

Diviser pour calculer plus vite

Fork - Join

riveill@unice.fr

<http://www.i3s.unice.fr/~riveill>



Comment mesurer l'accélération due au parallélisme

Accélération maximum → loi d'Amdhal

- p pourcentage du code exécutable en parallèle
 - $(1 - p)$: pourcentage du code exécuté en séquentiel
 - p / n : exécution du code parallèle sur n cœurs
- Le temps d'exécution théorique évolue donc en :
 - $T = (1 - p) + p / n$
- Accélération $A = 1/T$
 - $A = 1/((1 - p) + p/n)$
 - Si $n \rightarrow \infty$ alors $A \rightarrow 1/(1 - p)$

Quelques exemples issues du projet

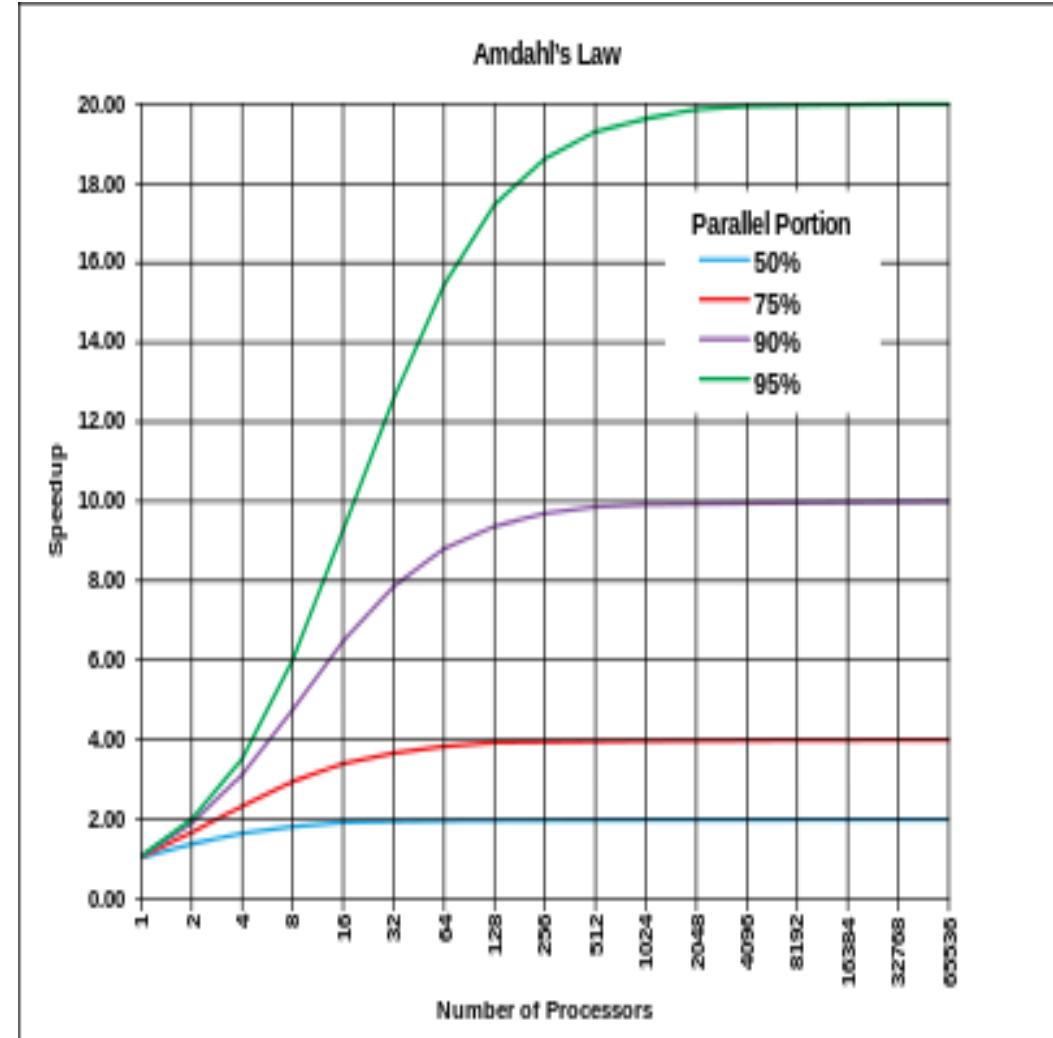
- Nombre de threads entre 2^0 et 2^9
- Pourcentage du code exécuté en parallèle
 - Chez certains $p = 0,05$
 - Chez d'autres $p = 0,95$
- Nombre de cœurs n (2, 4 ou 8 selon les stations)
- Temps d'exécution théorique
 - Temps pour faire sortir une personne du terrain constant en moyenne : t
 - Temps pour faire sortir X personnes en séquentiel : $X*t$
 - Temps pour faire sortir X personnes en parallèle : $T(X) = ((1-p) + p/n) * X * t$
- Avec 4 cœurs, dans un monde idéal, je devrais avoir pour X personnes
 - $T_{\text{seq}} = X*t$
 - $T_{4 \text{ cœurs}} = ((1-p) + p/4) * X / t$
 - Soit ration $T_{4\text{coeurs}} / T_{\text{seq}} = ((1-p) + p/4) == \text{constante}$
- Les questions
 - Combien vaut p dans votre code ?
 - Est-ce que ce calcul de p est conforme au p que l'on pourrait calculer en analysant le code ?
 - On essaie de trouver des explications raisonnables

Application numérique (sur d'autres données)

Application numérique :

- $p = 0,25$
 - $a = 3,7$ à 32 cœurs
 - $a \rightarrow 4$ quand $n \rightarrow \infty$
- $p = 0,95$
 - $a = 12,55$ à 32 cœurs
 - $a = 17,42$ à 128 cœurs
 - $a = 20$ quand $n \rightarrow \infty$
- $p = 0,96$
 - $a = 14,28$ à 32 cœurs : +12.1%
 - $a = 21,05$ à 128 cœurs : +17.2%
 - $a = 25$ quand $n \rightarrow \infty$

⇒ Ça vaut le coup de se battre pour paralléliser quelques pourcents



Comment augmenter le parallélisme

Option 1 : réduire la taille des sections critiques

- Passer du **verrouillage à gros grain** (coarse-grained locking)
➔ à un **verrouillage à grain fin** (fine-grained locking)
 - **Effet** : augmente le p dans la loi d'Amdahl
 - **Comment** : diviser les longues sections critiques par de nombreuses petites sections critiques
 - Augmente généralement la complexité du programme
- Travail nécessaire, complexe
 - Mais pas vraiment de technique générale pour le faire...
 - Par conséquent pas de quoi faire une séance de cours sauf à vous donner des sections critiques à reécrire.
- **Option 2** : améliorer les algorithmes de verrouillage
 - Cf cours TMPC en SI5
- **Option 3** : supprimer les verrous en adoptant une algorithmique non bloquante
 - Cf cours une prochaine semaine (introduction) + cours TMPC en SI5
- **Option 4** : changer complètement le modèle mémoire avec les mémoires transactionnelles
 - Cf cours TMPC en SI5

Diviser pour calculer plus vite

Fork - Join

Distinguer les tâches à faire et les threads qui vont les exécuter



Un peu d'algorithmique pour commencer

- Jusqu'à présent on a l'association
 - 1 tache est directement associé à 1 thread
 - Et on a autant de thread que de tache à exécuter
- Nous pouvons maintenant poser des questions de nature plus algorithmique :
 - Comment obtenir un gain de performance en adaptant le découpage tache / thread et en optimisant le nombre de thread
- Supposons que l'on souhaite additionner point par point deux tableaux de même taille :

```
// Semantics: C = A + B
static void add (int[] C, int[] A, int[] B) {
    for (int i = 0; i < C.length; i++)
        C[i] = A[i] + B[i] ;
}
```

 - La complexité est $O(n)$.
- Si on dispose de plusieurs processeurs, on peut espérer faire mieux, car les instructions $C[i] = A[i] + B[i]$ sont indépendantes les unes des autres.
 - Les architectures GPU sont parfaitement adaptées pour ce type de calcul

Un parallélisme forcé ?

- Une approche radicale consisterait à lancer n threads en parallèle :

```
static void add (int[] C, int[] A, int[] B) {  
    for (int i = 0; i < C.length; i++) {  
        fork { C[i] = A[i] + B[i]; }  
    }  
    joinAll ;  
}
```
- Est-ce une bonne idée ?
 - D'un point de vue pratique, non, ce n'est pas une bonne idée.
 - Créer un thread coute cher
 - Il faut au grand minimum plusieurs milliers d'instructions pour lancer et arrêter un processus « léger ».
 - Lancer un thread pour effectuer une tache très courte n'a aucun sens !

Une version moins agressive

- On peut lancer un nombre de threads moins important en découplant le tableau en une série d'intervalles :

```
static void addChunk (int[] C, int[] A, int[] B) {  
    final int CHUNK = ... ;  
    for (int c = 0; c < C.length; c += CHUNK) {  
        fork {  
            for (int i = c; i < c + CHUNK; i++)  
                C[i] = A[i] + B[i] ; }  
    }  
    joinAll;  
}
```

- Comment choisir la valeur de CHUNK ?

Pour bien faire, il faut s'adapter à la charge ?

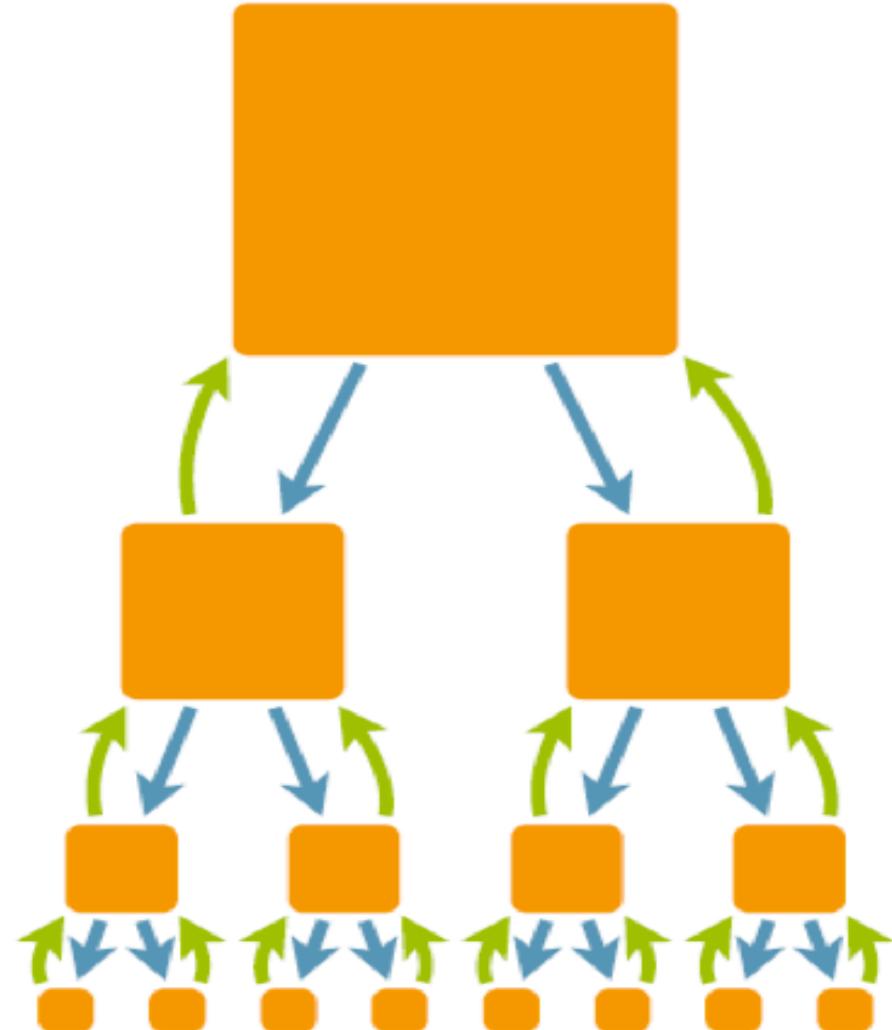
- Si on a p processeurs, on lance p threads et donc :
 - Choisir CHUNK = n/p
- Mais si certains processeurs sont déjà occupés par d'autres processus (lourds ou légers) :
 - Il y aura partage du temps, d'où inefficacité.
- Or le nombre de processeurs déjà occupés peut varier au cours du temps !
- Pas facile...

Vers une abstraction de plus : les tâches

- Pour faciliter la répartition dynamique du travail, on peut :
 - introduire une notion de tâche,
 - poser que fork/join créent/attendent une tâche,
 - utiliser un nombre fixe de processus légers (worker threads) pour exécuter ces tâches,
 - ce qui suppose un algorithme efficace de répartition des tâches entre travailleurs (« scheduling »)
- Plusieurs librairies implémentent cette idée :
 - Intel Thread Building Blocks
 - Microsoft Task Parallel Library
 - **Java 7 ForkJoin – c'est ce que nous allons étudier.**

Un exemple d'utilisation du modèle Fork / Join

- Lorsque le traitement est décomposable en sous-tâches indépendantes
 - Existence d'algorithmes parallèles spécifiques
- La décomposition en sous-tâches peut être :
 - statique : découper un tableau en zones fixes
 - dynamique : découvrir une arborescence de fichiers
 - Attention à maîtriser le volume de tâches créées
 - récursive

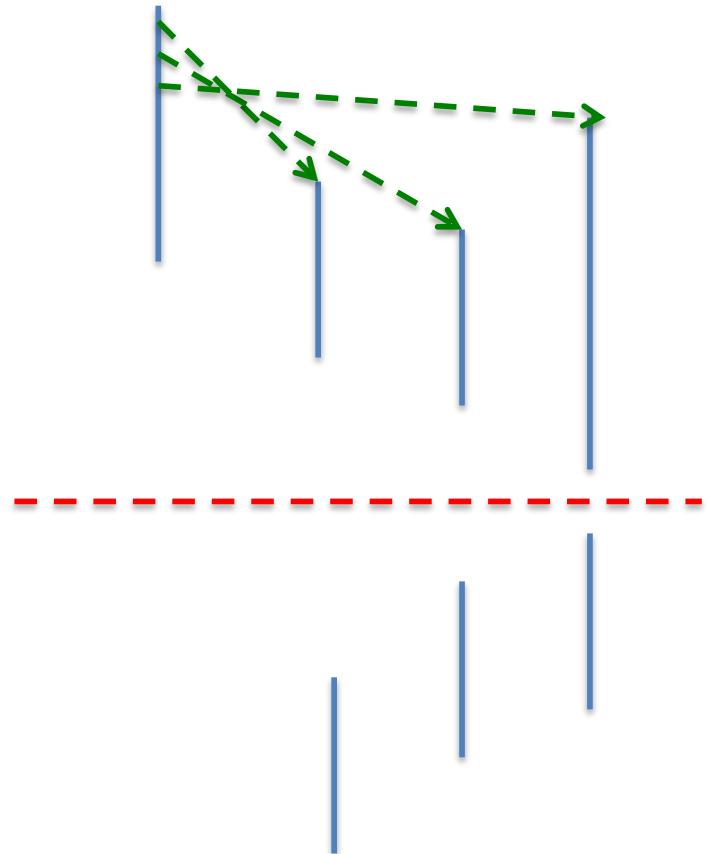


Etape 1

mise en œuvre du joinAll

JoinAll = barrière

- **Idée :** tous les processus franchissent au même moment “la barrière”
- **Implémentation :**
 - Facile à implémenter par un moniteur ou avec des sémaphores
 - Existe en Java 7, en posix
- **Principe en Java 7 :**
 - Un processus crée un objet barrière
 - il donne le nombre N de processus attendu
 - Chaque processus exécute « await » sur l'objet barrière
 - Quand N est atteint, la barrière s'ouvre



La barrière Java

```
public class MonRunnable implements Runnable {  
  
    private CyclicBarrier barrier;  
    string name = Thread.currentThread().getName();  
  
    public MonRunnable(CyclicBarrier barrier) {  
        this.barrier = barrier;  
    }  
  
    public void run() {  
        System.out.println(threadName + " is started");  
        Thread.sleep(400*new Random().nextInt(10));  
        System.out.println(threadName  
                           + " is waiting on barrier");  
        barrier.await();  
        System.out.println(threadName  
                           + " has crossed the barrier");  
    }  
}
```

La barrière Java

```
main(String args[]) {  
    //creating CyclicBarrier with 3 Threads needs to call await()  
    CyclicBarrier cb = new CyclicBarrier(3, new Runnable() {  
        public void run() {  
            //This task will be executed once all thread reaches barrier  
            System.out.println("All parties are arrived at barrier");  
        }  
    });  
  
    //starting each of thread  
    Thread t1 = new Thread(new MonRunnable(cb), "Thread 1");  
    Thread t2 = new Thread(new MonRunnable(cb), "Thread 2");  
    Thread t3 = new Thread(new MonRunnable(cb), "Thread 3");  
  
    System.out.println("Début démarrage des threads");  
    t1.start(); t2.start(); t3.start();  
    System.out.println("Fin démarrage des threads");  
}
```

La barrière Java

- Un exemple d'exécution

Début démarrage des threads

Fin démarrage des threads

Thread 3 is started

Thread 1 is started

Thread 2 is started

Thread 1 is waiting on barrier

Thread 2 is waiting on barrier

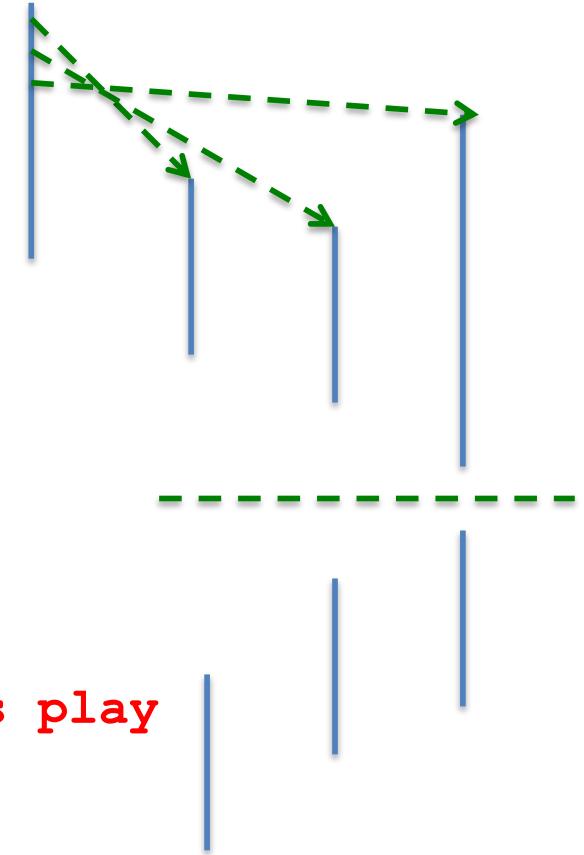
Thread 3 is waiting on barrier

All parties are arrived at barrier, lets play

Thread 3 has crossed the barrier

Thread 2 has crossed the barrier

Thread 1 has crossed the barrier



- Evidemment dans le cas de tâches cycliques, il faut réinitialiser la barrière
 - `reset()`

Mise en œuvre naïve de la barrière en Java

```
Class Barrier {  
    private final int N_THREADS;  
    int arrived;  
  
    public Barrier (int n) {  
        N_THREADS = n;  
    }  
  
    public synchronized void await ()  
    arrived++;  
    if (arrived < N_THREADS) {  
        wait();  
    } else {  
        arrived = 0;  
        notifyAll();  
    }  
}
```

Sans un phénomène bien curieux appelé → Spurious wakeup

Spurious wakeup

<http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

- A thread can also wake up without being notified, interrupted, or timing out, a so-called *spurious wakeup*.
- While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops, like this one:

```
synchronized (obj) {  
    while (<condition does not hold>)  
        obj.wait(timeout);  
    . . . // Perform action}
```

Arrive en particulier sur Linux car la mise en œuvre de la JVM repose sur les pthread qui engendre le phénomène de « Spurious wakeup »

- For more information on this topic, see Section 3.2.3 in Doug Lea's "Concurrent Programming in Java (Second Edition)" (Addison-Wesley, 2000), or Item 50 in Joshua Bloch's "Effective Java Programming Language Guide" (Addison-Wesley, 2001).

Mise en œuvre de la barrière Java (avant le java 7)

```
Class Barrier {  
    private final int N_THREADS;  
    int[] counts = new int[] {0, 0};  
    int current = 0;  
  
    public Barrier (int n) {  
        N_THREADS = n;  
    }  
  
    public synchronized void aWait () {  
        int my = current;  
        counts[my]++;  
        if (counts[my] < N_THREADS)  
            while (counts[my] < N_THREADS) wait();  
        else {  
            current = 1-current;  
            counts[current] = 0;  
            notifyAll();  
        }  
    }  
}
```

*Prise en compte du
Spurious wakeup*

Etape 2

Associer des tâches à des threads

Contrôle du parallélisme

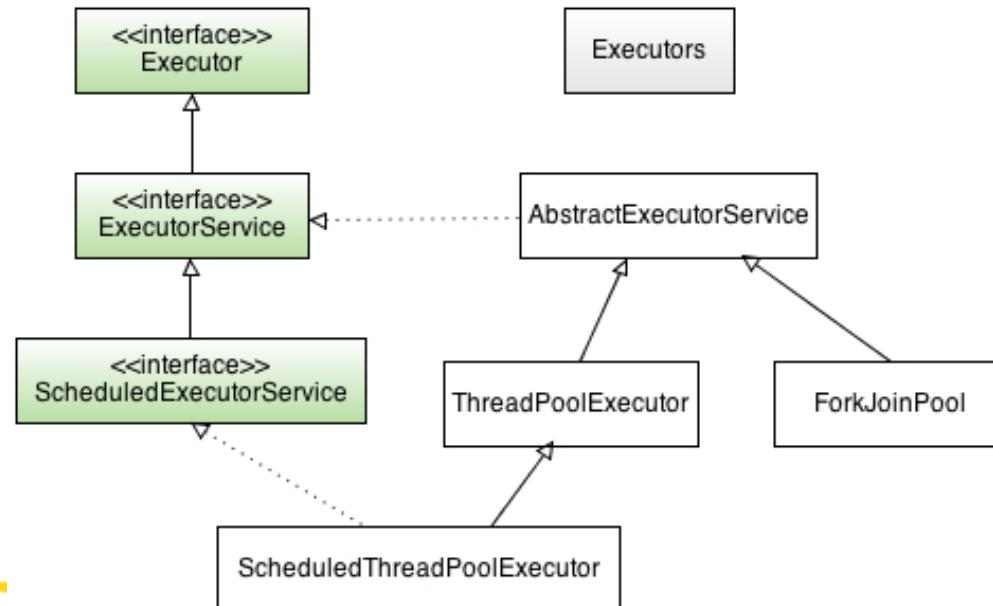
- Le parallélisme c'est bien mieux si on peut **effectivement** calculer en parallèle
 - Exploitation d'une l'architecture multi-cœur
 - Java
 - Nom du thread courant : `Thread.currentThread().getName()`
 - Nombre de cœur sur le processeur : `Runtime.getRuntime().availableProcessors();`
 - Quel est l'intérêt de lancer 10 thread si on a que 4 cœurs ?
- Jusqu'à présent on créait des threads parallèles sans jamais se poser la question du nombre de threads qui était effectivement exécuté en //
 - Un modèle : les « pools » de thread
 - Créer une thread coûte cher
 - Alors on recycle ☺

Pool de thread

Principe

- On crée un ensemble de N threads
 - Généralement moins que de processeurs/coeurs existants
 - Ou au contraire légèrement plus pour prendre en compte les opérations bloquantes (entrée/sortie, réseau)
- On crée un ensemble de tâche
 - Le nombre dépend généralement du problème à résoudre
- Les tâches sont successivement exécutées par les thread

Mise en œuvre en Java à l'aide d'une nouvelle hiérarchie : Executor



Classe Executors

une autre manière d'activer une thread

```
public class MonRunnable implements Runnable {  
    public void run() {  
        System.out.println("Debut execution dans le thread "  
                           +Thread.currentThread().getName());  
        //On simule un traitement long  
        Thread.sleep(4000);  
        System.out.println("Fin execution dans le thread "  
                           +Thread.currentThread().getName());  
    }  
}
```

- Solution déjà connue

```
Thread thread = new Thread(new MonRunnable());  
thread.start();
```

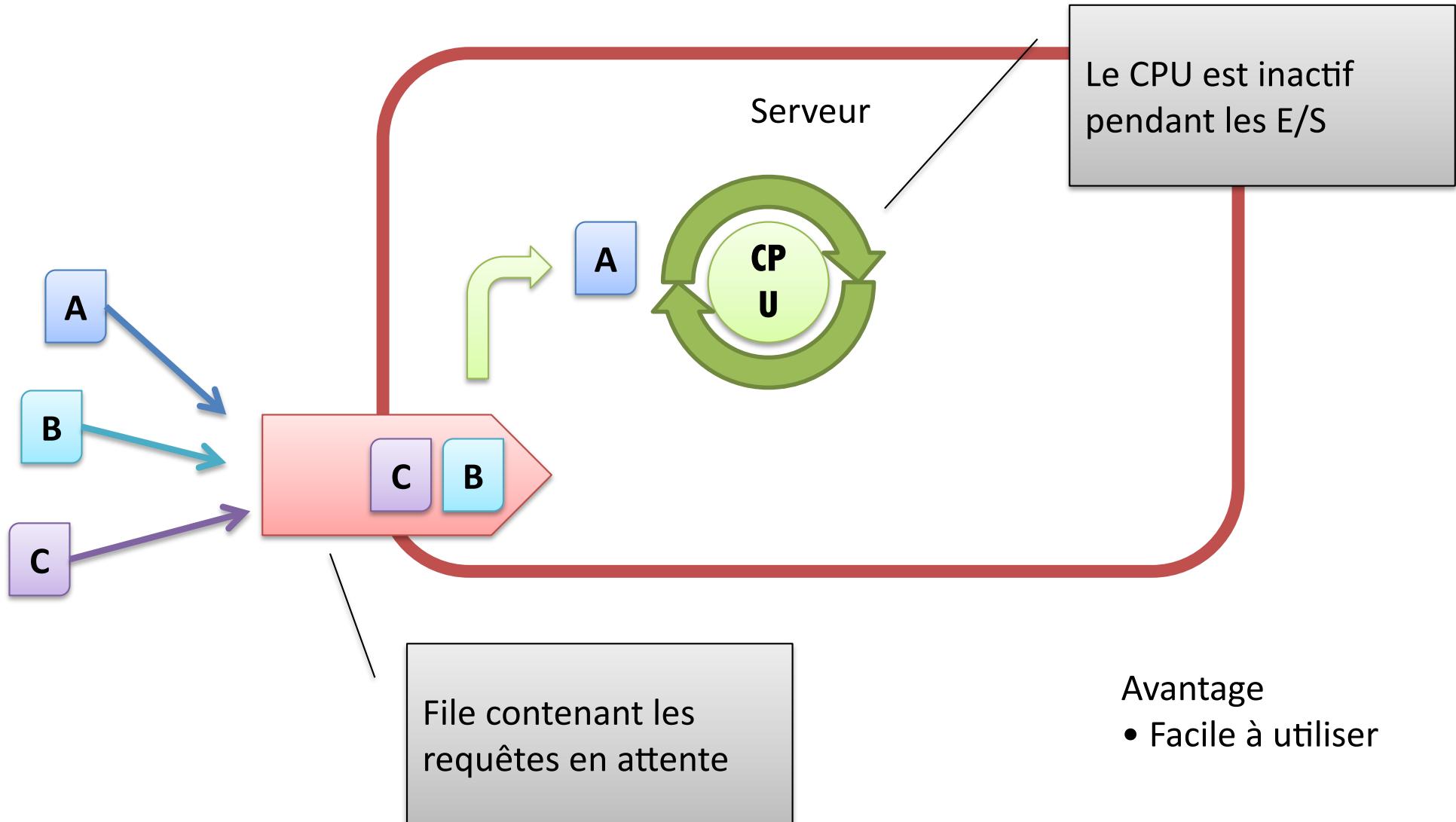
- En utilisant l'interface Executor

```
Executor executor = Executors.newSingleThreadExecutor();  
executor.execute(new MonRunnable());
```

Classe Executors : exécution séquentielle des tâches

```
List<Runnable> runnables = new ArrayList<Runnable>();  
//création de 4 tâches  
runnables.add(new MonRunnable());  
runnables.add(new MonRunnable());  
runnables.add(new MonRunnable());  
runnables.add(new MonRunnable());  
  
//création 'executor' mono-thread  
ExecutorService executor =  
    Executors.newSingleThreadExecutor();  
  
//exécution des tâches selon le modèle choisi  
for(Runnable r : runnables){  
    executor.execute(r);  
}  
//attente de la terminaison de toutes les tâches  
executor.shutdown();
```

Classe Executors : exécution séquentielle des tâches



Classe Executors : exécution séquentielle des tâches

- Evidemment les traces d'exécution sont :

Debut execution dans le thread pool-1-thread-1

Fin execution dans le thread pool-1-thread-1

Debut execution dans le thread pool-1-thread-1

Fin execution dans le thread pool-1-thread-1

Debut execution dans le thread pool-1-thread-1

Fin execution dans le thread pool-1-thread-1

Debut execution dans le thread pool-1-thread-1

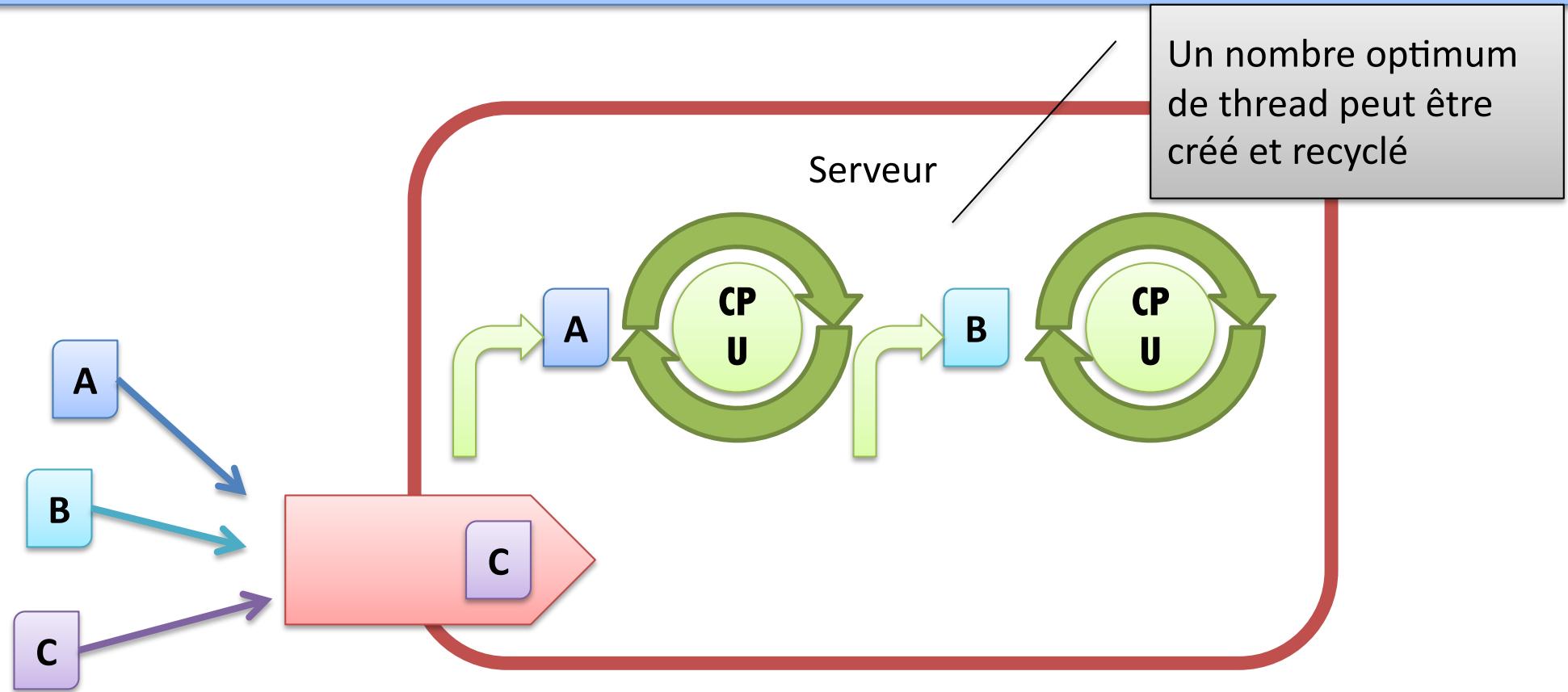
Fin execution dans le thread pool-1-thread-1

Classe Executors : exécution parallèle des tâches

```
List<Runnable> runnables = new ArrayList<Runnable>();  
//création de 4 tâches  
runnables.add(new MonRunnable());  
runnables.add(new MonRunnable());  
runnables.add(new MonRunnable());  
runnables.add(new MonRunnable());  
  
//création 'executor' pool de 2 threads  
ExecutorService executor =  
    Executors.newFixedThreadPool(2);  
    //Executors.newSingleThreadExecutor();  
  
//exécution des tâches selon le modèle choisi  
for(Runnable r : runnables) {  
    executor.execute(r);  
}  
  
//attente de la terminaison de toutes les tâches  
executor.shutdown();
```

Seul
changement

Classe Executors : exécution parallèle des tâches



Avantage

- Moins de requête dans la file d'attente
- Minimize la création /destruction de thread

Désavantage

- Nécessité de « tuning » pour tirer le meilleur parti du matériel
- Pas facile à mettre en œuvre avant Java 7.0

Classe Executors : exécution parallèle des tâches

- **Un exemple d'exécution des threads**

Debut execution dans le thread pool-1-thread-1

Debut execution dans le thread pool-1-thread-2

Fin execution dans le thread pool-1-thread-2

Fin execution dans le thread pool-1-thread-1

Debut execution dans le thread pool-1-thread-2

Debut execution dans le thread pool-1-thread-1

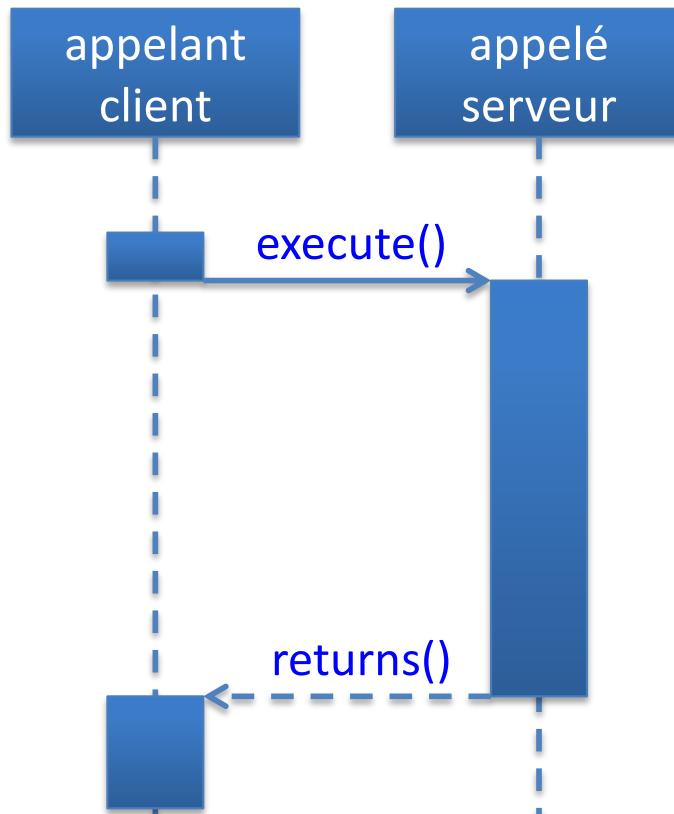
Fin execution dans le thread pool-1-thread-1

Fin execution dans le thread pool-1-thread-2

Etape 3. Appel de méthode avec futur (ou comment récupérer un résultat plus tard)

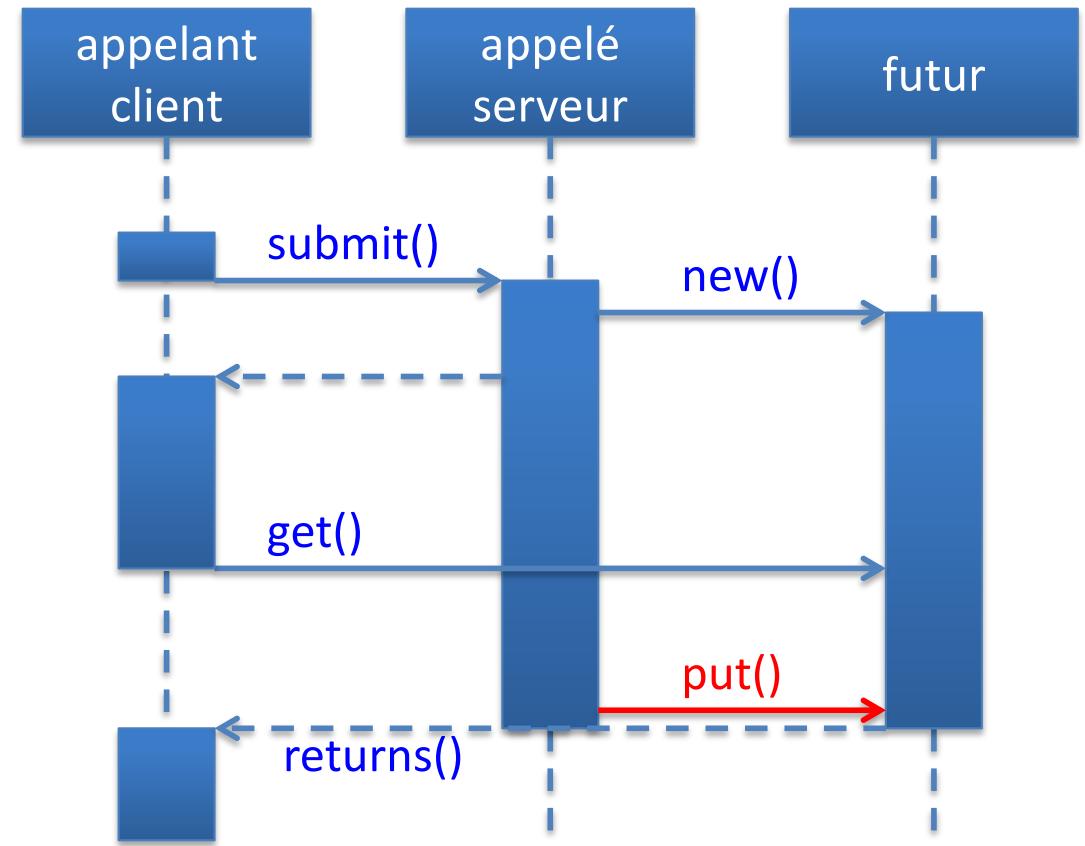
Un modèle : l'appel asynchrone avec future

- Appel synchrone



- Appel de méthode
- Appel de procédure à distance (RPC)
- Client-serveur

- Appel asynchrone avec futur



- Acteurs / objets actifs
- Appel asynchrone de procédure à distance (A-RPC)



Runnable versus Callable<T>

Runnable	Callable<T>
Introduced in Java 1.0	Introduced in Java 1.5 as part of java.util.concurrent library
Runnable cannot be parametrized	Callable is a parametrized type whose type parameter indicates the return type of its run method
Classes implementing Runnable needs to implement run() method	Classes implementing Callable needs to implement call() method
Runnable.run() returns no Value	Callable.call() returns a value of Type T
Can not throw Checked Exceptions	Can throw Checked Exceptions
<pre>public interface Runnable { void run(); }</pre>	<pre>public interface Callable<V> { V call() throwsException; }</pre>

Classe Callable

```
public class MonCallable
    implements Callable<Integer> {
    public Integer call() {
        System.out.println("Debut execution");
        //Simulation traitement long
        Thread.sleep(4000);
        System.out.println("Fin execution");
        return new Random().nextInt(10);
    }
}
```

La classe Future<T>

```
// création d'un "Executor"  
ExecutorService executor =  
    Executors.newSingleThreadExecutor();  
//Appel de la tache et récupération d'un Future<V>  
Future<Integer> future =  
    executor.submit(new MonCallable());  
System.out.println("Apres submit");  
//On a besoin du résultat on appelle : future.get()  
// appel bloquant  
System.out.println("Resultat=" + future.get());  
System.out.println("Avant shutdown");  
execute.shutdown();
```

Appel asynchrone avec futur

- Exemple de trace

Apres submit

Début execution

Fin execution

Résultat=6

Avant shutdown

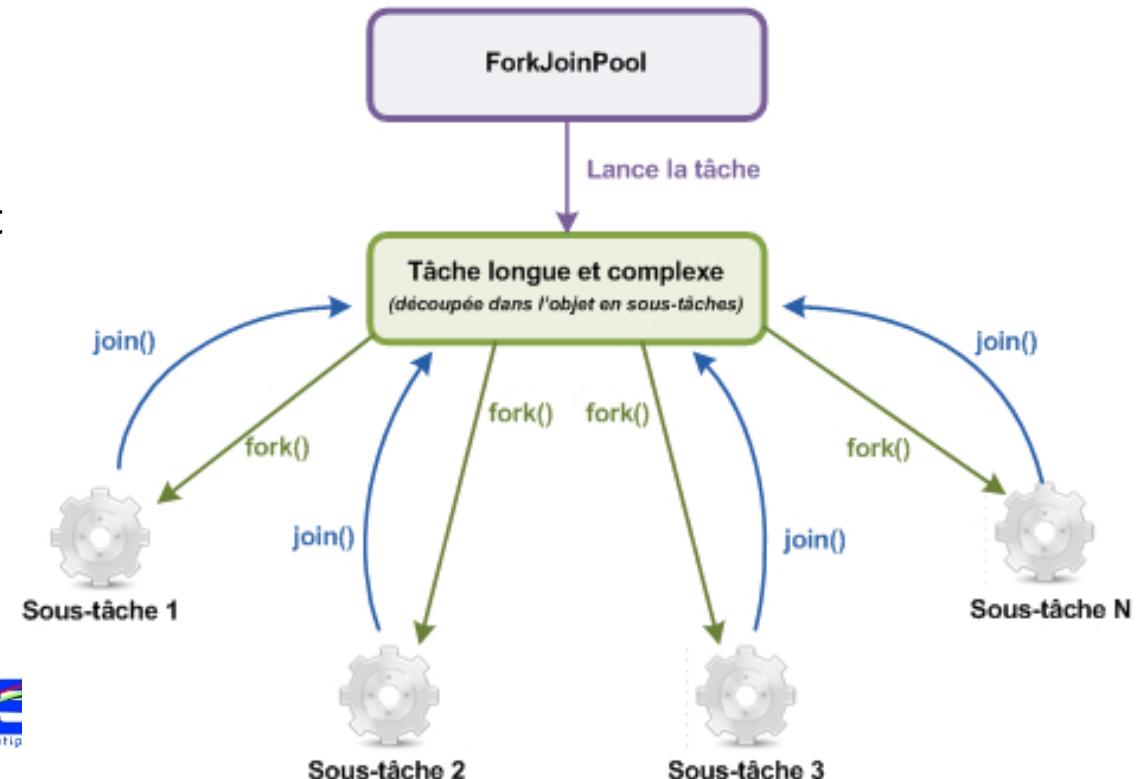
Etape 4. Mise en oeuvre du modèle fork-join en Java

Revoir les slides à partir de :

https://homes.cs.washington.edu/~dkg/teachingMaterials/spac/grossmanSPAC_forkJoinFramework.html

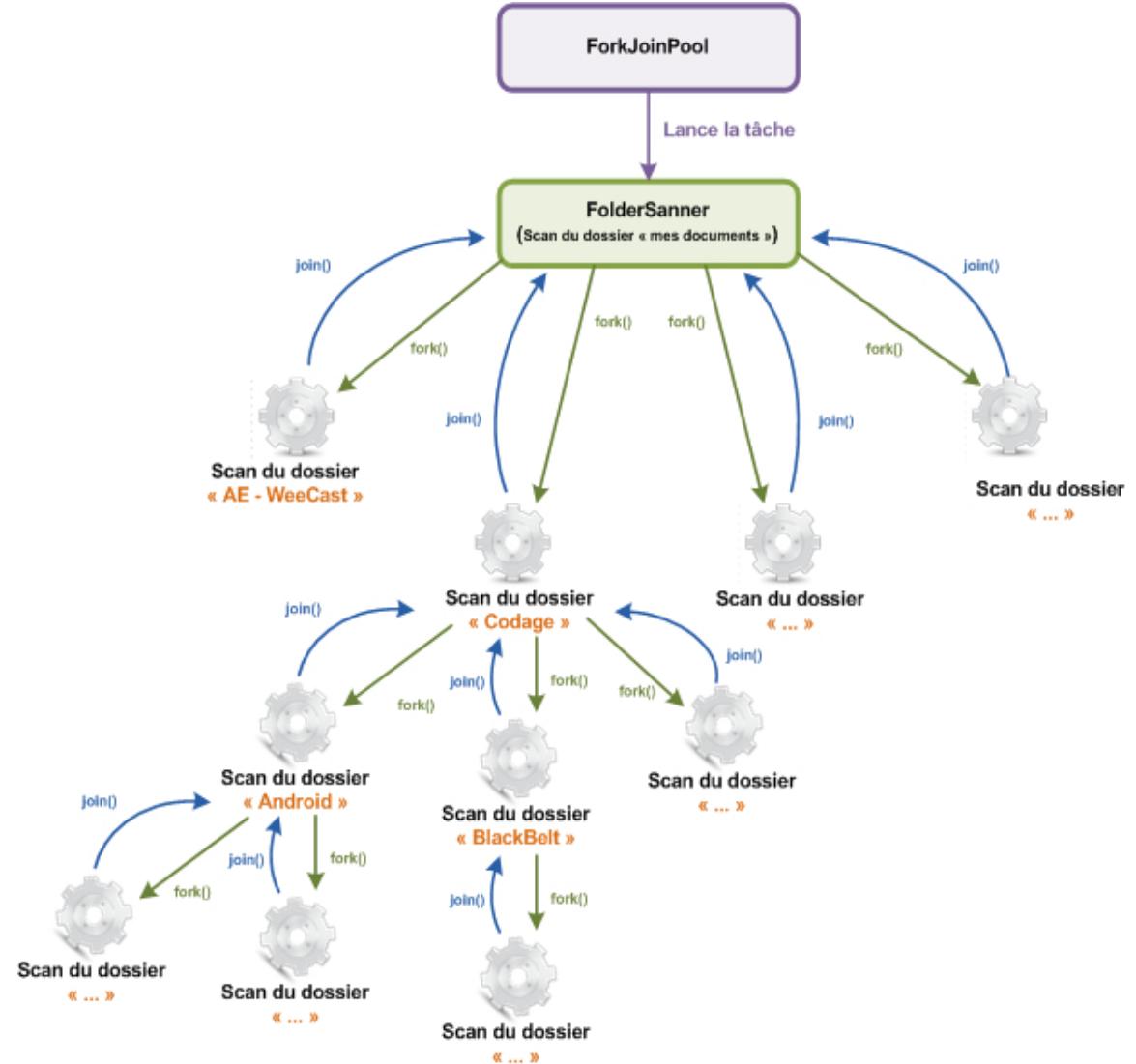
Mise en oeuvre du pattern ‘forkjoin’ en Java

- Exemple tiré de :
<http://fr.openclassrooms.com/informatique/cours/apprenez-a-programmer-en-java/depuis-java-7-le-pattern-fork-join>
- Le problème
 - Parcourir tous les répertoires de mon \$HOME et afficher le nombre de fichier contenant le suffixe ‘psd’
- Se décompose assez aisément en sous-problèmes indépendants
 - L’analyse d’un répertoire peut être réalisé de manière récursive par analyse des sous répertoire



Algorithme

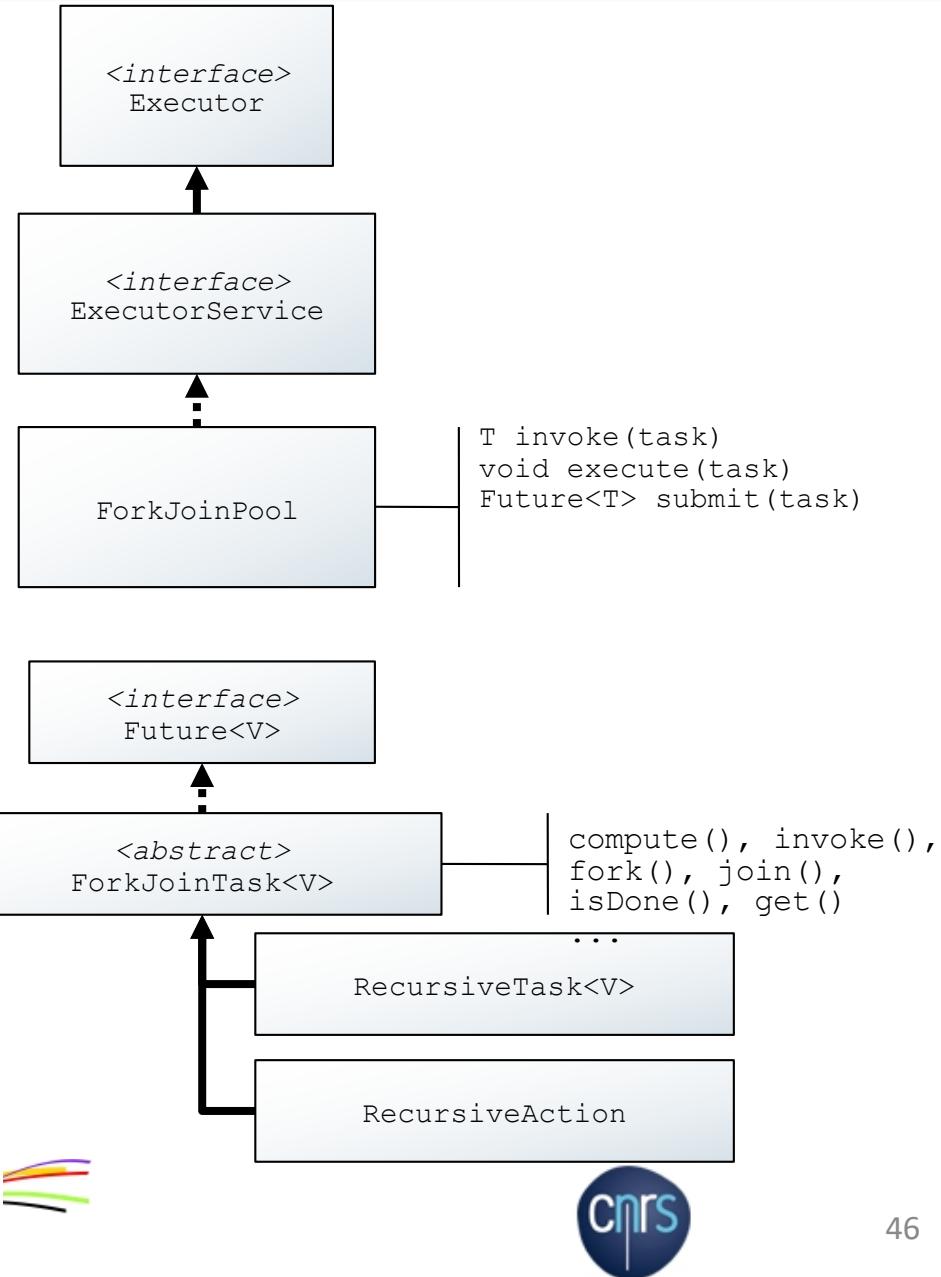
- Dans chaque répertoire :
 - On lance une nouvelle analyse pour chaque sous-répertoires
 - On compte le nombre de fichier correspondant au filtre
 - On attend la terminaison des analyses des sous répertoires et on additionne le nombre de fichier trouvé



- Approche séquentielle
- Il y a 92 fichier(s) portant l'extension *.psd
- Temps de traitement : 67723
- Utilisation de 4 coeurs
- Il y a 92 fichier(s) portant l'extension *.psd
- Temps de traitement : 78679
- Mais temps d'attente presque 3 fois plus cours

Le pattern forkjoin

- `ForkJoinPool` : représente un pool de thread qui reçoit la liste des tâches à réaliser
 - `invoke` : lance les tâches passées en paramètre de manière synchrones
 - `execute` : lance les tâches passées en paramètre de manière asynchrones
 - `submit` : lance la tâche passée en paramètre de manière asynchrone et renvoie un objet futur immédiatement
- `ForkJoinTask` : représente une tâche unique
 - `invoke()` : exécution de la tâche par la thread courante
 - `fork()` : lance une autre tâche dans le même pool que la thread courante
 - `join()` : retourne le résultat de l'exécution de cette tâche
- Un `ForkJoinPool` exécute des `ForkJoinTasks`



ForkJoinTask

- RecursiveAction
 - Implémente Future<Void>
 - Modélise un traitement qui ne renvoie pas de valeur
 - Peut néanmoins modifier les données passées en paramètre
 - Ex : trier un tableau "in-place"
- RecursiveTask<V>
 - Implémente Future<V>
 - Modélise un traitement récursif qui renvoie une valeur
 - Ex : calculer la taille totale d'une arborescence de fichiers

compute () est la méthode exécutée par la tâche

- équivalent à run () pour les thread

Les tâches peuvent être appelées de manière synchrone ou asynchrone

- Synchrone : invoke ()
- Asynchrone : fork ()

Main

```
// Nombre de processeurs disponibles
int processeurs =
    Runtime.getRuntime().availableProcessors();

// Création du pool de thread
ForkJoinPool pool =
    new ForkJoinPool(processeurs);

// Création de l'objet FolderScanner
FolderScanner fs =
    new FolderScanner(chemin, filtre);

// Analyse
resultat = fsSEQUENTIALScan();

// Un transparent qui suit décrit :
//      sequentialScan
// Nombre de processeurs disponibles
int processeurs =
    Runtime.getRuntime().availableProcessors();

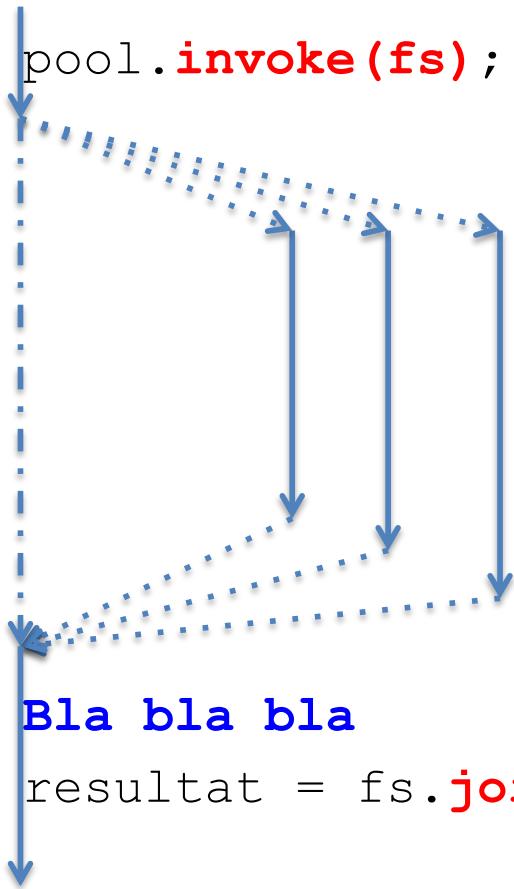
// Création du pool de thread
ForkJoinPool pool =
    new ForkJoinPool(processeurs);

// Création de l'objet FolderScanner
RecursiveTask<Long> fs =
    new FolderScanner(chemin, filtre);

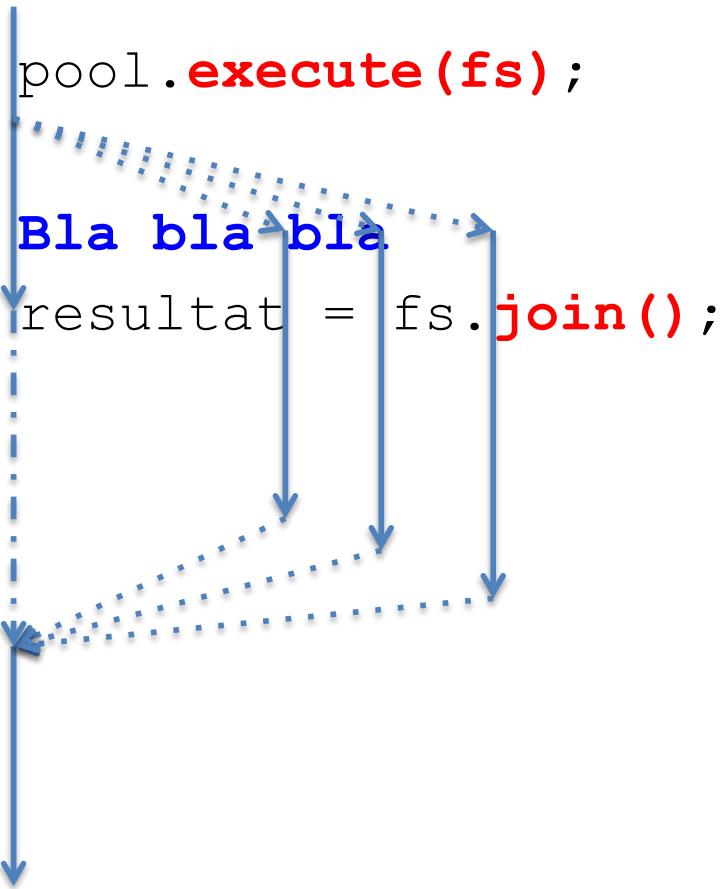
// Analyse
resultat = pool.invoke(fs); // exécute compute
// Un transparent qui suit décrit :
//      compute
```

Synchrone ou Asynchrone ? (ForkJoinPool)

Resultat = pool.**invoke**(fs);
est équivalent à :



Resultat = pool.**execute**(fs);
est équivalent à :



FolderScanner

```
public class FolderScanner {  
    public long sequentialScan() {  
        // Récupération de la liste des fichiers du répertoire d'analyse  
        DirectoryStream<Path> listing = Files.newDirectoryStream(path));  
        for (Path nom : listing) {  
            if (Files.isDirectory(nom.toAbsolutePath())) {  
                // S'il s'agit d'un dossier il est analysé  
                result += new FolderScanner(  
                    nom.toAbsolutePath(),  
                    this.filter)  
                    .sequentialScan();  
            }  
  
            // Récupération de la liste des fichiers répondant au filtre  
            DirectoryStream<Path> listing =  
                Files.newDirectoryStream(path, this.filter));  
            for (Path nom : listing){  
                // Pour chaque fichier, on incrémente le compteur  
                result++;  
            }  
  
            return result;  
    }
```

```
public class FolderScanner extends RecursiveTask<Long>  
{  
    public Long compute() {  
        // Récupération de la liste des fichiers du répertoire d'analyse  
        DirectoryStream<Path> listing = Files.newDirectoryStream(path));  
        for (Path nom : listing) {  
            if (Files.isDirectory(nom.toAbsolutePath())) {  
                // S'il s'agit d'un dossier il est analysé  
                subTasks.add(new FolderScanner(  
                    nom.toAbsolutePath(),  
                    this.filter)  
                    .fork());  
            }  
        }  
  
        // Récupération de la liste des fichiers répondant au filtre  
        DirectoryStream<Path> listing =  
            Files.newDirectoryStream(path, this.filter));  
        for (Path nom : listing) {  
            // Pour chaque fichier, on incrémente le compteur  
            result++;  
        }  
  
        // Récupération du résultat de toutes les sous-tâches  
        for (ForkJoinTask<Long> subTask : subTasks)  
            result += subTask.join();  
  
        return result;  
    }
```

Synchrone ou asynchrone ? (ForkJoinTask/RecursiveTask/RecursiveAction)

```
if (Files.isDirectory(nom.toAbsolutePath())) {  
    // S'il s'agit d'un dossier il est analysé  
    ForkJoinTask<Long> fs = new FolderScanner(  
        nom.toAbsolutePath(),  
        this.filter)  
    .invoke();  
}  
}
```

fs.invoke();

Bla bla bla
fs.join();

```
if (Files.isDirectory(nom.toAbsolutePath())) {  
    // S'il s'agit d'un dossier il est analysé  
    ForkJoinTask<Long> fs = new FolderScanner(  
        nom.toAbsolutePath(),  
        this.filter)  
    .fork();  
}  
}
```

fs.fork();

Bla bla bla
fs.join();

↓

Asynchronisme et tâches récursives (RecursiveTask)

// correct

```
protected Long compute() {  
    List<ForkJoinTask<Long>> subTasks = new ArrayList<>();  
    long size = 0;  
  
    for(File f : root.listFiles()) {  
        if (f.isDirectory()) {  
            subTasks.add(new Task(f).fork());  
        } else {  
            size += f.length();  
        }  
    }  
  
    for (ForkJoinTask<Long> subTask : subTasks) {  
        size += subTask.join();  
    }  
    return size;  
}
```

// Incorrect

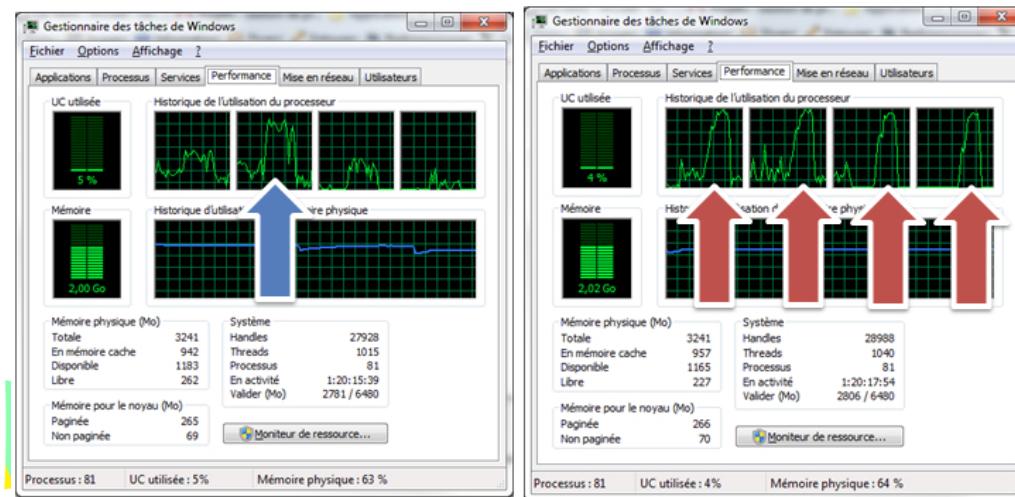
```
protected Long compute() {  
    ForkJoinTask<Long> subTask;  
    long size = 0;  
  
    for (File f : root.listFiles()) {  
        if (f.isDirectory()) {  
            subtask = new Task(f).fork();  
            size += subtask.join();  
        } else {  
            size += f.length();  
        }  
    }  
    return size;  
}  
  
/* L'asynchronisme souhaité (fork)  
n'est pas exploité : on attend tout de  
suite */
```

Bonnes pratiques

- Attention au coût de gestion
 - Le bénéfice obtenu en parallélisant un traitement doit être supérieur au coût de gestion par le framework
 - Trouver la bonne granularité
- Attention à la consommation mémoire
 - Grosses structures : préférer la modification "in-place"
 - Découverte dynamique des sous-tâches
- Attention à la complexité
 - Optimisation prématuée ?
 - Maintenance

Optimisations

- Framework Fork / Join
 - Threads \geq Degré de parallélisme
 - Les tâches en attente de join() sont mises de côté pour permettre à d'autres tâches d'être traitées
- Tâches
 - Eviter la synchronisation manuelle
 - Utiliser fork() et join() uniquement
 - Optimiser la granularité à l'aide des statistiques du pool
 - getQueuedSubmissionCount(), getStealCount()...
 - Possibilité d'optimisation dynamique



Quelques références qui m'ont aidées pour préparer ce cours

- API Java 7, package java.util.concurrent
<http://download.java.net/jdk7/docs/api/>
- Etude de Doug Lea
<http://gee.cs.oswego.edu/dl/papers/fj.pdf>
- Des blogs/tutoriaux
 - <http://www.oracle.com/technetwork/articles/java/fork-join-422606.html>
 - <http://blog.paumard.org/2011/07/05/java-7-fork-join/>
 - <http://sdz.tdct.org/sdz/le-framework-executor.html>
 - <http://fr.openclassrooms.com/informatique/cours/apprenez-a-programmer-en-java/depuis-java-7-le-pattern-fork-join>