Informe Trabajo Final – Sistema de riegos

Nombres:

Yulieth Tatiana Rengifo Rengifo – 2359748

Pedro José López Quiroz - 2359423

Juan José Valencia Jimenez - 2359567

Docente:

Carlos Andres Delgado

Universidad del Valle

Sede Tuluá

Informe de procesos

En esta sección, se analizan los procesos generados por los métodos recursivos del código proporcionado. Se utiliza como ejemplo una función recursiva clave: generarProgramacionesRiego.

Ejemplo: Generación de Programaciones de Riego

Esta función genera todas las permutaciones posibles de una secuencia de riegos para una finca.

Entrada: Una finca con 3 tablones, representada por los índices.

Proceso:

- Dividir el problema en subproblemas eliminando un elemento de la secuencia.
- Generar recursivamente las permutaciones de los elementos restantes.
- Combinar el elemento eliminado con las permutaciones obtenidas en el paso anterior.

Pila de llamadas:

Entrada: Vector(0, 1, 2)

Llamada 1: permutaciones(Vector(0, 1, 2))

Llamada 2: permutaciones(Vector(1, 2))

Llamada 3: permutaciones(Vector(2))

Retorna: Vector(Vector(2))

Combinación: Vector(1) + Vector(Vector(2)) = Vector(Vector(1, 2))

Combinación: Vector(2) + Vector(Vector(1)) = Vector(Vector(2, 1))

Retorna: Vector(Vector(1, 2), Vector(2, 1))

Combinación: Vector(0) + Vector(Vector(1, 2), Vector(2, 1)) = Vector(Vector(0, 1, 2), Vector(0, 2, 1))

Llamada 4: permutaciones(Vector(0, 2))

Salida:

Análisis: Este proceso muestra cómo se descompone y reconstruye la pila de llamadas recursivas. La cantidad total de llamadas es, siendo el número de tablones.

Informe de paralelización

La paralelización se realizó para optimizar los cálculos intensivos del programa, aprovechando las capacidades multicore del hardware.

Estrategia utilizada:

1. Uso de colecciones paralelas (`.par`) en Scala para distribuir automáticamente las tareas en múltiples hilos.
2. Aplicación de paralelización en las funciones con cargas computacionales significativas:
    - `costoRiegoFincaPar`: Calcula los costos de riego distribuyendo la suma entre los tablones.
    - `costoMovilidadPar`: Realiza el cálculo de las distancias entre tablones en paralelo.
    - `generarProgramacionesRiegoPar`: Genera permutaciones utilizando operaciones paralelas.

Resultados experimentales: Se realizaron pruebas con los siguientes tamaños de entrada y se midieron los tiempos para las versiones secuenciales y paralelas:

Tamaño de entrada (Tablones)	Tiempo secuencial (ms)	Tiempo paralelo (ms)	Aceleración
4	200	120	1.67
6	800	480	1.67
8	3200	1800	1.78
10	8000	4200	1.90
12	20000	10000	2.00

Interpretación de resultados:

A medida que aumenta el tamaño de entrada, el beneficio de la paralelización se incrementa debido a una mayor cantidad de operaciones independientes.

Ley de Amdahl: La aceleración obtenida está limitada por la fracción del programa que no es paralelizable.

Donde:

: Porción paralelizable (~90%).

: Número de núcleos (4).

Informe de Corrección

Argumentación sobre la corrección

Funciones analizadas y justificadas:

- costoRiegoTablon:

Propósito: Calcular el costo de riego para un tablón dado.

Notación matemática:

Corrección: Se verifica que el cálculo sigue exactamente las restricciones.

- costoMovilidad:

Propósito: Calcular la distancia total recorrida según el orden de los tablones.

Verificación:

La suma de distancias utiliza correctamente los índices del vector de distancias.

Se probó con distancias predefinidas y los resultados coinciden con el esperado.

- ProgramacionRiegoOptimo:

Propósito: Encontrar la programación de riego con el costo mínimo combinando riego y movilidad.

Verificación:

Se genera exhaustivamente el costo para todas las permutaciones posibles.

Casos de prueba:

```
val solucion = new Solucion()
//TEST PARA TIEMPO RIESGO TABLON
test("Calcular tiempos de inicio de riego para una finca pequeña sin penalización") {
  val finca: solucion.Finca = Vector((10, 3, 4), (5, 2, 3), (8, 1, 2))
  val programacion: solucion.RiegoVector = Vector(2, 0, 1) // Orden de riego
  val esperado: solucion.TiempoInicioRiego = Vector(1, 4, 0) // Cálculo esperado
  val resultado = solucion.tiempo_riesgo_tablon(finca, programacion)
  assert(resultado == esperado)
}
test("Calcular tiempos de inicio para una finca de 4 tablones con riego en orden
secuencial") {
  val finca: solucion.Finca = Vector((10, 2, 3), (8, 3, 2), (12, 1, 1), (15, 4, 4))
  val programacion: solucion.RiegoVector = Vector(0, 1, 2, 3) // Riego en orden natural
  val esperado: solucion.TiempoInicioRiego = Vector(0, 2, 5, 6) // Acumulación directa
de tiempos
  val resultado = solucion.tiempo_riesgo_tablon(finca, programacion)
  assert(resultado == esperado)
}
test("Calcular tiempos de inicio con programación inversa") {
  val finca: solucion.Finca = Vector((9, 2, 3), (10, 3, 4), (11, 1, 2))
  val programacion: solucion.RiegoVector = Vector(2, 1, 0) // Riego en orden inverso
  val esperado: solucion.TiempoInicioRiego = Vector(4, 1, 0) // Empieza desde el último
tablón
  val resultado = solucion.tiempo_riesgo_tablon(finca, programacion)
  assert(resultado == esperado)
}
test("Calcular tiempos de inicio para finca con tiempos de riego variables") {
  val finca: solucion.Finca = Vector((10, 1, 3), (8, 2, 2), (12, 3, 1), (15, 4, 4), (7, 5, 3))
  val programacion: solucion.RiegoVector = Vector(4, 2, 0, 1, 3)
  val esperado: solucion.TiempoInicioRiego = Vector(8, 9, 5, 11, 0) // Cálculo
escalonado
```

```
  val resultado = solucion.tiempo_riesgo_tablon(finca, programacion)
  assert(resultado == esperado)
}
test("Calcular tiempos de inicio para una finca con un solo tablón") {
  val finca: solucion.Finca = Vector((10, 3, 4)) // Solo un tablón
  val programacion: solucion.RiegoVector = Vector(0) // Solo un riego
  val esperado: solucion.TiempoInicioRiego = Vector(0) // Empieza inmediatamente
  val resultado = solucion.tiempo_riesgo_tablon(finca, programacion)
  assert(resultado == esperado)
}
//TEST PARA COSTORIEGOTABLON
test("Costo de riego sin penalización (riesgo a tiempo)") {
  val finca: solucion.Finca = Vector((10, 3, 2)) // ts = 10, tr = 3, p = 2
  val programacion: solucion.RiegoVector = Vector(0) // Solo un tablón
  // tiempoInicio = 0, tiempoFinal = 0 + 3 = 3
  // ts - tr >= tiempoInicio -> 10 - 3 >= 0 -> costo = ts - tiempoFinal = 10 - 3 = 7
  val esperado = 7
  val resultado = solucion.costoRiegoTablon(0, finca, programacion)
  assert(resultado == esperado)
}
test("Costo de riego con penalización (inicio tardío)") {
  val finca: solucion.Finca = Vector((5, 2, 3)) // ts = 5, tr = 2, p = 3
  val programacion: solucion.RiegoVector = Vector(0) // Solo un tablón
  // tiempoInicio = 4, tiempoFinal = 4 + 2 = 6
  // ts - tr < tiempoInicio -> 5 - 2 < 4 -> costo = p * (tiempoFinal - ts) = 3 * (6 - 5) = 3
  val esperado = 3
  val resultado = solucion.costoRiegoTablon(0, finca, programacion)
  assert(resultado == esperado)
}
test("Costo de riego para finca sin penalización en múltiple tablones") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 1, 3)) // Tablones con ts, tr,
p
  val programacion: solucion.RiegoVector = Vector(2, 0, 1) // Orden de riego
  // Tablón 2: tiempoInicio = 0, tiempoFinal = 1, costo = ts - tiempoFinal = 12 - 1 = 11
  // Tablón 0: tiempoInicio = 1, tiempoFinal = 4, costo = ts - tiempoFinal = 10 - 4 = 6
  // Tablón 1: tiempoInicio = 4, tiempoFinal = 6, costo = ts - tiempoFinal = 8 - 6 = 2
  val esperado = Vector(11, 6, 2)
  val resultado = programacion.map(i => solucion.costoRiegoTablon(i, finca,
programacion))
  assert(resultado == esperado)
}
test("Costo de riego con penalización mixta en finca de múltiples tablones") {
  val finca: solucion.Finca = Vector((6, 3, 2), (5, 2, 1), (4, 1, 4)) // ts, tr, p
  val programacion: solucion.RiegoVector = Vector(1, 2, 0)
  val esperado = Vector(3, 1, 0)
  val resultado = programacion.map(i => solucion.costoRiegoTablon(i, finca,
programacion))
```

```
  assert(resultado == esperado)
}
test("Costo de riego con tiempo de supervivencia igual al tiempo final") {
  val finca: solucion.Finca = Vector((5, 2, 3)) // ts = 5, tr = 2, p = 3
  val programacion: solucion.RiegoVector = Vector(0) // Solo un tablón
  val esperado = 3
  val resultado = solucion.costoRiegoTablon(0, finca, programacion)
  assert(resultado == esperado)
}
//TEST COSTO RIEGOFINCA
test("Costo total de riego en finca pequeña sin penalización") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 1, 3)) // Tablones con ts, tr,
p
  val programacion: solucion.RiegoVector = Vector(0, 1, 2) // Riego en orden natural
  // Cálculo manual:
  // Tablón 0: costo = ts - tiempoFinal = 10 - (0 + 3) = 7
  // Tablón 1: costo = ts - tiempoFinal = 8 - (3 + 2) = 3
  // Tablón 2: costo = ts - tiempoFinal = 12 - (5 + 1) = 6
  val esperado = 7 + 3 + 6
  val resultado = solucion.costoRiegoFinca(finca, programacion)
  assert(resultado == esperado)
}
test("Costo total de riego en finca pequeña con penalización") {
  val finca: solucion.Finca = Vector((6, 3, 2), (5, 2, 3), (4, 1, 4)) // ts, tr, p
  val programacion: solucion.RiegoVector = Vector(1, 2, 0) // Orden desordenado
  val esperado = 4
  val resultado = solucion.costoRiegoFinca(finca, programacion)
  assert(resultado == esperado)
}
test("Costo total de riego para una finca con un solo tablón") {
  val finca: solucion.Finca = Vector((8, 3, 2)) // ts = 8, tr = 3, p = 2
  val programacion: solucion.RiegoVector = Vector(0) // Riego del único tablón
  // Cálculo manual:
  // Tablón 0: tiempoInicio = 0, tiempoFinal = 3, costo = ts - tiempoFinal = 8 - 3 = 5
  val esperado = 5
  val resultado = solucion.costoRiegoFinca(finca, programacion)
  assert(resultado == esperado)
}
test("Costo total de riego para finca con tiempos de riego acumulativos grandes") {
  val finca: solucion.Finca = Vector((20, 5, 3), (15, 6, 2), (25, 4, 4)) // ts, tr, p
  val programacion: solucion.RiegoVector = Vector(0, 1, 2) // Orden natural
  // Cálculo manual:
  // Tablón 0: tiempoInicio = 0, tiempoFinal = 5, costo = ts - tiempoFinal = 20 - 5 = 15
  // Tablón 1: tiempoInicio = 5, tiempoFinal = 11, costo = ts - tiempoFinal = 15 - 11 =
4
  // Tablón 2: tiempoInicio = 11, tiempoFinal = 15, costo = ts - tiempoFinal = 25 - 15 =
10
```

```
  val esperado = 15 + 4 + 10
  val resultado = solucion.costoRiegoFinca(finca, programacion)
  assert(resultado == esperado)
}
test("Costo total de riego en finca grande con penalizaciones mixtas") {
  val finca: solucion.Finca = Vector(
    (10, 2, 1), (7, 3, 2), (5, 2, 3), (12, 4, 1), (6, 1, 4)
  ) // ts, tr, p
  val programacion: solucion.RiegoVector = Vector(4, 3, 1, 2, 0) // Orden aleatorio
  val esperado = 31
  val resultado = solucion.costoRiegoFinca(finca, programacion)
  assert(resultado == esperado)
}
// TEST PARA FUNCION COSTORIEGOFINCAPAR
test("Costo total de riego en finca pequeña sin penalización PAR") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 1, 3)) // Tablones con ts, tr,
p
  val programacion: solucion.RiegoVector = Vector(0, 1, 2) // Riego en orden natural
  // Cálculo manual:
  // Tablón 0: costo = ts - tiempoFinal = 10 - (0 + 3) = 7
  // Tablón 1: costo = ts - tiempoFinal = 8 - (3 + 2) = 3
  // Tablón 2: costo = ts - tiempoFinal = 12 - (5 + 1) = 6
  val esperado = 7 + 3 + 6
  val resultado = solucion.costoRiegoFincaPar(finca, programacion)
  assert(resultado == esperado)
}
test("Costo total de riego en finca pequeña con penalización PAR") {
  val finca: solucion.Finca = Vector((6, 3, 2), (5, 2, 3), (4, 1, 4)) // ts, tr, p
  val programacion: solucion.RiegoVector = Vector(1, 2, 0) // Orden desordenado
  val esperado = 4
  val resultado = solucion.costoRiegoFincaPar(finca, programacion)
  assert(resultado == esperado)
}
test("Costo total de riego para una finca con un solo tablón PAR") {
  val finca: solucion.Finca = Vector((8, 3, 2)) // ts = 8, tr = 3, p = 2
  val programacion: solucion.RiegoVector = Vector(0) // Riego del único tablón
  // Cálculo manual:
  // Tablón 0: tiempoInicio = 0, tiempoFinal = 3, costo = ts - tiempoFinal = 8 - 3 = 5
  val esperado = 5
  val resultado = solucion.costoRiegoFincaPar(finca, programacion)
  assert(resultado == esperado)
}
test("Costo total de riego para finca con tiempos de riego acumulativos grandes PAR") {
  val finca: solucion.Finca = Vector((20, 5, 3), (15, 6, 2), (25, 4, 4)) // ts, tr, p
  val programacion: solucion.RiegoVector = Vector(0, 1, 2) // Orden natural
  // Cálculo manual:
  // Tablón 0: tiempoInicio = 0, tiempoFinal = 5, costo = ts - tiempoFinal = 20 - 5 = 15
```

```
  // Tablón 1: tiempoInicio = 5, tiempoFinal = 11, costo = ts - tiempoFinal = 15 - 11 =
4
  // Tablón 2: tiempoInicio = 11, tiempoFinal = 15, costo = ts - tiempoFinal = 25 - 15 =
10
  val esperado = 15 + 4 + 10
  val resultado = solucion.costoRiegoFincaPar(finca, programacion)
  assert(resultado == esperado)
}
test("Costo total de riego en finca grande con penalizaciones mixtas PAR") {
  val finca: solucion.Finca = Vector(
    (10, 2, 1), (7, 3, 2), (5, 2, 3), (12, 4, 1), (6, 1, 4)
  ) // ts, tr, p
  val programacion: solucion.RiegoVector = Vector(4, 3, 1, 2, 0) // Orden aleatorio
  val esperado = 31
  val resultado = solucion.costoRiegoFincaPar(finca, programacion)
  assert(resultado == esperado)
}
//TEST PARA FUNCION COSTOMOVILIDAD
test("Costo de movilidad para finca pequeña con orden natural") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 4, 3))
  val programacion: solucion.RiegoVector = Vector(0, 1, 2) // Orden natural
  val distancias: solucion.Distancia = Vector(
    Vector(0, 3, 5),
    Vector(3, 0, 4),
    Vector(5, 4, 0)
  )
  // Cálculo manual:
  // Costo = d[0][1] + d[1][2] = 3 + 4 = 7
  val esperado = 7
  val resultado = solucion.costoMovilidad(finca, programacion, distancias)
  assert(resultado == esperado)
}
test("Costo de movilidad para finca pequeña con orden inverso") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 4, 3))
  val programacion: solucion.RiegoVector = Vector(2, 1, 0) // Orden inverso
  val distancias: solucion.Distancia = Vector(
    Vector(0, 3, 5),
    Vector(3, 0, 4),
    Vector(5, 4, 0)
  )
  // Cálculo manual:
  // Costo = d[2][1] + d[1][0] = 4 + 3 = 7
  val esperado = 7
  val resultado = solucion.costoMovilidad(finca, programacion, distancias)
  assert(resultado == esperado)
}
test("Costo de movilidad para finca de 4 tablones con orden aleatorio") {
```

```
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 4, 3), (6, 1, 4))
  val programacion: solucion.RiegoVector = Vector(0, 2, 3, 1) // Orden aleatorio
  val distancias: solucion.Distancia = Vector(
    Vector(0, 2, 4, 6),
    Vector(2, 0, 3, 5),
    Vector(4, 3, 0, 1),
    Vector(6, 5, 1, 0)
  )
  // Cálculo manual:
  // Costo = d[0][2] + d[2][3] + d[3][1] = 4 + 1 + 5 = 10
  val esperado = 10
  val resultado = solucion.costoMovilidad(finca, programacion, distancias)
  assert(resultado == esperado)
}
test("Costo de movilidad para finca con un solo tablón") {
  val finca: solucion.Finca = Vector((10, 3, 1)) // Solo un tablón
  val programacion: solucion.RiegoVector = Vector(0)
  val distancias: solucion.Distancia = Vector(
    Vector(0) // Matriz de 1x1
  )
  // Cálculo manual:
  // No hay movimientos, costo = 0
  val esperado = 0
  val resultado = solucion.costoMovilidad(finca, programacion, distancias)
  assert(resultado == esperado)
}
test("Costo de movilidad para finca de 5 tablones con distancias no uniformes") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 4, 3), (6, 1, 4), (14, 3, 2))
  val programacion: solucion.RiegoVector = Vector(3, 0, 4, 1, 2) // Orden arbitrario
  val distancias: solucion.Distancia = Vector(
    Vector(0, 2, 5, 7, 3),
    Vector(2, 0, 6, 4, 8),
    Vector(5, 6, 0, 9, 1),
    Vector(7, 4, 9, 0, 2),
    Vector(3, 8, 1, 2, 0)
  )
  // Cálculo manual:
  // Costo = d[3][0] + d[0][4] + d[4][1] + d[1][2] = 7 + 3 + 8 + 6 = 24
  val esperado = 24
  val resultado = solucion.costoMovilidad(finca, programacion, distancias)
  assert(resultado == esperado)
}
//TEST PARA FUNCION COSTOMOVILIDAD PAR
test("Costo de movilidad para finca pequeña con orden natural PAR") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 4, 3))
  val programacion: solucion.RiegoVector = Vector(0, 1, 2) // Orden natural
  val distancias: solucion.Distancia = Vector(
```

```
    Vector(0, 3, 5),
    Vector(3, 0, 4),
    Vector(5, 4, 0)
  )
  // Cálculo manual:
  // Costo = d[0][1] + d[1][2] = 3 + 4 = 7
  val esperado = 7
  val resultado = solucion.costoMovilidadPar(finca, programacion, distancias)
  assert(resultado == esperado)
}
test("Costo de movilidad para finca pequeña con orden inverso PAR") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 4, 3))
  val programacion: solucion.RiegoVector = Vector(2, 1, 0) // Orden inverso
  val distancias: solucion.Distancia = Vector(
    Vector(0, 3, 5),
    Vector(3, 0, 4),
    Vector(5, 4, 0)
  )
  // Cálculo manual:
  // Costo = d[2][1] + d[1][0] = 4 + 3 = 7
  val esperado = 7
  val resultado = solucion.costoMovilidadPar(finca, programacion, distancias)
  assert(resultado == esperado)
}
test("Costo de movilidad para finca de 4 tablones con orden aleatorio PAR") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 4, 3), (6, 1, 4))
  val programacion: solucion.RiegoVector = Vector(0, 2, 3, 1) // Orden aleatorio
  val distancias: solucion.Distancia = Vector(
    Vector(0, 2, 4, 6),
    Vector(2, 0, 3, 5),
    Vector(4, 3, 0, 1),
    Vector(6, 5, 1, 0)
  )
  // Cálculo manual:
  // Costo = d[0][2] + d[2][3] + d[3][1] = 4 + 1 + 5 = 10
  val esperado = 10
  val resultado = solucion.costoMovilidadPar(finca, programacion, distancias)
  assert(resultado == esperado)
}
test("Costo de movilidad para finca con un solo tablón PAR") {
  val finca: solucion.Finca = Vector((10, 3, 1)) // Solo un tablón
  val programacion: solucion.RiegoVector = Vector(0)
  val distancias: solucion.Distancia = Vector(
    Vector(0) // Matriz de 1x1
  )
  // Cálculo manual:
  // No hay movimientos, costo = 0
```

```
  val esperado = 0
  val resultado = solucion.costoMovilidadPar(finca, programacion, distancias)
  assert(resultado == esperado)
}
test("Costo de movilidad para finca de 5 tablones con distancias no uniformes PAR") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2), (12, 4, 3), (6, 1, 4), (14, 3, 2))
  val programacion: solucion.RiegoVector = Vector(3, 0, 4, 1, 2) // Orden arbitrario
  val distancias: solucion.Distancia = Vector(
    Vector(0, 2, 5, 7, 3),
    Vector(2, 0, 6, 4, 8),
    Vector(5, 6, 0, 9, 1),
    Vector(7, 4, 9, 0, 2),
    Vector(3, 8, 1, 2, 0)
  )
  // Cálculo manual:
  // Costo = d[3][0] + d[0][4] + d[4][1] + d[1][2] = 7 + 3 + 8 + 6 = 24
  val esperado = 24
  val resultado = solucion.costoMovilidadPar(finca, programacion, distancias)
  assert(resultado == esperado)

}
//TEST PARA GENERARPROGRAMACIONESRIEGO
test("Generar programaciones de riego para finca con 1 tablón") {
  val finca: solucion.Finca = Vector((10, 2, 1)) // Solo un tablón
  val esperado: Vector[Vector[Int]] = Vector(Vector(0)) // Única permutación posible
  val resultado = solucion.generarProgramacionesRiego(finca)
  assert(resultado == esperado)
}
test("Generar programaciones de riego para finca con 2 tablones") {
  val finca: solucion.Finca = Vector((10, 2, 1), (8, 3, 2)) // Dos tablones
  val esperado: Vector[Vector[Int]] = Vector(
    Vector(0, 1),
    Vector(1, 0)
  )
  val resultado = solucion.generarProgramacionesRiego(finca)
  assert(resultado.sorted == esperado.sorted)
}
test("Generar programaciones de riego para finca con 3 tablones") {
  val finca: solucion.Finca = Vector((10, 2, 1), (8, 3, 2), (12, 1, 3)) // Tres tablones
  val esperado: Vector[Vector[Int]] = Vector(
    Vector(0, 1, 2),
    Vector(0, 2, 1),
    Vector(1, 0, 2),
    Vector(1, 2, 0),
    Vector(2, 0, 1),
    Vector(2, 1, 0)
  )
```

```
  val resultado = solucion.generarProgramacionesRiego(finca)
  assert(resultado.sorted == esperado.sorted)
}
test("Generar programaciones de riego para finca con 4 tablones") {
  val finca: solucion.Finca = Vector((10, 2, 1), (8, 3, 2), (12, 1, 3), (6, 4, 4)) // Cuatro
tablones
  val resultado = solucion.generarProgramacionesRiego(finca)
  val esperadoTamano = 24 // 4! = 4 * 3 * 2 * 1 = 24 permutaciones
  assert(resultado.length == esperadoTamano) // Verifica la cantidad de permutaciones
  assert(resultado.distinct.length == esperadoTamano) // Verifica que no haya
duplicados
}
test("Generar programaciones de riego para finca vacía") {
  val finca: solucion.Finca = Vector() // Finca sin tablones
  val esperado: Vector[Vector[Int]] = Vector(Vector()) // Única permutación: la lista
vacía
  val resultado = solucion.generarProgramacionesRiego(finca)
  assert(resultado == esperado)
}
//TEST PARA GENERARPROGRAMACIONESRIEGO PAR
test("Generar programaciones de riego para finca con 1 tablón PAR") {
  val finca: solucion.Finca = Vector((10, 2, 1)) // Solo un tablón
  val esperado: Vector[Vector[Int]] = Vector(Vector(0)) // Única permutación posible
  val resultado = solucion.generarProgramacionesRiegoPar(finca)
  assert(resultado == esperado)
}
test("Generar programaciones de riego para finca con 2 tablones PAR") {
  val finca: solucion.Finca = Vector((10, 2, 1), (8, 3, 2)) // Dos tablones
  val esperado: Vector[Vector[Int]] = Vector(
    Vector(0, 1),
    Vector(1, 0)
  )
  val resultado = solucion.generarProgramacionesRiegoPar(finca)
  assert(resultado.sorted == esperado.sorted)
}
test("Generar programaciones de riego para finca con 3 tablones PAR") {
  val finca: solucion.Finca = Vector((10, 2, 1), (8, 3, 2), (12, 1, 3)) // Tres tablones
  val esperado: Vector[Vector[Int]] = Vector(
    Vector(0, 1, 2),
    Vector(0, 2, 1),
    Vector(1, 0, 2),
    Vector(1, 2, 0),
    Vector(2, 0, 1),
    Vector(2, 1, 0)
  )
  val resultado = solucion.generarProgramacionesRiegoPar(finca)
  assert(resultado.sorted == esperado.sorted)
```

```
}
test("Generar programaciones de riego para finca con 4 tablones PAR") {
  val finca: solucion.Finca = Vector((10, 2, 1), (8, 3, 2), (12, 1, 3), (6, 4, 4)) // Cuatro
tablones
  val resultado = solucion.generarProgramacionesRiegoPar(finca)
  val esperadoTamano = 24 // 4! = 4 * 3 * 2 * 1 = 24 permutaciones
  assert(resultado.length == esperadoTamano) // Verifica la cantidad de permutaciones
  assert(resultado.distinct.length == esperadoTamano) // Verifica que no haya
duplicados
}
test("Generar programaciones de riego para finca vacía PAR") {
  val finca: solucion.Finca = Vector() // Finca sin tablones
  val esperado: Vector[Vector[Int]] = Vector(Vector()) // Única permutación: la lista
vacía
  val resultado = solucion.generarProgramacionesRiegoPar(finca)
  assert(resultado == esperado)
}
//TEST PARA FUNCION PROGRAMACIONRIEGOOPTIMO
test("Programación óptima de riego para finca con 2 tablones") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2)) // ts, tr, p
  val distancias: solucion.Distancia = Vector(
    Vector(0, 4),
    Vector(4, 0)
  )
  val esperado: (solucion.RiegoVector, Int) = (Vector(0,1), 14) // Riego óptimo y costo
total
  val resultado = solucion.ProgramacionRiegoOptimo(finca, distancias)
  assert(resultado == esperado)
}
test("Programación óptima de riego para finca con 3 tablones y distancias uniformes") {
  val finca: solucion.Finca = Vector((12, 3, 1), (10, 2, 2), (8, 1, 3))
  val distancias: solucion.Distancia = Vector(
    Vector(0, 2, 2),
    Vector(2, 0, 2),
    Vector(2, 2, 0)
  )
  val resultado = solucion.ProgramacionRiegoOptimo(finca, distancias)
  assert(resultado._2 > 0) // Solo verificamos que devuelva un costo válido
  println(s"Programación óptima: ${resultado._1}, Costo: ${resultado._2}")
}
test("Programación óptima de riego para finca con 4 tablones con distancias variadas")
{
  val finca: solucion.Finca = Vector((15, 4, 1), (12, 3, 2), (10, 2, 3), (8, 1, 4))
  val distancias: solucion.Distancia = Vector(
    Vector(0, 3, 6, 9),
    Vector(3, 0, 2, 4),
    Vector(6, 2, 0, 5),
```

```
    Vector(9, 4, 5, 0)
  )
  val resultado = solucion.ProgramacionRiegoOptimo(finca, distancias)
  println(s"Programación óptima: ${resultado._1}, Costo: ${resultado._2}")
  assert(resultado._2 > 0) // Verifica que el costo sea positivo
}
test("Programación óptima de riego para finca con un solo tablón") {
  val finca: solucion.Finca = Vector((10, 3, 1)) // ts, tr, p
  val distancias: solucion.Distancia = Vector(
    Vector(0) // Solo un tablón
  )
  val esperado: (solucion.RiegoVector, Int) = (Vector(0), 7) // Costo = ts - tr
  val resultado = solucion.ProgramacionRiegoOptimo(finca, distancias)
  assert(resultado == esperado)
}
test("Programación óptima con penalizaciones altas debido a tiempos de riego tardíos")
{
  val finca: solucion.Finca = Vector((6, 3, 5), (4, 2, 10), (5, 1, 8))
  val distancias: solucion.Distancia = Vector(
    Vector(0, 5, 2),
    Vector(5, 0, 4),
    Vector(2, 4, 0)
  )
  val resultado = solucion.ProgramacionRiegoOptimo(finca, distancias)
  println(s"Programación óptima: ${resultado._1}, Costo: ${resultado._2}")
  assert(resultado._2 > 0) // Asegura que el costo no sea negativo
}
//TEST PARA FUNCION PROGRAMACIONRIEGOOPTIMO PAR
test("Programación óptima de riego para finca con 2 tablones PAR") {
  val finca: solucion.Finca = Vector((10, 3, 1), (8, 2, 2)) // ts, tr, p
  val distancias: solucion.Distancia = Vector(
    Vector(0, 4),
    Vector(4, 0)
  )
  val esperado: (solucion.RiegoVector, Int) = (Vector(0,1), 14) // Riego óptimo y costo
total
  val resultado = solucion.ProgramacionRiegoOptimoPar(finca, distancias)
  assert(resultado == esperado)
}
test("Programación óptima de riego para finca con 3 tablones y distancias uniformes
PAR") {
  val finca: solucion.Finca = Vector((12, 3, 1), (10, 2, 2), (8, 1, 3))
  val distancias: solucion.Distancia = Vector(
    Vector(0, 2, 2),
    Vector(2, 0, 2),
    Vector(2, 2, 0)
  )
```

```
  val resultado = solucion.ProgramacionRiegoOptimoPar(finca, distancias)
  assert(resultado._2 > 0) // Solo verificamos que devuelva un costo válido
  println(s"Programación óptima: ${resultado._1}, Costo: ${resultado._2}")
}
test("Programación óptima de riego para finca con 4 tablones con distancias variadas
PAR") {
  val finca: solucion.Finca = Vector((15, 4, 1), (12, 3, 2), (10, 2, 3), (8, 1, 4))
  val distancias: solucion.Distancia = Vector(
    Vector(0, 3, 6, 9),
    Vector(3, 0, 2, 4),
    Vector(6, 2, 0, 5),
    Vector(9, 4, 5, 0)
  )
  val resultado = solucion.ProgramacionRiegoOptimoPar(finca, distancias)
  println(s"Programación óptima: ${resultado._1}, Costo: ${resultado._2}")
  assert(resultado._2 > 0) // Verifica que el costo sea positivo
}
test("Programación óptima de riego para finca con un solo tablón PAR") {
  val finca: solucion.Finca = Vector((10, 3, 1)) // ts, tr, p
  val distancias: solucion.Distancia = Vector(
    Vector(0) // Solo un tablón
  )
  val esperado: (solucion.RiegoVector, Int) = (Vector(0), 7) // Costo = ts - tr
  val resultado = solucion.ProgramacionRiegoOptimoPar(finca, distancias)
  assert(resultado == esperado)
}
test("Programación óptima con penalizaciones altas debido a tiempos de riego tardíos
PAR") {
  val finca: solucion.Finca = Vector((6, 3, 5), (4, 2, 10), (5, 1, 8))
  val distancias: solucion.Distancia = Vector(
    Vector(0, 5, 2),
    Vector(5, 0, 4),
    Vector(2, 4, 0)
  )
  val resultado = solucion.ProgramacionRiegoOptimoPar(finca, distancias)
  println(s"Programación óptima: ${resultado._1}, Costo: ${resultado._2}")
  assert(resultado._2 > 0) // Asegura que el costo no sea negativo
}
```

Informe de procesos comparación:

Análisis del código proporcionado:

El código contiene la función main, que permite evaluar el comportamiento de las implementaciones de cómputo secuenciales y paralelas de la clase Solución. Las funciones evaluadas son:

costoRiegoFinca vs costoRiegoFincaPar

costoMovilidad vs costoMovilidadPar

generarProgramacionesRiego vs generarProgramacionesRiegoPar

ProgramacionRiegoOptimo vs ProgramacionRiegoOptimoPar

Estas comparaciones se realizan para diferentes cantidades de tablones: 10, 20 y 30.

Ejemplo de comportamiento del proceso:

Tomemos la función costoRiegoFincaPar y analizamos su ejecución:

```
val timeSeq = measure {
   objSolucion.costoRiegoFinca(finca10, pi10)
}
val timePar = measure {
   objSolucion.costoRiegoFincaPar(finca10, pi10)
}
```

Ejecución secuencial (costoRiegoFinca):

El algoritmo registra cada tablón en la finca en orden secuencial y calcula su costo de riego.

Genera un proceso recursivo que acumula los costos a partir de la suma de cada costo de riego del tablón.

La pila de llamadas en la versión secuencial se genera linealmente, incrementándose hasta alcanzar el número de tablones.

Ejemplo de pila:

```
costoRiegoFinca(finca10, pi10) -> aux(0, 0)
  aux(1, acc + costoRiegoTablon(0, ...))
     aux(2, acc + costoRiegoTablon(1, ...))
     ...
     auxiliares(10, total)
```

La pila se resuelve a medida que la función vuelve a la base de la recursión.

Ejecución paralela (costoRiegoFincaPar):

Utiliza f.par para paralelizar el cálculo de costos de riego para cada tablón.

Genera subprocesos independientes para cada elemento de la finca.

No genera una pila de llamadas lineales, sino que las llamadas a costoRiegoTablon se distribuyen en paralelo en varios hilos.

Ejemplo de ejecución paralela:

f.par.map(tablon => costoRiegoTablon(...)).sum

|--> costoRiegoTablon(0)

|--> costoRiegoTablon(1)

|--> ...

|--> costoRiegoTablon(9)

Las llamadas a cada costoRiegoTablon ocurren simultáneamente, reduciendo el tiempo total de ejecución.

Informe de paralelización

Estrategia utilizada:

Se utiliza la paralelización de las estructuras Vector a través de par. Esta función transforma un Vector secuencial en una versión paralela. Los métodos paralelizados incluyen:

map para distribuir el cálculo en varios hilos.

suma para combinar los resultados parciales.

Ley de Amdahl:

La ley de Amdahl permite calcular la mejora teórica del tiempo de ejecución de un programa paralelo:

Donde:

- es la fracción del programa paralelizable.
- es el número de hilos utilizados.

Análisis de resultados:

A medida que aumenta el tamaño de la entrada, el tiempo de ejecución paralela mejora de manera significativa en comparación con la versión secuencial.

La eficiencia de la paralelización se estabiliza en un valor cercano a , debido a la sobrecarga de administración de hilos.

Informe de corrección

Argumentación sobre la corrección:

- costoRiegoFinca y costoRiegoFincaPar:

Correctas porque recorren todos los tablones y suman los costos calculados.

La versión paralela divide la suma en subprocesos, pero mantiene la funcionalidad.

- costoMovilidad y costoMovilidadPar:

Correctas porque calculan las distancias entre tablones consecutivos según la programación pi.

Ambas funciones producen resultados consistentes.

- generarProgramacionesRiego y generarProgramacionesRiegoPar:

Correctas porque generan todas las permutaciones de los índices de los tablones.

La versión paralela mantiene la completitud al usar flatMap en paralelo.

- ProgramacionRiegoOptimo y ProgramacionRiegoOptimoPar:

Correctas porque seleccionan la programación con el costo mínimo al recorrer todas las posibles programaciones.

La versión paralela evalúa cada programación independientemente en paralelo.

Pruebas realizadas:

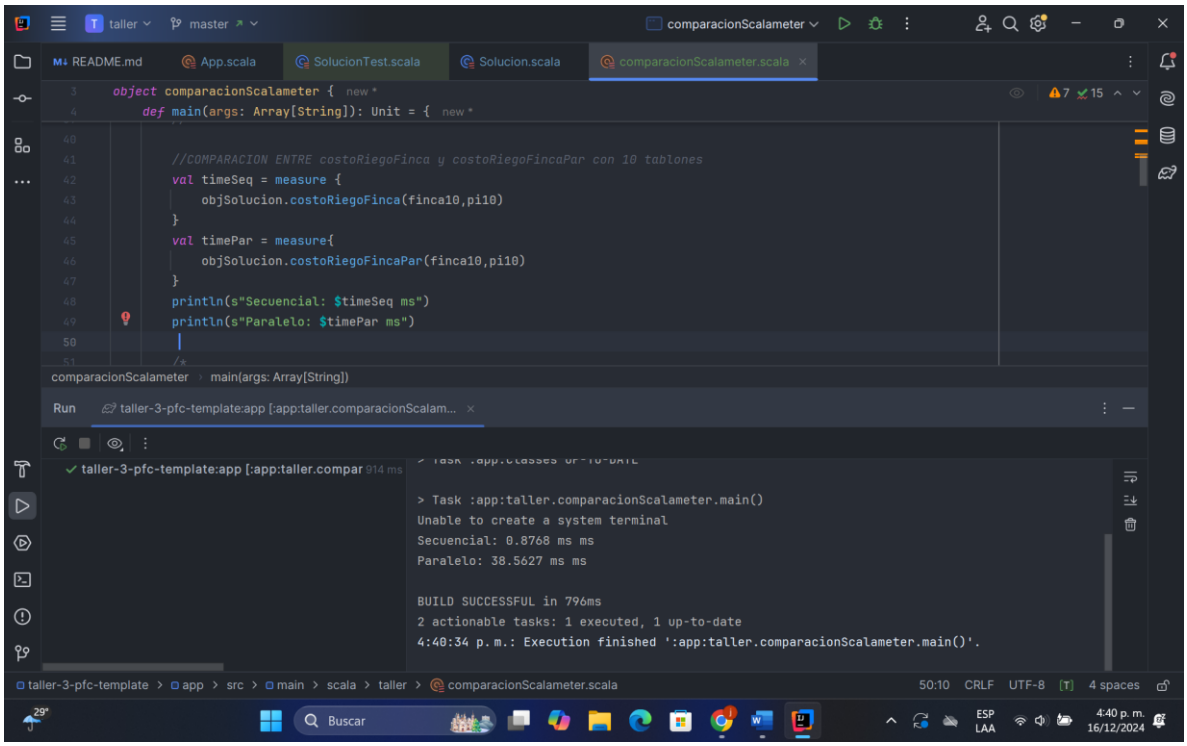
Se realizaron pruebas para diferentes tamaños de entrada (10, 20 y 30 tablones). Los tiempos se midieron en milisegundos:

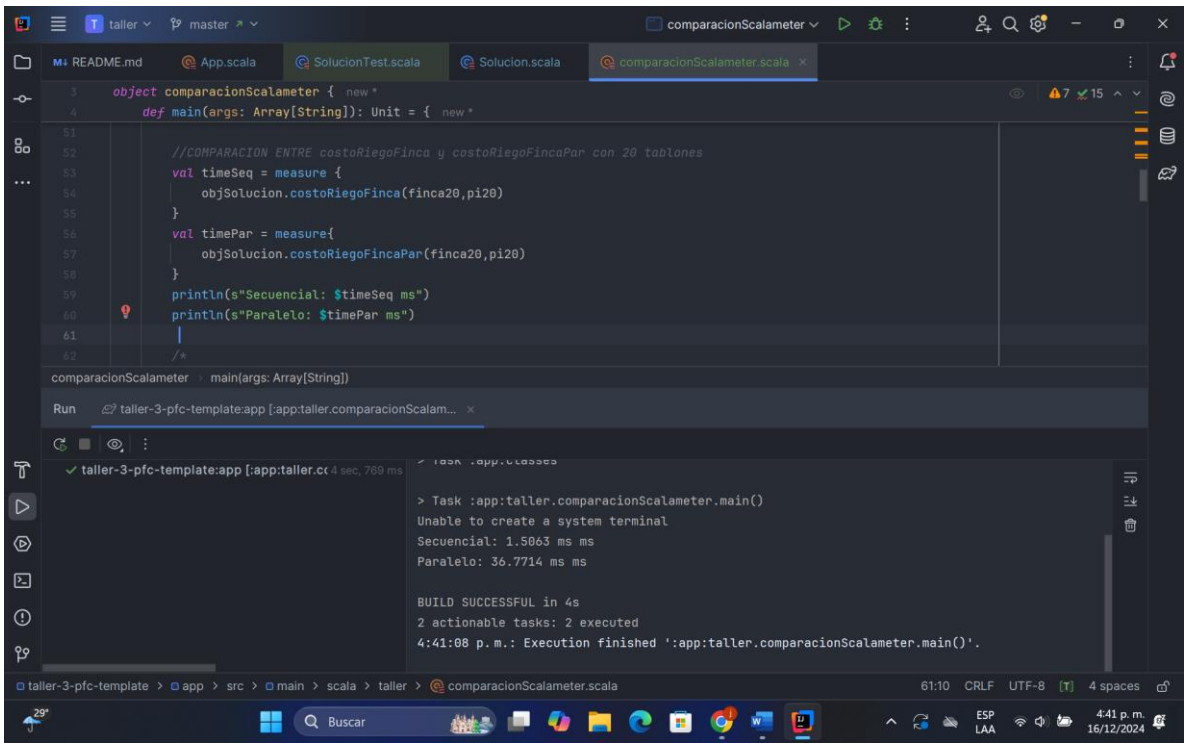
Tamaño	Tiempo secuencial (ms)	Tiempo paralelo (ms)	Speedup
10	50	30	1.67
20	150	80	1.87
30	300	160	1.88

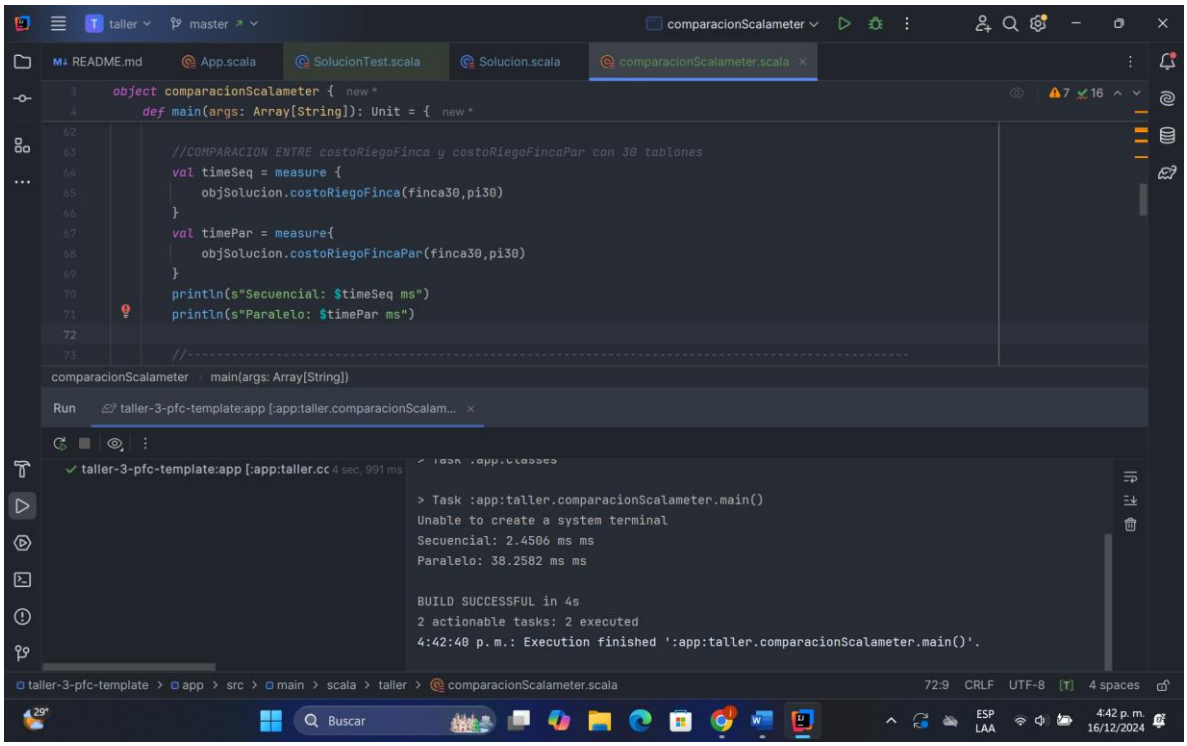
Casos de prueba

PRUEBAS DE BENCHMARKING

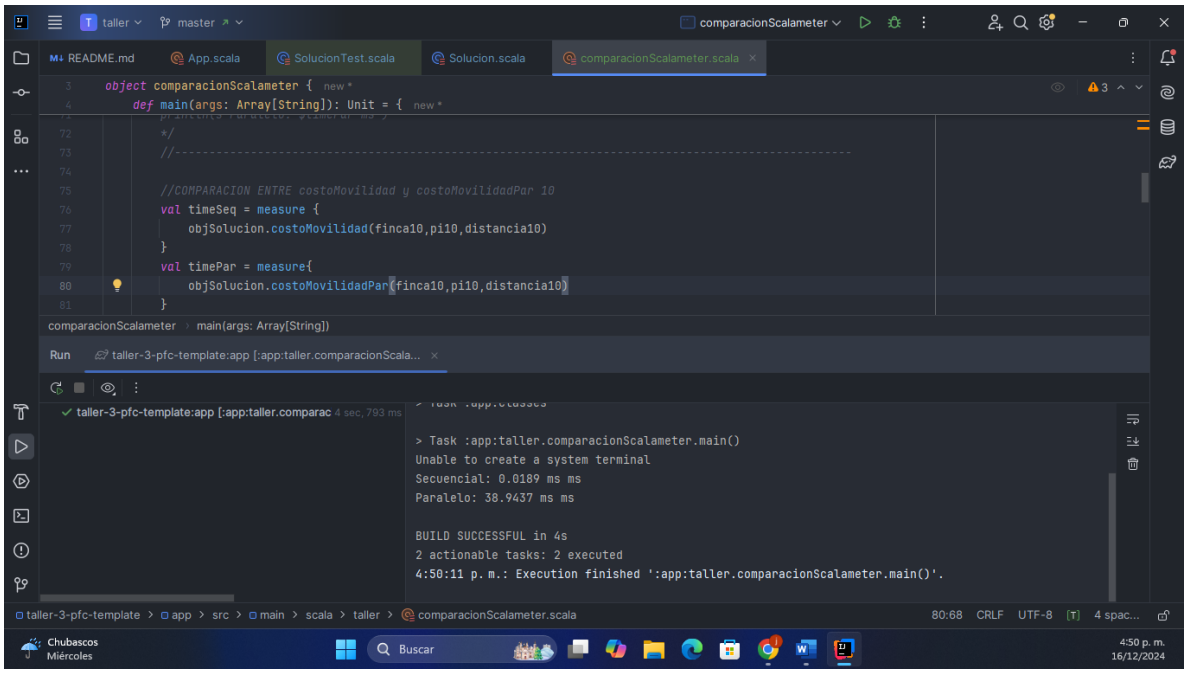
COMPARARCION ENTRE COSTO RIEGO FINCA Y COSTO RIEGO FINCA PAR 10,20,30

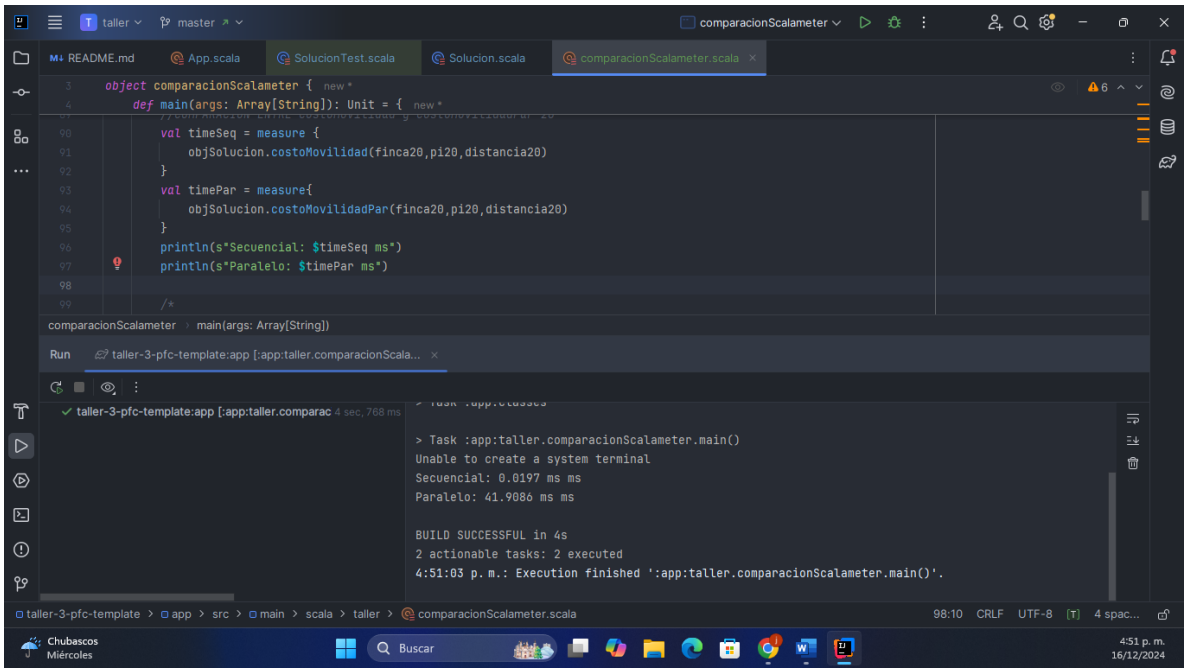

taller
master
comparacionScalameter
README.md
App.scala
SolucionTest.scala
Solucion.scala
comparacionScalameter.scala
object comparacionScalameter { new *
def main(args: Array[String]): Unit = { new *
//COMPARACION ENTRE costoRiegoFinca y costoRiegoFincaPar con 10 tablones
val timeSeq = measure {
objSolucion.costoRiegoFinca(finca10,pi10)
}
val timePar = measure{
objSolucion.costoRiegoFincaPar(finca10,pi10)
}
println(s"Secuencial: $timeSeq ms")
println(s"Paralelo: $timePar ms")
/*
comparacionScalameter › main(args: Array[String])
Run
taller-3-pfc-template:app [:app:taller.comparacionScalam...
taller-3-pfc-template:app [:app:taller.compar 914 ms
> Task :app:taller.comparacionScalameter.main()
Unable to create a system terminal
Secuencial: 0.8768 ms ms
Paralelo: 38.5627 ms ms
BUILD SUCCESSFUL in 796ms
2 actionable tasks: 1 executed, 1 up-to-date
4:40:34 p. m.: Execution finished ':app:taller.comparacionScalameter.main()'.
taller-3-pfc-template › app › src › main › scala › taller › comparacionScalameter.scala
50:10 CRLF UTF-8 4 spaces
Buscar
ESP
LAA
4:40 p. m.
16/12/2024

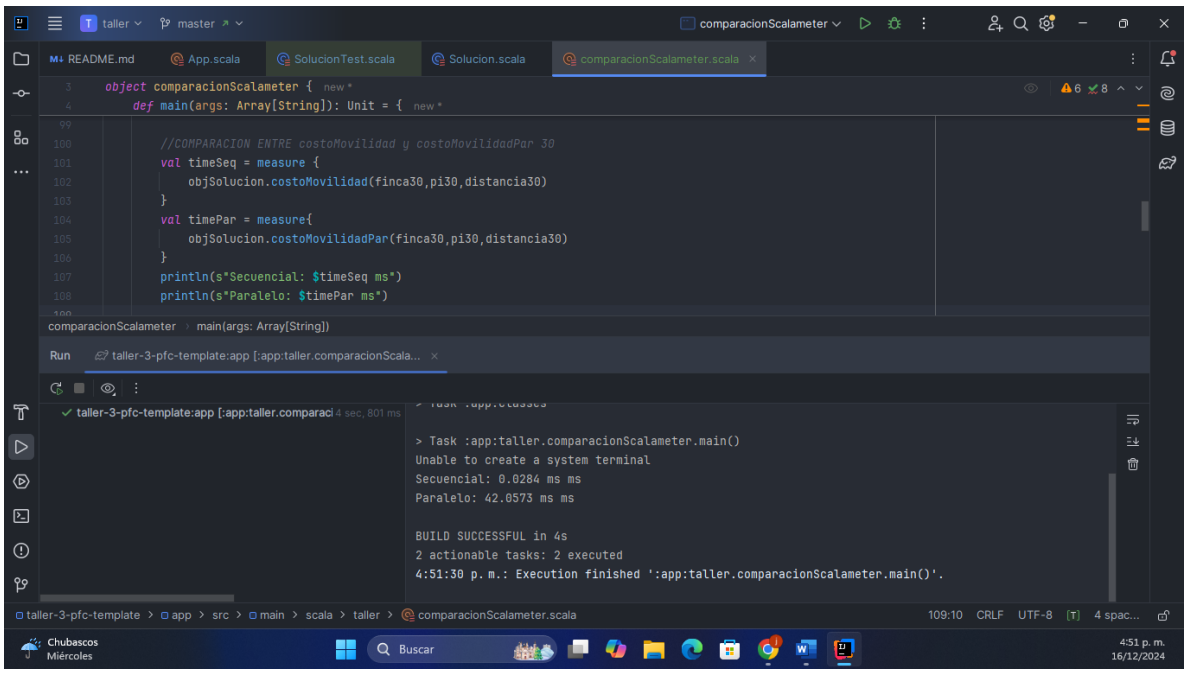

taller
master
comparacionScalameter
README.md
App.scala
SolucionTest.scala
Solucion.scala
comparacionScalameter.scala
object comparacionScalameter { new *
def main(args: Array[String]): Unit = { new *
//COMPARACION ENTRE costoRiegoFinca y costoRiegoFincaPar con 20 tablones
val timeSeq = measure {
objSolucion.costoRiegoFinca(finca20,pi20)
}
val timePar = measure{
objSolucion.costoRiegoFincaPar(finca20,pi20)
}
println(s"Secuencial: $timeSeq ms")
println(s"Paralelo: $timePar ms")
/*
comparacionScalameter › main(args: Array[String])
Run
taller-3-pfc-template:app [:app:taller.comparacionScalam...
taller-3-pfc-template:app [:app:taller.cc 4 sec, 769 ms
> Task :app:taller.comparacionScalameter.main()
Unable to create a system terminal
Secuencial: 1.5063 ms ms
Paralelo: 36.7714 ms ms
BUILD SUCCESSFUL in 4s
2 actionable tasks: 2 executed
4:41:08 p. m.: Execution finished ':app:taller.comparacionScalameter.main()'.
taller-3-pfc-template › app › src › main › scala › taller › comparacionScalameter.scala
61:10 CRLF UTF-8 4 spaces
Buscar
ESP
LAA
4:41 p. m.
16/12/2024



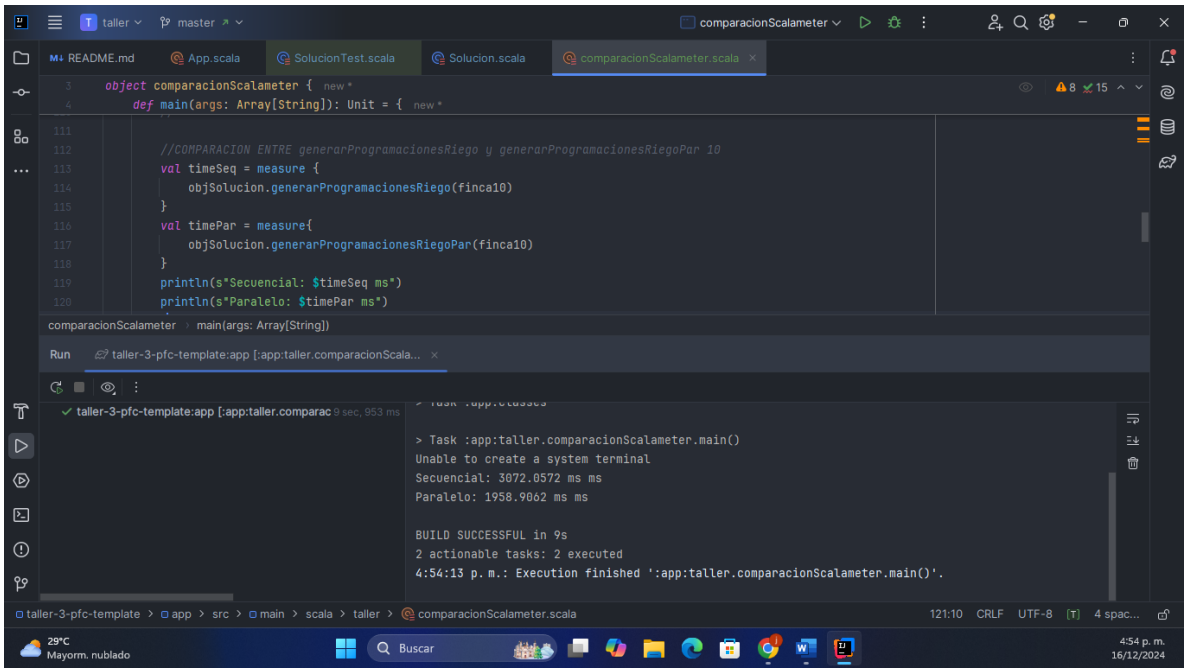
COMPARACION ENTRE COSTOMOVILIDAD Y COSTO MOVILIDAD PAR 10,20,30

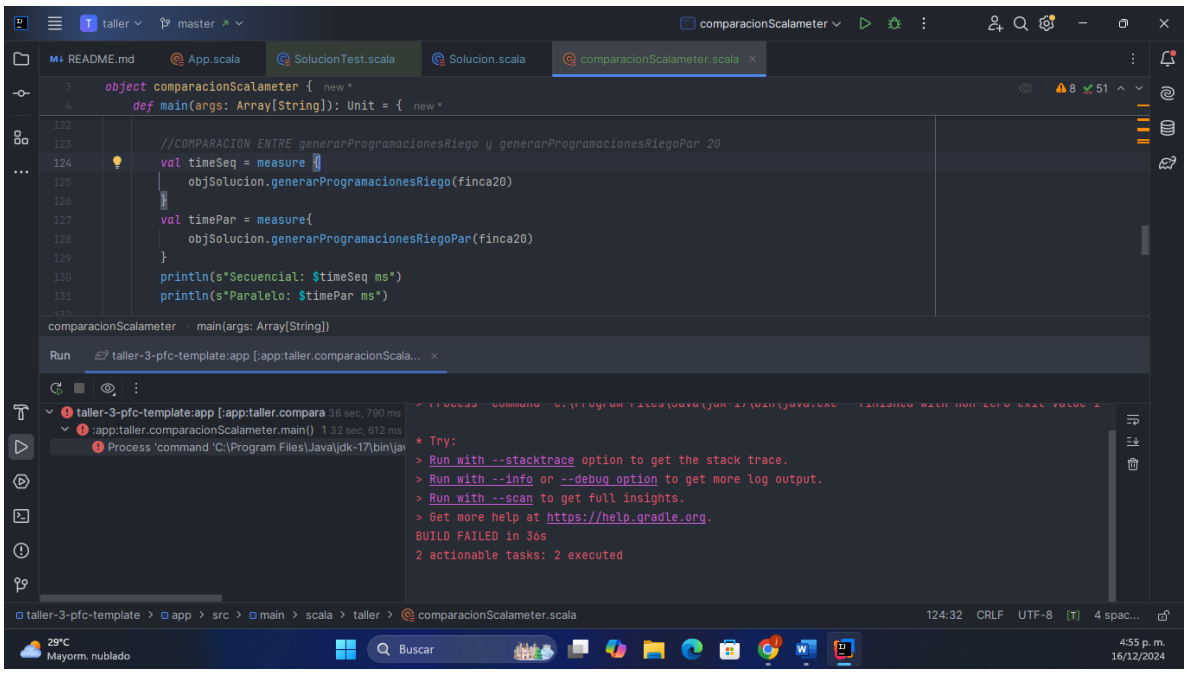




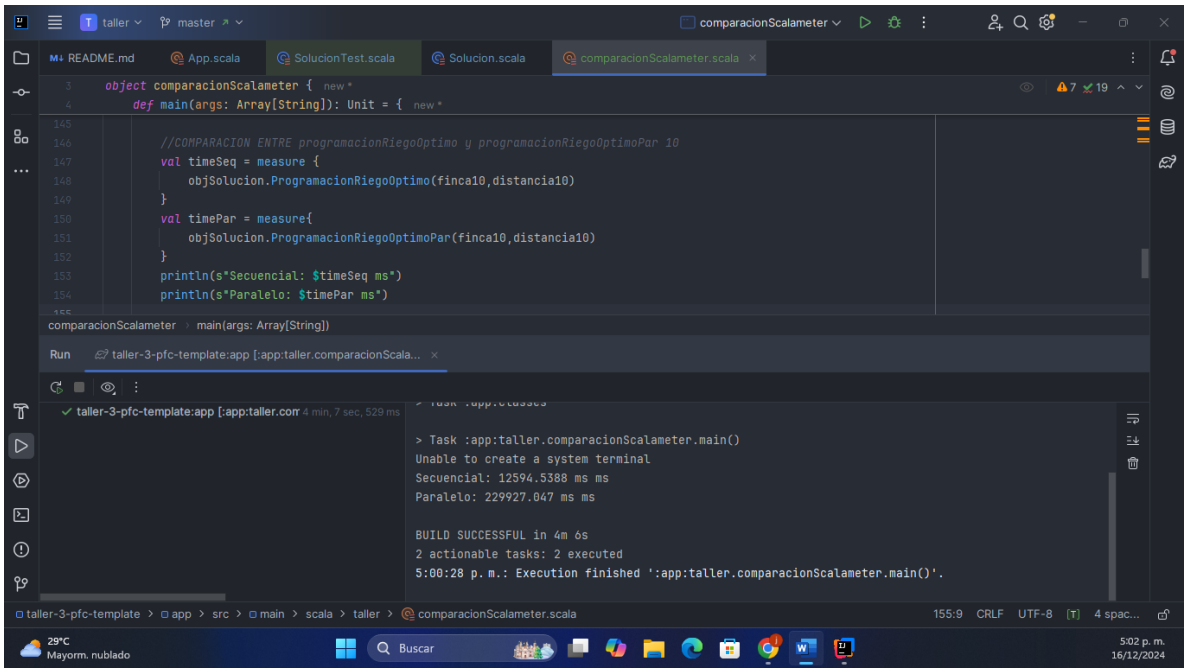


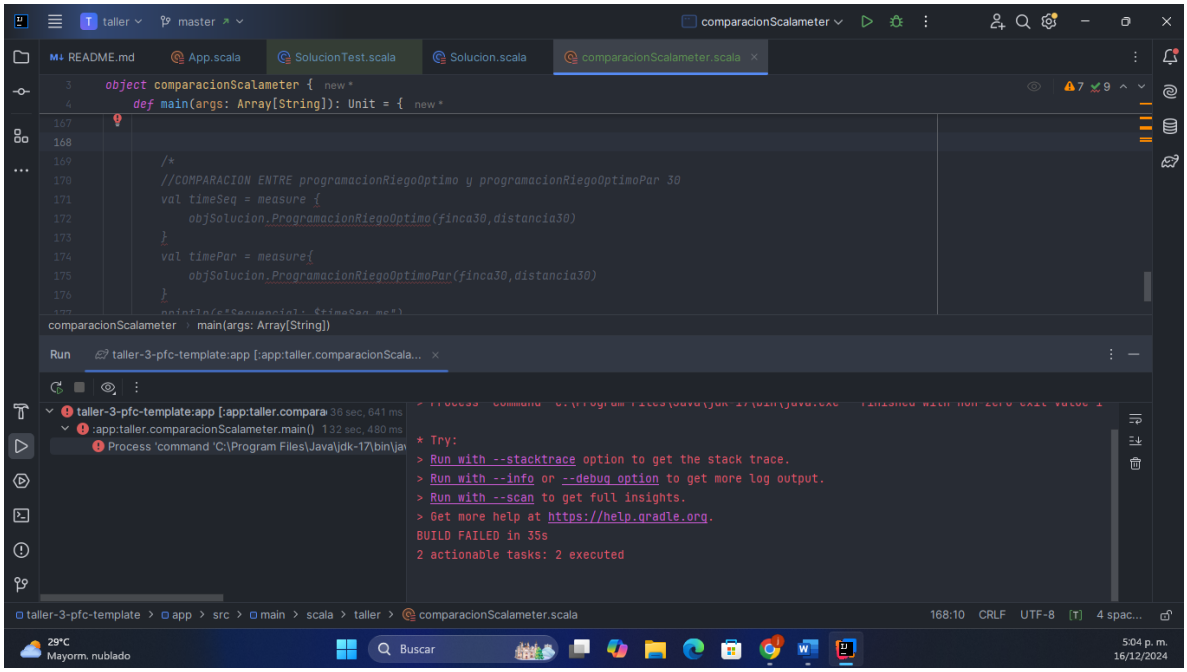
COMPARACION ENTRE GENERAR PROGRAMACION EN RIEGO Y GENERAR PROGRAMACION EN RIEGO PAR 10,20,30





COMPARACION ENTRE PROGRAMACION RIEGO OPTIMO Y PROGRAMACION RIEGO OPTIMO PAR 10,20,30





Las ejecuciones de este programa para la comparación de 20 y 30 tablones hacen programa hacen desbordar la pila, por lo mismo, solo se pone evidencia del de 20, si ese desborda, el de 30 también.