Algoritmos paralelos II
Fork / Join

PROGRAMACIÓN CONCURRENTE EN JAVA - TEMA 5

Universidad
Rey Juan Carlos

Divide y vencerás

- Técnica muy **versátil** que se adapta muy bien al **diseño paralelo** de algoritmos.
- Permite crear **tareas paralelas** de una manera natural.
- Se basa en **dividir** un problema complejo para convertirlo en varios problemas más **sencillos** de resolver.
- Se suele resolver utilizando **recursividad**.

Divide y vencerás

- Caso base
  - Si el problema es lo suficientemente pequeño, se resuelve.
- Caso recursivo
  - Dividir el problema en sub-problemas más sencillos
- Finalmente, se combinan las soluciones obtenidas para generar la solución del problema original.

Ejercicio

- Implementa un algoritmo de divide y vencerás secuencial que encuentre el máximo valor almacenado en un array de enteros.
  - Se considera que hemos llegado al caso base cuando estamos procesando un array de dos elementos.
  - Se recomienda utilizar recursividad para la implementación.

Fork / Join

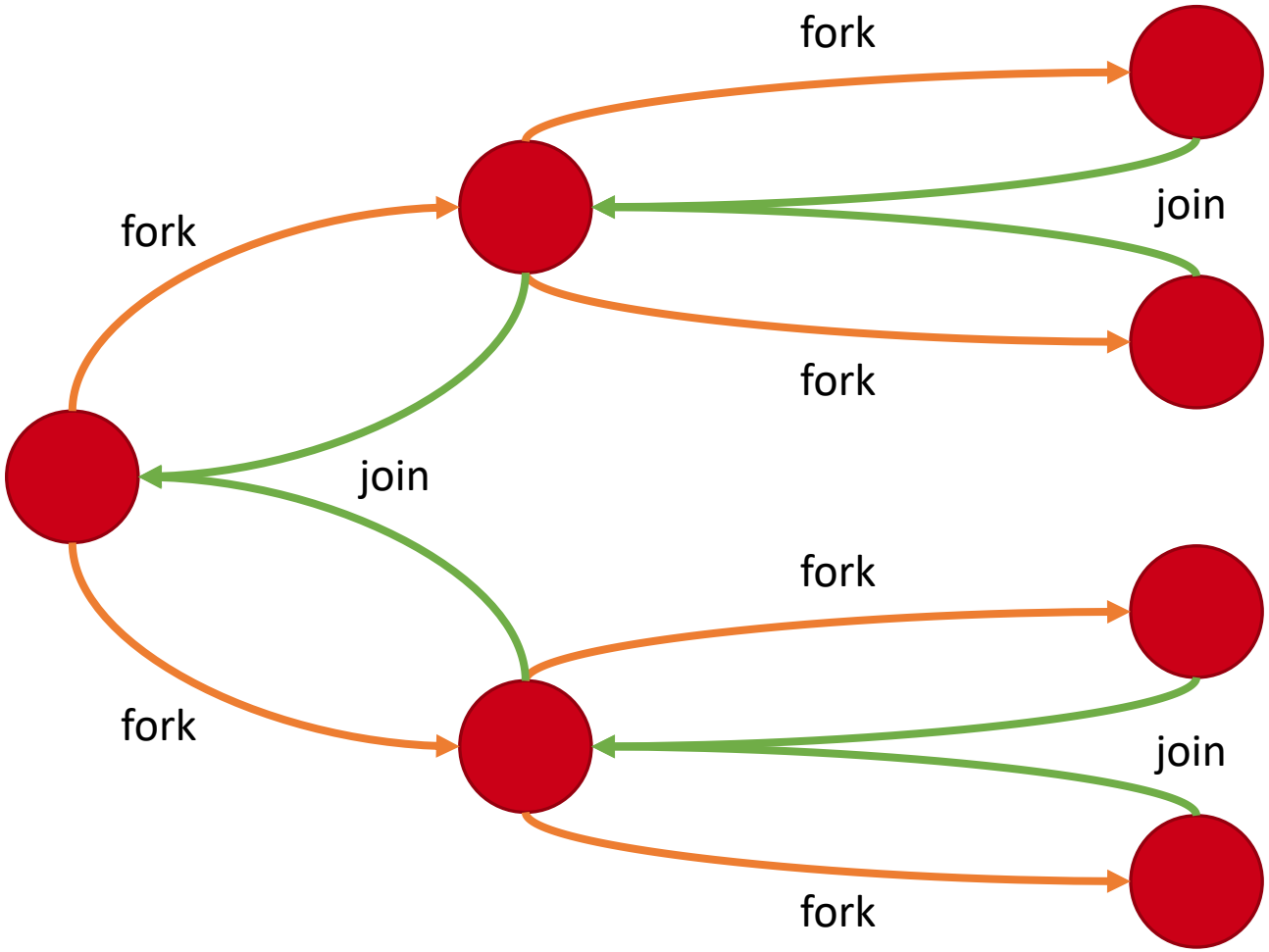
- En Java, esta estrategia es una implementación de la interfaz *ExecutorService*.
- Trata de aprovechar todo el potencial del hardware disponible para mejorar el rendimiento de la aplicación.
- Como cualquier implementación de *ExecutorService*, se basa en la distribución de tareas a trabajadores de un pool de threads.

Fork / Join

- Fork / Join es una de las técnicas de diseño más simples y efectivas para obtener un buen rendimiento en aplicaciones paralelas.
- Básicamente son versiones paralelas de algoritmos clásicos de divide y vencerás.
- Los algoritmos Fork / Join suelen seguir siempre la misma base:
  - Si el problema es pequeño
    - Resuelve directamente el problema.
  - Si el problema es grande:
    - Divide el problema y crea una nueva tarea (fork) para cada subproblema.
    - Recupera los resultados (join) de cada una de las subtareas y combínalos para generar el resultado final.

Fork / Join

Algoritmos Paralelos



Fork / Join

- **Fork**
  - Dividir una tarea en sub-tareas más pequeñas que pueden ser ejecutadas de manera concurrente.
  - Al dividir una tarea, puede ser ejecutada en paralelo por diferentes CPU o diferentes threads.
  - Una tarea solo se divide si es lo suficientemente compleja, ya que existe un coste adicional por dividir una tarea.

- **Join**
  - Cuando una tarea se ha dividido, debe esperar a que terminen las sub-tareas para poder continuar.
  - Cuando las sub-tareas terminan, la tarea original une (*join*) todos los resultados en un único valor.
  - A veces las sub-tareas no devuelven ningún valor, en cuyo caso la tarea original tan solo espera a que finalicen, sin hacer el *join*.

Fork / Join

- El diseño Fork / Join puede ser utilizado en cualquier lenguaje o framework que soporte la ejecución de subtareas en paralelo.
  - Además, debe ofrecer algún mecanismo para esperar a que las tareas generadas terminen.
- Sin embargo, los threads de Java no son eficientes en el desarrollo de aplicaciones basadas en Fork / Join.
- Las tareas fork / join necesitan una sincronización simple, de manera que su gestión es mucho más sencilla que la de los threads tradicionales.
  - Por ejemplo, las tareas fork / join nunca deben ser bloqueadas excepto para esperar a las subtareas. Por lo tanto, la sobrecarga debida al seguimiento de hilos genéricos bloqueados no se aprovecha.
- Dada una granularidad de tarea razonable, el coste de construir y gestionar un thread puede ser mayor que el tiempo de cómputo de la propia tarea.

Fork / Join

- Por lo tanto, los frameworks tradicionales de threads son demasiado complejos para aprovechar el diseño fork / join.
- La solución utilizada en Java es establecer un pool de trabajadores que llevarán a cabo las tareas.
  - Cada trabajador es un thread genérico tradicional.
  - Lo normal es tener un trabajador por cada procesador disponible.
- Las tareas fork / join son instancias de clases ejecutables muy sencillas (Runnable), no de threads.
- Se utiliza un método de gestión de tareas basado en colas especialmente diseñado para la metodología.
- Una clase de gestión se encarga de inicializar a los trabajadores y la ejecución de nuevas tareas cuando se necesite.

Fork / Join

Algoritmos Paralelos

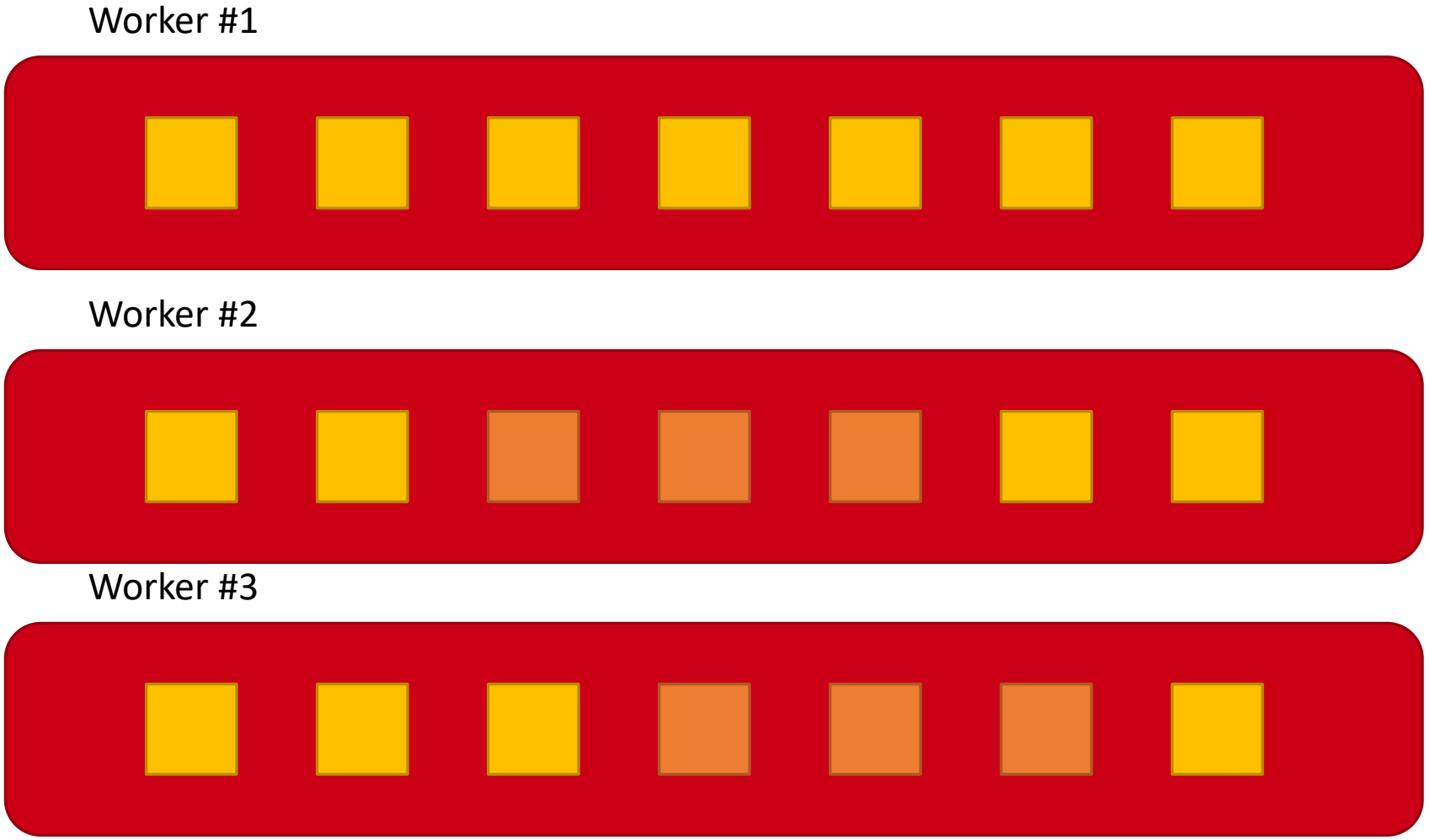
- Cada trabajador mantiene tareas ejecutables en su propia cola de ejecución.
  - Cada cola es una cola doblemente enlazada.
- Cada subtarea generada en una tarea que ejecuta un trabajador se añade (push) a la cola del propio trabajador.
- Los trabajadores procesan cada cola siguiendo un orden LIFO (pop).
- Cuando un trabajador no tiene más tareas que ejecutar, intenta robar una tarea de otro trabajador al azar, siguiendo un orden FIFO.
- Cuando un trabajador se encuentra con una operación *join*, procesa las otras tareas hasta que la tarea que ha hecho *join* pueda ser terminada (*isDone*).
- Cuando un trabajador no tiene más tareas y no puede robar de otros trabajadores, se duerme y lo intenta más tarde, hasta que todos los threads se encuentren en el mismo estado, en cuyo caso deben esperar una nueva tarea.

Fork / Join

- La utilización del orden LIFO para cada trabajador, y FIFO para robar tareas es óptima para un amplio rango de diseños fork / join recursivos.
- Tiene dos ventajas principales:
  1. Reduce los conflictos que se producen al tener a los trabajadores robando tareas en el otro extremo de la cola.
  2. Explota la propiedad recursiva de los algoritmos de divide y vencerás generando las tareas grandes al principio.
     - De esta manera, una tarea robada llevará tiempo en la cola y, por lo tanto, será una tarea grande, generando bastante trabajo para el thread que la ha robado.

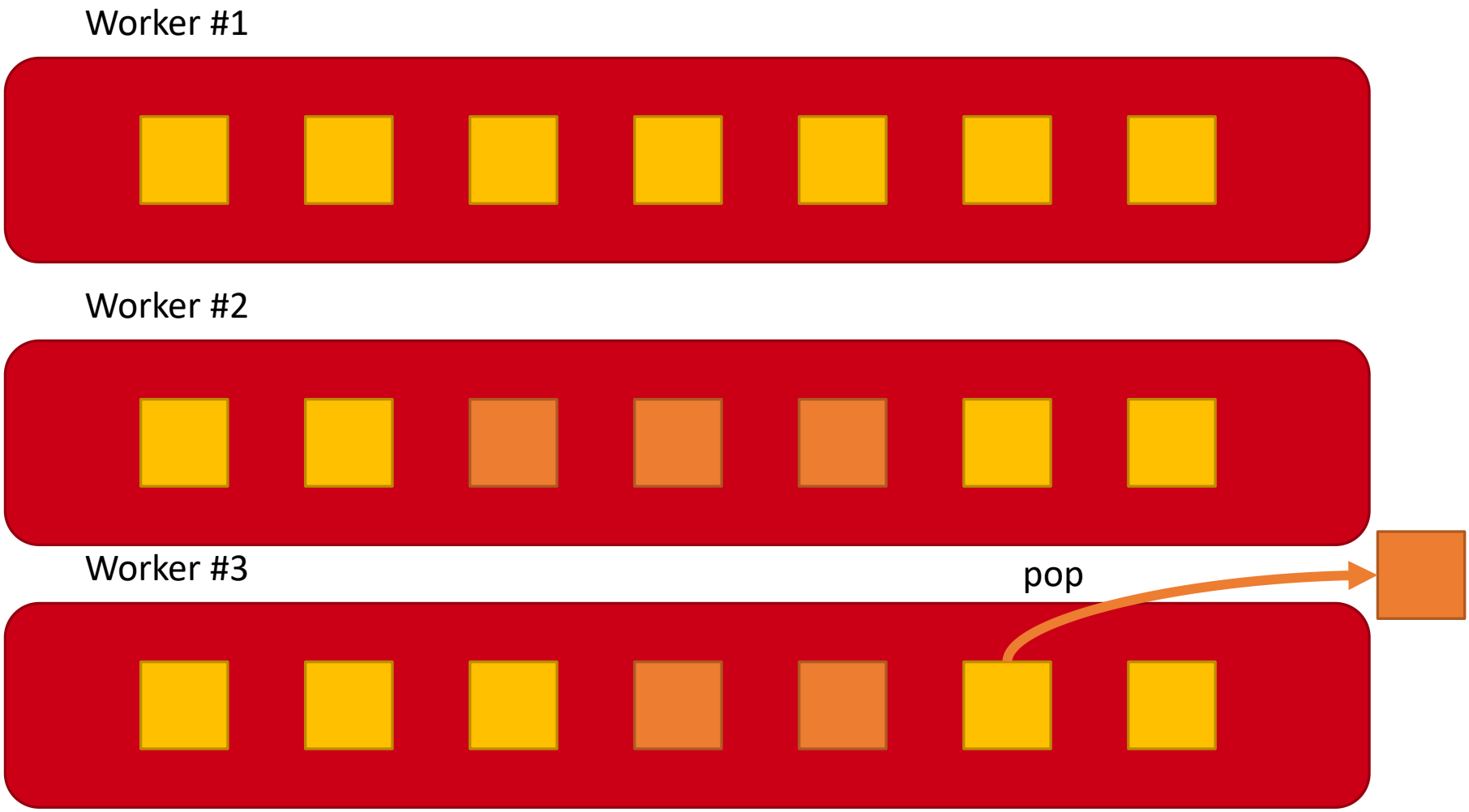
Fork / Join

Algoritmos Paralelos



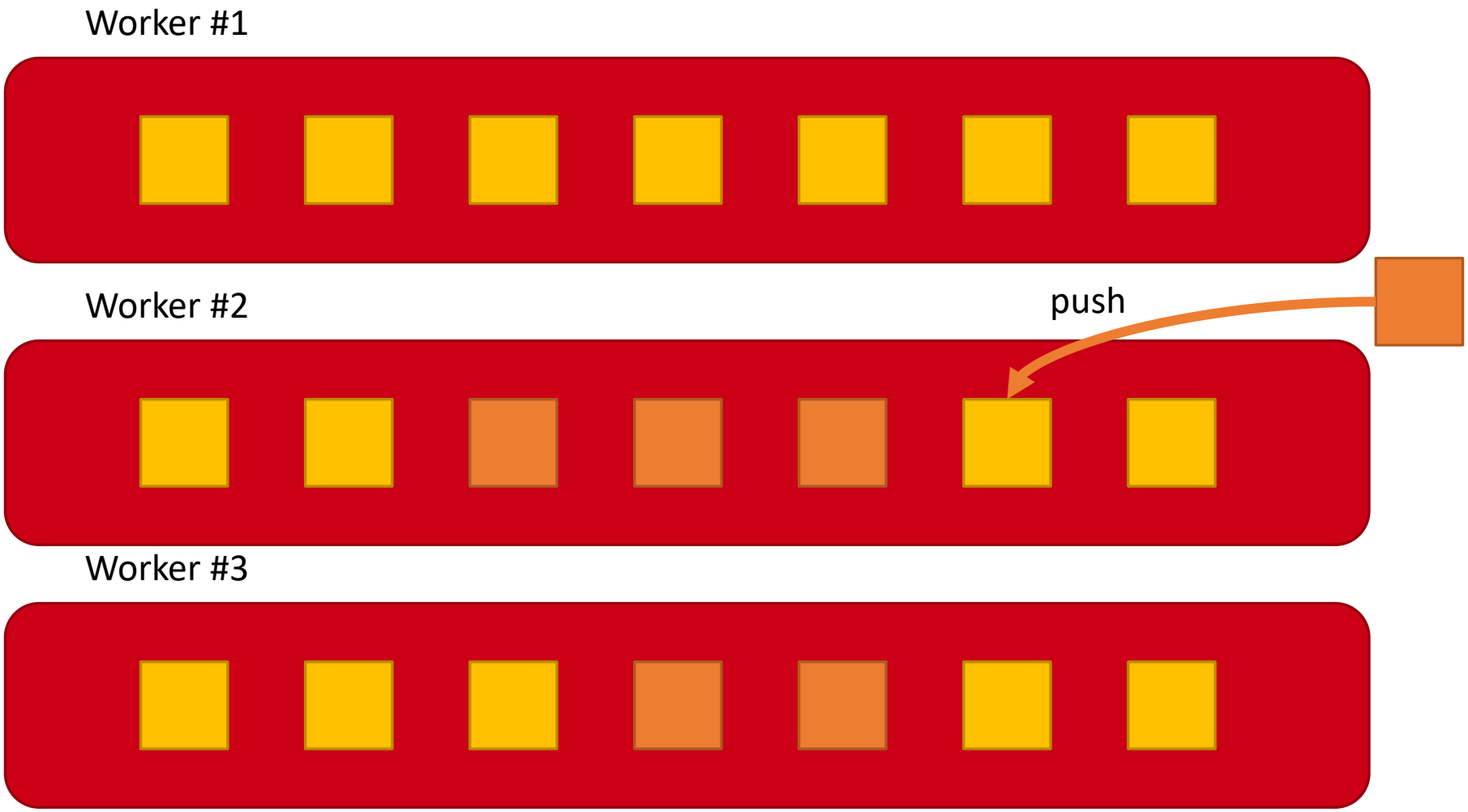
Fork / Join

Algoritmos Paralelos



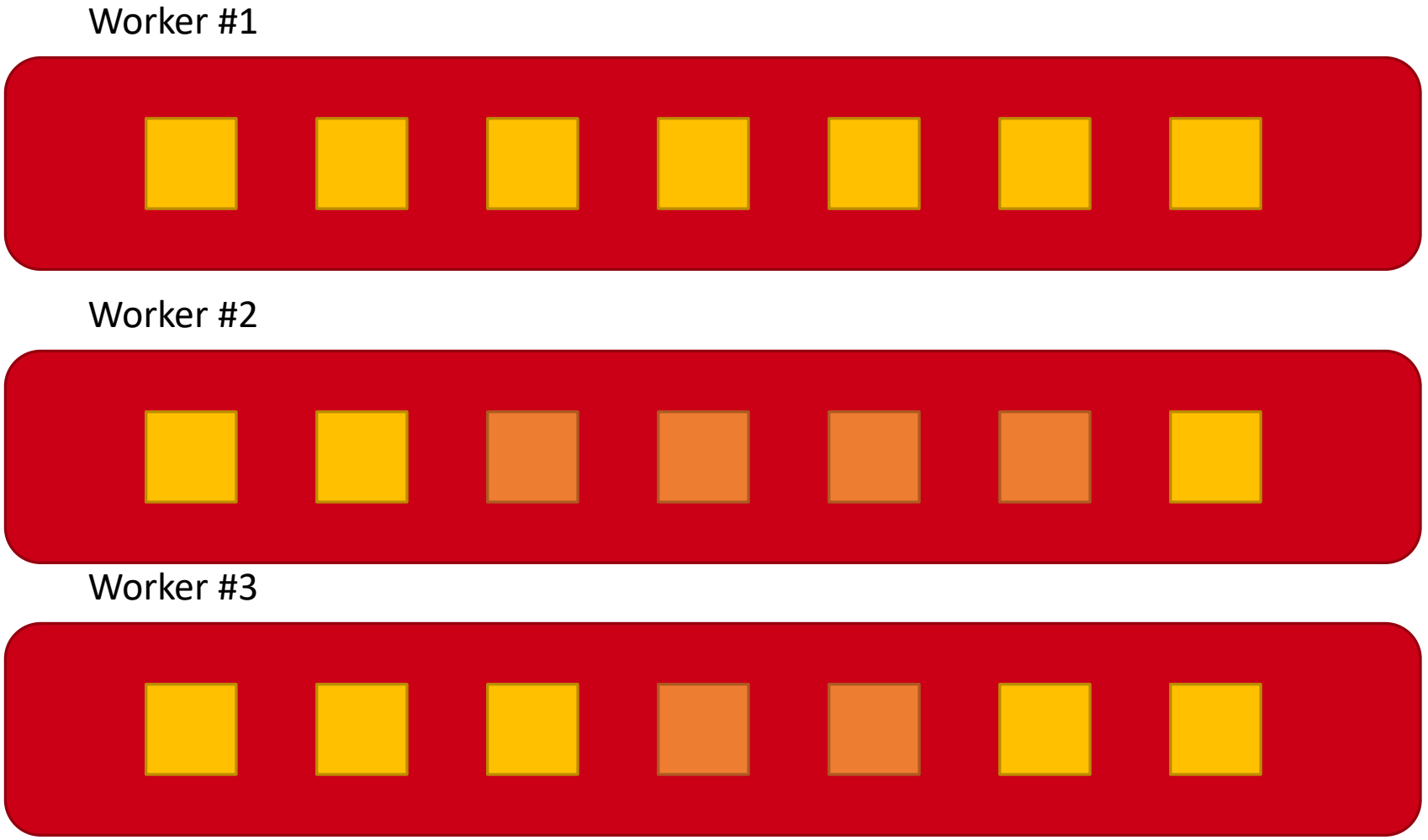
Fork / Join

Algoritmos Paralelos



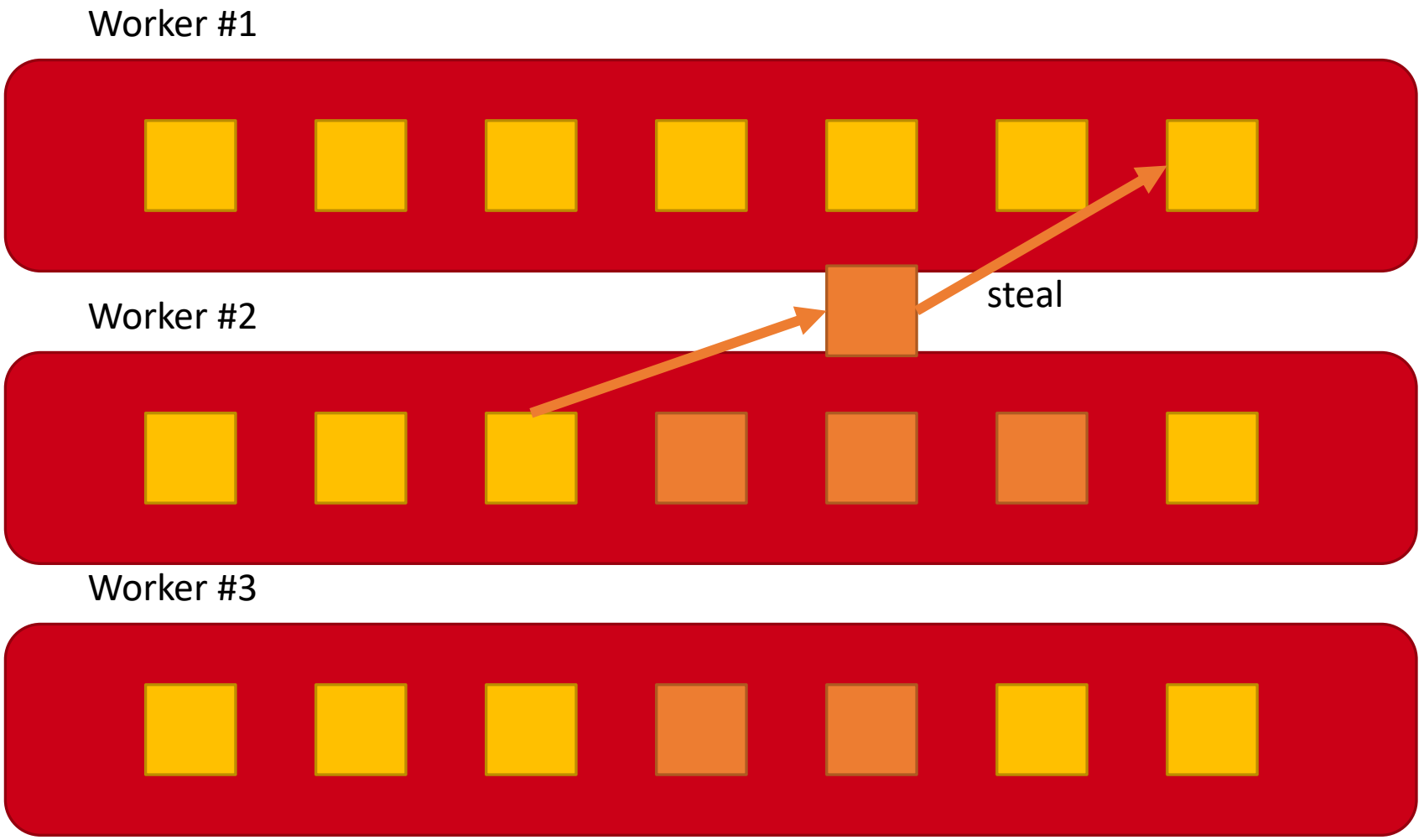
Fork / Join

Algoritmos Paralelos



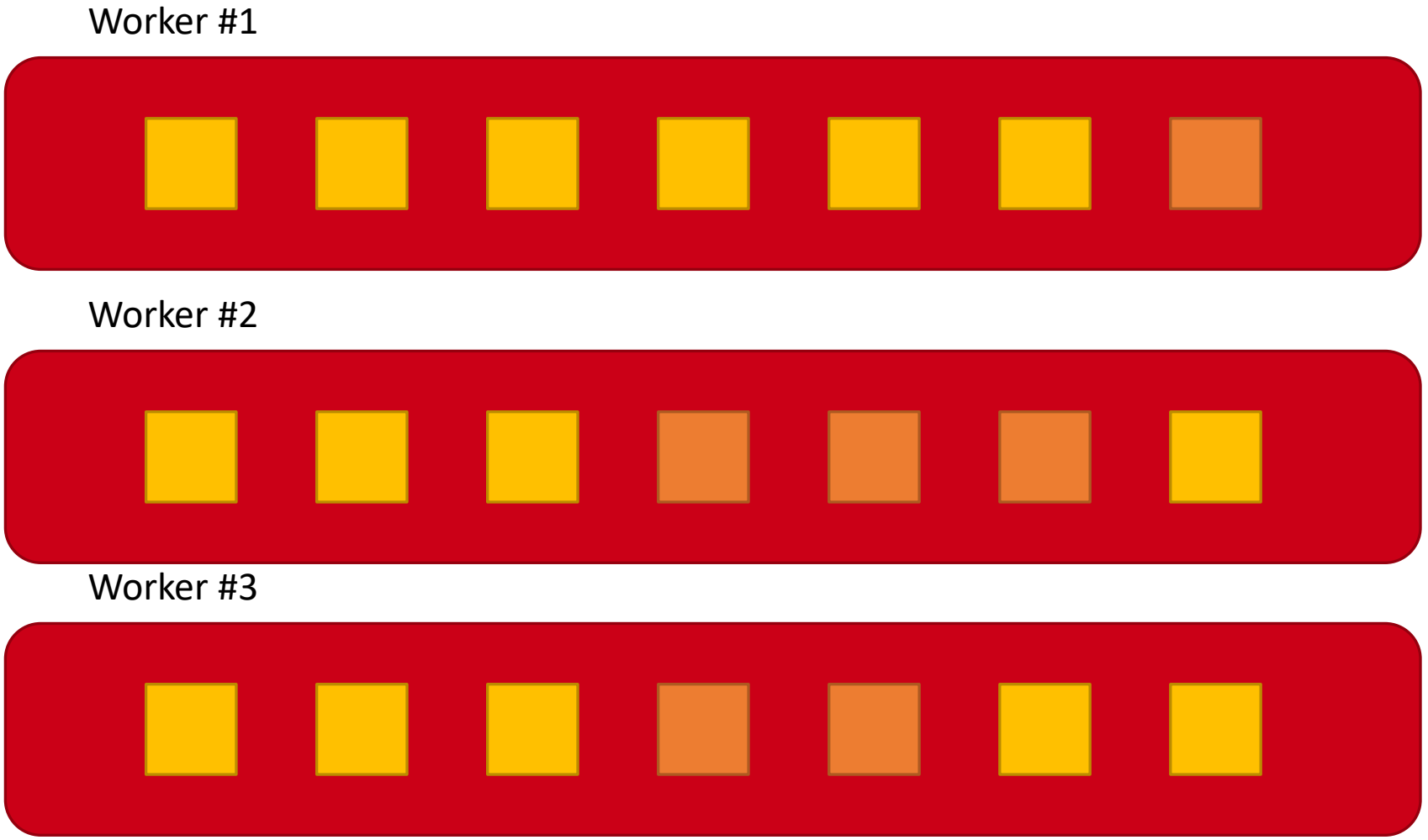
Fork / Join

Algoritmos Paralelos



Fork / Join

Algoritmos Paralelos



Fork / Join

- *ForkJoinPool* es un pool de threads diseñado para trabar de manera eficiente con división de tareas de tipo *fork / join*.
- Funciona de manera similar a los *ExecutorService* vistos hasta ahora, solo que mejora la gestión de tareas de tipo *fork / join*.
- Un *ForkJoinPool* admite dos tipos de tareas, que heredan de *ForkJoinTask*:
  - Tareas que no devuelven ningún resultado
    - RecursiveAction
  - Tareas que devuelven un resultado
    - RecursiveTask

Fork / Join

- Creación de un pool de *threads* para tareas *fork / join*
  - ForkJoinPool pool = new ForkJoinPool()
    - El constructor por defecto creará un pool con tantos threads como procesadores haya disponibles.
  - ForkJoinPool pool = new ForkJoinPool(int paralellism)
    - También podemos indicar cuántos threads queremos que tenga el pool.
- Java nos ofrece atajos para crear un pool de threads común.
  - ForkJoinPool.commonPool()
    - Método estático que devuelve un pool con el paralelismo por defecto de la máquina.

Recursive Action

- Es una tarea que no devuelve ningún valor.
  - Realiza tareas como consultar una base de datos, escribir en disco, etc.
- Puede ser que necesitemos dividirla en sub-tareas que sean ejecutadas de manera paralela.
- Para implementarla, nuestra clase debe heredar de *RecursiveAction*
  - Esto nos obliga a implementar el método *compute*

Recursive Action


```
public class ActionExample extends RecursiveAction {
	@Override
	protected void compute() {
		// Fork / Join code
	}
}
```

Recursive Task

- Es una tarea que devuelve un valor al finalizar.
- Puede ser que necesitemos dividirla en sub-tareas que sean ejecutadas de manera paralela.
  - En este caso, necesitamos combinar los resultados devueltos por cada una de las sub-tareas.
- Para implementarla, nuestra clase debe heredar de *RecursiveTask*
  - Esto nos obliga a implementar el método *compute*

Recursive Action


```
public class TaskExample extends RecursiveTask<Integer> {
	@Override
	protected Integer compute() {
		// Fork / Join code
		return null;
	}
}
```

Fork / Join: implementación

- Una tarea fork / join siempre se compone de dos partes básicas, al igual que un método recursivo.
  - **Caso base**: contiene el código que se ejecutará cuando el tamaño del problema sea lo suficientemente pequeño.
  - **Caso general**: contiene el código que se encarga de dividir la tarea en subtareas más sencillas y espera a su ejecución.

Ejemplo: máximo de un array

- Creamos la estructura de nuestra tarea:

```
public class FJTaskMax extends RecursiveTask<Integer> {
    private static final int TH_SEQ = 10;

    @Override
    protected Integer compute() {
        if (ARRAY_SIZE < TH_SEQ) {
            // Caso base
        } else {
            // Caso general
        }
    }
}
```

Ejemplo: máximo de un array

- Constructor y atributos de la tarea:

```
private int[] array;
private int low; // Límite inferior del array
private int high; // Límite superior del array

public FJTaskMax(int[] array, int low, int high) {
    this.array = array;
    this.low = low;
    this.high = high;
}
```

Ejemplo: máximo de un array

- Caso base:
  - Si el tamaño del array que debo comprobar es menor que el umbral que he fijado.

```
if (high - low < TH_SEQ) {
    int max = Integer.MIN_VALUE;
    for (int i=low;i<high;i++) {
        max = Math.max(max, array[i]);
    }
    return max;
}
```

Ejemplo: máximo de un array

- Caso general:
  - La tarea es aún compleja para ser resuelta de manera secuencial, por lo que la dividimos en sub-tareas.
  - Para ello, creamos *N* nuevos objetos de nuestra tarea, siendo *N* el número de sub-tareas que queremos crear.
  - Para ejecutar cada nueva sub-tarea en paralelo, invocamos el método *fork()* en cada una de ellas.
  - Para esperar el resultado, invocamos el método *join()*.
    - Esta llamada es bloqueante, por lo que debemos hacer todos los *fork* antes de empezar con los *join*

Ejemplo: máximo de un array

- Caso general:
  - Dividimos en dos tareas

```
int mid = low + (high - low) / 2;
FJTaskMax left = new FJTaskMax(array, low, mid);
FJTaskMax right = new FJTaskMax(array, mid, high);
```

  - Las ejecutamos en paralelo, esperamos que terminen y devolvemos el resultado

```
left.fork();
right.fork();
int leftRes = left.join();
int rightRes = right.join();
return Math.max(leftRes, rightRes);
```

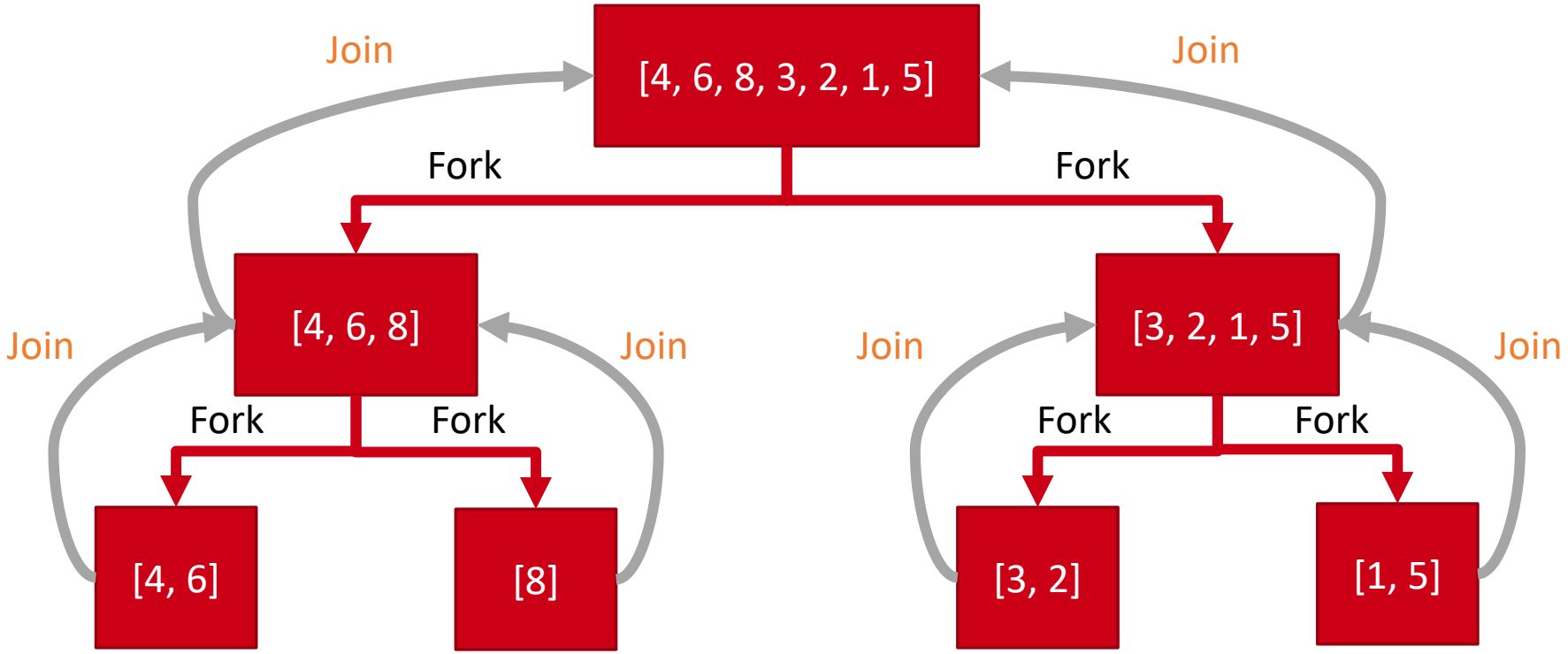
Ejemplo: máximo de un array

- Programa principal:

```
private static final int SIZE = 100000000;

public static void main(String[] args) {
    ForkJoinPool pool = new ForkJoinPool();
    int[] array = new int[SIZE];
    for (int i=0;i<array.length;i++) {
        array[i] = new Random().nextInt(SIZE*2);
    }
    int result = pool.invoke(new FJTaskMax(array, 0, SIZE));
    System.out.println("MAX: "+result);
    pool.shutdown();
}
```

Ejemplo: máximo de un array



Mejoras en el rendimiento

- Aunque el ejemplo anterior muestra la implementación natural de *fork / join* en Java, tiene un problema.
- Cuando dividimos la tarea en sub-tareas, la original se queda sin hacer nada, solamente esperando el *join* de las sub-tareas.
- Podríamos aprovechar esa tarea original para que también haga parte del trabajo.
- De esta manera estaríamos creando una tarea menos en cada una de las llamadas que se hicieran (que no fueran el caso base).

Mejoras en el rendimiento

- La modificación consiste en:
  1. Hacer *fork* de una sub-tarea.
  2. Realizar la otra sub-tarea sin hacer un *fork*.
  3. Esperar a que termine la sub-tarea lanzada.

```
int mid = low + (high - low) / 2;
FJTaskMaxOpt left = new FJTaskMaxOpt(array, low, mid);
FJTaskMaxOpt right = new FJTaskMaxOpt(array, mid, high);
left.fork();
int rightRes = right.compute();
int leftRes = left.join();
return Math.max(leftRes, rightRes);
```

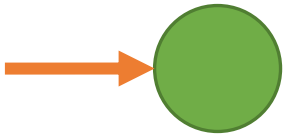
Aspectos a considerar

- La mejora propuesta siempre incrementa el rendimiento de la aplicación, ya que minimiza la creación de tareas.
- Es importante recordar que join se bloquea hasta que la respuesta esté lista.
- El orden de ejecución es muy importante, pudiendo bloquear la ejecución o convirtiéndola en secuencial.

Aspectos a considerar

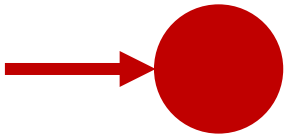
- El orden de ejecución es muy importante

```
left.fork();
int rightRes = right.compute();
int leftRes = left.join();
return Math.max(leftRes, rightRes);
```



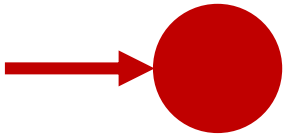
Correcto

```
left.fork();
int leftRes = left.join();
int rightRes = right.compute();
return Math.max(leftRes, rightRes);
```



No paralelo

```
int rightRes = right.compute();
left.fork();
int leftRes = left.join();
return Math.max(leftRes, rightRes);
```



No paralelo

Aspectos a considerar

- En ningún caso debemos utilizar el método *invoke* del pool de threads dentro de una *RecursiveTask* o *RecursiveAction*.
- Si no capturamos las excepciones dentro del método *compute*, no va a ser fácil depurar desde el exterior una excepción que se haya producido aquí.
  - La excepción se habrá producido en otro thread, de manera que el depurador no va a poder controlar dónde se originó si no la hemos capturado.

Crítica a la implementación

- Analizando la implementación del *fork / join* en Java, se han encontrado una serie de errores que hacen que no suela ser recomendable su uso.
- Las principales críticas[1] que ha recibido este *framework* son:
  - Complejidad excesiva
  - Fallos de diseño
  - Muy académico
  - Desaprovechamiento del paralelismo
  - Ineficiente
  - Lento y poco escalable
  - No es una API

1 http://coopsoft.com/ar/CalamityArticle.html

Algoritmos paralelos II
Fork / Join

PROGRAMACIÓN CONCURRENTE EN JAVA - TEMA 5

Universidad
Rey Juan Carlos