Fork / Join

Rémi Forax

Vocabulaire

Concurrence

- Exécuter plusieurs calculs en même temps

Parallèle

- Un calcul est décomposé et exécuté sur plusieurs threads

Vectorisé

- Une opération simple (+, min, etc) est exécutée par un même thread sur plusieurs données adajcences en mémoire (sur 128 bits, 256 bits, etc)

Distribué

- Un calcul est décomposé et exécuté sur plusieurs machines

Start / Join

Historiquement, le modèle utilisé est start / join

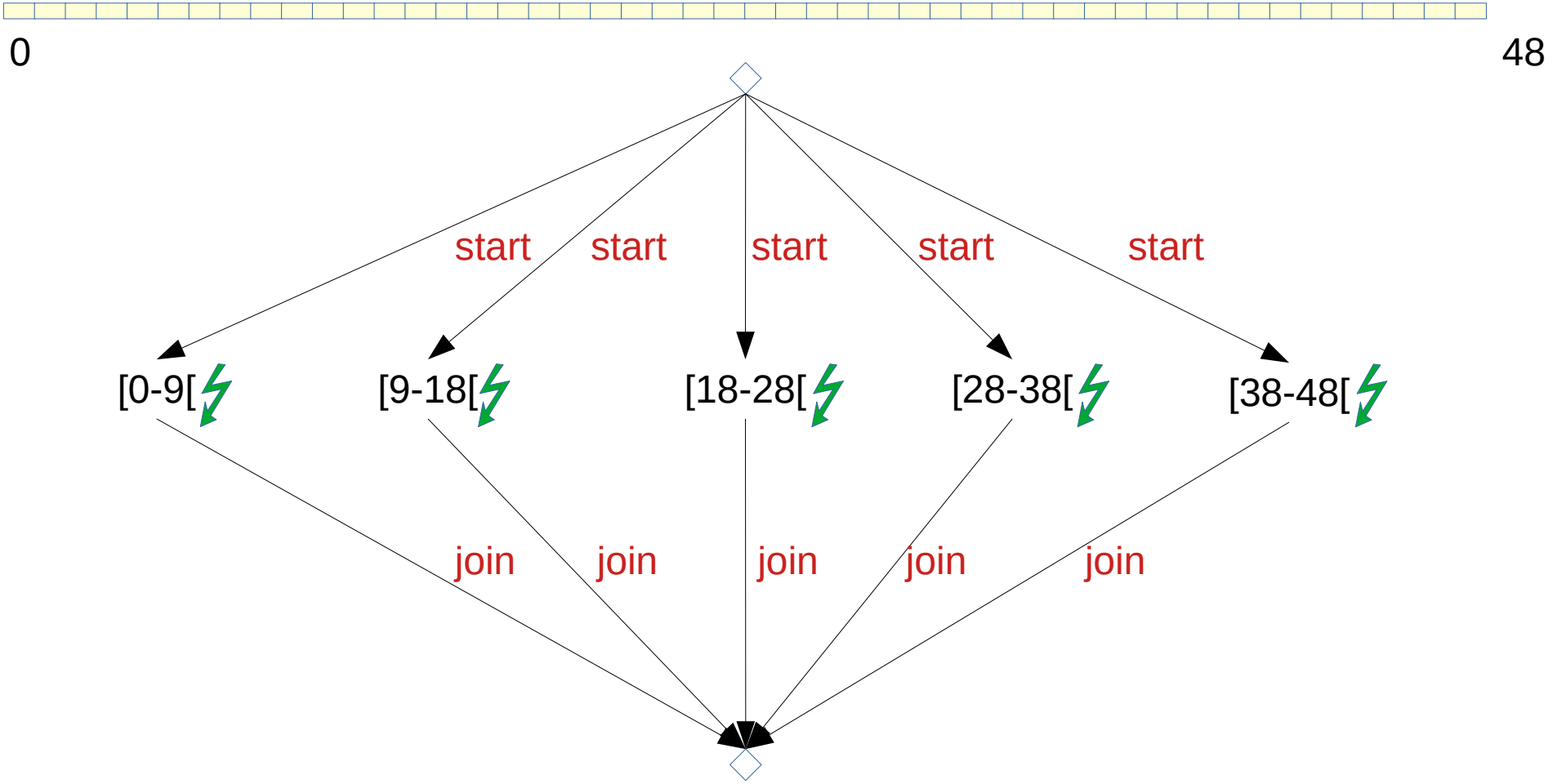
Exemple, faire *2 aux valeurs d'un tableau avec 5 threads

```
var threads = IntStream.range(0, 5)
  .mapToObj(i -> new Thread(() -> {
    var startIndex = ...
    var endIndex = ...
    for(var i = startIndex; i < endIndex; i++) {
       array[i] *= 2;
    }
  }))
  .toList();
for(var thread: threads) { thread.start(); }
for(var thread: threads) { thread.join();  }
```

Recopié du modèle fork() / waitpid() sous Unix

Le modèle start / join

Avec un tableau de 48 cases



Problèmes du modèle start / join

Le modèle start / join marche bien

- si toutes les données sont connus en amont (*upfront*)

  Marche bien pour un tableau, moins bien pour un arbre rouge/noir car il faut tout parcourir pour voir toutes les données

- si le calcul est homogène en temps

  Si le calcul est long seulement pour certain index, une thread peut être encore entrain de faire des calculs alors que les autres ont finis

  => problème de répartition de la charge de calcul

Divide and Conquer

Solution naive “divide and conquer”,

évite d’avoir à connaitre tout les données en amont

```
process(int startIndex, int endIndex)
  if endIndex – startIndex < SMALL
    // for(...)
    return

  var middle = (startIndex + endIndex) >>> 1;
  start process(startIndex, middle);
  start process(middle, endIndex);
  join
  join
}
```

Fork / Join

Evolution de la solution naive “divide and conquer”,

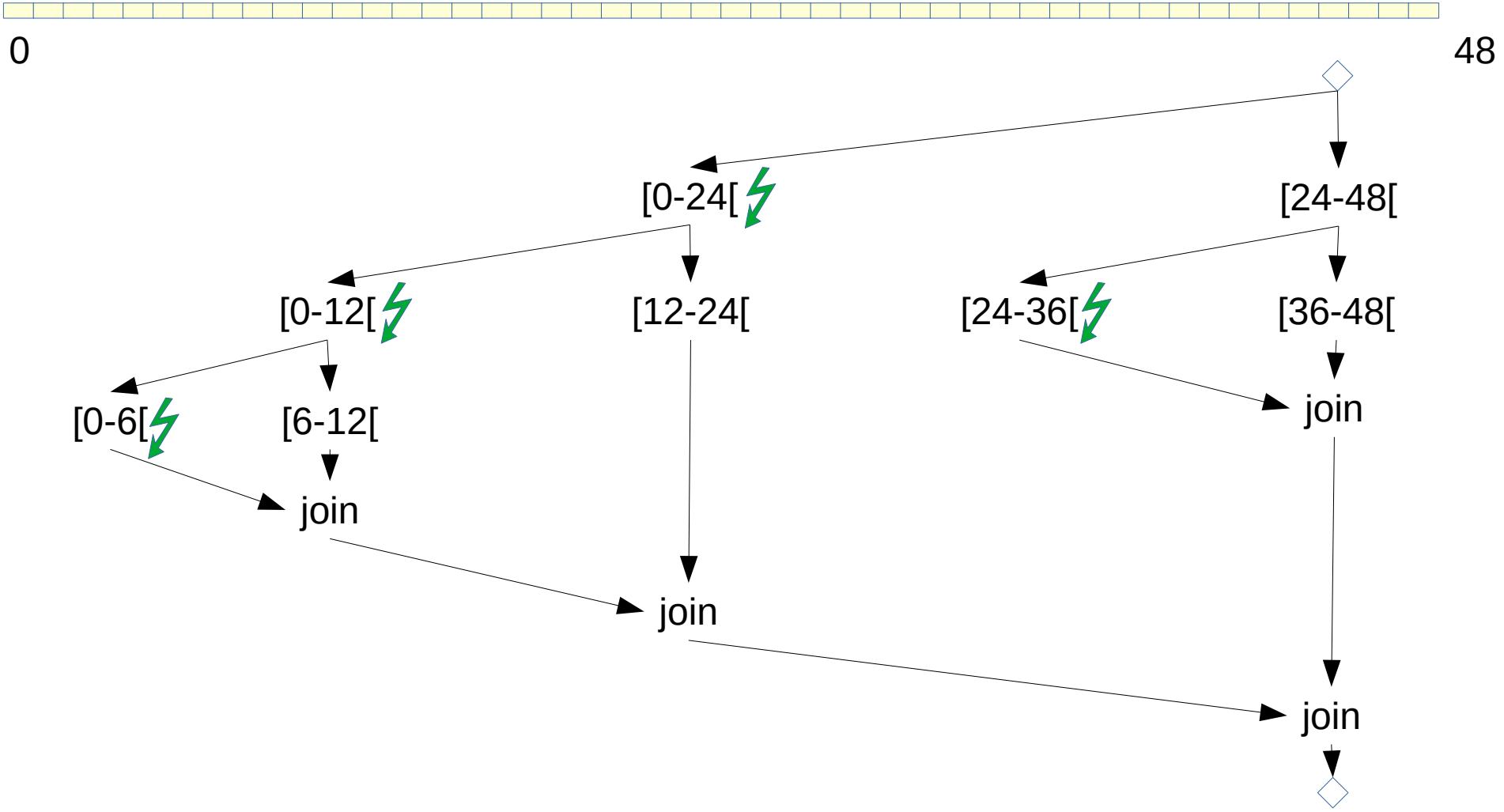
on évite d’avoir des threads en attentes

```
process(int startIndex, int endIndex
  if endIndex – startIndex < SMALL
    // for(...)
    return ...

  var middle = (startIndex + endIndex) >>> 1;
  fork process(startIndex, middle);
  var result2 = process(middle, endIndex);
  var result1 = join
  return combine result1 result2
}
```

Le modèle fork / join

Avec un tableau de 48 cases



Problème du modèle fork / join

On a toujours le problème de la répartition de la charge de calcul si le calcul est pas homogène en temps

Solution: découpler la partie de calcul effectuer de la thread qui effectue le calcul

=> pool de thread + work stealing

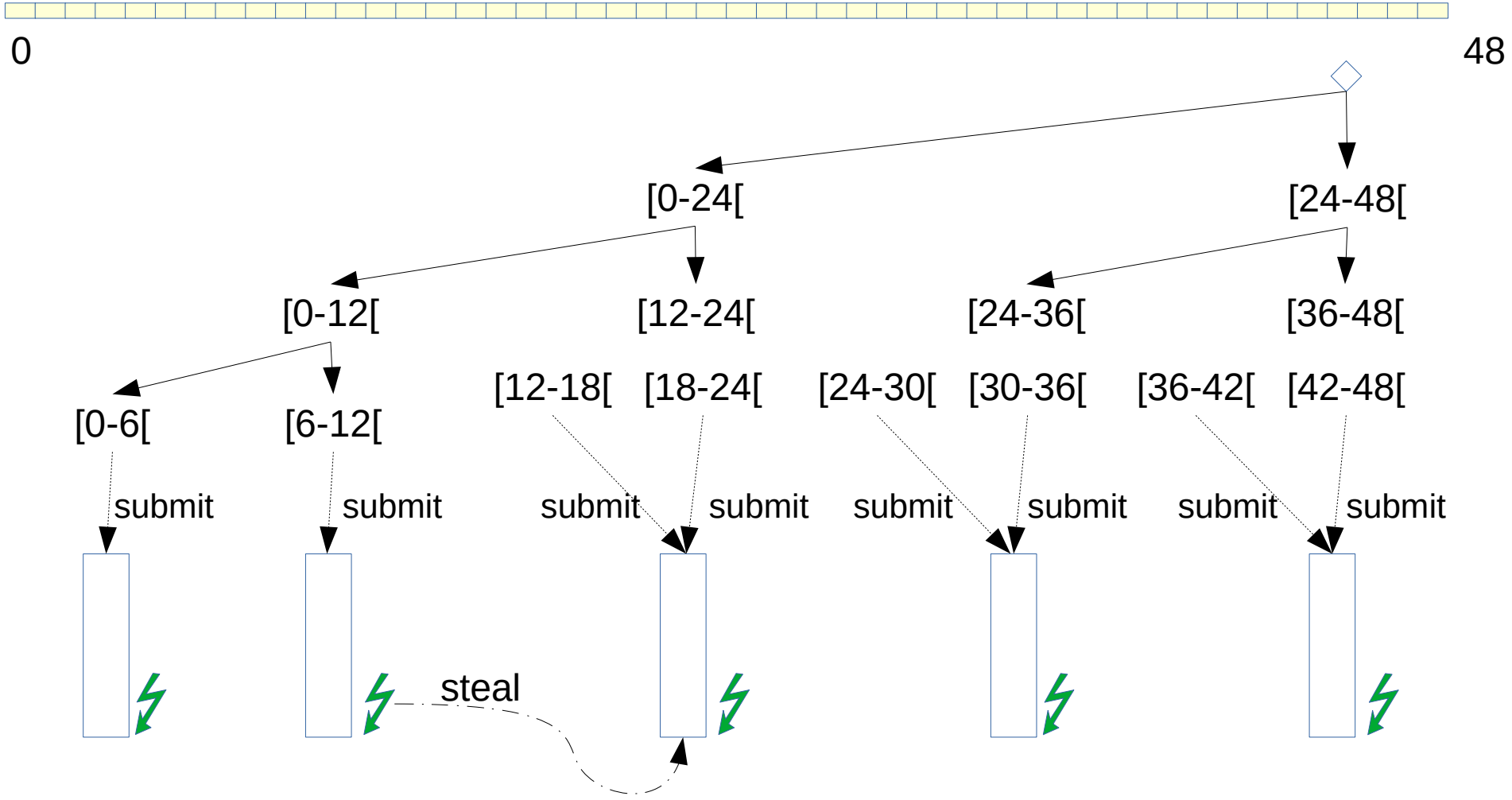
Thread Pool avec Work Stealing

- Le nombre de threads est fixe
- Chaque thread à sa *queue* de tâche
- Si un thread a fini toutes les tâches de sa *queue* (si la *queue* est vide), il va voler (*steal*) des tâches **à la fin** dans les queues des autres threads

  => Le fait d'aller voler des tâches permet d'équilibrer la charge de travail

fork / join + work stealing

Avec un tableau de 48 cases



Régler fork / join

Sélection de SMALL

- Si SMALL trop gros, pas assez de tâches, pas assez de parallèlisme
- Si SMALL trop petit, trop de tâches, trop d’allocations + queue trop grande

```
process(int startIndex, int endIndex
  if endIndex – startIndex < SMALL
    // for(...)
    return ...

  ...
}
```

Régler fork / join (2)

Combine doit être une opération **associative**

On ne peut pas décomposer le calcul
si combine(combine(a, b), c) != combine (a, combine(b, c))

```
process(int startIndex, int endIndex
  ...

  var middle = (startIndex + endIndex) >>> 1;
  fork process(startIndex, middle);
  var result2 = process(middle, endIndex);
  var result1 = join
  return combine result1 result2
}
```

La moyenne (a, b) → (a + b) / 2 est pas associative !

Fork / Join en Java

java.util.concurrent.ForkJoinPool

Le pool de threads (l'ExecutorService) qui fait du *work stealing* s'appel ForkJoinPool

Pour créer un ForkJoinPool

- **new** ForkJoinPool(5) // avec 5 threads

Il existe déjà un ForkJoinPool par défaut,

- ForkJoinPool.commonPool()

Tâches du ForkJoinPool

Le ForkJoinPool exécute des ForkJoinTask

- RecursiveAction pour les tâches sans valeur
- RecursiveTask<V> pour les tâches avec une valeur

Exécuter la tâche initiale sur le ForkJoinPool

- execute(RecursiveAction)
  - L'exécution est asynchrone
- invoke(RecursiveAction|RecursiveTask)
  - Bloque la thread courante et attend l'exécution de la tâche
- submit(RecursiveAction|RecursiveTask) → Future
  - Renvoie un Future (Future.get() permet d'avoir la valeur)

Rappel sur la suite de Fibonacci

Calcul de la suite de Fibonacci en récursif (stupide mais simple)

```
int fibonacci(int n)
  if (n <= 1) {
    return n;
  }
  return fibonacci(n – 1) + fibonacci(n – 2);
}
```

Idée: on peut utiliser le pattern Fork / Join pour faire le calcul en parallèle

Exemple de RecursiveTask

Calcul de Fibonacci en récursif (stupide mais simple)

```
private static class Fibonacci extends RecursiveTask<Integer> {
  private final int n;
  private Fibonacci(int n) { this.n = n; }
  @Override
  protected Integer compute() {
    if (n <= 1) {
      return 1;
    }
    var fib1 = new Fibonacci(n - 1);
    var fib2 = new Fibonacci(n - 2);
    f1.fork();                          // soumet au ForkJoinPool
    var result2 = fib2.compute();       // calcul dans la thread courante
    var result1 = f1.join();            // attend le résultat de la tâche
    return result1 + result2;           // le + est bien associatif
  }
}
```

Exemple de RecursiveTask (2)

Calcul de Fibonacci en récursif (stupide mais simple)

```
private static class Fibonacci extends RecursiveTask<Integer> {
  private final int n;

  private Fibonacci(int n) { this.n = n; }

  @Override
  protected Integer compute() {
     …
  }
}
…
var pool = ForkJoinPool.commonPool();
var fibo = new Fibonacci(15);
var result = pool.invoke(fibo);      // on attend la fin du calcul
```

Exemple de RecursiveAction

Multiplier par 2 les valeurs du tableaux

```
private static class MultiplyBy2 extends RecursiveAction {
  private final int[] array;
  private final int start;
  private final int end;

  private MultiplyBy2(...) { this.array = array; this.start = start; this.end = end; }

  @Override
  protected void compute() {
    if (end – start < 1024) {
      for(var i = start; I < end; i++) { array[i] *= 2; }
      return;
    }
    var middle = (start + end) >>> 1;
    var mul1 = new MultiplyBy2(array, start, middle);
    var mul2 = new MultiplyBy2(array, middle, end);
    mul2.fork();
    mul1.compute();
    mul2.join();
  }
}
```

Exemple de RecursiveAction (2)

Multiplier par 2 les valeurs du tableaux

```
private static class MultiplyBy2 extends RecursiveAction {
  private final int[] array;
  private final int start;
  private final int end;
  private MultiplyBy2(...) { this.array = array; this.start = start; this.end = end;
}
  @Override
  protected void compute() {
    ...
  }
}
…
var pool = ForkJoinPool.commonPool();
var fibo = new MultiplyBy2(array, 0, array.length);
var future = pool.submit(fibo);      // on récupère un future
…
System.out.println(future.get());     // on récupère la valeur
```

Relation avec les Streams

Lorsque l’on utilise un Stream parallèle

stream.parallel(). …

le calcul est fait dans le ForkJoinPool par défaut

ForkJoinPool.commonPool()

L’implantation utilise la méthode Spliterator.trySplit() du Spliterator du Stream pour faire la décomposition récursive