

Diviser pour calculer plus vite

Fork - Join

riveill@unice.fr

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



Diviser pour calculer plus vite

Fork - Join

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



Association tâche - thread

- Jusqu'à présent on a l'association
 - 1 tâche est directement associé à 1 thread
 - Et on a autant de thread que de tâche à exécuter
- On peut se poser des questions de nature plus algorithmique :
 - Comment obtenir un gain de performance en adaptant le découpage tâche / thread et en optimisant le nombre de thread ?
- Exemple : somme de 2 tableaux de même taille

```
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 plusieurs cœur
 - On peut 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
 - Possibilité d'un très grand nombre de thread

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 ?

- Sur GPU, on pourrait le faire... attention néanmoins au transfert mémoire entre le CPU et le GPU qui prend du temps
- Sur CPU, ce n'est pas une bonne idée.
 - **Créer un thread coute cher**
 - Il faut au grand 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 (appelé **chunk**)

```
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 coeurs, on pourrait lancer p threads et donc :
 - $\text{CHUNK} = \text{int}(n/p)$
- Mais si certains coeurs 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... Et pourtant c'est ce que nous allons essayer de faire à travers le modèle fork / join (ou d'une manière plus générale map / reduce)

Vers une abstraction de plus : les tâches

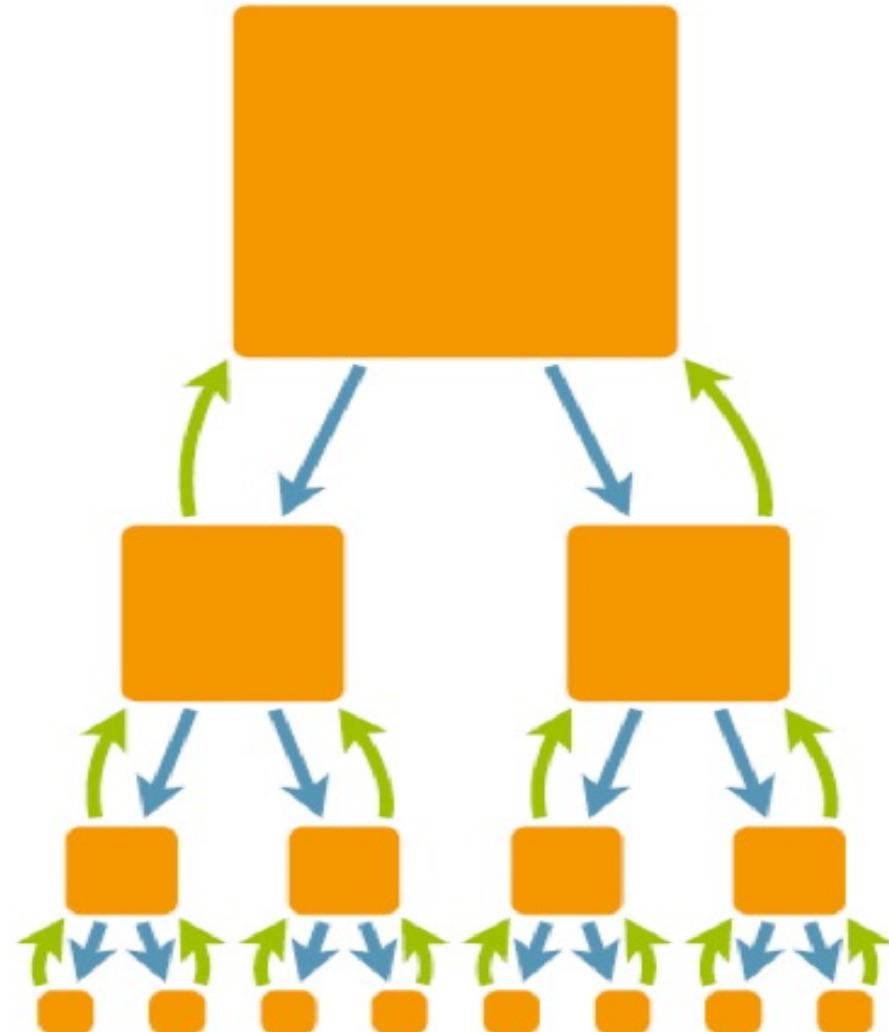
- Pour faciliter la répartition dynamique du travail, on peut :
 - introduire une notion de tâche,
 - poser que fork/join créent/attendent une tâche,
 - utiliser un nombre fixe de processus légers (worker threads) pour exécuter ces tâches,
 - ce qui suppose un algorithme efficace de répartition des tâches 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

Possible lorsque le traitement est décomposable en **sous-tâches indépendantes**

Exemple de découpage :

- **statique** : découper un tableau en zones fixes
- **dynamique** : découvrir une arborescence de fichiers
 - Attention à maîtriser le volume de tâches créées
- **récursif**



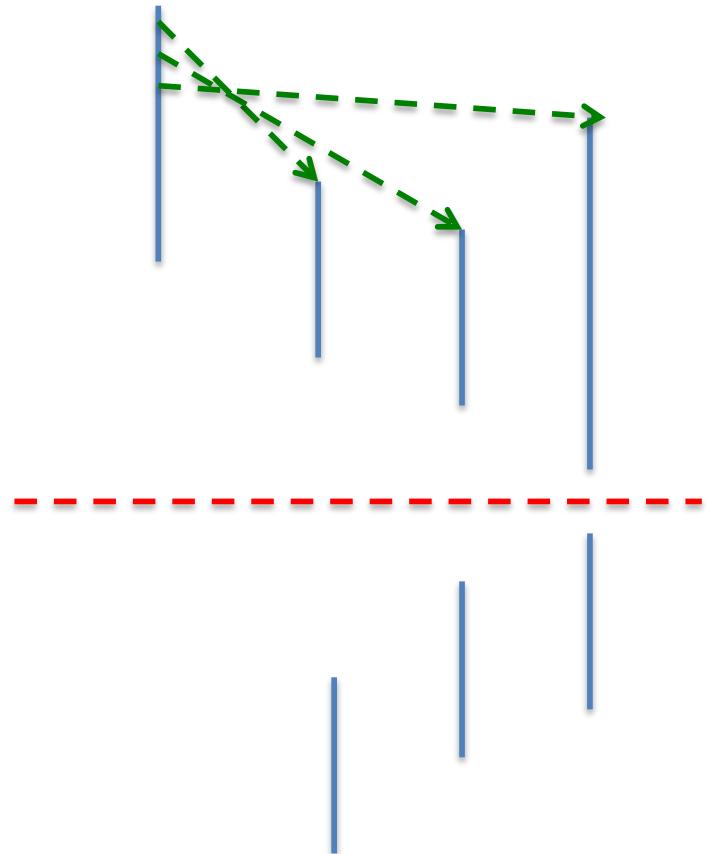


Etape 1

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

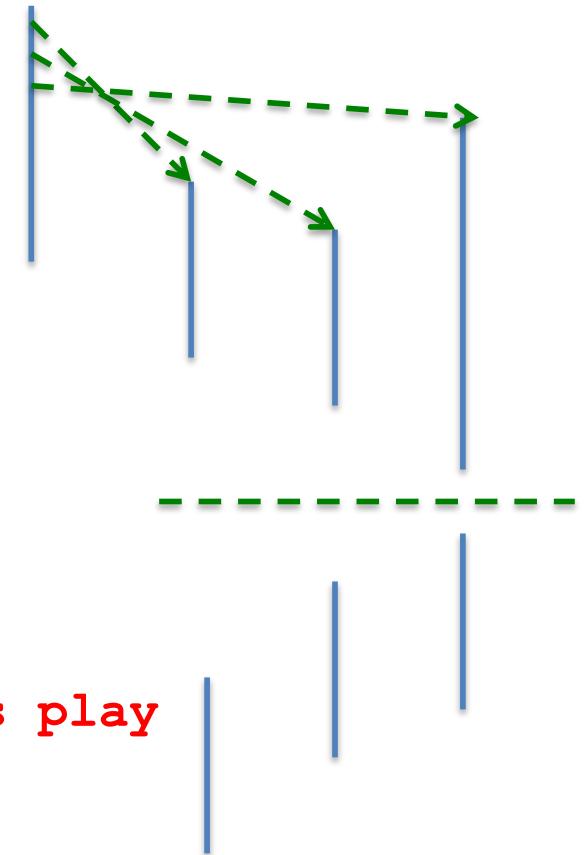
Thread 3 is waiting on barrier

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

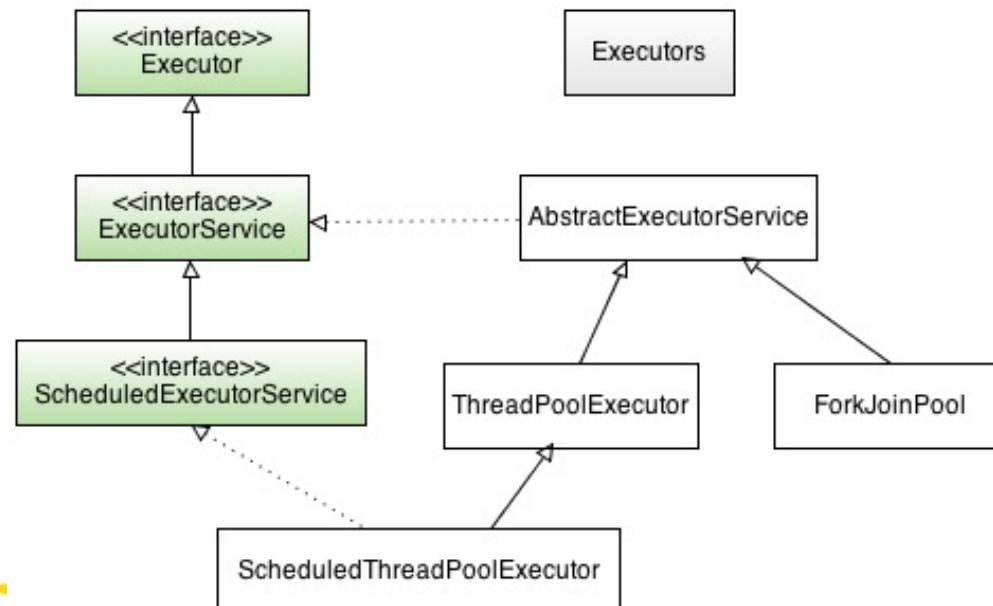
- Le parallélisme c'est bien mieux si on peut **effectivement** calculer en parallèle
 - Exploitation d'une l'architecture multi-cœur
 - Java
 - Nom du thread courant :
Thread.currentThread().getName()
 - Nombre de cœur sur le processeur :
Runtime.getRuntime().availableProcessors()
 - Quel est l'intérêt 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

- Création d'une tache

```
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());  
    }  
}
```

- Association d'une tache à une thread

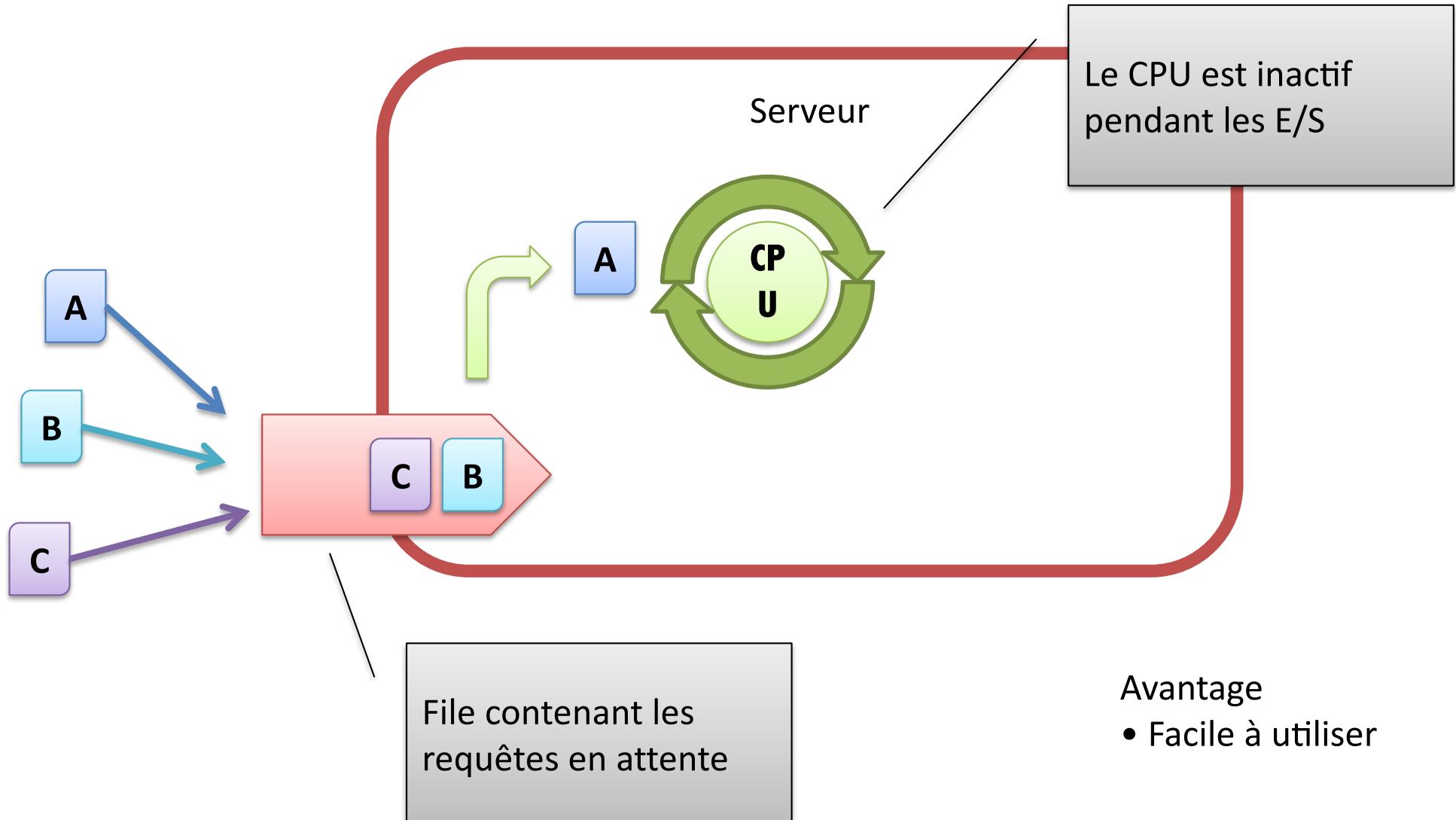
```
Thread thread = new Thread(new MonRunnable());  
thread.start();
```

```
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

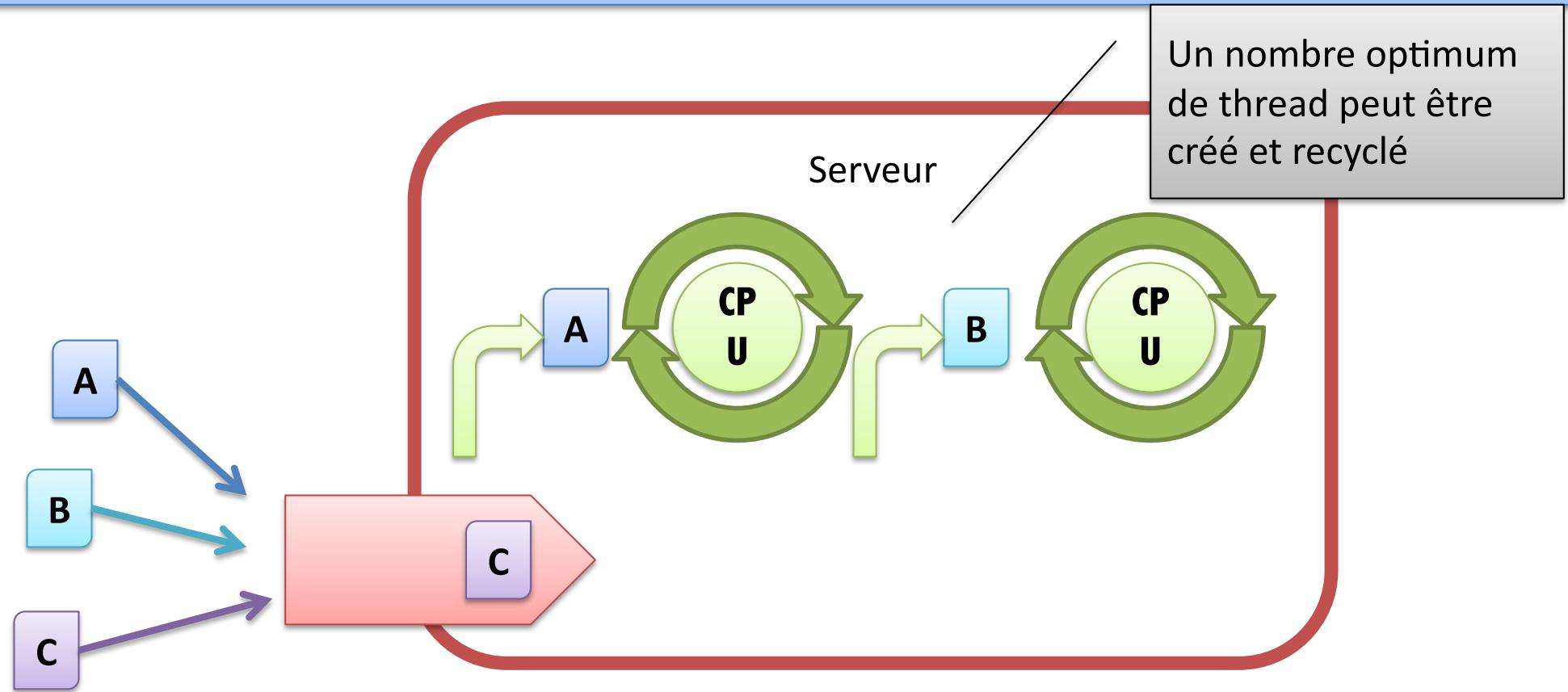
Fin execution dans le thread pool-1-thread-1

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 taches selon le modèle choisi  
for(Runnable r : runnables) {  
    executor.execute(r);  
}  
  
//attente de la terminaison de toutes les taches  
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

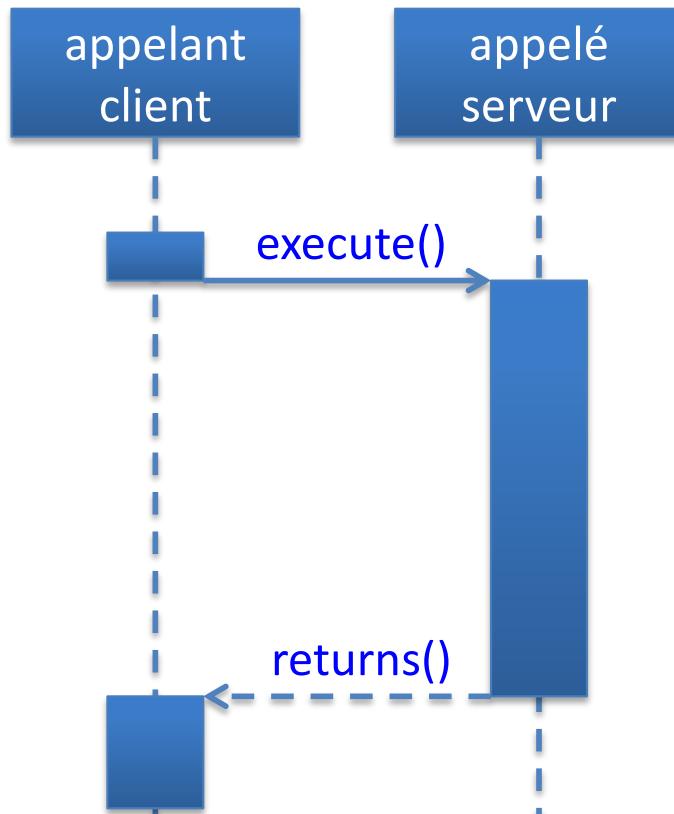
Fin execution dans le thread pool-1-thread-1

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

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

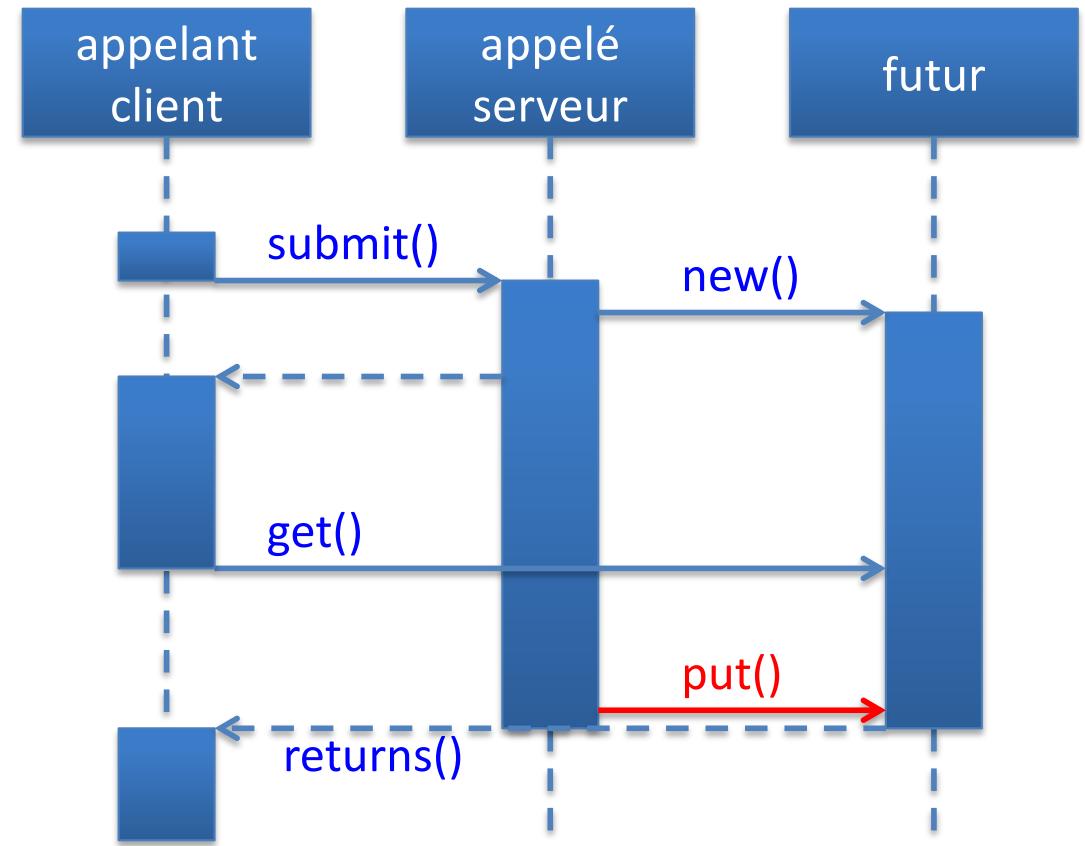
Un modèle : l'appel asynchrone avec future

- 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
<pre>public interface Runnable { void run(); }</pre>	<pre>Public interface Callable<V>{ V call() throwsException; }</pre>
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

Classe Callable

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

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

La classe Future<T>

Un scénario :

- Je vais à Mac Do
- Je passe ma commande
- Je paye et je reçois un ticket

Jusqu'à présent

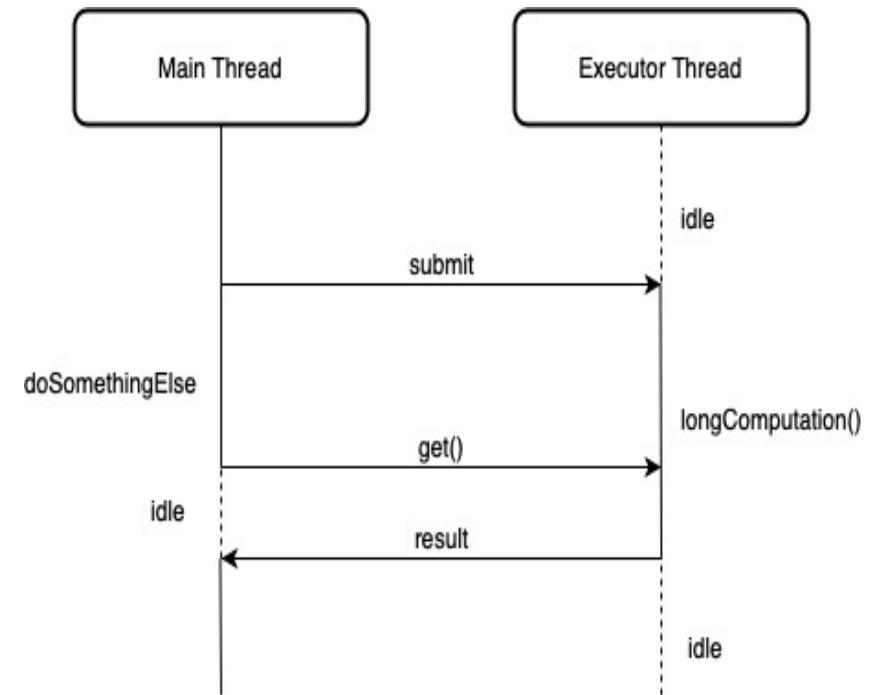
- Je vais au comptoir et j'attends que ma commande est prête

Avec un futur

- Je vais m'asseoir
- Je sors mon téléphone et je joue
- Quand je suis affamé, je vais chercher ma commande au comptoir
- Je la reçois si elle est prête, sinon j'attends
- Je peux même être prévenu que ma commande est prête

La classe Future<T>

- Permet de récupérer un résultat quand on en a besoin et de faire autre chose tant que l'on en a pas besoin
- Le ticket reçu lors de la commande est important.
- Avec un futur, il permet :
 - `get()` récupère le résultat de la fonction. Si le traitement est toujours en cours, le thread qui appelle la fonction `get` reste bloqué jusqu'à obtenir la réponse.
 - `cancel()` permet d'annuler le traitement en cours
 - `isDone()` permet de vérifier si le traitement asynchrone est terminé. Ceci est utile lorsque l'on veut faire de l'*active-pooling*
 - `isCancelled()` permet de vérifier si le traitement a été annulé



Exemple d'utilisation

```
public final static main(String... args) {  
    ExecutorService es = Executors.newSingleThreadExecutor();  
  
    Future<Integer> f1 = es.submit(()->myComputation(2, 500));  
    Future<Integer> f2 = es.submit(()->myComputation(3, 1000));  
    // Do something  
    System.out.println(f1.get() + f2.get());  
}  
  
private Integer myComputation(Integer value, Long sleepTime)  
{  
    // Simuler une opération lente  
    Thread.sleep(sleepTime);  
    return value * value;  
}
```

CompletableFuture<T>

Objectif:

- Enchaîner des opérations asynchrone de manière simple

Les principales méthodes de CompletableFuture<T>

- *join()*
 - bloque le thread courant.
 - Équivalent au get de la classe Future<T>
- *runAsync()*
 - Créer un CompletableFuture futur à partir d'un Runnable
- *thenCombineAsync()*
 - permet de composer un nouveau CompletableFuture à partir des deux CompletableFuture indépendants
- *thenApplyAsync()*
 - permet de lancer un traitement asynchrone à partir d'un autre CompletableFuture.
 - Ce nouveau traitement sera lancé lorsque le premier CompletableFuture sera complété.

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

Un bon tutorial:

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

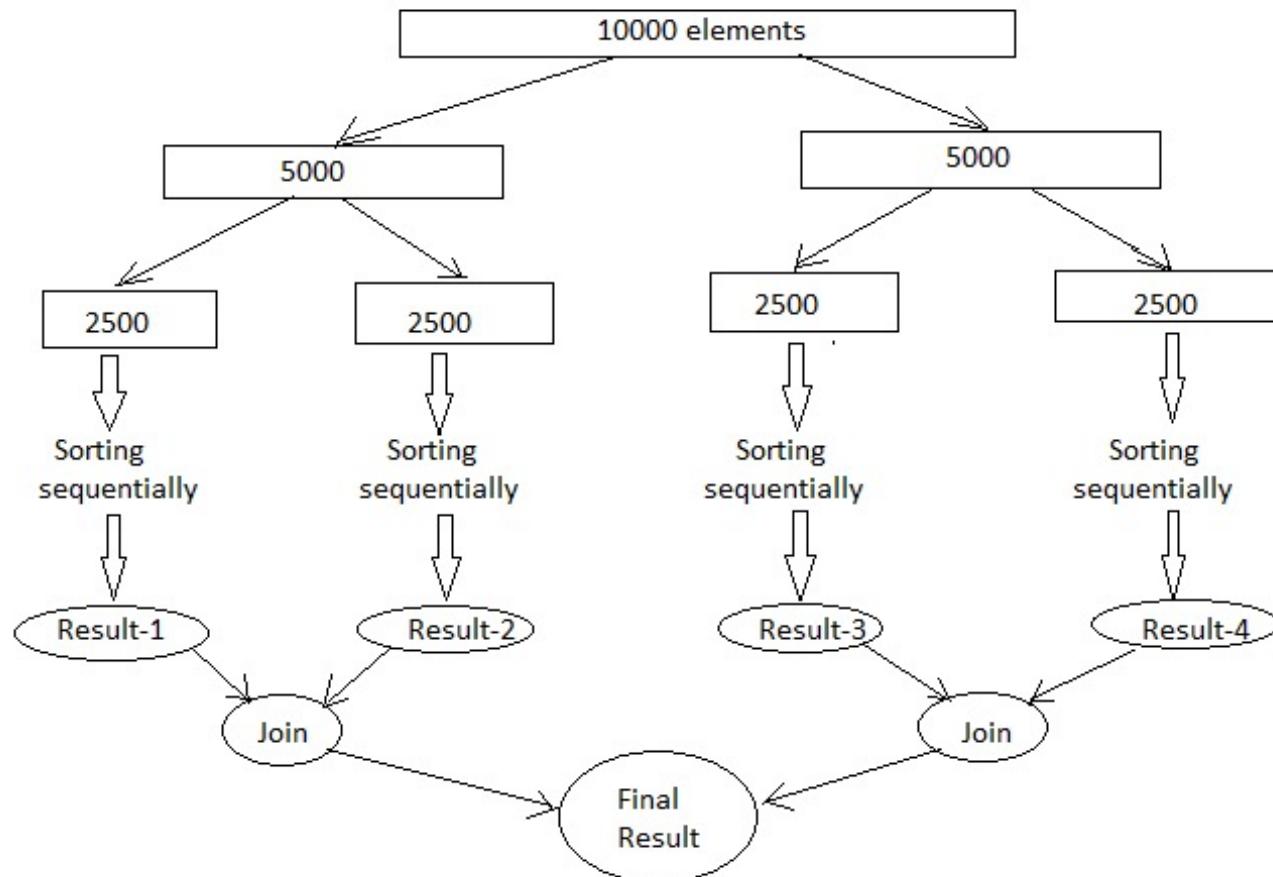
Le pattern Fork/Join

- Introduit dans Java 7, simplifié en Java 8
- Fournit des outils permettant d'accélérer le traitement parallèle en essayant d'utiliser tous les cœurs de processeur disponibles
 - Utilisation d'une approche diviser pour mieux régner
- En pratique
 - On commence par décomposer récursivement une tâche en sous-tâches
 - Sous tâches : plus petites, indépendantes, exécutées de manière asynchrone.
 - Ensuite, on construit le résultat
 - En assemblant les résultats de chacune des sous-tache
- Pour assurer une exécution parallèle efficace, utilisation depuis Java 8
 - D'un pool de threads appelé **ForkJoinPool**
 - Qui gère des threads de type **ForkJoinWorkerThread**.

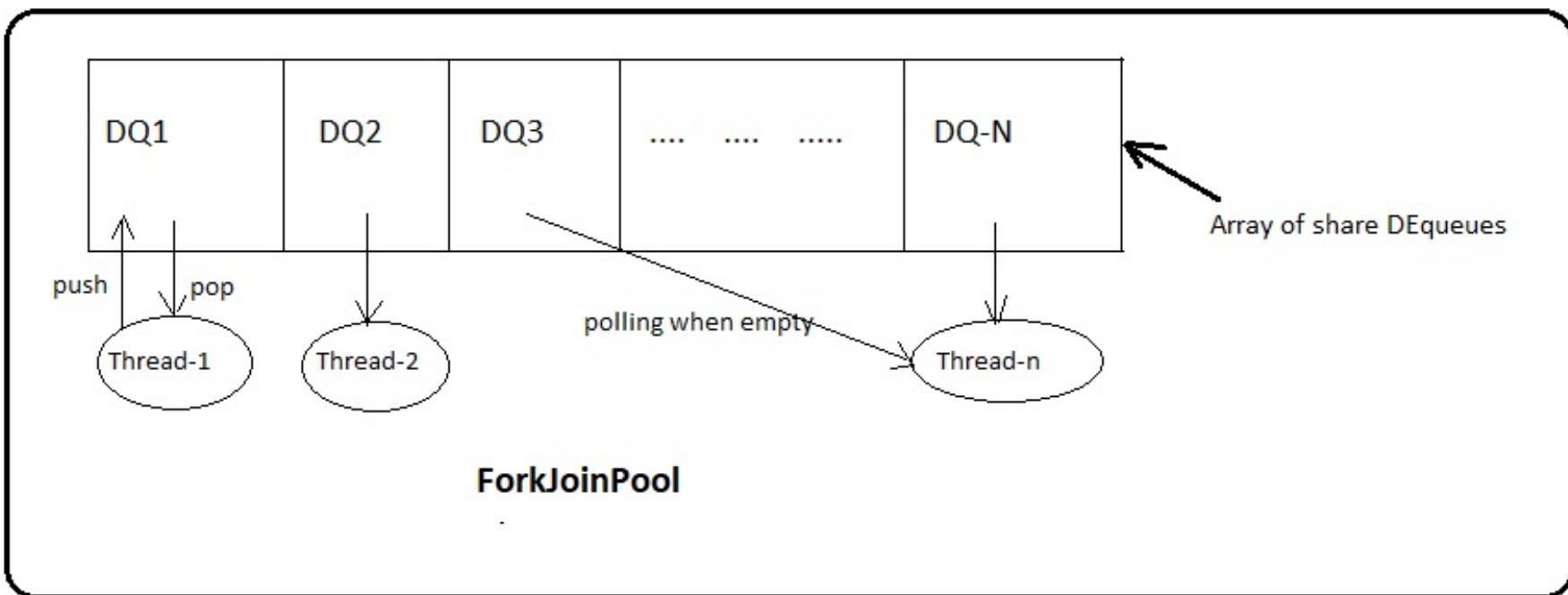
Fork/Join pattern in pseudo code

```
If (Task is small) {  
    Execute the task  
} else {  
    //Split the task into smaller chunks  
    ForkJoinTask first = getFirstHalfTask();  
    ForkJoinTask second = getSecondHalfTask();  
    first.fork();  
    second.fork();  
    result = join(first.getResult(), second.getResult());  
}
```

Un exemple: recherche du plus grand élément d'un tableau



Un exemple: recherche du plus grand élément d'un tableau



Un exemple: recherche du plus grand élément d'un tableau

```
public class ForkJoinPoolTest {  
    public static void main(String[] args) {  
        int[] array = yourMethodToGetData();  
        ForkJoinPool pool = new ForkJoinPool();  
  
        Integer max = pool.invoke(new FindMaxTask(array, 0, array.length));  
  
        System.out.println(max);  
    }  
  
    static class FindMaxTask extends RecursiveTask<Integer> {  
        ...  
    }  
}
```

Un exemple: recherche du plus grand élément d'un tableau

```
static class FindMaxTask extends RecursiveTask<Integer> {  
    private int[] array;  
    private int start, end;  
    public FindMaxTask(int[] array, int start, int end) {  
        this.array = array;  
        this.start = start;  
        this.end = end;  
    }  
    @Override protected Integer compute() {  
        if (end - start <= 3000) {  
            // easy task → do it  
            int max = -99;  
            for (int i = start; i < end; i++) { max = Integer.max(max, array[i]); }  
            return max;  
        } else {  
            // Sophisticated task → divide it  
            ...  
        }  
    }  
}
```

Un exemple: recherche du plus grand élément d'un tableau

```
} else {  
    // Sophisticated task → divide it  
    int mid = (end - start) / 2 + start;  
  
    FindMaxTask left = new FindMaxTask(array, start, mid);  
    FindMaxTask right = new FindMaxTask(array, mid + 1, end);  
  
    ForkJoinTask.invokeAll(right, left);  
  
    int leftRes = left.getRawResult();  
    int rightRes = right.getRawResult();  
  
    return Integer.max(leftRes, rightRes);  
}
```

Q&A

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

