Bucles Paralelos

Un **bucle paralelo** permite que múltiples iteraciones se ejecuten **al mismo tiempo**, aprovechando varios núcleos del procesador. Esto es útil cuando las iteraciones **no dependen unas de otras** y se quiere mejorar el rendimiento.

Se distinguen dos tipos:

Bucle while paralelo: adecuado cuando no se conoce cuántas iteraciones habrá (ej. recorrer una lista enlazada).

```
finish {
  for (p = head; p != null; p = p.next)
    async compute(p);
}
```

Cada compute(p) se ejecuta en paralelo.

Bucle contado-para paralelo (forall): adecuado cuando sí sabemos cuántas iteraciones hay (por ejemplo, recorrer un arreglo con índices conocidos).

```
forall (i : [0:n-1]) a[i] = b[i] + c[i];
```

Cada suma b[i] + c[i] se hace en paralelo.

Los flujos de Java pueden ser una forma elegante de

especificar cálculos de bucles paralelos que producen una única matriz de salida, por

ejemplo, reescribiendo la sentencia de suma de vectores de la siguiente manera:

Ejemplo concreto en Java

Supongamos que tienes tres arreglos:

```
int[] b = {1, 2, 3, 4};

int[] c = {5, 6, 7, 8};
```

Y quieres calcular a[i] = b[i] + c[i] en paralelo. Usamos:

```
int[] a = IntStream.range(0, b.length)
                .parallel()
                .map(i -> b[i] + c[i])
                .toArray();
```

¿Qué hace esto?

- `range(0, b.length)` crea los índices.
- `.parallel()` indica que se deben ejecutar en paralelo.
- `.map(i -> b[i] + c[i])` hace la suma para cada `i`.
- `.toArray()` devuelve el nuevo arreglo con los resultados.

¿Cuándo usar `forall`?

Usamos `forall` o estructuras equivalentes cuando:

- Sabemos cuántas iteraciones hay.
- Cada iteración es independiente.
- Se busca eficiencia computacional.

En resumen, los flujos son una notación conveniente para bucles paralelos con como máximo una matriz de salida, pero la notación forall es más conveniente para bucles que crean/actualizan múltiples matrices de salida, como ocurre en muchos cálculos científicos. Por razones de generalidad, utilizaremos la notación forall para los bucles paralelos en el resto de este módulo.

Algoritmo clásico para la multiplicación de matrices:

```
for (i : [0:n-1]) {
    for (j : [0:n-1]) {
        c[i][j] = 0;
        for (k : [0:n-1]) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

Vamos a analizar **cuáles bucles se pueden paralelizar usando `forall`** (es decir, convertirlos a bucles paralelos).

Análisis de dependencia

Para que un bucle se pueda paralelizar (convertir a `forall`), **las iteraciones deben ser independientes** entre sí, es decir, que no se compartan ni modifiquen las mismas posiciones de memoria.

1. **Bucle externo (`for i`) y medio (`for j`)**:
    - Cada par `(i, j)` escribe en una **celda única** `c[i][j]`.
    - Por tanto, **no hay dependencia entre distintas iteraciones** de `i` y `j`.
    - **Ambos se pueden paralelizar.**
2. **Bucle interno (`for k`)**:
    - Modifica `c[i][j]` **acumulativamente**, lo cual introduce una **dependencia** entre las iteraciones de `k`.
    - **No se puede paralelizar directamente** (a menos que uses una reducción paralela).

¿Cómo sería la versión con forall?

En pseudocódigo paralelo con forall, podemos escribir:

```
forall (i : [0:n-1]) {
    forall (j : [0:n-1]) {
        c[i][j] = 0;
        for (k : [0:n-1]) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

Incluso podrías fusionar ambos forall en uno doble:

```
forall (i,j : [0:n-1] x [0:n-1]) {
    c[i][j] = 0;
    for (k : [0:n-1]) {
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
```

```
}
```

El bucle k debe seguir siendo secuencial (a menos que lo paralelices con una operación de reducción).

Alternativa con reducción paralela (avanzado)

Si tu entorno permite **reducciones paralelas**, podrías incluso hacer que el for k sea paralelo:

```
forall (i,j : [0:n-1] x [0:n-1]) {
    c[i][j] = reduce(+, k in [0:n-1]) a[i][k] * b[k][j];
}
```

Esto suma en paralelo todos los productos a[i][k]*b[k][j].

Implementación en Java con IntStream.parallel()

Podemos usar IntStream.range(...).parallel() de Java 8+.

```
int n = ...;
int[][] a = ...;
int[][] b = ...;
int[][] c = new int[n][n];
IntStream.range(0, n).parallel().forEach(i -> {
    for (int j = 0; j < n; j++) {
        int sum = 0;
        for (int k = 0; k < n; k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] = sum;
    }
});
```

- Creamos un stream paralelo sobre i.
- Cada **hilo** se encarga de computar una fila i completa.
- Dentro de cada hilo, j y k siguen siendo secuenciales.
- Como cada hilo **escribe en una fila distinta de c**, no hay conflictos.

¿Y si queremos paralelizar también j?

Podemos hacer un flatMap sobre los pares (i,j):

```
IntStream.range(0, n).boxed().flatMap(i ->
    IntStream.range(0, n).mapToObj(j -> new int[]{i, j})
).parallel().forEach(pair -> {
    int i = pair[0], j = pair[1];
    int sum = 0;
    for (int k = 0; k < n; k++) {
        sum += a[i][k] * b[k][j];
    }
    c[i][j] = sum;
});
```

- Paralelizamos **cada elemento c[i][j] por separado**, no solo filas.
- Esto puede aprovechar aún más núcleos, útil para matrices **muy grandes**.
- Sigue siendo seguro porque cada hilo accede a **una única posición c[i][j]**.

Barreras en Bucles Paralelos

Una **barrera** es un punto de sincronización en un bucle paralelo donde **todas las iteraciones deben llegar antes de continuar**.

Sirve para **dividir un bucle paralelo en fases secuenciales**, garantizando cierto orden de ejecución entre las partes del bucle.

En la práctica, una barrera se implementa con **sincronización de hilos**. Veamos cómo se hace en **Java**, que es un lenguaje común para programación paralela.

BarreraEjemplo.java

```java
import java.util.concurrent.*;

public class BarreraEjemplo {
    static final int N = 4;
    static CyclicBarrier barrier = new CyclicBarrier(N);

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(N);

        for (int i = 0; i < N; i++) {
            final int id = i;
            executor.submit(() -> {
                String myId = "Thread-" + id;
                System.out.println("HELLO " + myId);

                try {
                    barrier.await(); // barrera
                } catch (Exception e) {
                    e.printStackTrace();
                }

                System.out.println("BYE " + myId);
            });
        }

        executor.shutdown();
    }
}
```

input

```
HELLO Thread-0
HELLO Thread-2
HELLO Thread-3
HELLO Thread-1
BYE Thread-1
BYE Thread-0
BYE Thread-2
BYE Thread-3
```

Proceso:

Crea 4 hilos (`N = 4`)
Cada hilo imprime `"HELLO Thread-x"`
Luego **espera en la barrera**
Cuando **todos llegan**, imprimen `"BYE Thread-x"`

Promedio iterativo unidimensional

Un **esténcil** es un patrón computacional usado en muchos algoritmos numéricos (como simulaciones físicas, soluciones de ecuaciones diferenciales, procesamiento de imágenes, etc.) donde el valor de una celda se calcula como función de sus **vecinos**.

Xi =(Xi−1 + Xi+1)/2 con condiciones de
contorno, X0 = 0 y Xn = 1. Aunque podemos derivar fácilmente una solución analítica para este ejemplo, (Xi = i/n), la mayoría de los códigos esténcil en la práctica no tienen soluciones analíticas conocidas y dependen del cálculo para obtener una solución.

Método de Jacobi

Se usa una **doble matriz**:

- oldX[]: contiene los valores actuales.
- newX[]: guarda los nuevos valores calculados.

```
for (iter = 0; iter < nsteps; iter++) {
   for (i = 1; i < n; i++) {
      newX[i] = (oldX[i-1] + oldX[i+1]) / 2;
   }
   swap(oldX, newX);
}
```

Proceso:

[0, 0, 0, 0, 1] (oldX inicial)

iteración 1: [0, 0, 0, 0.5, 1]

iteración 2 [0, 0, 0.25, 0.5, 1]

iteracion 3 `[0, 0.125, 0.25, 0.625, 1]`

iteracion 4 [0, 0.125, 0.375, 0.625, 1]

iteracion 5 [0, 0.1875, 0.375, 0.6875, 1]

… [0,0.25,0.5,0.75,1] (Tras varias iteraciones se aproxima a la solución)

Cada paso iter representa una **iteración de refinamiento** de la solución.

Versión paralela ingenua (demasiadas tareas)

```
for (iter = 0; iter < nsteps; iter++) {
    forall (i = 1 to n-1) {
        newX[i] = (oldX[i-1] + oldX[i+1]) / 2;
    }
    swap(newX, oldX);
}
```

Cada una de las nsteps iteraciones crea **n-1 tareas**, lo que da como resultado nsteps × (n-1) tareas paralelas. Esto es **ineficiente**, porque:

Crear tareas cuesta tiempo.
Es difícil manejar tantos hilos concurrentes.
El sistema se sobrecarga.

Solución eficiente: tareas persistentes + barreras

```
forall (i = 1 to n-1) {
    localNewX = newX;
    localOldX = oldX;
    for (iter = 0; iter < nsteps; iter++) {
        localNewX[i] = (localOldX[i-1] + localOldX[i+1]) / 2;
        NEXT; // ⛔ barrera
        swap(localNewX, localOldX);
    }
}
```

- **Sólo se crean n−1 tareas** paralelas (una por cada índice i de 1 a n-1).
- Estas tareas se **ejecutan durante todo el proceso**.
- Después de cada iteración, hay una **barrera** (NEXT) que **espera a que todos terminen** antes de continuar.
- Esto sincroniza correctamente el cálculo y mantiene la coherencia de los datos.