code URJC

Programación Concurrente

Tema 7

Diseño de algoritmos paralelos

Micael Gallego
micael.gallego@urjc.es
@micael_gallego

Universidad Rey Juan Carlos

CC BY SA

Diseño de Algoritmos Paralelos

code URJC

- **Introducción**
  - **Paralelización de algoritmos**
  - Tiempo de ejecución de un algoritmo
  - Metodología de diseño de algoritmos paralelos
  - Modelos de algoritmos paralelos
  - Criterios de evaluación de un algoritmo paralelo
  - Reglas para diseñar un algoritmo paralelo
- Algoritmos sobre arrays
- Esquema ForkJoin

Paralelización de algoritmos

code URJC

- Existen muchos tipos de programas informáticos
- Su **estructura** puede llegar a ser muy **diferente** porque tienen objetivos también diferentes
- Se puede hacer una gran **distinción entre aplicaciones:**
  - Aplicaciones que prestan un **servicio interactivo**. Atienden a uno o varios usuarios/clientes
  - Aplicaciones que ejecutan un **algoritmo** para obtener un resultado partiendo de unos datos de entrada

Paralelización de algoritmos

- Las **aplicaciones de red** y las **bases de datos** son aplicaciones interactivas
- El aprovechamiento de recursos se obtiene atendiendo a varios usuarios simultáneamente
- Las aplicaciones con **interfaz de usuario** atienden a un único usuario, pero procesan peticiones en **primer plano** (una cada vez) y peticiones en **segundo plano** (varias)

Paralelización de algoritmos

- Las aplicaciones que ejecutan un **algoritmo**, habitualmente sólo tienen que realizar una tarea: **obtener el resultado** del algoritmo
- En esta situación para **aprovechar** los recursos es necesario **dividir el algoritmo en tareas** que puedan ejecutarse en paralelo
- Es habitual que los algoritmos **no** hagan uso intensivo de **entrada/salida**, por lo que el tiempo de ejecución total sólo se reducirá si se ejecuta en un sistema **multiprocesador (en paralelo)**

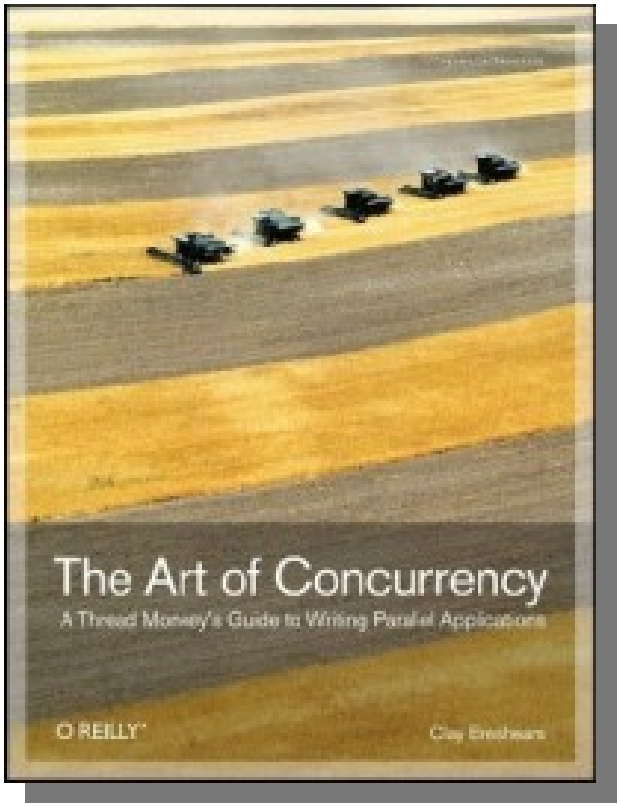
Paralelización de algoritmos

code URJC

- Algunos ejemplos de aplicaciones que principalmente ejecutan un algoritmo
  - Compiladores
  - Filtros en imágenes
  - Generación de imágenes 3D
  - Montaje de vídeos
  - Simulación meteorológica
  - Búsqueda de caminos más cortos en un grafo
  - Cálculo de rutas de vehículos

Paralelización de algoritmos

- En este tema nos basaremos en el libro

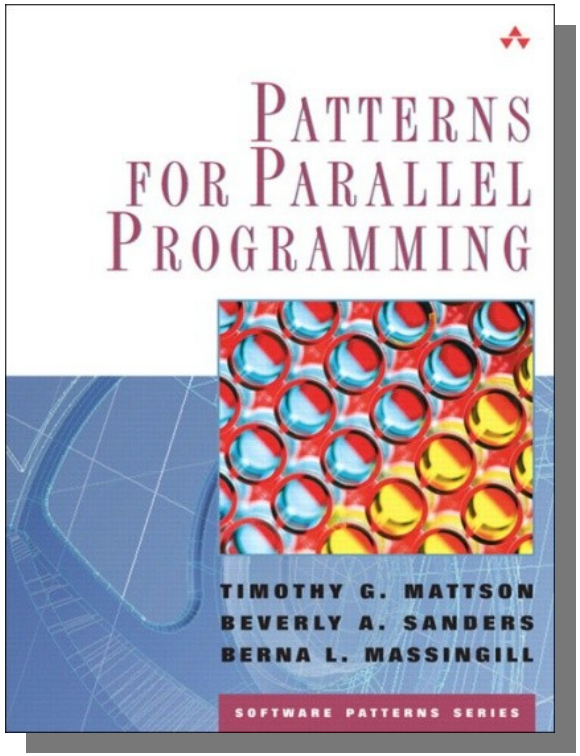


The Art of Concurrency - A Thread Monkey's Guide to Writing Parallel Applications
Clay Breshears
O'REILLY 2009

Paralelización de algoritmos

code URJC

- **Otros libros interesantes…**



Patterns for Parallel Programming, by Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill

http://software.intel.com/en-us/articles/technical-books-for-multi-core-software-developers/

Paralelización de algoritmos

- En este tema estudiaremos las técnicas para **paralelizar un algoritmo**, es decir:
  - Identificar la **concurrencia del problema** que resuelve el algoritmo secuencial
  - Crear un **nuevo algoritmo** estructurado de forma que la concurrencia se pueda explotar
  - **Implementar** el algoritmo en una plataforma de desarrollo que lo pueda ejecutar en paralelo

Paralelización de algoritmos

code URJC

- Los **algoritmos** se paralelizan (y se ejecutan en paralelo) por varios **motivos**:
  - **Reducir el tiempo** de ejecución total
  - Ejecutar el algoritmo **con más calidad** (predicción, renderizado, optimización…)
  - Si la plataforma paralela es un clúster, para resolver **problemas más grandes** de los que puede gestionar una única máquina.

Diseño de Algoritmos Paralelos

code URJC

- **Introducción**
  - Paralelización de algoritmos
  - **Tiempo de ejecución de un algoritmo**
  - Metodología de diseño de algoritmos paralelos
  - Modelos de algoritmos paralelos
  - Criterios de evaluación de un algoritmo paralelo
  - Reglas para diseñar un algoritmo paralelo
- Algoritmos sobre arrays
- Esquema ForkJoin

Tiempo de ejecución de un algoritmo

- Uno de los objetivos de la paralelización de un algoritmo suele ser la reducción del **tiempo de ejecución**
- Se puede **estudiar de forma analítica** la reducción del tiempo en función del número de **procesadores** (o **cores**) disponibles.
- Eso permite obtener algunos datos sobre la **eficiencia** de la paralelización

Tiempo de ejecución de un algoritmo

- Tiempo de ejecución de un algoritmo

$$T_{total}(1) = T_{preparacion} + T_{computacion} + T_{finalizacion}$$

- Si suponemos que la preparación y la finalización no pueden paralelizarse. El tiempo de ejecución con P procesadores es:

$$T_{total}(P) = T_{preparacion} + \frac{T_{computacion}(1)}{P} + T_{finalizacion}$$

Tiempo de ejecución de un algoritmo

- El ***speedup* relativo** es el incremento en la velocidad al ejecutarse en paralelo con P procesadores

$$S(P) = \frac{Ttotal(1)}{Ttotal(P)}$$

- El objetivo es que el *speedup* sea igual a P, lo que se llama *speedup* lineal perfecto.
- Este valor es muy difícil de conseguir por el tiempo de preparación y finalización y por la sobrecarga de la coordinación entre tareas

Tiempo de ejecución de un algoritmo

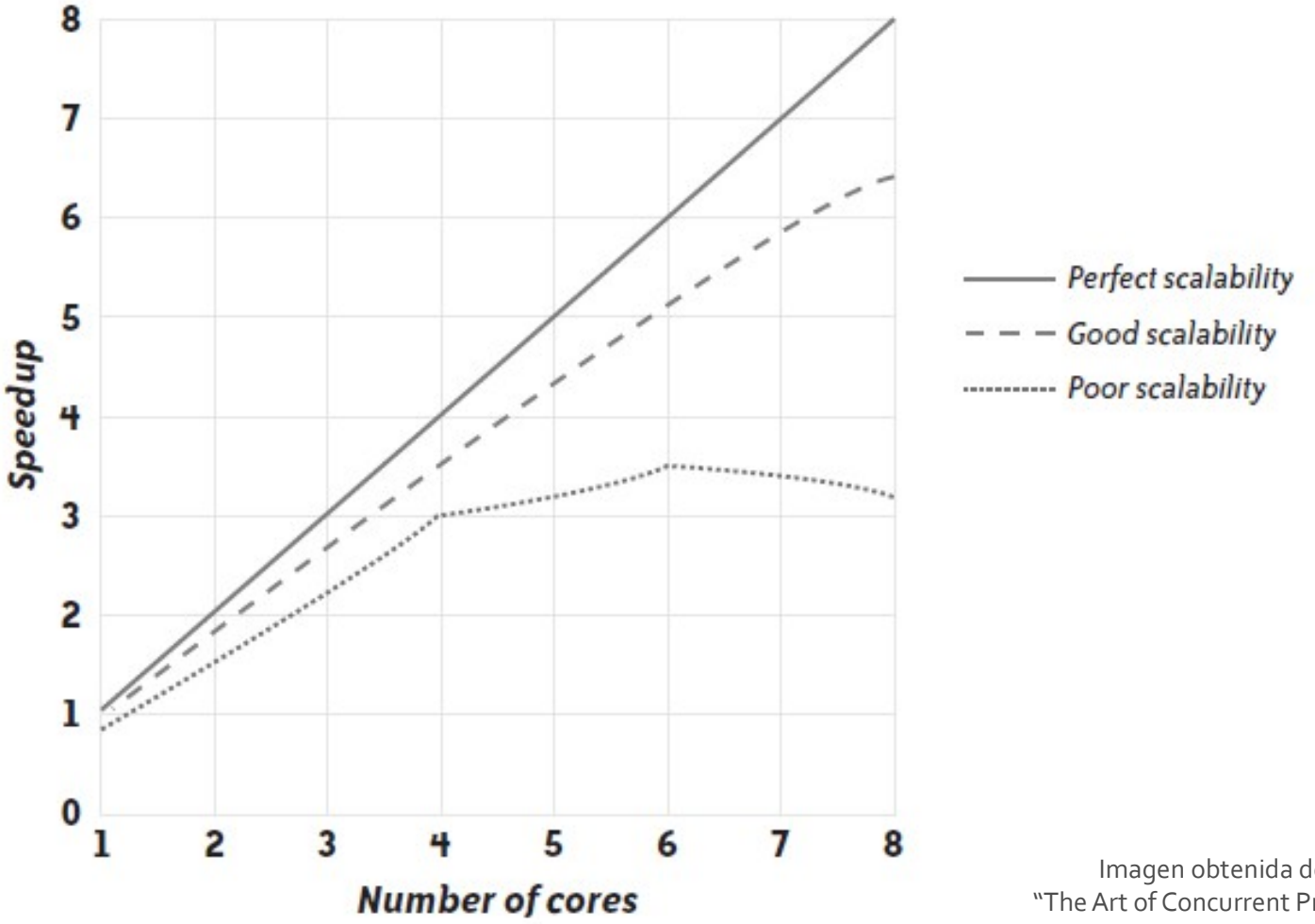


Imagen obtenida de la pág 67 de
"The Art of Concurrent Programming"
de Breshears, O'Reilly 2009

Tiempo de ejecución de un algoritmo

- La **eficiencia en la paralelización** se calcula como el *speedup* dividido por el número de procesadores

$$E(P) = \frac{S(P)}{P}$$

- Con una eficiencia del 100% tendríamos *speedup* lineal perfecto.

Tiempo de ejecución de un algoritmo

- El *speedup* está limitado por la parte secuencial del algoritmo (esa que no se puede paralelizar)

$$Pserie = \frac{Tpreparacion + Tfinalizacion}{Ttotal(1)}$$

- El *speedup* se puede calcular en función de la parte secuencial del programa y en función de P (**Ley de Amdahl's**)

$$S(P) = \frac{1}{Pserie + \frac{1 - Pserie}{P}}$$

Tiempo de ejecución de un algoritmo

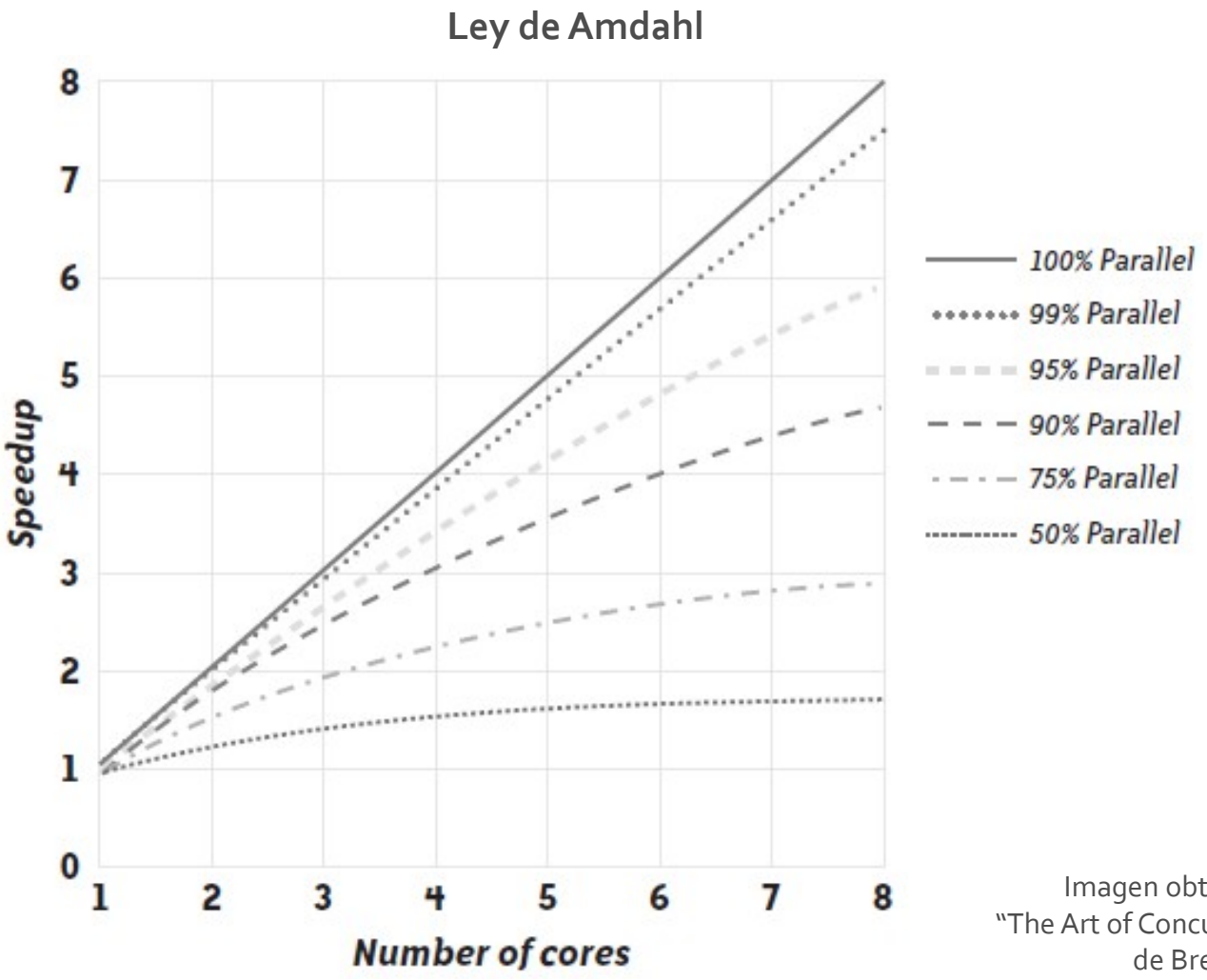
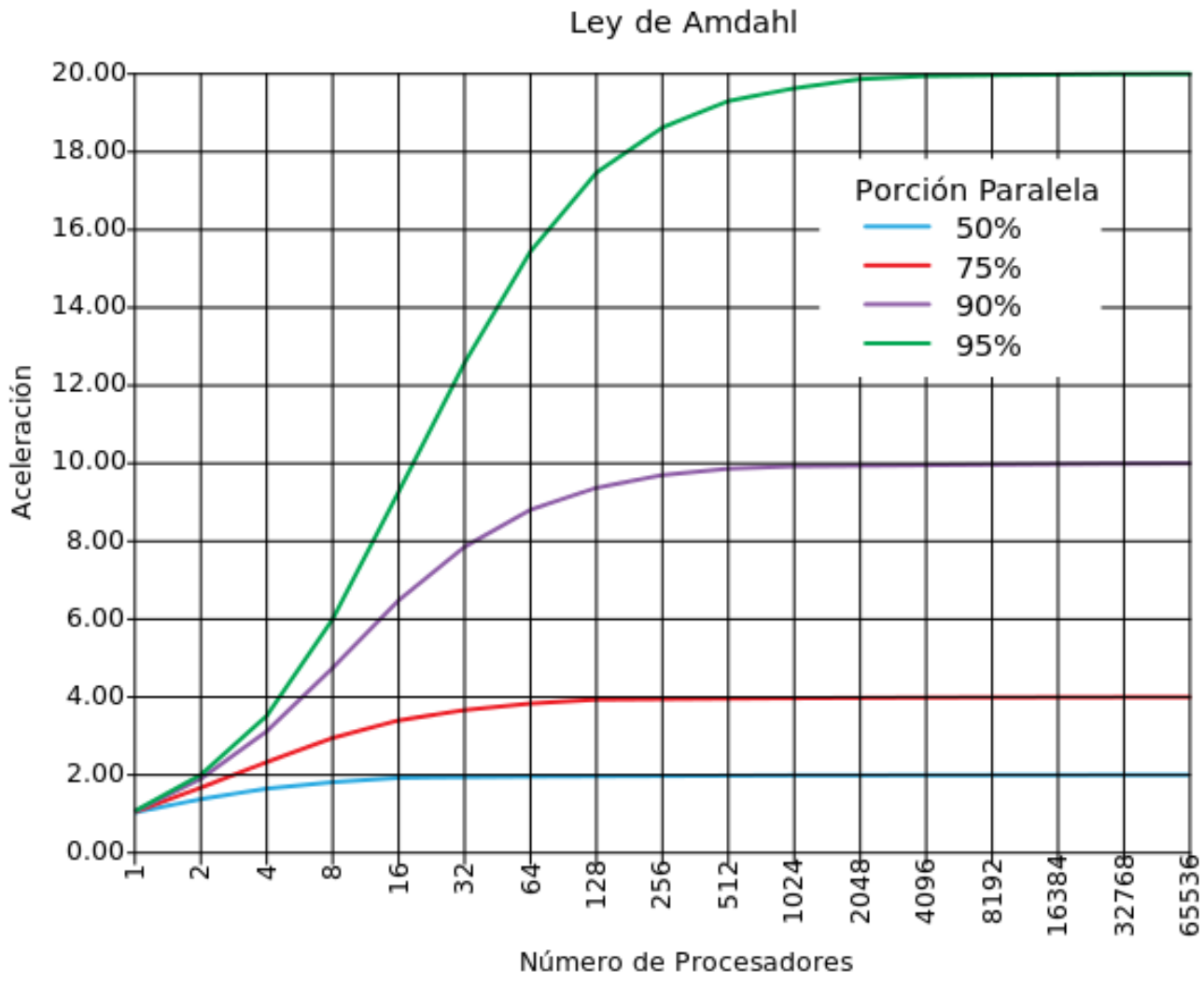


Imagen obtenida de la pág 69 de
"The Art of Concurrent Programming"
de Breshears, O'Reilly 2009

Tiempo de ejecución de un algoritmo



Wikipedia

Diseño de Algoritmos Paralelos

code URJC

- **Introducción**
  - Paralelización de algoritmos
  - Tiempo de ejecución de un algoritmo
  - **Metodología de diseño de algoritmos paralelos**
  - Modelos de algoritmos paralelos
  - Criterios de evaluación de un algoritmo paralelo
  - Reglas para diseñar un algoritmo paralelo
- Algoritmos sobre arrays
- Esquema ForkJoin

Metodología de diseño de algoritmos paralelos

- Algunos autores recomiendan que se sigan la siguiente metodología:
  - **Análisis:** Identificar las partes paralelizables en el algoritmo secuencial
  - **Diseño e implementación:** Implementar el algoritmo, sin preocuparse de la eficiencia
  - **Pruebas y corrección de bugs:** Todos los códigos tienen errores, especialmente los algoritmos concurrentes
  - **Optimización:** Revisar qué partes del algoritmo se podrían optimizar

Metodología de diseño de algoritmos paralelos

- **No se recomienda** el diseño y la implementación de un algoritmo **paralelo desde el principio**
- La programación concurrente es una disciplina **muy compleja y propensa a errores**
- El tiempo **invertido** en implementar la versión secuencial reduce el tiempo empleado en **validar** e implementar la versión paralela

Diseño de Algoritmos Paralelos

code URJC

- **Introducción**
  - Paralelización de algoritmos
  - Tiempo de ejecución de un algoritmo
  - Metodología de diseño de algoritmos paralelos
  - **Modelos de algoritmos paralelos**
  - Criterios de evaluación de un algoritmo paralelo
  - Reglas para diseñar un algoritmo paralelo
- Algoritmos sobre arrays
- Esquema ForkJoin

Modelos de algoritmos paralelos

code URJC

- Para paralelizar un algoritmo habitualmente se parte de un **algoritmo secuencial**
- De esa forma se tiene un algoritmo de referencia que funciona, lo que permite comprobar que el **algoritmo paralelo es correcto**
- También se puede **medir experimentalmente el *speedup*** de la versión paralela comparado con la versión secuencial

Modelos de algoritmos paralelos

code URJC

- Para convertir un algoritmo secuencial en un algoritmo paralelo primero hay que identificar qué cálculos pueden ejecutarse **concurrentemente** y cuales **secuencialmente**
- Hay dos estrategias para paralelizar un algoritmo
  - **Descomposición en tareas:** Cada hilo puede ejecutar concurrentemente un conjunto de tareas
  - **Descomposición de datos:** Cada hilo puede procesar una parte del conjunto total de datos.

Descomposición en tareas

- Algunos ejemplos de lo que puede ser una tarea:
  - La ejecución de un método
  - Cada una de las iteraciones de un bucle
  - Un grupo de sentencias

> Por ejemplo, procesar una carpeta del disco es una tarea dentro un algoritmo de búsqueda de ficheros duplicados

- Hay que intentar que las tareas sean lo más **independientes entre sí**

Descomposición en tareas

- El esquema básico de un algoritmo basado en **descomposición de tareas**:
  - Un hilo principal encargado de crear las tareas
  - Las tareas se ejecutan en diferentes hilos
  - El hilo principal espera a que todas las tareas han terminado su ejecución
  - El hilo principal muestra, guarda o envía el resultado obtenido por los demás hilos

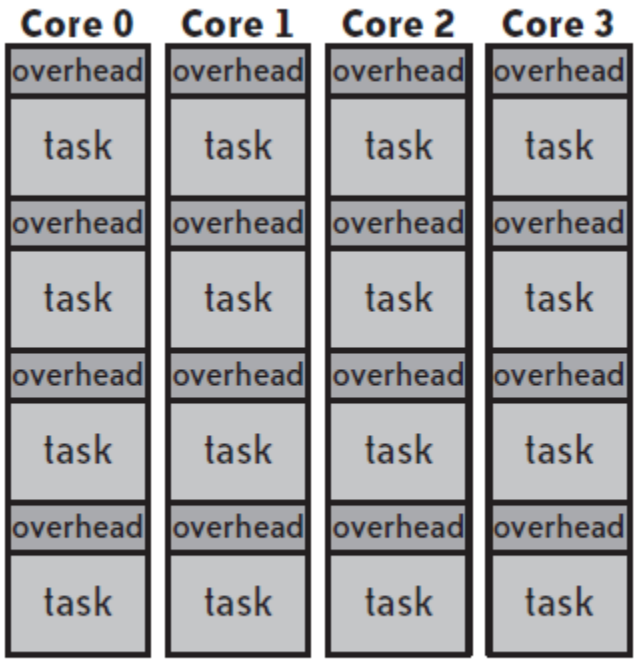
Descomposición en tareas

- Puede haber muchas **variaciones** al esquema dependiendo del algoritmo, por ejemplo:
  - Es posible que no todas las tareas se ejecuten si el objetivo del algoritmo es buscar un elemento. Cuando el elemento es encontrado por alguna tarea, el algoritmo puede finalizar.
  - Es posible que las tareas se vayan descubriendo a medida que otras tareas se van ejecutando (como en el ejemplo de búsqueda de ficheros duplicados)
  - Si los recursos son limitados, el hilo principal podría también ejecutar tareas en vez de simplemente esperar al resto de hilos

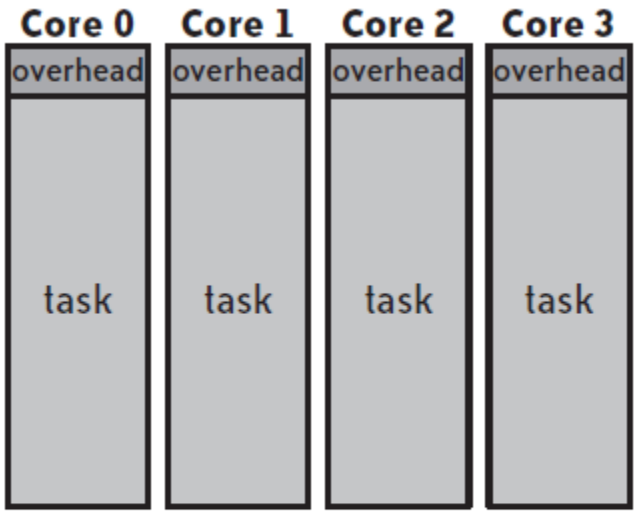
Descomposición en tareas

- El **tamaño** sí importa:
  - Debe haber el **suficiente número de tareas** para que todos los procesadores disponibles estén ejecutando tareas
  - Las tareas deben ser lo **suficientemente grandes** como para no sobrecargar a los hilos con código de gestión de tareas y realizar trabajo útil la mayor parte del tiempo
- Es un cuestión de **granularidad**:
  - Pocas tareas de **grano grueso** pueden dejar recursos ociosos (idle)
  - Muchas tareas de **grano fino** puede ser una sobrecarga de sincronización y gestión de tareas

Descomposición en tareas



(a) Fine-grained decomposition



(b) Coarse-grained decomposition

Imagen obtenida de la pág 26 de "The Art of Concurrent Programming" de Breshears, O'Reilly 2009

Descomposición en tareas

- **Asignación de tareas a hilos**
  - Intenta definir las tareas de forma que las tareas sean **independientes**
  - Hay veces que una tarea **depende** de los **resultados** de otra tarea o que deba realizarse **después** de otra tarea por otro motivo
  - En ese caso hay dos opciones:
    - Usar herramientas de **sincronización** entre hilos: sincronización condicional, exclusión mutua, colas, etc…)
    - Asignar al **mismo hilo** tareas que deban ejecutarse en un orden determinado (pero no es siempre posible)

Descomposición en tareas

- **Asignación de tareas a hilos**
  - Las tareas se pueden asignar a hilos:
    - **Asignación estática:** Se divide el conjunto total de tareas entre los hilos disponibles.
    - **Asignación dinámica:** Los hilos ejecutan las tareas disponibles cuando terminan la tarea anterior.
  - En general es mejor la **asignación dinámica** porque permite un mejor balanceo de carga, aunque requiere que haya bastantes **más tareas que hilos**
  - En Java la asignación dinámica se consigue con un **Executor**

Descomposición de datos

- Un algoritmo se pueden paralelizar con una descomposición de datos si…
  - …procesa una gran cantidad de datos y
  - realiza cálculos basados en cada uno de los elementos de esos datos y
  - el resultado se escribe en otra estructura de datos (o en la misma) y
  - el cálculo de cada dato es independiente entre sí

Descomposición de datos

- El **esquema básico** de un algoritmo basado en descomposición de **datos**:
  - Un hilo principal se encarga de asignar partes del conjunto de datos (**chunks**) a cada uno de los hilos
  - El **hilo principal espera** a que todos los hilos hayan terminado su ejecución
  - El **hilo principal muestra**, guarda o envía el resultado obtenido por los demás hilos

Descomposición de datos

- **Tamaño de los chunks y asignación a hilos**
  - Si el tiempo de cómputo de cada **chunk es similar**, es mejor tener una **asignación estática** de chunks a hilos
  - Si los **chunks** tienen tiempos de procesamiento **diferentes** (por tamaño, por complejidad, etc…) es mejor tener varios chunks por hilo y usar la **asignación dinámica** de chunks a hilos

Descomposición de datos

- **División de datos en chunks**
  - Diferentes estrategias de división de una matriz de 2 dimensiones

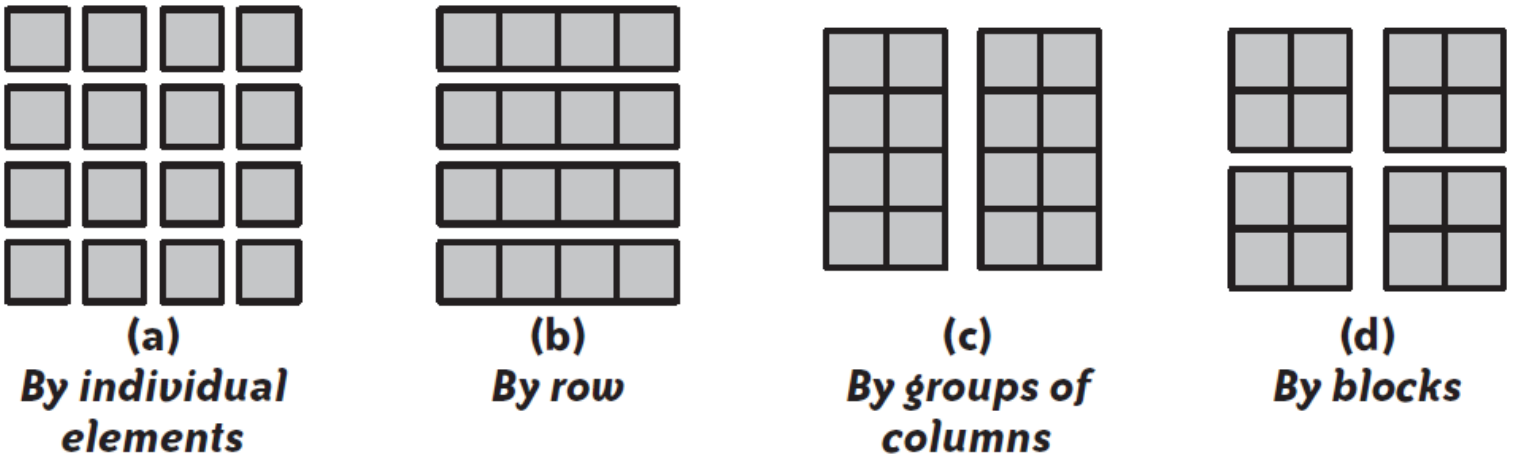


Imagen obtenida de la pág 34 de "The Art of Concurrent Programming" de Breshears, O'Reilly 2009

Diseño de Algoritmos Paralelos

code URJC

- **Introducción**
  - Paralelización de algoritmos
  - Tiempo de ejecución de un algoritmo
  - Metodología de diseño de algoritmos paralelos
  - Modelos de algoritmos paralelos
  - **Criterios de evaluación de un algoritmo paralelo**
  - Reglas para diseñar un algoritmo paralelo
- Algoritmos sobre arrays
- Esquema ForkJoin

Criterios de evaluación de un algoritmo paralelo

- Los algoritmos paralelos se evalúan en base a los siguientes criterios:
  - Eficiencia
  - Simplicidad
  - Portabilidad
  - Escalabilidad

Criterios de evaluación de un algoritmo paralelo

- **Eficiencia**
  - Se evalúa la sobrecarga añadida por la sincronización entre hilos
  - Cuando menor es la sobrecarga, más eficiente es el algoritmo
  - Siempre existen diferentes alternativas, algunas más eficientes que otras

Criterios de evaluación de un algoritmo paralelo

- **Simplicidad**
  - Cuanto más sencillo es el algoritmo, más fácil es de diseñar, implementar, mantener, entender, corregir errores, etc…
  - También hace referencia a la parte del algoritmo paralelo que sigue igual que en el algoritmo secuencial. Cuanto más parecido, más simple.

Criterios de evaluación de un algoritmo paralelo

- **Portabilidad**
  - Si existen dos soluciones igual de eficientes y sencillas, siempre será mejor aquella que se pueda aplicar con facilidad a otros modelos de concurrencia
  - Nosotros estudiamos algoritmos en memoria compartida, pero es bueno saber cómo de portable en el algoritmo en un modelo de paso de mensajes

Criterios de evaluación de un algoritmo paralelo

- **Escalabilidad**
  - Los algoritmos paralelos tienen que ser escalables, de forma que aprovechen el número de procesadores disponibles
  - Algunos algoritmos tienen cuellos de botella, puntos únicos de control, que pueden limitar la escalabilidad

Diseño de Algoritmos Paralelos

code URJC

- **Introducción**
  - Paralelización de algoritmos
  - Tiempo de ejecución de un algoritmo
  - Metodología de diseño de algoritmos paralelos
  - Modelos de algoritmos paralelos
  - Criterios de evaluación de un algoritmo paralelo
  - **Reglas para diseñar un algoritmo paralelo**
- Algoritmos sobre arrays
- Esquema ForkJoin

Reglas para diseñar un algoritmo paralelo

- **Regla 1:** Identifica los cálculos realmente independientes
- **Regla 2:** Implementa la concurrencia con el mayor nivel de abstracción posible
- **Regla 3:** Diseño pensando en la escalabilidad (para aprovechar los recursos)
- **Regla 4:** Reutiliza librerías y componentes concurrentes

Reglas para diseñar un algoritmo paralelo

- **Regla 5:** Utiliza el modelo de concurrencia adecuado (Paso de mensajes, memoria transaccional, estado compartido…)
- **Regla 6:** Nunca asumas un orden de ejecución determinado
- **Regla 7:** Reduce la sincronización lo más posible
- **Regla 8:** Evalúa otros algoritmos secuenciales para resolver el problema como base para paralelizar (se pueden obtener mejores resultados)

Diseño de Algoritmos Paralelos

code URJC

- Introducción
- Algoritmos sobre arrays
  - **Suma**
  - Ordenación
- Esquema ForkJoin

Suma

- Los algoritmos más usados en la programación son aquellos que procesan arrays (o listas) de elementos
- En este tema se paralelizarán los algoritmos básicos sobre arrays:
  - Suma
  - Ordenación

Algoritmo secuencial

- Se pretende sumar un array de números enteros de forma paralela
- El algoritmo secuencial es muy sencillo

```
int[] valores = new int[1000];
int suma = 0;
for(int i=0; i<valores.length; i++){
  suma += valores[i];
}
System.out.println("Suma = "+suma);
```

- Al paralelizarse se pueden estudiar las bases para la paralelización de otros tipos de algoritmos más complejos

Algoritmo secuencial

- La operación de suma tiene las propiedades:
  - **Asociativa:** Se pueden sumar dos operandos y el resultado sumarle a otro operando en cualquier orden
  - **Conmutativa:** El orden de los operandos no afecta al resultado
- Cualquier operación con estas propiedades puede paralelizarse con el mismo esquema

Diseño del algoritmo paralelo

- La suma se puede paralelizar como un problema de **descomposición de datos**
- El conjunto de datos se puede **dividir en chunks iguales** y la computación es **similar** por cada elemento
- Esto permite hacer una **asignación estática**: Asignar a cada hilo una parte del array
- Cada hilo suma en una **variable local** y al terminar hace accesible el valor al hilo **main**
- El hilo **main** suma las sumas **parciales** de cada hilo

Introducción

```
Thread[] threads = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
  final int numThread = i;
  Thread t = new Thread(()-> suma(numThread));
  threads[i] = t;
  t.start();
}

int suma = 0;
for (int i = 0; i < NUM_THREADS; i++) {
  threads[i].join();
  suma += sumaCompar[i];
}

System.out.println("Suma de los elementos del array:"+suma);
```

Creación de hilos para que ejecuten el método suma(...)

Espera por cada hilo y suma del valor que ha calculado para obtener la suma completa

Implementación del algoritmo paralelo


```
private double[] valores;
private double[] sumaCompar = new double[NUM_THREADS];

private void suma(int numThread) {
  int sumaLocal = 0;
  int size = valores.length / NUM_THREADS;
  int start = numThread * size;
  int end = start + size;
  for (int i = start; i < end; i++) {
    sumaLocal += valores[i];
  }
  sumaCompar[numThread] = sumaLocal;
}
```

Suma de los valores del chunk asignado a cada hilo

Asignar la suma local al array compartida

Evaluación

- **Eficiencia**
  - El cálculo de los límites del chunk es sobrecarga (código de coordinación incorporado al algoritmo secuencial)
  - Esta paralelización tiene sentido si los chunks son lo suficientemente grandes para que ese tiempo sea despreciable

Evaluación

- **Simplicidad**
  - Conceptualmente la paralelización es muy sencilla
  - Hay mucho código nuevo porque:
    - La librería Java es muy explícita en muchas operaciones (creación de hilo, espera para la terminación de varios, etc…)
    - El código secuencial es muy sencillo

Evaluación

- **Portabilidad**
  - Las ideas aplicadas se pueden usar en un modelo de paso de mensajes por el array compartido para las sumas de cada chunk se puede sustituir por el envío de la suma al proceso principal
- **Escalabilidad**
  - El código escala bien con un mayor número de cores
  - Si el chunk es demasiado pequeño, la sobrecarga de coordinar los hilos puede ser importante

Diseño de Algoritmos Paralelos

code URJC

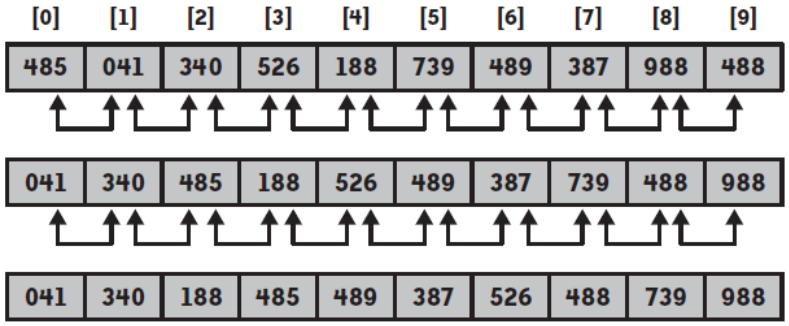
- Introducción
- Algoritmos sobre arrays
  - Suma
  - **Ordenación**
- Esquema ForkJoin

Ordenación

- Existen muchos algoritmos para ordenar arrays de elementos
- Estudiaremos cómo paralelizar el más básico, el de la **burbuja**

Algoritmo secuencial

- Se pretende ordenar un array de números enteros con el **método de la burbuja**
- Con este método se recorre el array verificando si cada par de elementos consecutivos están **ordenados correctamente** y si no es así, **se intercambian** entre sí
- Se realizan pasadas hasta que cada par de elementos consecutivos está **ordenado**



ORDENACIÓN BURBUJA
Algoritmo secuencial

```
for (int i = NUM_VALS - 1; i > 0; i--) {

  boolean intercambio = false;

  for (int j = 0; j < i; j++) {
    if (valores[j] > valores[j + 1]) {

      int temp = valores[j];
      valores[j] = valores[j + 1];
      valores[j + 1] = temp;
      intercambio = true;
    }
  }

  if(!intercambio){
    break;
  }
}
```

Se hacen tantas pasadas como elementos en el array.

En cada pasada se omite un elemento porque se ordenó en la pasada anterior

Si dos elementos adyacentes no están bien ordenados, se intercambian entre sí

Si en una pasada no se hace ningún intercambio, el array está ordenado

ORDENACIÓN BURBUJA
Algoritmo secuencial

```
int i = NUM_VALS;
while (true) {
  i--;
  if (i <= 0) { break; }

  boolean intercambio = false;

  for (int j = 0; j < i; j++) {
    if (valores[j] > valores[j + 1]) {

      int temp = valores[j];
      valores[j] = valores[j + 1];
      valores[j + 1] = temp;
      intercambio = true;
    }
  }

  if (!intercambio) { break; }
}
```

Otra forma de escribir el algoritmo con un while(true) en vez de un for

Diseño del algoritmo paralelo

- Para paralelizar el algoritmo de la burbuja se pretende que varios hilos puedan realizar las **pasadas** sobre el array **concurrentemente**
- **Sincronización** entre hilos
  - Si **no hay sincronización**, los hilos se pueden "adelantar" entre sí (provocando condiciones de carrera)
  - Si se **sincroniza cada par** de elementos puede haber demasiada **sobrecarga** por esa sincronización

Diseño del algoritmo paralelo

- Hay que **sincronizar** los hilos, pero no sincronizarlos en cada elemento
- Solución:
  - **Dividir** el array en varias **zonas** iguales
  - En cada zona sólo puede trabajar un hilo (poner las **zonas bajo exclusión mutua**)
  - Para aprovechar los recursos y que todos los hilos puedan trabajar: crear **más zonas que hilos**

ORDENACIÓN BURBUJA
Implementación del algoritmo paralelo

```
int tamannoZona = (NUM_VALS / NUM_ZONAS) + 1;

while (!fin) {  //Inicializado a false

  int numZona = 0; boolean intercambio = false;

  locks[numZona].lock();

  int i = iCompartido--;//Inicializado a NUM_VALS-1
  if (i <= 0) { fin = true; locks[numZona].unlock(); break; }

  int finZona = tamannoZona;
  for (int j = 0; j < i; j++) {
    if (valores[j] > valores[j + 1]) {
      int temp = valores[j];
      valores[j] = valores[j + 1];
      valores[j + 1] = temp;
      intercambio = true;
    }
    if (j == finZona) {
      locks[numZona].unlock();
      numZona++;
      locks[numZona].lock();
      finZona += tamannoZona;
    }
  }
  locks[numZona].unlock();
  if (!intercambio) { fin = true; }
}
```

Cada zona está bajo EM.
En la zona o se determina el último elemento a evaluar

Variable global que cuenta las pasadas y guarda el último elemento que hay que evaluar

El algoritmo finaliza cuando el último elemento a evaluar es o

Al llegar al final de cada zona, se libera la EM y se pasa a la siguiente EM (que habrá liberado el hilo anterior)

Evaluación

- **Eficiencia**
  - El algoritmo original no es muy eficiente, y la versión paralela tampoco lo es porque comparte la misma esencia
  - La paralelización impone bastante sobrecarga porque cada hilo, en cada pasada, debe entrar en la sección crítica de cada zona (y puede haber muchas pasadas)
- **Simplicidad**
  - El código de separación en zonas dificulta la comprensión del algoritmo original

Evaluación

- **Portabilidad**
  - El algoritmo paralelo propuesto para memoria compartida no se puede usar en un modelo de paso de mensajes porque habría mucha información que enviar y recibir
- **Escalabilidad**
  - Mayor número de hilos implica mas zonas y más sobrecarga en el acceso a cada una de ellas.
  - Cuando el algoritmo avanza, las pasadas son más cortas (y tienen menos zonas) y muchos hilos no conseguiran una "zona libre"

Java Fork Join

code URJC

- **Fork Join** es un conjunto de clases de Java que permiten implementar algoritmos en paralelo
- Ofrecen facilidades para:
  - Dividir un algoritmo en partes (tareas)
  - Ejecutar las tareas en paralelo (en varios hilos)
  - Fusionar el resultado de cada tarea cuando haya terminado
  - Devolver el resultado final

Java Fork Join

code URJC

- El principio de funcionamiento de **Fork Join** es el siguiente:

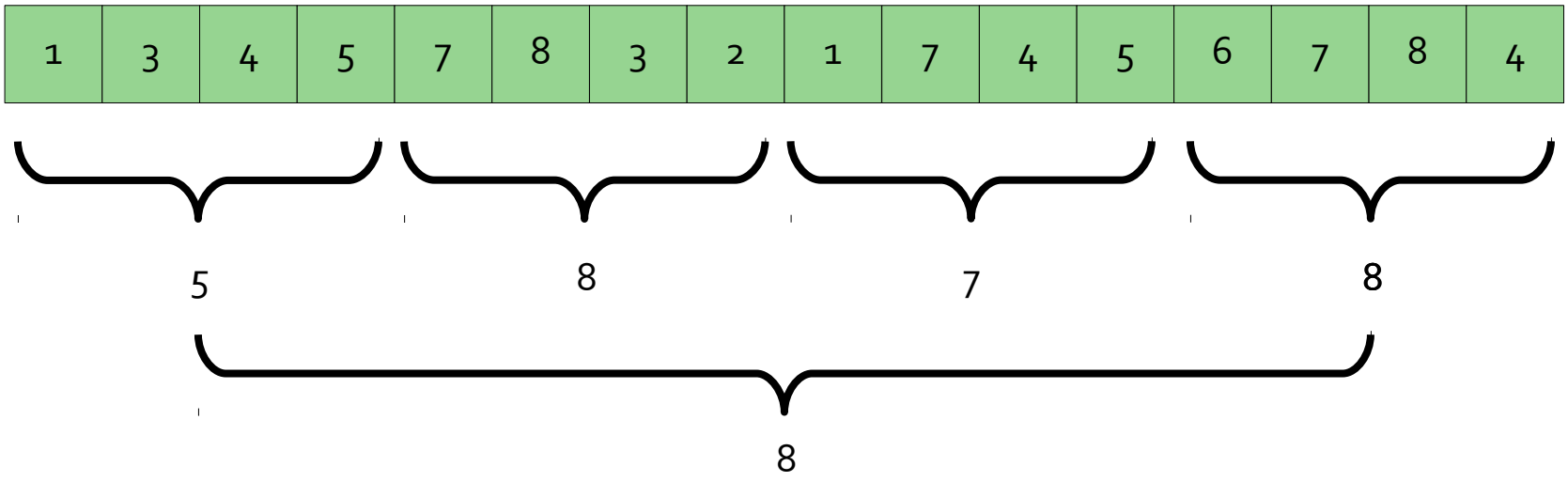
```
Resultado resolver(Problema problema) {

  if (problema es pequeño)
     Resolver el problema
     Devolver Resultado
  } else {
     Dividir el problema en problemas más pequeños
     Resolver cada sub-problema recursivamente y en paralelo
     Esperar a que acabe de resolverse cada subproblema
     Fusionar los resultados de cada sub-problema
     Devolver el resultado final
   }
}
```

Java Fork Join

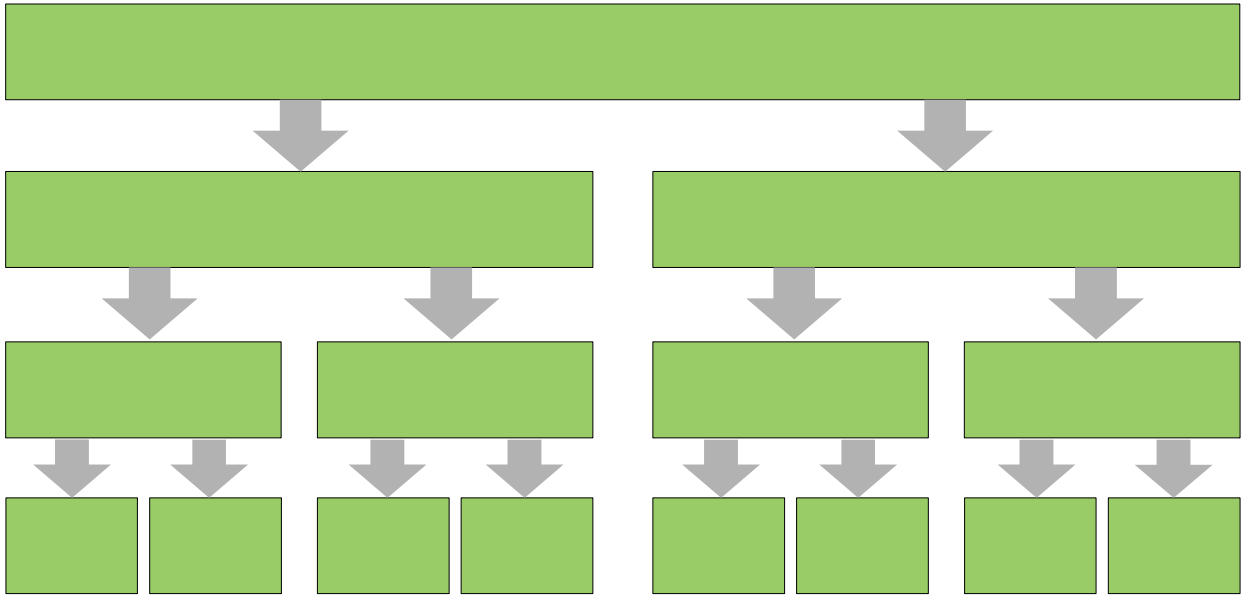
code URJC

- **Ejemplo:** Buscar el máximo en un array
  - Se divide el array en partes y se busca en cada parte
  - Cuando han terminado las partes, se calcula el máximo entre los resultados



Java Fork Join

- Fork Join permite crear tareas que se dividen a su vez en tareas más pequeñas formando un **árbol** con **varios niveles**



Java Fork Join

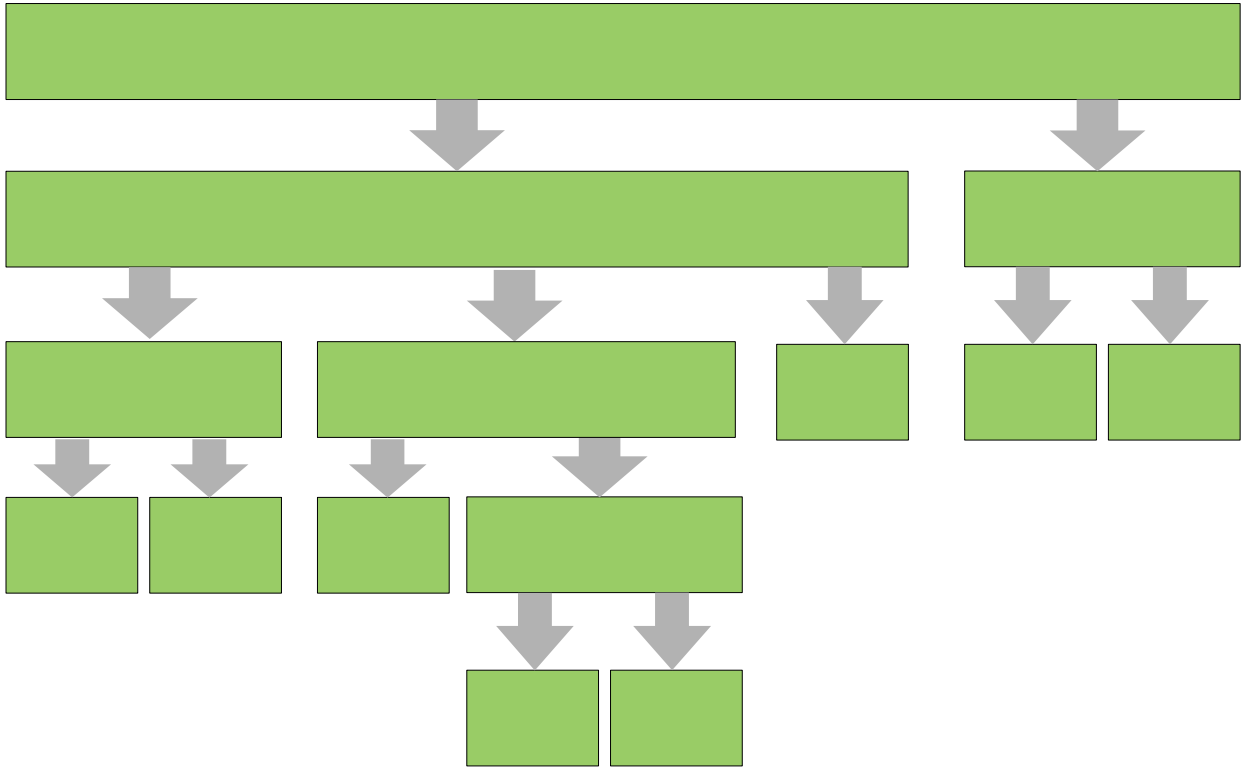
code URJC

- El **tamaño de las hojas** (las tareas más pequeñas) tiene que ser el adecuado para que no haya mucha sobrecarga y tampoco se desperdicien los recursos
- Las tareas pueden tener **diferentes tamaños** (coste de ejecución)
- El árbol **no tiene por qué ser equilibrado**, algunas ramas pueden ser más profundas que otras

Java Fork Join

code URJC

- Árbol desequilibrado



Java Fork Join

code URJC

- Para usar Fork Join
  - Se implementa una tarea como una clase hija de **java.util.concurrent.RecursiveTask<R>**
  - Se implementa el método **<R> compute()**
    - Si la tarea es pequeña, se ejecuta y se devuelve el resultado
    - Si la tarea es grande, se divide el trabajo creando tareas más pequeñas
      - Se ejecutan con el método fork()
      - Se espera a que acaben con el método join()
      - Se combinan los resultados y se devuelve

Java Fork Join

code URJC

- Ejemplo
  - Calcular el máximo de un array
  - Tarea:
    - Referencia al **array** y la **región** que tiene que procesar (desde la posición **start** a la posición **end**)
    - En el método **compute** comprueba el tamaño de la **región:**
      - Si es pequeña (<5) se procesa secuencialmente
      - Si es grande, se divide en 2 subtareas, se ejecutan en paralelo (fork), se espera su resultado (join) y se procesa

Java Fork Join

code URJC

```
public class MaxArrayTask extends RecursiveTask<Integer> {

    private int[] data;
    private int start;
    private int end;

    public MaxArrayTask(int[] data, int start, int end) {
        this.data = data;
        this.start = start;
        this.end = end;
    }

    protected Integer compute(){ … }
}
```

Java Fork Join

code URJC

```
protected Integer compute() {

    int length = end - start;

    if (length < 5) {
        return computeDirectly();
    } else {
        return computeInSubtasks();
    }
}
```

Java Fork Join

code URJC

```
private Integer computeDirectly() {

    int max = Integer.MIN_VALUE;

    for (int i = start; i < end; i++) {
        if (data[i] > max) {
            max = data[i];
        }
    }

    return max;
}
```

Java Fork Join

code URJC

```
private Integer computeInSubtasks() {

  int mid = (end - start) / 2;

  MaxArrayTask left = new MaxArrayTask(data, start, start + mid);
  MaxArrayTask right = new MaxArrayTask(data, start + mid, end);

  left.fork();
  right.fork();

  return Math.max(right.join(), left.join());

}
```

Java Fork Join

code URJC

- Para ejecutar estas tareas de ForkJoin se usa un Executor especial llamado **java.util.concurrent.ForkJoinPool**
- Al construir se puede indicar el **grado de paralelismo** (número de threads ejecutando tareas y no bloqueados)
- Las tareas se envían con el método **invoke()** para esperar por el resultado

Java Fork Join

code URJC

```
int[] data = new int[1000];

//Fill with random data

ForkJoinPool pool = new ForkJoinPool(4);

MaxArrayTask task = new MaxArrayTask(data, 0, data.length);

System.out.println("Max: " + pool.invoke(task));
```

Java Fork Join

code URJC

- **Ejemplo**
  - Búsqueda de ficheros cuyo nombre incluya un patrón
  - Por cada sub-directorio se crea una nueva tarea
  - Los ficheros se tratan secuencialmente
  - Cada tarea se divide en un número de subtareas que depende de los datos (las divisiones no son siempre iguales)
  - El árbol de tareas no es simétrico

```
protected List<File> compute() {

    List<File> filesFound = new ArrayList<>();
    List<FileTask> taskList = new ArrayList<>();

    for (File file : dir.listFiles()) {

        if (file.isDirectory()) {
            FileTask task = new FileTask(file,pattern);
            task.fork();
            taskList.add(task);

        } else {
            if (file.getName().contains(pattern)) {
                filesFound.add(file);
            }
        }
    }

    // Wait for tasks
    for (FileTask task : taskList) {
        filesFound.addAll(task.join());
    }

    return filesFound;
}
```

Java Fork Join

code URJC

- **Ejercicio 1:**
  - Implementa el ejercicio del parallelForSum usando ForkJoin