Ejercicio 4

Nombre REDACTED

1 Descripción de las pruebas de desempeño

El archivo `OneDimAveragingPhaserTest.java` contiene un conjunto de pruebas unitarias destinadas a validar la **correctitud** y el **rendimiento** de dos implementaciones paralelas del método de *promedio iterativo unidimensional*:

- `runParallelBarrier` — versión que usa una barrera global mediante `Phaser.arriveAndAwaitAdvance()`, donde todos los hilos deben sincronizarse antes de avanzar a la siguiente iteración.
- `runParallelFuzzyBarrier` — versión con *barrera difusa* o *fuzzy barrier*, donde cada hilo sólo espera a sus vecinos directos utilizando las llamadas `arrive()` y `awaitAdvance(phase)` sobre un conjunto de phasers locales.

El objetivo de estas pruebas es doble:

1. Verificar que ambas implementaciones producen los mismos resultados numéricos.
2. Evaluar si la implementación con *fuzzy barrier* logra una mejora de velocidad (*speedup*) respecto a la versión con barrera global.

1.1 Estructura general de la prueba

El método auxiliar `parTestHelper(N, ntasks)` ejecuta el mismo problema de tamaño N con ambas implementaciones, midiendo el tiempo de ejecución mediante `System.currentTimeMillis()`. Posteriormente calcula el cociente:

$$\text{speedup} = \frac{T_{\text{barrier}}}{T_{\text{fuzzy}}}$$

donde T_{barrier} es el tiempo de la versión con barrera global y T_{fuzzy} el de la versión difusa.

Después, se valida que los resultados numéricos sean idénticos. Dependiendo del número de iteraciones (`niterations`), la comparación se realiza sobre `myNew` o `myVal`:

`checkResult(myNewRef, myNew)` o `checkResult(myValRef, myVal)`.

1.2 La prueba principal: `testFuzzyBarrier`

La función de prueba:

```
public void testFuzzyBarrier() {
    final double expected = 1.1;
    final double speedup = parTestHelper(4 * 1024 * 1024, getNCores() * 16);
    assertTrue(errMsg, speedup >= expected);
}
```

verifica que el *fuzzy barrier* sea al menos un 10% más rápido que la versión con barrera global, es decir, que speedup ≥ 1.1. Si la condición no se cumple, se lanza un `AssertionFailedError` con un mensaje que indica el *speedup* real obtenido.

2 Explicación del código

El método `runParallelFuzzyBarrier` implementa una versión paralela del promedio iterativo unidimensional utilizando una *barrera difusa* o *fuzzy barrier*. Esta permite que cada hilo sólo espere a sus vecinos directos. Lo cual busca reducir el tiempo de espera y permitir un mayor solapamiento de computación.

A continuación se presenta el código completo seguido de una explicación comentada de cada uno de sus componentes.

2.1 Código

```
public static void runParallelFuzzyBarrier(final int iterations,
        final double[] myNew, final double[] myVal, final int n,
        final int tasks) {

    Phaser[] phs = new Phaser[tasks];
    for (int i = 0; i < tasks; i++) {
        phs[i] = new Phaser(1);
    }

    Thread[] threads = new Thread[tasks];

    for (int ii = 0; ii < tasks; ii++) {
        final int i = ii;

        threads[ii] = new Thread(() -> {
            double[] threadPrivateMyVal = myVal;
            double[] threadPrivateMyNew = myNew;

            for (int iter = 0; iter < iterations; iter++) {
                int phase = phs[i].getPhase();

                final int left  = i * (n / tasks) + 1;
                final int right = (i + 1) * (n / tasks);

                for (int j = left; j <= right; j++) {
                    threadPrivateMyNew[j] = (threadPrivateMyVal[j - 1]
                                           + threadPrivateMyVal[j + 1]) / 2.0;
                }

                phs[i].arrive();

                if (i > 0) {
                    phs[i - 1].awaitAdvance(phase);
                }
                if (i < tasks - 1) {
                    phs[i + 1].awaitAdvance(phase);
                }

                double[] temp = threadPrivateMyNew;
                threadPrivateMyNew = threadPrivateMyVal;
                threadPrivateMyVal = temp;
            }
        });
        threads[ii].start();
    }

    for (int ii = 0; ii < tasks; ii++) {
        try {
            threads[ii].join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

2.2 Explicación del código

2.2.1 Creación de los phasers

```
Phaser[] phs = new Phaser[tasks];
for (int i = 0; i < tasks; i++) {
    phs[i] = new Phaser(1);
}
```

Se crea un `Phaser` por cada hilo. Cada phaser inicia con una sola "party" registrada: el propio hilo asociado. Esto permite que cada hilo tenga su propio punto de sincronización, el cual podrá ser observado por sus vecinos.

2.2.2 Creación y lanzamiento de hilos

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

Se crea un arreglo de hilos, uno para cada tarea.
Cada hilo recibe un segmento del arreglo global y ejecuta el cálculo en paralelo.

2.2.3 Variables privadas por hilo

```
double[] threadPrivateMyVal = myVal;
double[] threadPrivateMyNew = myNew;
```

Se mantienen referencias locales (no copias) a los arreglos. Aunque todos los hilos comparten los mismos arreglos, cada uno escribe únicamente en su propio segmento definido por `left` y `right`.

2.2.4 Lectura de la fase actual

```
int phase = phs[i].getPhase();
```

Antes de anunciar que terminó su iteración, el hilo registra la fase actual. Esta fase será utilizada para verificar que los vecinos hayan avanzado al menos hasta ese punto.

2.2.5 Cálculo local del promedio iterativo

```
final int left  = i * (n / tasks) + 1;
final int right = (i + 1) * (n / tasks);

for (int j = left; j <= right; j++) {
    threadPrivateMyNew[j] = (threadPrivateMyVal[j - 1]
                           + threadPrivateMyVal[j + 1]) / 2.0;
}
```

Cada hilo calcula el promedio únicamente para su segmento del arreglo global.

2.2.6 Notificación de finalización

```
phs[i].arrive();
```

El hilo anuncia que ha completado la iteración correspondiente.

2.2.7 Sincronización con vecinos

```
if (i > 0) {
    phs[i - 1].awaitAdvance(phase);
}
if (i < tasks - 1) {
    phs[i + 1].awaitAdvance(phase);
}
```

Cada hilo sólo espera a que sus vecinos inmediatos avancen a una fase superior a `phase`. Esto evita una barrera global y permite que se logre mayor solapamiento entre hilos, siempre preservando las dependencias de datos del promedio iterativo.

2.2.8 Intercambio de arreglos

```
double[] temp = threadPrivateMyNew;
threadPrivateMyNew = threadPrivateMyVal;
threadPrivateMyVal = temp;
```

Al final de cada iteración se realiza el intercambio clásico de roles entre `myVal` y `myNew` para el siguiente paso.

2.2.9 Unión de hilos

```
threads[ii].join();
```

El hilo principal espera a que todos los hilos terminen antes de retornar.

3 Resultados

Durante la ejecución del conjunto de pruebas, se presenta el error:

`co.edu.unal.paralela.OneDimAveragingPhaserTest#testFuzzyBarrier AssertionFailedError`

Este fallo no indica un error de *correctitud numérica*, sino un problema relacionado con el **rendimiento relativo** entre las implementaciones paralelas evaluadas.

3.1 Naturaleza de la prueba

La prueba `testFuzzyBarrier` compara el tiempo de ejecución de dos métodos:

1. `runParallelBarrier`: utiliza una barrera global (`arriveAndAwaitAdvance`), donde todos los hilos deben sincronizarse en cada iteración.
2. `runParallelFuzzyBarrier`: utiliza una "barrera difusa" (`arrive` + `awaitAdvance`), donde cada hilo espera sólo a sus vecinos directos, buscando mayor solapamiento de cómputo.

El objetivo de la prueba es verificar que la versión con *fuzzy barrier* sea significativamente más rápida. El test exige que:

$$\text{speedup} \geq 1.1.$$

Si esta condición no se cumple, se lanza un `AssertionFailedError`.

3.2 Causa del fallo

El fallo de la prueba significa que:

$$T_{\text{fuzzy}} \geq \frac{T_{\text{barrier}}}{1.1},$$

es decir, que la implementación con *fuzzy barrier* no logró ser al menos un 10% más rápida que la implementación con barrera global.

Este comportamiento suele deberse a uno o varios de los siguientes factores:

1. **Sincronización insuficientemente granular.** Aunque el método busca sincronizarse solo con los vecinos, un uso incorrecto de `awaitAdvance` (por ejemplo, esperando una fase errónea) provoca que los hilos se bloqueen más de lo esperado, recreando en la práctica una barrera global encubierta.
2. **Sobrecarga excesiva por creación y planificación de hilos.** La prueba utiliza $\texttt{tasks} = 16 \times \texttt{nCores}$. Este número puede ser tan alto que el costo del *thread scheduling* supere cualquier ganancia de solapamiento.
3. **Desbalance de carga entre hilos.** La partición del arreglo mediante $n/\texttt{tasks}$ puede dejar segmentos muy pequeños, haciendo que la sobrecarga de sincronización domine sobre el tiempo de cómputo útil.
4. **Falta de paralelismo efectivo.** La estructura de dependencias del problema (promedio iterativo) obliga a que los bordes de cada bloque dependan siempre del progreso de los bloques vecinos. Si la sincronización no está optimizada, el *fuzzy barrier* puede comportarse prácticamente igual que la barrera global.
5. **Contención en la memoria.** Todos los hilos escriben sobre segmentos contiguos de dos arreglos compartidos. Para un número grande de hilos, esto puede generar intensa competencia en la jerarquía de caché, reduciendo el rendimiento.