# TP Parallélisme régulé

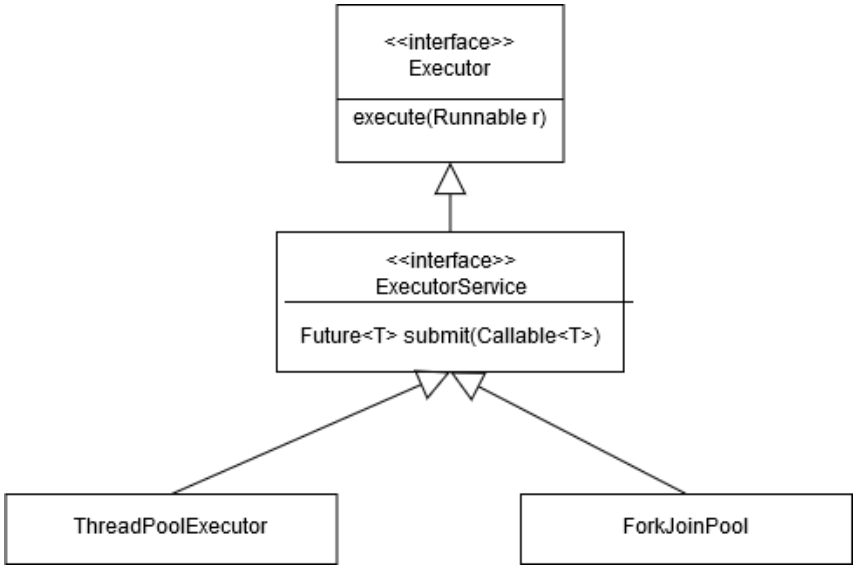
Julien Desvergnes

## 1 Objectifs

Dans ce TP on va chercher à mettre en place deux types d'algorithmes :

- pool de thread,
- pool fork-join

Les classes et notions utilisées jusqu'ici étaient destinées à définir et gérer la concurrence explicitement, et à un niveau fin : le choix de lancer, d'attendre et de terminer une tâche appartient entièrement au programmeur. De même, le programmeur a la charge des choix en termes de gestion de la cohérence (variables volatile, classes atomiques...) et du type d'attente (blocs synchronized, verrous, attente active).




La plateforme Java fournit la classe Executor, destinée à séparer la gestion des activités des aspects purement applicatifs. Le principe est qu'un objet de la classe Executor ( exécuteur ) fournit un service de gestion et d'ordonnancement d'activités, auquel on soumet des tâches à traiter. Une application est donc vue comme un ensemble de tâches qui sont fournies à l'exécuteur. L'exécuteur gère alors l'exécution des tâches qui lui sont soumises de manière indépendante et transparente pour l'application.

Si il y a des questions plus pointues, **https://www.jmdoudoux.fr/java/dej/chap-executor.htm**

# 2 Présentation des algorithmes

## 2.1 Pool de thread

### 2.1.1 En quelques mots

On a un ensemble d'ouvrier qui peuvent exécuter une tâche. Cet ensemble d'ouvrier forme la **pool**. Il est ensuite possible de soumettre une tâche à la pool. Un ouvrier qui n'a pas de tâche à effectuer est alors réquisitionné pour effectuer la tâche.

### 2.1.2 Intérêts

Les pools de threads évitent la création de nouveaux threads pour chaque tâche à traiter, puisque qu'un même ouvrier est réutilisé pour traiter une suite de tâches, ce qui présente plusieurs avantages :

- éviter la création de threads apporte un gain (significatif lorsque les tâches sont nombreuses) en termes de consommation de ressources mémoire et processeur,
- le délai de prise en charge des requêtes est réduit du temps de la création du traitant de la requête,
- le contrôle du nombre d'ouvriers va permettre de réguler et d'adapter l'exécution en fonction des ressources matérielles disponibles.

Les pools de threads sont bien adaptés au **traitement de problèmes réguliers**, c'est à dire aux problèmes décomposables en sous-problèmes de taille équivalente, ce qui garantit une bonne répartition des tâches entre ouvriers.

### 2.1.3 En pratique : un exemple avec le calcul d'une somme

**Imports**

```
1  import java.util.concurrent.Future;
2  import java.util.concurrent.Callable;
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
```

**Main : Création de la pool et lancement des tâches**

```
1  public class Somme {
2      public static void main(String[] args) throws Exception {
3
4          //Creation de la pool
5          ExecutorService poule = Executors.newFixedThreadPool(2);
6
7          //Soumission des tches
8          Future<Long> f1 = poule.submit(new SigmaC(0L,1_000_000_000L));
9          Future<Long> f2 = poule.submit(new SigmaC(0L,4_000_000_000L));
10         poule.execute(new SigmaR(900_000L,1_000_000_000L));
11         Future<Long> f3 = poule.submit(new SigmaC(1,100));
12         Future<Long> f4 = poule.submit(new SigmaC(0L,3_000_000_000L));
13
14         //On interdit la soumission de nouvelles tches
15         poule.shutdown();
16
17         // Get permet de recuperer les resultats
18         System.out.println("Resultat obtenu. f1 = "+f1.get());
19         System.out.println("Resultat obtenu. f2 = "+f2.get());
20         System.out.println("Resultat obtenu. f3 = "+f3.get());
21         System.out.println("Resultat obtenu. f4 = "+f4.get());
22     }
23 }
```

**Implémentation version Callable**

```
1  class SigmaC implements Callable<Long> {
2      private long dbut;
3      private long fin;
4
5      SigmaC(long d, long f) { dbut = d; fin = f;}
6
7      @Override
8      // Quand on fait un callable, la methode call DOIT renvoyer un objet
9      public Long call() {
10         long s = 0;
11         for (long i = dbut; i <= fin; i++) s = s + i;
12         return s;
13     }
14 }
```

**Implémentation version Runnable**

```
1  class SigmaR implements Runnable {
2      private long dbut;
3      private long fin;
4
5      SigmaR(long d, long f) { dbut = d; fin = f;}
6
7      @Override
8      public void run() {
9          long s = 0;
10         for (long i = dbut; i <= fin; i++) s = s + i;
11         System.out.println("Calcul termine. ("+debut+","+fin+") = "+s);
12     }
13 }
```

## 2.2 Pool Fork-Join

### 2.2.1 En quelques mots

Le principe du schéma est récursif. Il a le comportement suivant :

- si la taille de mon problème est suffisamment petite pour être traitée directement, je la traite,
- sinon, on divise le problèmes en sous-problèmes de plus petite taille qui seront traités en parallèle (fork) et dont les résultats seront attendus et agrégés (join).

### 2.2.2 Intérêts

Ce schéma de programmation permet de créer dynamiquement un nombre de tâches adapté à la taille de chacun des (sous)-problèmes rencontrés, chacune des tâches créées représentant une charge de travail équivalente. Ce schéma est donc bien adapté au **traitement de problèmes irréguliers**, **de grande taille**. L'ordonnanceur de la classe ForkJoinPool comporte en outre une régulation (vol de tâches) qui permet l'adaptation de l'exécution aux capacités de calcul disponibles.

Il est important de noter que ce schéma repose sur le fait que les sous-tâches créées s'exécutent en parallèle, et donc sur l'hypothèse qu'**elles sont complètement indépendantes**. Tout conflit d'accès aux ressources, ou synchronisation compromet l'efficacité de ce schéma. Le schéma Fork/Join est donc idéalement et principalement destiné aux calculs intensifs, irréguliers, en mémoire pure (sans E/S).

### 2.2.3 En pratique : un exemple

**Imports**

```
1  import java.util.concurrent.RecursiveTask;
2  import java.util.concurrent.ForkJoinPool;
```

**Main : Creation de la pool et lancement**

```
1  public class FJG {
2   static ForkJoinPool fjp = new ForkJoinPool();
3   static final int TAILLE = 1024; //Attention : necessairement une puissance de 2
4
5   public static void main(String[] args) throws Exception {
6      // Creation de la pool
7      TraiterProblme monProblme = new TraiterProblme(TAILLE);
8
9      // Lancement initial sur la totalite des donnes
10     int resultat = fjp.invoke(monProblme);
11
12     System.out.println("Resultat final = " + rsultat);
13  }
14 }
```

**Implémentation de la tâche récursive**

```
1  class TraiterProbleme extends RecursiveTask<Integer> {
2
3   private int resteAFaire = 0;
4   private int rsultat = 0;
5   static final int SEUIL = 10;
6
7   TraiterProblme(int resteAFaire) {
8      this.resteAFaire = resteAFaire;
9   }
10
11  protected Integer compute() {
12
13     //si la tache est trop grosse, on la decompose en 2
14     if(this.resteAFaire > SEUIL) {
15        System.out.println("Dcomposition de resteAFaire : " + this.resteAFaire);
16
17        TraiterProblme sp1 = new TraiterProblme(this.resteAFaire / 2);
18        TraiterProblme sp2 = new TraiterProblme(this.resteAFaire / 2);
19
20        // On fait un fork
21        sp1.fork();
22        sp2.fork();
23
24        // On join sur les resultats
25        rsultat = sp1.join()+ sp2.join();
26
27        return resultat;
28
29     // Sinon on effectue le traitement
30     } else {
31        System.out.println("Traitement direct de resteAFaire : " + this.resteAFaire);
32        return resteAFaire * 3;
33     }
34  }
35 }
```

# 3 Exercices

L'archive fournie propose différents exercices. Chaque exercice comporte un calcul séquentiel (itératif ou récursif), qu'il faut paralléliser en utilisant un pool fixe et/ou un pool Fork/Join. Chaque exercice comporte une méthode main permettant de lancer et comparer les différentes versions. Des commentaires // ********* A compléter ou // ********* A corriger signalent **les seuls endroits du code où vous devez intervenir** pour implanter les versions parallèles du calcul séquentiel fourni.

## 3.1 Générer des données pour les exercices du maximum et du tri fusion

Certains exercices (max et tri-fusion) utilisent des tableaux d'entiers stockés sur disque. L'application GCVT.java propose une classe TableauxDisque permettant de générer, charger en mémoire, sauvegarder ou comparer de tels tableaux.

Cette application pourra en particulier être utilisée pour générer les jeux de données utiles aux tests. En effet, pour que le gain apporté par les versions parallèles soit sensible, il est nécessaire que les volumes de données traités soient significatifs, ce qui implique ici de travailler (pour l'évaluation de performances) sur des tableaux de 1 à 100 millions d'entrées, ce qui aurait alourdi inutilement l'archive. Vous devrez donc générer vos jeux de données avec cette application, sans oublier de supprimer les fichiers créés une fois le TP passé, sans quoi vous risquez d'épuiser votre quota d'espace disque.

**Utilisation de l'outil GCVT** Compiler le fichier GCVT.

- Générer un fichier : **GCVT -g nomDuFichier nombreElements valeurMaximum**,
- Comparer deux fichiers : **GCVT -c fichier1 fichier2 indiceDeDebutDeComparaison**,
- Visualiser un fichier : **GCVT -v nomDuFichier indiceDebut indiceFin**

## 3.2 Exercice 1 : Maximum d'un tableau

### 3.2.1 Problématique

Le calcul d'un opérateur associatif et commutatif sur un ensemble de données est une application canonique de la parallélisation. Cet exercice permet de mettre simplement et directement en pratique les deux schémas (pool fixe et map/reduce) présentés dans l'énoncé.

### 3.2.2 Lancer l'application

**java MaxTab fichier nbEssais nbTachesPool tailleTroncon nbOuvriersPool**

### 3.2.3 Méthodes à compléter

- méthode call de la classe MaxLocal : calcul le maximum sur le tronçon de tableau entre début et fin,
- méthode compute de la classe TraiterFragment : si la taille est inférieure au seuil on peut traiter sinon on décompose,
- méthode maxPoolFixe : soumission des tâches et récupération des résultats,
- méthode maxForkJoin : soumission des tâches.

### 3.2.4 Petites questions :

- Le calcul séquentiel à paralléliser est une itération. A votre avis, quel sera le schéma de parallélisation le plus naturel ? le plus efficace ?
- Notez que le calcul étant très simple, il est important pour évaluer les performances de cet exercice de travailler avec un grand tableau.
- Comparer les deux versions (pool fixe et Fork/join) avec la version séquentielle. Les mesures confirment-elles vos a priori ? Commentez.

## 3.3 Exercice 2 : Tri Fusion

### 3.3.1 Problématique

Tri d'un tableau selon le schéma tri-fusion. Même s'il est régulier, le schéma récursif le prête parfaitement à l'utilisation du schéma map/reduce, et d'autant mieux qu'il est organisé en 2 phases (tri, puis fusion).

### 3.3.2 Lancer l'application

**java TF fichier nbEssais nbTachesPool tailleTroncon nbOuvriersPool**

### 3.3.3 Méthodes à compléter

- méthode compute de la classe TraiterFragment : si la taille est inférieure au seuil on peut traiter sinon on décompose,
- méthode TFPool : soumission des tâches et récupération des résultats,
- méthode TFForkJoin : soumission des tâches.

### 3.3.4 Petites questions :

- Paralléliser l'algorithme récursif proposé en utilisant les deux schémas (pool fixe et Fork/Join)
- Comparer ces deux versions avec la version séquentielle, en termes de facilité de conception, et de performances. Pour cet exercice, un tableau d'un million d'entrées devrait suffire.

## 3.4 Exercice 3 : Comptage de mots dans un répertoire

Cet exercice est plus libre que les deux précédents. Il faut simplement écrire la parallélisation avec le schéma Fork-Join.

### 3.4.1 Petite question

Comparer cette version avec la version séquentielle, en termes de facilité de conception, et de performances. Pour le test, on pourra prendre un répertoire contenant des fichiers sources, et rechercher un mot clé du langage.