Proyecto Final - Fundamentos de programación funcional y concurrente

Carlos Andres Delgado S
carlos.andres.delgado@correounivalle.edu.co

Noviembre 2024

El presente proyecto tiene por objeto observar el logro de los siguientes resultados de aprendizaje del curso por parte de los estudiantes:

- Utiliza un lenguaje de programación funcional y/o concurrente, usando las técnicas adecuadas, para implementar soluciones a un problema dado.
- Aplica conceptos fundamentales de la programación funcional y concurrente, utilizando un lenguaje de programación adecuado como SCALA, para analizar un problema, modelar, diseñar y desarrollar su solución.
- Construye argumentaciones formalmente correctas, usando las técnicas de argumentación apropiadas, para sustentar la corrección de programas funcionales y para sustentar la mejora en el desempeño de programas concurrentes frente a las soluciones secuenciales.
- Trabaja en equipo, desempeñando un rol específico y llevando a cabo un conjunto de actividades, usando mecanismos de comunicación efectivos con sus compañeros y el profesor, para desarrollar un proyecto de curso.
- Desarrolla un programa funcional concurrente utilizando un lenguaje de programación adecuado como SCALA para resolver en grupo un proyecto de programación planteado por el profesor.
- Escribe un informe de proyecto, presentando los aspectos más relevantes del desarrollo realizado, para que un lector pueda evaluar el proyecto.
- Desarrolla una presentación digital, con los aspectos más relevantes del desarrollo realizado, para sustentar el trabajo ante los compañeros y el profesor.
- Desarrolla programas funcionales puros con estructuras de datos inmutables utilizando recursión, reconocimiento de patrones, mecanismos de encapsulación, funciones de alto orden e iteradores para resolver problemas de programación.
- Combina la programación funcional con la POO, entendiendo las limitaciones y ventajas de cada enfoque para resolver problemas de programación.
- Razona sobre la estructura de programas funcionales utilizando la inducción como mecanismo de argumentación para demostrar propiedades de los programas que construye.

- Paraleliza algoritmos escritos en estilo funcional usando técnicas de paralelización de tareas y datos para lograr acelerar sus tiempos de ejecución.
- Razona sobre programas con paralelismo en datos y tareas en un contexto funcional, concluyendo sobre las ganancias en tiempo con respecto a las versiones secuenciales.
- Aplica técnicas de análisis de desempeño de programas paralelos a los programas funcionales paralelizados para concluir sobre el grado de aceleración de los tiempos de ejecución con respecto a las versiones secuenciales.

1. El problema del riego óptimo

Como todos lo saben, el Valle del Cauca es una región agrícola en general, y cañera en particular. Los cultivos de caña son inmensamente grandes; a veces se pueden recorrer kilómetros y kilómetros de carretera sin dejar de ver a lado y lado cultivos de caña de azúcar.

Los cultivos se organizan normalmente en fincas, y cada una de éstas en tablones. Un tablón es la unidad básica de estructuración de los cultivos de caña. Cada tablón tiene asociadas características como:

- Tamaño del tablón.
- Días desde que inició el cultivo.
- Etapa actual del cultivo.
- Capacidad hídrica del tablón, medida en términos de la cantidad de agua almacenada.

Una característica importante es el tiempo de supervivencia, que indica el número máximo de días que un tablón puede permanecer sin ser regado sin que el cultivo sufra. Cada tablón también tiene asociado un tiempo de regado, el número de días necesarios para restablecer su capacidad hídrica. Finalmente, cada tablón tiene una prioridad asignada entre 1 y 4, donde 4 representa la prioridad más alta.

1.1. Programación del riego

Debido a los altos costos y la naturaleza del trabajo, es imposible tener un sistema fijo de riego para cada tablón. En cambio, se utilizan sistemas móviles de riego, que pueden movilizarse de un tablón a otro con un costo de movilidad menor al costo de tener sistemas fijos en cada tablón.

El objetivo de este proyecto es calcular las fechas de inicio de riego para cada tablón de una finca, minimizando tanto el tiempo de sufrimiento de los cultivos por falta de agua como los costos de movilidad del sistema móvil de riego. Se asume que:

- Se dispone de un solo sistema de riego para toda la finca.
- El consumo de agua es proporcional al tiempo que lleva el tablón sin ser regado y a su capacidad hídrica inicial.

1.2. Formalización

Una finca F es una secuencia de tablones $F = \langle T_0, \ldots, T_{n-1} \rangle$. Cada T_i es una tupla $\langle ts_i^F, tr_i^F, p_i^F \rangle$, donde:

- ts_i^F: tiempo de supervivencia.
- tr_i^F: tiempo de regado.
- p_i^F: prioridad del tablón i de la finca F, con $0 \leq i < n$.

Una programación de riego de la finca F es una permutación $\Pi = \langle \pi_0, \pi_1, \ldots, \pi_{n-1} \rangle$ de $\{0, 1, 2, \ldots, n-1\}$, que indica el orden en el que se regarán los tablones. Primero el tablón T_{π_0}, luego T_{π_1}, y así sucesivamente hasta $T_{\pi_{n-1}}$.

1.2.1. Cálculo del tiempo de inicio de riego

Dada una programación de riego Π, se puede calcular, para cada tablón T_i, el tiempo en que se iniciará su riego t_i^Π según:

$$\begin{aligned} t_{\pi_0}^\Pi &= 0, \\ t_{\pi_j}^\Pi &= t_{\pi_{j-1}}^\Pi + tr_{\pi_{j-1}}^F, \quad j = 1, \ldots, n-1. \end{aligned}$$

1.2.2. Costo de riego de un tablón

El costo de riego de un tablón T_i de la finca F, dada una programación de riego Π, se define como:

$$CR_F^\Pi[i] = \begin{cases} ts_i^F - (t_i^\Pi + tr_i^F), & \text{si } ts_i^F - tr_i^F \geq t_i^\Pi, \\ p_i^F \cdot ((t_i^\Pi + tr_i^F) - ts_i^F), & \text{de lo contrario.} \end{cases}$$

1.2.3. Costo total de riego

El costo total de riego de la finca F, dada una programación de riego Π, es:

$$CR_F^\Pi = \sum_{i=0}^{n-1} CR_F^\Pi[i].$$

1.2.4. Costo de movilidad

El costo de movilidad del sistema móvil de riego está dado por una matriz de distancias D_F, donde $D_F[i, j]$ representa el costo de mover el sistema móvil de T_i a T_j. Este costo es proporcional a la distancia entre los tablones. Para este ejercicio, se asume que $D_F[i, j] = D_F[j, i]$ y $D_F[i, i] = 0$.

El costo de movilidad para una programación Π se define como:

$$CM_F^\Pi = \sum_{j=0}^{n-2} D_F[\pi_j, \pi_{j+1}].$$

1.2.5. Problema del riego óptimo

El problema del riego óptimo se define como:

- **Entrada:** Una finca F y una matriz D_F.
- **Salida:** Una programación Π tal que $CR_F^{\Pi} + CM_F^{\Pi}$ sea mínimo.

1.3. ¿Entendimos el problema?

1.3.1. Ejemplo 1

Entrada:

$$F_1 = \langle\langle 10, 3, 4\rangle, \langle 5, 3, 3\rangle, \langle 2, 2, 1\rangle, \langle 8, 1, 1\rangle, \langle 6, 4, 2\rangle\rangle$$

$$D_{F_1} = \begin{bmatrix} 0 & 2 & 2 & 4 & 4 \\ 2 & 0 & 4 & 2 & 6 \\ 2 & 4 & 0 & 2 & 2 \\ 4 & 2 & 2 & 0 & 4 \\ 4 & 6 & 2 & 4 & 0 \end{bmatrix}$$

Programación 1: $\Pi_1 = \langle 0, 1, 4, 2, 3\rangle$

- $t^{\Pi_1} = \langle 0, 3, 10, 12, 6\rangle$
- $CR_{F_1}^{\Pi_1} = 7 + 1 \cdot 3 + 10 \cdot 1 + 5 \cdot 1 + 4 \cdot 2 = 33$
- $CM_{F_1}^{\Pi_1} = 2 + 6 + 2 + 2 = 12$
- $CR_{F_1}^{\Pi_1} + CM_{F_1}^{\Pi_1} = 33 + 12 = 45$

Programación 2: $\Pi_2 = \langle 2, 1, 4, 3, 0\rangle$

- $t^{\Pi_2} = \langle 10, 2, 0, 9, 5\rangle$
- $CR_{F_1}^{\Pi_2} = 3 \cdot 4 + 0 + 0 + 2 \cdot 1 + 3 \cdot 2 = 20$
- $CM_{F_1}^{\Pi_2} = 4 + 6 + 4 + 4 = 18$
- $CR_{F_1}^{\Pi_2} + CM_{F_1}^{\Pi_2} = 20 + 18 = 38$

En este caso, Π_2 es mejor que Π_1. ¿Habrá mejores?

1.3.2. Ejemplo 2

Entrada:

$$F_2 = \langle\langle 9, 3, 4\rangle, \langle 5, 3, 3\rangle, \langle 2, 2, 1\rangle, \langle 8, 1, 1\rangle, \langle 6, 4, 2\rangle\rangle$$

$$D_{F_2} = \begin{bmatrix} 0 & 2 & 2 & 4 & 4 \\ 2 & 0 & 4 & 2 & 6 \\ 2 & 4 & 0 & 2 & 2 \\ 4 & 2 & 2 & 0 & 4 \\ 4 & 6 & 2 & 4 & 0 \end{bmatrix}$$

Programación 1: $\Pi_1 = \langle 2, 1, 4, 3, 0\rangle$

- $t^{\Pi_1} = \langle 10, 2, 0, 9, 5 \rangle$
- $CR_{F_2}^{\Pi_1} = 4 \cdot 4 + 0 + 0 + 2 \cdot 1 + 3 \cdot 2 = 24$
- $CM_{F_2}^{\Pi_1} = 2 + 6 + 2 + 2 = 12$
- $CR_{F_2}^{\Pi_1} + CM_{F_2}^{\Pi_1} = 24 + 12 = 36$

Programación 2: $\Pi_2 = \langle 2, 1, 4, 0, 3 \rangle$

- $t^{\Pi_2} = \langle 9, 2, 0, 12, 5 \rangle$
- $CR_{F_2}^{\Pi_2} = 3 \cdot 4 + 0 + 0 + 5 \cdot 1 + 3 \cdot 2 = 23$
- $CM_{F_2}^{\Pi_2} = 4 + 6 + 4 + 4 = 18$
- $CR_{F_2}^{\Pi_2} + CM_{F_2}^{\Pi_2} = 23 + 18 = 41$

En este caso, Π_1 es mejor que Π_2. ¿Habrá mejores?

2. Implementando una solución funcional al problema

Para implementar una solución funcional a este problema, se definen los siguientes tipos de datos:

```
// Un tablon es una tripleta con el tiempo de supervivencia,
// el tiempo de riego y la prioridad del tablon
type Tablon = (Int, Int, Int)

// Una finca es un vector de tablones
type Finca = Vector[Tablon]
// Si f : Finca, f(i) = (tsi, tri, pi)

// La distancia entre dos tablones se representa por
// una matriz
type Distancia = Vector[Vector[Int]]

// Una programación de riego es un vector que asocia
// cada tablon i con su turno de riego (0 es el primer turno,
// n-1 es el último turno)
type ProgRiego = Vector[Int]
// Si v : ProgRiego, y v.length == n, v es una permutación
// de {0, ..., n-1} v(i) es el turno de riego del tablon i
// para 0 <= i < n

// El tiempo de inicio de riego es un vector que asocia
// cada tablon i con el momento del tiempo en que se riega
type TiempoInicioRiego = Vector[Int]
// Si t : TiempoInicioRiego y t.length == n, t(i) es la hora a
// la que inicia a regarse el tablon i
```

2.1. Generación de entradas aleatorias

Para generar entradas al azar para el problema, se definen las siguientes funciones:

```
val random = new Random()

def fincaAlAzar(long: Int): Finca = {
  // Crea una finca de long tablones,
  // con valores aleatorios entre 1 y long * 2 para el tiempo
  // de supervivencia, entre 1 y long para el tiempo
  // de regado y entre 1 y 4 para la prioridad
  val v = Vector.fill(long)(
    (random.nextInt(long * 2) + 1,
     random.nextInt(long) + 1,
     random.nextInt(4) + 1)
  )
  v
}

def distanciaAlAzar(long: Int): Distancia = {
  // Crea una matriz de distancias para una finca
  // de long tablones, con valores aleatorios entre
  // 1 y long * 3
  val v = Vector.fill(long, long)(random.nextInt(long * 3) + 1)
  Vector.tabulate(long, long)((i, j) =>
    if (i < j) v(i)(j)
    else if (i == j) 0
    else v(j)(i))
}
```

2.2. Exploración de entradas

Para explorar las entradas generadas, se definen las siguientes funciones:

```
def tsup(f: Finca, i: Int): Int = {
  f(i)._1
}

def treg(f: Finca, i: Int): Int = {
  f(i)._2
}

def prio(f: Finca, i: Int): Int = {
  f(i)._3
}
```

2.3. Calculando el tiempo de inicio de riego

Implemente una función `tIR` que reciba de entrada una finca f y una programación de riego específica π, y devuelva el tiempo de inicio de riego de cada tablón de la finca f según π:

```
def tIR(f: Finca, pi: ProgRiego): TiempoInicioRiego = {
  // Dada una finca f y una programación de riego pi,
  // y f.length == n, tIR(f, pi) devuelve t: TiempoInicioRiego
  // tal que t(i) es el tiempo en que inicia el riego del
  // tablon i de la finca f según pi
  val tiempos = Array.fill(f.length)(0)
  for (j <- 1 until pi.length) {
    val prevTablon = pi(j - 1)
    val currTablon = pi(j)
    tiempos(currTablon) = tiempos(prevTablon) + treg(f, prevTablon)
  }
  tiempos.toVector
}
```

2.4. Calculando costos

Implemente las siguientes funciones para calcular los costos relacionados con el riego:

- `costoRiegoTablon`: Recibe i el identificador de un tablón, la finca f y una programación de riego específica π, y devuelve el costo de regar el tablón i de la finca f con la programación π.
- `costoRiegoFinca`: Recibe una finca f y una programación de riego específica π, y devuelve el costo de regar la finca f con la programación π.
- `costoMovilidad`: Recibe una finca f, una programación de riego específica π, y una matriz de distancias d, y devuelve el costo asociado a la movilidad del sistema de riego de la finca f según la matriz d.

```
def costoRiegoTablon(i: Int, f: Finca, pi: ProgRiego): Int = {
  val tiempoInicio = tIR(f, pi)(i)
  val tiempoFinal = tiempoInicio + treg(f, i)
  if (tsup(f, i) - treg(f, i) >= tiempoInicio) {
    tsup(f, i) - tiempoFinal
  } else {
    prio(f, i) * (tiempoFinal - tsup(f, i))
  }
}

def costoRiegoFinca(f: Finca, pi: ProgRiego): Int = {
  (0 until f.length).map(i => costoRiegoTablon(i, f, pi)).sum
}

def costoMovilidad(f: Finca, pi: ProgRiego, d: Distancia): Int = {
  (0 until pi.length - 1).map(j => d(pi(j))(pi(j + 1))).sum
}
```

2.5. Generando programaciones de riego

Implemente una función `generarProgramacionesRiego`, que dada una finca f, genere todas las posibles programaciones de riego y las devuelva en un vector.

```
def generarProgramacionesRiego(f: Finca): Vector[ProgRiego] = {
  // Dada una finca de n tablones, devuelve todas las
  // posibles programaciones de riego de la finca
  val indices = (0 until f.length).toVector
  indices.permutations.toVector
}
```

2.6. Calculando una programación de riego óptima

Implemente una función `ProgramacionRiegoOptimo`, que dada una finca f y una matriz de distancias d, devuelva una programación de riego óptima. Para resolver este punto, utilice la función `generarProgramacionesRiego` del punto anterior.

```
def ProgramacionRiegoOptimo(f: Finca, d: Distancia): (ProgRiego, Int) = {
  // Dada una finca devuelve la programación
  // de riego óptima
  val programaciones = generarProgramacionesRiego(f)
  val costos = programaciones.map(pi =>
    (pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d))
  )
  costos.minBy(_._2)
}
```

3. Acelerando los cálculos con paralelismo de tareas y de datos

Una vez terminada la etapa donde se han programado las versiones secuenciales de las funciones `tIR`, `costoRiegoTablon`, `costoRiegoFinca`, `costoMovilidad`, `generarProgramacionesRiego`, y `ProgramacionRiegoOptimo`, se implementan las versiones paralelas de algunas de ellas para mejorar los tiempos de cálculo.

3.1. Paralelizando el cálculo de los costos de riego y de movilidad

Implemente las funciones `costoRiegoFincaPar` y `costoMovilidadPar` que paralelicen los cálculos de las respectivas funciones secuenciales.

```
def costoRiegoFincaPar(f: Finca, pi: ProgRiego): Int = {
  // Devuelve el costo total de regar una finca f dada una
  // programación de riego pi, calculando en paralelo
  (0 until f.length).par.map(i => costoRiegoTablon(i, f, pi)).sum
}

def costoMovilidadPar(f: Finca, pi: ProgRiego, d: Distancia): Int = {
  // Calcula el costo de movilidad de manera paralela
```

```
  (0 until pi.length - 1).par.map(j => d(pi(j))(pi(j + 1))).sum
}
```

3.2. Paralelizando la generación de programaciones de riego

Implemente la función `generarProgramacionesRiegoPar`, que paralelice el cálculo de las programaciones posibles.

```
def generarProgramacionesRiegoPar(f: Finca): Vector[ProgRiego] = {
  // Genera las programaciones posibles de manera paralela
  val indices = (0 until f.length).toVector
  indices.permutations.toVector.par.toVector
}
```

3.3. Paralelizando la programación de riego óptima

Implemente la función `ProgramacionRiegoOptimoPar`, que paralelice el cálculo de la programación óptima.

```
def ProgramacionRiegoOptimoPar(f: Finca, d: Distancia): (ProgRiego, Int)
    ↪ = {
  // Dada una finca, calcula la programación óptima de riego
  val programaciones = generarProgramacionesRiegoPar(f)
  val costos = programaciones.par.map(pi =>
    (pi, costoRiegoFincaPar(f, pi) + costoMovilidadPar(f, pi, d))
  )
  costos.minBy(_._2)
}
```

3.4. Produciendo datos para hacer la evaluación comparativa

Un punto esencial en estos proyectos de paralelización es realizar el análisis comparativo y concluir en qué casos es beneficioso usar la paralelización y en qué casos no. Para ello, es importante generar tablas con los tiempos que toman las dos versiones (secuencial y paralela) para:

- El cálculo de los costos de riego y de movilidad (`costoRiegoFinca`, `costoRiegoFincaPar`, `costoMovilidad`, y `costoMovilidadPar`).
- La generación de programaciones de riego (`generarProgramacionesRiego` y `generarProgramacione`
- La programación de riego óptima (`ProgramacionRiegoOptimo` y `ProgramacionRiegoOptimoPar`).

Tamaño de las fincas: Determine los tamaños de las fincas (en número de tablones) para los que vale la pena utilizar las versiones paralelas.

3.4.1. Generación de datos

Para generar los datos necesarios para el análisis, se recomienda usar la biblioteca `org.scalameter` y expresiones `for` para medir los tiempos de ejecución.

```
import org.scalameter._

val timeSeq = measure {
  ProgramacionRiegoOptimo(finca, distancia)
}

val timePar = measure {
  ProgramacionRiegoOptimoPar(finca, distancia)
}

println(s"Secuencial: $timeSeq ms")
println(s"Paralelo: $timePar ms")
```

3.4.2. Presentación de resultados

Presente los resultados obtenidos en tablas como esta:

Tamaño de la finca (tablones)	Versión secuencial (ms)	Versión paralela (ms)	Aceleración (%)
10	120	80	33,33
20	500	300	40,00
30	1200	700	41,67

3.4.3. Análisis de los resultados

Analice los resultados obtenidos:

- Identifique los tamaños de fincas donde el paralelismo genera ganancias significativas.
- Evalúe si las versiones paralelas introducen sobrecarga en casos pequeños.
- Concluya sobre los beneficios del paralelismo para diferentes escenarios.

3.5. Produciendo datos para hacer la evaluación comparativa

Un punto esencial en estos proyectos de paralelización es realizar el análisis comparativo y concluir en qué casos es beneficioso usar la paralelización y en qué casos no. Para ello, es importante generar tablas con los tiempos que toman las dos versiones (secuencial y paralela) para:

- El cálculo de los costos de riego y de movilidad (`costoRiegoFinca`, `costoRiegoFincaPar`, `costoMovilidad`, y `costoMovilidadPar`).
- La generación de programaciones de riego (`generarProgramacionesRiego` y `generarProgramacione`
- La programación de riego óptima (`ProgramacionRiegoOptimo` y `ProgramacionRiegoOptimoPar`).

Tamaño de las fincas: Determine los tamaños de las fincas (en número de tablones) para los que vale la pena utilizar las versiones paralelas.

3.5.1. Generación de datos

Para generar los datos necesarios para el análisis, se recomienda usar la biblioteca `org.scalameter` y expresiones `for` para medir los tiempos de ejecución.

```
import org.scalameter._

val timeSeq = measure {
  ProgramacionRiegoOptimo(finca, distancia)
}

val timePar = measure {
  ProgramacionRiegoOptimoPar(finca, distancia)
}

println(s"Secuencial: $timeSeq ms")
println(s"Paralelo: $timePar ms")
```

3.5.2. Presentación de resultados

Presente los resultados obtenidos en tablas como esta:

Tamaño de la finca (tablones)	Versión secuencial (ms)	Versión paralela (ms)	Aceleración (%)
10	120	80	33,33
20	500	300	40,00
30	1200	700	41,67

3.5.3. Análisis de los resultados

Analice los resultados obtenidos:

- Identifique los tamaños de fincas donde el paralelismo genera ganancias significativas.
- Evalúe si las versiones paralelas introducen sobrecarga en casos pequeños.
- Concluya sobre los beneficios del paralelismo para diferentes escenarios.

4. Entrega

4.1. Informe del proyecto - secciones

Todo proyecto debe venir acompañado de un informe en formato PDF. El informe debe contener la información explícita solicitada en el enunciado.

Para este caso, el informe del proyecto debe contener al menos cuatro secciones: informe de procesos, informe de corrección, informe de paralelización y conclusiones.

No olvide colocar los nombres completos y códigos de todos los integrantes del grupo.

4.1.1. Informe de procesos

Tal como se ha visto en clase, los procesos generados por los programas recursivos, genere un ejemplo para cada ejercicio, y muestre cómo se comporta el proceso generado por cada uno de los programas. Para ello, debe mostrar la pila de llamadas que se genera en cada caso, y cómo se va desplegando la pila de llamadas a medida que se resuelve el problema. Use ejemplos con valores pequeños.

4.1.2. Informe de parelización

Debe explicar que estrategia utilizó para parelelizar y determinar las ganancias de acuerdo a la ley de Ahmdal. Debe correr al menos 10 pruebas para para 5 tamaños de entrada diferentes, adjunte capturas de pantalla cómo evidencia y explique los resultados con sus propias palabras.

4.1.3. Informe de corrección

Es muy importante reflexionar sobre la corrección del código entregado. Para ello se deberá argumentar sobre la corrección de los programas entregados, y también deberá entregar un conjunto de pruebas. Todo esto lo consigna en esta sección del informe, dividida de la siguiente manera:

Argumentación sobre la corrección Para cada función, `pertenece`, `grande`, `complemento`, `union`, `interseccion`, `inclusion` e `igualdad`, argumente si la implementación es correcta o no. Use notación matemática para argumentar la corrección de los algoritmos.

Casos de prueba Para cada función se requieren mínimo 5 casos de prueba donde se conozca claramente el valor esperado, y se pueda evidenciar que el valor calculado por la función corresponde con el valor esperado en toda ocasión. Agreguelos como pruebas de software. Estas pruebas debe ser disponibles en el paquete proyecto, de test.

4.2. Condiciones de entrega

La fecha de entrega máxima es lunes 16 de Diciembre a las 23:59:59, se aplicará una penalización de 0.15 por cada hora o fracción de retraso. Por ejemplo, si se entrega a partir a las 00:00:01, la nota máxima será de 4.85, si se entrega a partir las 01:00:00, la nota máxima será de 4.70, y así sucesivamente. La inscripción de grupos estará disponible hasta el 28 de Octubre de 2024 a las 23:59:59, los grupos son de máximo 4 personas.

Las condiciones de entrega son:

1. Los estudiantes deben inscribirse a un grupo en el campus virtual, no hacerlo implica 0.0 en la nota del proyecto.

2. Debe hacerse un fork del repositorio: `https://github.com/cardel/plantilla-funcional`, en caso de que lo haya trabajado cree una nueva rama con nombre **proyecto3**, así mismo agregar como colaboradores a sus compañeros.

3. Debe entregar el enlace de su repositorio en el Campus Virtual antes del cierre indicado por el docente

4. Las fecha del último commit debe ser antes del cierre indicado por el docente

5. Incluya el informe en la raíz del repositorio en formato PDF

6. El docente verificará con herramientas antiplagio la originalidad de los proyectoes entregados, así mismo se tomará en cuenta la interacción de commits dentro del repositorio de github y su trazabilidad (flujo de trabajo).

7. El proyecto debe sustentarse la nota del proyecto será individual y será entre 0 y 1, la cual se multiplica por la nota grupal. Por ejemplo, si usted obtiene 0.5 y la nota de su grupo es 5.0, su nota de proyecto será 2.5.

8. No tocar los archivos relacionados con el Gradle, workflows en Github y el check, sólo lo relacionado a archivos de solución del problema y pruebas, es causal de rebaja de nota si se detecta que se han tocado estos archivos para dar la impresión que la entrega es funcional.

5. Rubrica de evaluación

5.1. Regla del proyecto

Importante: Debe utilizar el paquete common visto en clase para la parelización o pararelización de datos con la directiva `par`, no se aceptan soluciones con otros paquetes.

Criterio	**No cumple (0 puntos)**	**Cumple (10 puntos)**
Informe en formato PDF y disponible en la raíz del proyecto		
El proyecto es un fork del indicado por el docente		
El informe tiene los nombres completos y códigos de los estudiantes		

En total se pueden obtener 30 puntos en la regla del proyecto.

5.2. Rubrica de evaluación puntos

Importante El proyecto debe entregarse en su totalidad, en caso de que no se entregue algún punto o este no es funcional, máximo se asignará nivel 1 en la rubrica de evaluación.

Criterio	**Nivel 0 (0 puntos)**	**Nivel 1 (5 puntos)**	**Nivel 2 (10 puntos)**	**Nivel 3 (15 puntos)**
Solución del problema funcional	No se entrega o es no funcional.	La implementación no es correcta y produce resultados erróneos	La implementación es correcta pero no se sigue el enfoque indicado en el enunciado	La implementación es correcta y se resuelve como se indica en el enunciado

Criterio	**Nivel 0 (0 puntos)**	**Nivel 1 (5 puntos)**	**Nivel 2 (10 puntos)**	**Nivel 3 (15 puntos)**
Solución del problema paralela	No se entrega o no es funcional o usar una librería diferente a parallel o task	La implementación no utiliza una buena estrategia de paralelización	La implementación usa una buena estrategia de paralelización	
Informe de proceso	No se entrega o es no funcional.	No se explica correctamente el enfoque de funciones de alto orden en la solución del problema	Se explica el enfoque de funciones de alto orden en la solución del problema, pero la explicación carece de profundidad y análisis de la solución presentada	Se explica el enfoque de funciones de alto orden en la solución del problema, es profunda, utiliza los conceptos vistos en clase y se analiza correctamente la solución presentada
Informe de corrección	No se entrega o es no funcional.	Argumenta la corrección de los algoritmos de forma parcial; la notación matemática tiene errores menores.	Argumenta claramente la corrección de los algoritmos; la notación matemática es en su mayoría correcta.	Argumenta claramente utilizando notación matemática para demostrar que los algoritmos implementados son correctos; muestra cómo son los llamados. La notación es correcta y clara.

Criterio	**Nivel 0 (0 puntos)**	**Nivel 1 (5 puntos)**	**Nivel 2 (10 puntos)**	**Nivel 3 (15 puntos)**
Informe de paralelización	No se entrega o es no funcional.	No explica claramente la estrategia de paralelización o no evidencia el benchmarking con capturas de pantalla.	Explica claramente la estrategia de paralelización y evidencia el benchmarking con capturas de pantalla, pero los tamaños de entrada no son significativos, es decir no se diferencian entre sí y no permiten estudiar la parelización correctamente.	Explica claramente la estrategia de paralelización y evidencia el benchmarking con capturas de pantalla, pero los tamaños de entrada son significativos, es decir se diferencian entre sí y no permiten estudiar la parelización correctamente.
Casos de prueba*	No se entrega o es no funcional.	Incluye algunos casos de prueba, pero no son significativos o no se ejecutan al construir el proyecto.	Incluye 5 ejemplos para algunos puntos del proyecto; se ejecutan al construir el proyecto y son casos mayormente significativos.	Incluye en el código 5 ejemplos para cada punto del proyecto como pruebas de software; estas se ejecutan al construir el proyecto y son casos significativos (no triviales) de la solución del problema.

* Los casos de prueba deben ser pruebas de software dentro de la carpeta test, estas se deben ejecutar dentro de las rutinas de Gradle.

En total se pueden obtener 60 puntos en la rubrica de evaluación.

5.3. Calificación

La calificación final del proyecto será la suma de los puntos obtenidos en la regla del proyecto y la rubrica de evaluación. La calificación máxima es de 105 puntos.

La nota grupal N, se pondera entre 0 y 5, de acuerdo a los puntos P obtenidos de la siguiente manera:

$$\nu = \frac{P}{105} \times 5$$

Tener en cuenta que esta es la nota grupal, y que la nota individual en caso de requerir sustentación se multiplica por esta nota grupal.