'ensemble des nombres complexes c pour lesquels la suite $(|z_k^c|)_{k\in\mathbb{N}}$ est bornée, où $(z_k^c)_{k\in\mathbb{N}}$ est définie par récurrence ci-dessous :

$$\begin{cases} z_0^c &= 0 \\ z_{k+1}^c &= (z_k^c)^2 + c \end{cases}$$

Par exemple, pour $c = -1$, on a $z_0^{-1} = 0$, $z_1^{-1} = -1$, $z_2^{-1} = 0$, puis une alternance de -1 et de 0. -1 appartient donc à l'ensemble. Inversement, pour $c = 1$, on a $z_0^1 = 0$, $z_1^1 = 1$, $z_2^1 = 2$, $z_3^1 = 5$, puis z_k^1 continue de croître indéfiniment. 1 ne fait donc pas partie de l'ensemble.

Pour calculer si un point c appartient à l'ensemble, nous utilisons la propriété suivante : si, pour un k donné, $|z_k^c| > 2$, alors $(|z_k^c|)_{k\in\mathbb{N}}$ tend vers l'infini. On calcule alors z_k^c pour toutes les valeurs de $k \leq \mathit{threshold}$, où *threshold* est une valeur choisie à l'avance.

- Si $|z_{\mathit{threshold}}^c| \leq 2$, on déclare que $(|z_k^c|)_{k\in\mathbb{N}}$ doit être bornée et on dessine le point c en noir.
- Sinon, on dessine le point c dans une couleur qui dépend du plus petit k tel que $|z_k^c| > 2$: plus k est grand, plus le point est lumineux.

De plus, on teste si le point appartient à la cardioide principale ou au plus grand disque pour optimiser le calcul. Cet algorithme est implémenté dans la méthode `Mandelbrot.mandelbrot(double x, double y, int threshold)`.

3.2 Architecture du programme

Le programme à télécharger sur Madoc est organisée selon une architecture client-serveur. Le serveur implémente l'interface `Server`, qui contient une méthode `getBlock()`, dont l'argument de type `ScreenArea` décrit la zone de l'écran à dessiner qui retourne un objet de type `Block`. Les blocs ont la responsabilité de calculer, puis dessiner, des images dans un objet graphique. Leur méthode principale est la méthode `draw()`, qui retourne un booléen indiquant si l'image a pu être dessinée. Enfin, le client, qui suit le code de la classe `Client`, gère la fenêtre et délègue le dessin aux blocs. Le client mesure également le temps nécessaire pour que l'image complète s'affiche.

```
interface Server { Block getBlock(ScreenArea area); }
interface Block { boolean draw(Graphics2D graphics); }
```

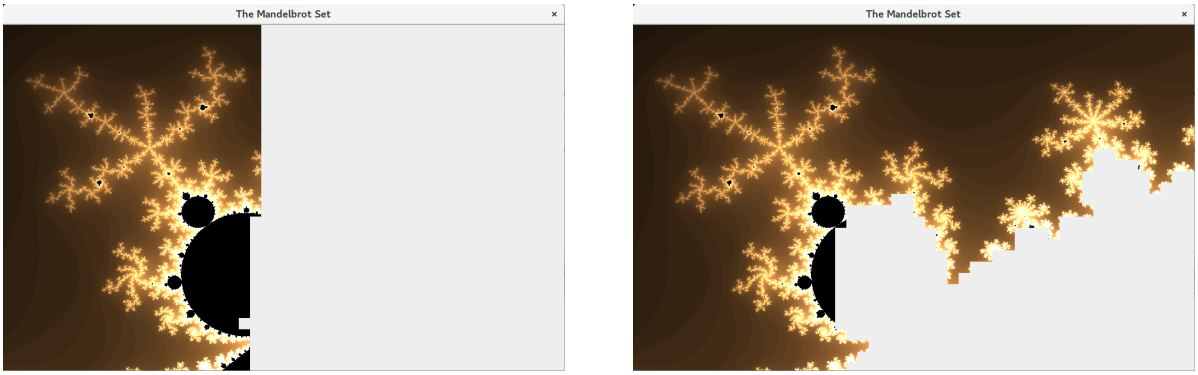
4 Travail à réaliser

4.1 Utilisation d'un thread pool

Dans la version actuelle du programme, le serveur est implémenté dans la class `NaiveServer`. Il s'agit d'une implémentation très naive dans laquelle les images sont calculées par le serveur au moment de la création des blocs. Il n'y a donc pas de parallélisme.

a. Vous devez implémenter une nouvelle classe qui implémente l'interface `Server` et qui utilise un thread pool de taille fixe pour calculer la couleur des points. Chaque bloc devra être calculé comme une tâche indépendante. Éditez la ligne `Server server = new NaiveServer();` dans la fonction `Main.main()` pour remplacer le serveur naïf par le votre.

Pour vous aider à réaliser le projet en autonomie, la figure 1a montre une capture d'écran obtenue pendant l'exécution du programme. Si vous ne pouvez pas observer un état similaire, votre programme est probablement incorrect.



(a) Sans utilisation des priorités.

(b) Avec utilisation des priorités.

FIGURE 1: Capture d'écran en cours d'exécution, avec 10 threads.

b. Mesurez et tracez le graphe du temps nécessaire pour obtenir l'image complète en fonction du nombre de threads (faites varier avec un pas de un entre un seul thread au total jusqu'à environ 20 threads, puis vous pouvez espacer les mesures jusqu'à un thread par bloc). Vous pouvez modifier les paramètres de la méthode `Main.main()` pour que les temps de calcul soient mieux adaptés à votre machine. Un temps autour de 30 secondes devrait convenir pour observer des effets intéressants.

c. Expliquez les résultats graphiques et les mesures de performance.

4.2 Implémentation d'un thread pool

En faisant la partie précédente, vous avez dû remarquer que les blocs en dehors de l'ensemble de Mandelbrot étaient calculés beaucoup plus rapidement que ceux à l'intérieur de l'ensemble. Pour obtenir plus de points plus rapidement, on décide d'exécuter en priorité les blocs en dehors de l'ensemble. Pour cela, l'interface `ImageDrawer` (implémentée par la classe `Mandelbrot`) a une méthode `boolean hasPriority(ScreenArea area)` qui retourne `true` si le bloc doit être calculé en priorité.

Pour tenir compte des priorités, il ne sera pas possible d'utiliser un service d'exécution de la bibliothèque standard. Jakob Jenkov explique comment les thread pools sont implémentés dans son excellent tutoriel : https://jenkov.com/tutorials/java-concurrency/thread-pools.html. La lecture de son billet fait partie de l'exercice et est obligatoire.

Vous devez implémenter un deuxième serveur, basé sur l'implémentation des thread pools présentée par Jakob Jenkov, qui n'assigne aux threads des tâches non-prioritaires que quand il n'y a plus de tâche prioritaire à exécuter. Concrètement, il faudra remplacer la file par un nouveau moniteur qui gère les deux types de tâches avec deux files différentes. N'imposez pas de limite sur la taille des files.

Pour vous aider à réaliser le projet en autonomie, la figure 1b montre une capture d'écran obtenue pendant l'exécution du programme. Si vous ne pouvez pas observer un état similaire, votre programme est probablement incorrect.

5 Pour aller plus loin

- https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html
- https://jenkov.com/tutorials/java-concurrency/thread-pools.html
- https://www.javaworld.com/article/2078809/java-concurrency/java-concurrency-java-101-the-next-generation-java-concurrency-without-the-pain-part-1.html
- http://www.baeldung.com/java-completablefuture
- https://www.journaldev.com/1090/java-callable-future-example