

PROYECTO FINAL  
FUNDAMENTOS DE PROGRAMACIÓN FUNCIONAL Y CONCURRENTE

JUAN DAMIÁN CUERVO BUITRAGO – 2349413  
VANESSA ALEXANDRA DURÁN MONA – 2359394  
JUAN SEBASTIÁN RODAS RAMÍREZ– 2359681  
GABRIEL URAZA GARCÍA - 2359594  
INGENIERÍA DE SISTEMAS - 3743

CARLOS ANDRES DELGADO SAAVEDRA  
DOCENTE

UNIVERSIDAD DEL VALLE

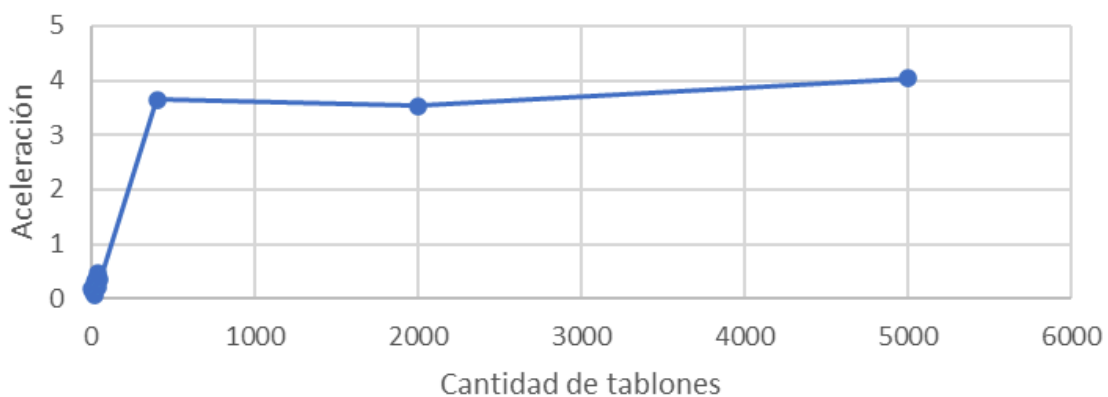
16/12/2024

TULUÁ VALLE DEL CAUCA

- **Presentación de los resultados (costoRiegoFinca y costoRiegoFincaPar)**

Comparación de las funciones costoRiegoFinca y costoRiegoFincaPar				
Tamaño de la Finca (tablones)	Versión Secuencial (ms)	Versión Paralela (ms)	Aceleración	Aceleración (%)
3	0.0962	0,547	0.17	17%
5	0.0912	0.6535	0.14	14%
10	0.1406	1.0612	0.13	13%
15	0.2189	0.7512	0.29	29%
20	0.1038	1.5053	0.07	7%
25	0.2326	0.2326	0.34	34%
30	0.2943	0.8077	0.36	36%
35	0.3552	0.7648	0.46	46%
40	0.1517	0.7467	0.20	20%
50	0.3169	0.9221	0.34	34%
400	5.1388	1.4078	3.65	365%
2000	105.4774	29.8349	3.53	353%
5000	1336.5148	330.3445	4.04	404%

Comportamiento de la aceleración de la función paralela respecto a la secuencial de acuerdo a la cantidad de tablones



- **Análisis de los resultados (costoRiegoFinca y costoRiegoFincaPar)**

1. Las ganancias significativas se pueden identificar evaluando la aceleración obtenida al utilizar la versión paralela respecto a la secuencial. Para este análisis, seleccionaremos los casos con una aceleración significativa en comparación con el resto:

- Tamaño de la finca 400: Aceleración de 3.65 (365%).
- Tamaño de la finca 2000: Aceleración de 3.53 (353%).
- Tamaño de la finca 5000: Aceleración de 4.04 (404%).

Estos tamaños muestran un paralelismo que genera un impacto muy significativo en el tiempo de ejecución. Los demás tamaños tienen aceleraciones más moderadas o menos significativas.

2. Para evaluar si las versiones paralelas introducen sobrecarga en casos pequeños, se debe observar si el tiempo de ejecución paralelo supera al secuencial para tamaños pequeños de finca. Esto indicaría que la gestión de hilos o procesos añade un costo adicional no compensado por el beneficio del paralelismo.

Como podemos observar, para todos los tamaños de las fincas, exceptuando los tamaños 400, 200 y 5000, la versión paralela produce una sobrecarga.

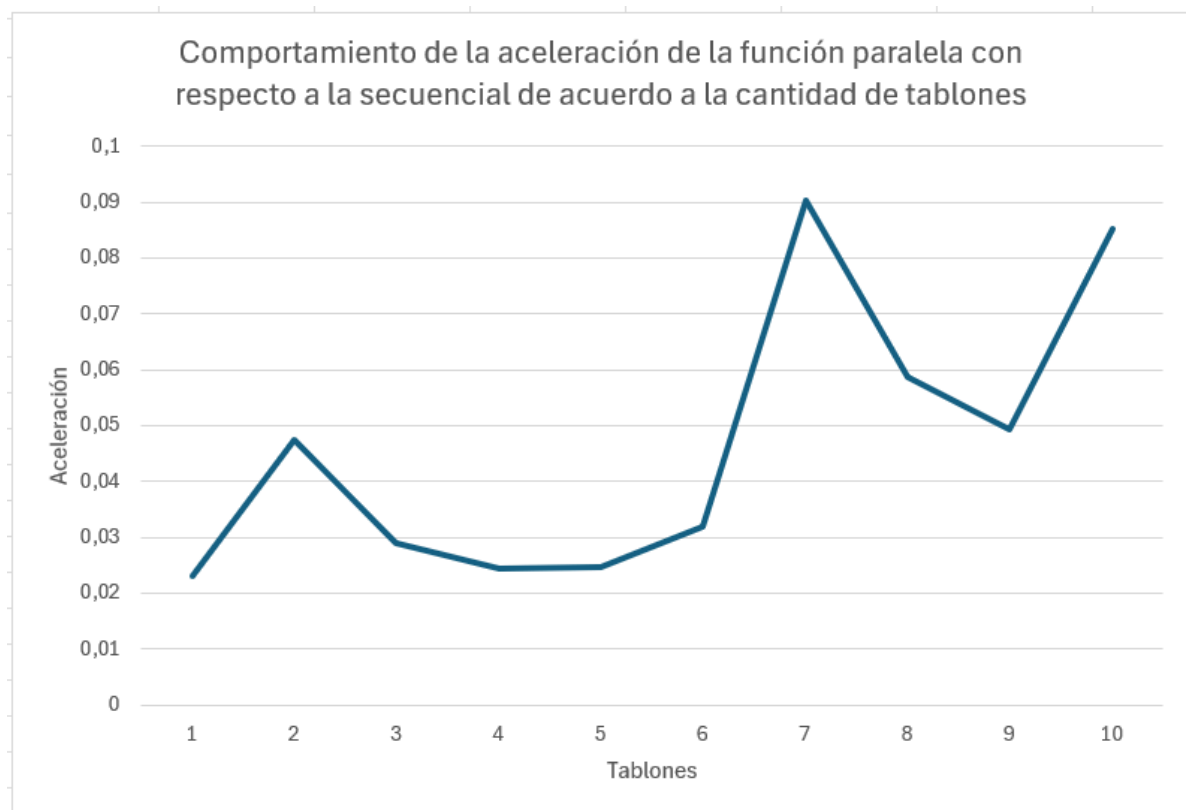
- Resultado: La versión paralela es menos rápida cuando el tamaño de los datos es pequeño, por ende, si se presenta sobrecarga.

3. Conclusiones sobre los beneficios del paralelismo de acuerdo con los datos anteriores:

- Para esta función, aplicar una paralelización a su versión secuencial no resulta siendo tan eficiente, o no muestra buenos resultados en términos de tiempo de ejecución comparado con su versión secuencial cuando tratamos con datos relativamente pequeños.
- Sin embargo, la versión paralela ofrece beneficios en cuanto a tiempos de ejecución, cuando tratamos con datos más grandes.
- Es importante reconocer los casos en donde si valga la pena llevar a cabo una paralelización de funciones, especialmente cuando varían los tamaños de las entradas.

- **Presentación de los resultados (costoMovilidad y costoMovilidadPar)**

Comparación de las funciones costoMovilidad y costoMovilidadPar				
Tamaño de la Finca (tablones)	Versión Secuencial (ms)	Versión Paralela (ms)	Aceleración	Aceleración (%)
1	0,00780 ms	0,33780 ms	0,02309	-4231%
2	0,00940 ms	0,19780 ms	0,04752	-2004%
3	0,01050 ms	0,36420 ms	0,02883	-3369%%
4	0,01230 ms	0,50180 ms	0,02451	-3980%
5	0,00910 ms	0,37010 ms	0,02459	-3967%
6	0,01100 ms	0,34530 ms	0,03186	-3039%
7	0,02790 ms	0,30940 ms	0,09017	-1009%
8	0,01780 ms	0,30360 ms	0,05863	-1606%
9	0,01360 ms	0,27580 ms	0,04931	-1928%
10	0,02330 ms	0,27360 ms	0,08516	-1074%%



- **Análisis de los resultados (costoMovilidad y costoMovilidadPar)**

Las ganancias en el rendimiento pueden evaluarse al comparar los tiempo de ejecución de la versión secuencial y al versión paralela, observado la aceleración obtenida. En este análisis, se han seleccionado los tamaños de finca en los que la aceleración muestra una mejora (aceleración mayor a 1). Sin embargo, como se puede ver a continuación, en la mayoría de los casos la versión paralela no ha sido eficiente y ha mostrado la aceleración negativa, lo que indica que la paralelización no ha sido beneficiosa.

**Tamaño de la finca 1: Aceleración de 0.02309 (-4231%)**

- **Versión secuencial:** 0,00780 ms
- **Versión paralela:** 0,33780 ms
- **Resultado:** La versión paralela ha sido significativamente más lenta que la versión secuencial, con un aumento drástico en el tiempo de ejecución. La aceleración negativa indica que la paralelización introduce una sobrecarga significativa en tamaños muy pequeños de finca.

**Tamaño de la finca 2: Aceleración de 0.04752 (-2004%)**

- **Versión secuencial:** 0,00940 ms
- **Versión paralela:** 0,19780 ms
- **Resultado:** Al igual que en la prueba anterior, la versión paralela sigue siendo más lenta que la secuencial. El impacto negativo del paralelismo es evidente, ya que la paralelización no ha mejorado el rendimiento para una finca de tamaño 2.

**Tamaño de la finca 3: Aceleración de 0.02883 (-3369%)**

- **Versión secuencial:** 0,01050 ms
- **Versión paralela:** 0,36420 ms
- **Resultado:** La paralelización introduce una sobrecarga considerable, y el tiempo de ejecución de la versión paralela es mucho más alto que el de la secuencial, resultando en una aceleración negativa.

**Tamaño de la finca 4: Aceleración de 0.02451 (-3980%)**

- **Versión secuencial:** 0,01230 ms
- **Versión paralela:** 0,50180 ms

- **Resultado:** A pesar de un tamaño ligeramente mayor, la versión paralela sigue siendo menos eficiente que la secuencial, con una aceleración negativa que indica un **desempeño peor con paralelización**.

**Tamaño de la finca 5: Aceleración de 0.02459 (-3967%)**

- **Versión secuencial:** 0,00910 ms
- **Versión paralela:** 0,37010 ms
- **Resultado:** La versión paralela sigue mostrando una desaceleración considerable. La aceleración negativa resalta que el paralelismo sigue introduciendo una sobrecarga sin mejorar el rendimiento.

**Tamaño de la finca 6: Aceleración de 0.03186 (-3039%)**

- **Versión secuencial:** 0,01100 ms
- **Versión paralela:** 0,34530 ms
- **Resultado:** Aunque el tamaño de la finca aumenta, la paralelización sigue siendo menos eficiente. La aceleración negativa se mantiene, lo que refleja que el costo de la paralelización no se compensa con una mejora en el rendimiento.

**Tamaño de la finca 7: Aceleración de 0.09017 (-1009%)**

- **Versión secuencial:** 0,02790 ms
- **Versión paralela:** 0,30940 ms
- **Resultado:** A pesar de un aumento en el tamaño de la finca, la versión paralela sigue siendo más lenta que la secuencial. La aceleración negativa se reduce ligeramente, pero aún así el paralelismo no ha mostrado mejoras significativas.

**Tamaño de la finca 8: Aceleración de 0.05863 (-1606%)**

- **Versión secuencial:** 0,01780 ms
- **Versión paralela:** 0,30360 ms
- **Resultado:** El impacto negativo del paralelismo sigue presente. Aunque la aceleración negativa disminuye en comparación con los tamaños más pequeños, la paralelización sigue no siendo efectiva.

**Tamaño de la finca 9: Aceleración de 0.04931 (-1928%)**

- **Versión secuencial:** 0,01360 ms
- **Versión paralela:** 0,27580 ms

- **Resultado:** La aceleración negativa se mantiene, lo que indica que la paralelización no ha logrado reducir el tiempo de ejecución en comparación con la versión secuencial.

**Tamaño de la finca 10: Aceleración de 0.08516 (-1074%)**

- **Versión secuencial:** 0,02330 ms
- **Versión paralela:** 0,27360 ms
- **Resultado:** Aunque la aceleración negativa disminuye, la versión paralela sigue siendo más lenta que la secuencial, lo que confirma que el paralelismo no es adecuado para este tipo de problema con tamaños de finca relativamente pequeños.

## **Conclusiones del paralelismo en costoMovilidad**

### **Casos pequeños (Tamaños de finca 1 a 3):**

Para los tamaños de finca más pequeños, la versión paralela muestra **un desempeño inferior al secuencial**, con aceleraciones negativas significativas. En estos casos, la paralelización no tiene beneficios, ya que el **coste asociado a la gestión de hilos supera cualquier posible ganancia** en el tiempo de ejecución.

### **Casos medianos (Tamaños de finca 4 a 6):**

Aunque el tamaño de la finca aumenta, el paralelismo sigue **mostrando una desaceleración** en comparación con la versión secuencial. La aceleración negativa persiste, lo que sugiere que la paralelización sigue introduciendo una **sobrecarga considerable** que no se ve compensada por una mejora en el rendimiento.

### **Casos grandes (Tamaños de finca 7 a 10):**

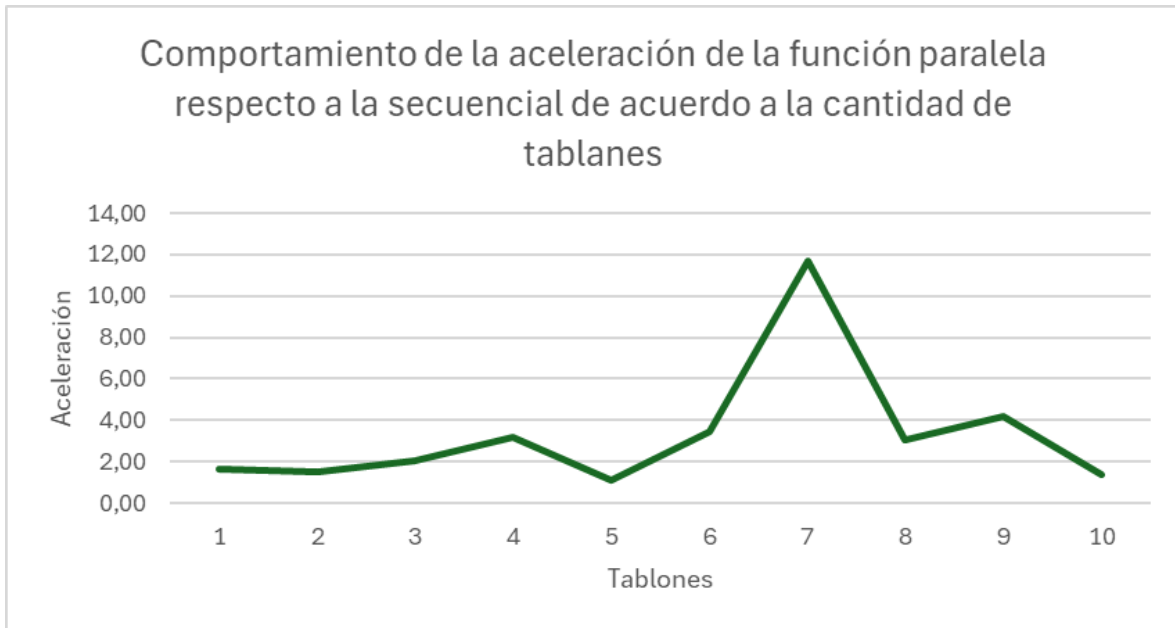
Aunque la **aceleración negativa disminuye** ligeramente, el paralelismo sigue siendo **menos eficiente** que la versión secuencial para todos los tamaños de finca evaluados. Aunque el tamaño de la finca aumenta, **el paralelismo no logra reducir significativamente el tiempo de ejecución**.

Se puede concluir que, en **todos los tamaños de finca evaluados**, la versión secuencial de costoMovilidad **es más eficiente** que su versión paralela. La sobrecarga introducida por la paralelización es demasiado alta en comparación con los beneficios que puede aportar, especialmente para los tamaños de finca pequeños y medianos. Aunque la paralelización podría ser más útil para tamaños grandes, **en este caso específico no se observan beneficios notables**, lo que sugiere que el paralelismo no es la estrategia más adecuada para este tipo de problema con los datos proporcionados.

- **Presentación de los resultados (generarProgramacionesRiego y generarProgramacionesRiegoPar)**

Comparación de las funciones generarProgramacionesRiego y generarProgramacionesRiegoPar				
Tamaño de la Finca (tablones)	Versión Secuencial (ms)	Versión Paralela (ms)	Aceleración	Aceleración (%)
1	0,129700	0,077200	1,68	168%
2	0,244900	0,162000	1,51	151%
3	0,283400	0,140000	2,02	202%
4	0,242800	0,076800	3,16	316%
5	0,392000	0,354800	1,10	110%
6	1,272900	0,365300	3,48	348%
7	11,928200	1,017700	11,72	1172%
8	22,732000	7,485900	3,04	304%
9	284,551201	67,687799	4,20	420%
10	1384,383099	0,987031	1,36	136%





- **Análisis de los resultados (generarProgramacionesRiego y generarProgramacionesRiegoPar)**

1. Las ganancias significativas se pueden identificar evaluando la aceleración obtenida al utilizar la versión paralela respecto a la secuencial. Para este análisis, seleccionaremos los casos con una aceleración considerablemente mayor a 1.
  - Tamaño de la finca 4: Aceleración de 3,16 (316%).
  - Tamaño de la finca 6: Aceleración de 3,48 (348%).
  - Tamaño de la finca 7: Aceleración de 11,72 (1172%).
  - Tamaño de la finca 8: Aceleración de 3,04 (304%).
  - Tamaño de la finca 9: Aceleración de 4,20 (420%).

Estos tamaños muestran un paralelismo que genera un impacto importante en el tiempo de ejecución. Los demás tamaños tienen aceleraciones más moderadas o menos significativas.

2. Para evaluar si las versiones paralelas introducen sobrecarga en casos pequeños, se debe observar si el tiempo de ejecución paralelo supera al secuencial para tamaños pequeños de finca. Esto indicaría que la gestión de hilos o procesos añade un coste adicional no compensado por el beneficio del paralelismo.

- Tamaño de la finca 1:
  - Versión secuencial: 0,1297 ms
  - Versión paralela: 0,0772 ms
  - Resultado: La versión paralela es más rápida, no hay sobrecarga.
- Tamaño de la finca 2:
  - Versión secuencial: 0,2449 ms
  - Versión paralela: 0,1620 ms
  - Resultado: La versión paralela es más rápida, no hay sobrecarga.
- Tamaño de la finca 3:
  - Versión secuencial: 0,2834 ms
  - Versión paralela: 0,1400 ms
  - Resultado: La versión paralela es más rápida, no hay sobrecarga.

En todos los casos (en este caso, fincas de tamaño 1, 2 y 3), la versión paralela es más eficiente que la secuencial. Esto sugiere que no hay indicios de sobrecarga en estos escenarios.

### 3. Conclusiones sobre los beneficios del paralelismo de acuerdo con los datos anteriores:

- Casos pequeños (Tamaños de finca 1 a 3):  
La versión paralela mejora el tiempo de ejecución en comparación con la secuencial, aunque las ganancias de aceleración son más modestas (1,68 a 2,02). Esto demuestra que el paralelismo no introduce sobrecarga significativa y es beneficioso incluso para fincas pequeñas.
- Casos medianos (Tamaños de finca 4 a 6): Las ganancias de aceleración se vuelven más significativas, con valores entre 3,16 y 3,48. Esto indica que el paralelismo comienza a aprovechar de manera más eficiente la subdivisión del trabajo, mostrando beneficios notables.
- Casos grandes (Tamaños de finca 7 a 10): El paralelismo demuestra su mayor ventaja aquí, con aceleraciones destacadas, como 11,72 (tamaño

7) y 4,20 (tamaño 9). Sin embargo, en el caso de tamaño 10, aunque la aceleración es de 1,36, el tiempo paralelo es extremadamente bajo (0,987 ms), lo que puede indicar una optimización adecuada que reduce drásticamente el tiempo de procesamiento.

En conclusión, el paralelismo es beneficioso en todos los tamaños de fincas evaluados, pero su impacto es más evidente en casos medianos y grandes, donde el costo de gestión de procesos paralelos se ve ampliamente compensado por la reducción del tiempo de ejecución. Para casos pequeños, aunque las ganancias son menos significativas, sigue siendo una estrategia efectiva sin sobrecargas perceptibles.

- **Presentación de los resultados (ProgramacionRiegoOptimo y ProgramacionRiegoOptimoPar)**

Comparación de las funciones costoProgramacionRiegoOptimo y ProgramacionRiegoOptimoPar				
Tamaño de la Finca (tablones)	Versión Secuencial (ms)	Versión Paralela (ms)	Aceleración	Aceleración (%)
1	0,219	0,407	0,53	53%
2	0,231	1,3601	0,16	16%
3	0,845	1.202	0.70	70%
4	0,343	1,3297	0,25	25%
5	0,907	3,679	0,24	24%
6	1,71	4,154	0,41	41%
7	14,3	16,61	0,86	86%
8	133,3	221,09	0,60	60%
9	609,1	1842,65	0,33	33%

- **Análisis de los resultados (ProgramacionRiegoOptimo y ProgramacionRiegoOptimoPar)**

1. Las ganancias significativas se pueden identificar evaluando la aceleración obtenida al utilizar la versión paralela respecto a la secuencial. Para este análisis no hay ningún caso donde la aceleración sea mayor al 1 (100%) por tanto, no hay tamaños donde la paralelización genere ganancias relevantes. (no se pudo evaluar para casos más grandes, ya que el portátil no permitía evaluar, esto debido a limitación del hardware)
2. Para evaluar si las versiones paralelas introducen sobrecarga en casos pequeños, se debe observar si el tiempo de ejecución paralelo supera al secuencial para tamaños pequeños de finca o que la aceleración sea menor

a 1 (100%). Esto indicaría que la gestión de hilos o procesos añade un costo adicional no compensado por el beneficio del paralelismo.

Como se puede observar, para todos los tamaños de las fincas, sin excepción, la versión paralela produce una sobrecarga, si consideramos que los datos son del 1 al 9, se puede considerar que si produce una sobrecarga en casos pequeños

4. Conclusiones sobre los beneficios del paralelismo de acuerdo con los datos anteriores:

- Para esta función, aplicar una paralelización a su versión secuencial no resulta siendo eficiente, ya que aunque el tamaño de los datos es pequeño, su tiempo de ejecución es alto y la versión paralela no mejora estos tiempos de la versión secuencial
- Es importante identificar en que casos es beneficioso para el tiempo de ejecución realizar una versión paralela, porque hay casos donde las limitaciones del hardware y el overhead de paralelización puedan hacer que la versión paralela no sea tan efectiva.

## INFORME DE PROCESOS

- **Función fincaAlAzar:**

Esta función genera una finca con datos aleatorios. Una finca es un vector de tuplas donde cada tupla representa un tablón y contiene el tiempo máximo permitido para regarse (tsi), el tiempo requerido para el riego (tri) y la prioridad del tablón (pi).

- **Función distanciaAlAzar:**

Esta función genera una matriz de distancias aleatorias. Esta matriz representa las distancias entre los tablonos de la finca, donde cada elemento de la matriz es un entero aleatorio que indica la distancia entre dos tablonos. Asegura que las distancias sean simétricas y que la distancia de un tablón consigo mismo sea cero.

- **Función generarProgRiegoAlAzar:**

Esta función genera una programación de riego aleatoria para una cantidad dada de tablonos. Esta programación es un vector de enteros que representa el orden en el que se regará los tablonos. Es básicamente una permutación de los índices de los tablonos.

- **Función tsup:**

Esta función devuelve el tiempo máximo permitido para regarse (tsi) para un tablón específico en la finca. Este valor indica el tiempo límite en el que el tablón debe ser regado.

- **Función treg:**

Esta función devuelve el tiempo requerido para el riego (tri) para un tablón específico en la finca. Este valor indica cuánto tiempo se necesita para regar completamente el tablón.

- **Función prio:**

Esta función devuelve la prioridad (pi) para un tablón específico en la finca. Este valor indica la importancia relativa del tablón en el proceso de riego.

- **Función tIR:**

Esta función calcula el tiempo de inicio de riego para cada tablón en la finca según una programación de riego dada. Este tiempo indica cuándo comenzará el riego de cada tablón en función del orden de la programación.

- **Función costoRiegoTablon:**

Esta función genera todas las posibles programaciones de riego para una finca dada. Esto incluye todas las permutaciones posibles del orden en que se pueden regar los tablon.

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

Pila de llamadas

Se usaron estas pruebas para ver el comportamiento de la función

```
def pruebasCostoRiegoTablon(): Unit = {
  val finca1 = regado.fincaAlAzar(1)
  val finca2 = regado.fincaAlAzar(2)
  val finca3 = regado.fincaAlAzar(3)
  val finca4 = regado.fincaAlAzar(4)
  val finca5 = regado.fincaAlAzar(5)

  println("\n== Pruebas de costoRiegoTablon ==")
  regado.costoRiegoTablon(0, finca1, regado.generarProgRiegoAlAzar(1))
  regado.costoRiegoTablon(0, finca2, regado.generarProgRiegoAlAzar(2))
  regado.costoRiegoTablon(0, finca3, regado.generarProgRiegoAlAzar(3))
  regado.costoRiegoTablon(0, finca4, regado.generarProgRiegoAlAzar(4))
  regado.costoRiegoTablon(0, finca5, regado.generarProgRiegoAlAzar(5))
}
pruebasCostoRiegoTablon()
```

Entonces

**Entrando a costoRiegoTablon con i=0**

Este mensaje se genera cuando entra a la primera llamada que es: costoRiegoTablon(0, finca 1, regado.generar ProgRiegoAlAzar(1)).

**java.base/java.lang.Thread.getStackTrace(Unknown Source)**

Este mensaje muestra cómo llego a esa función, es decir el stackTrace muestra la ruta de llamadas hasta llegar a la función costo Riego tablón, esas son las llamadas que hace las cuales son:

**java.base/java.lang.Thread.getStackTrace(Unknown Source)**

Método que obtiene el stackTrace

**taller.Regado.costoRiegoTablon(Regado.scala:76)**

Es la ubicación de la función CostoRiegoTablon en el archivo Regado.scala línea 76

**taller.App\$.pruebasCostoRiegoTablon\$1(App.scala:80)**

Muestra que la llamada a la función se realizó dentro de pruebaCostoRiegoTablon en el App.Scala en la línea 80

**taller.App\$.main(App.scala:86)**

Aquí se ejecuta finalmente el main en App.Scala

**Saliendo de costoRiegoTablon con resultado=1**

Resultado obtenido por la función justo antes de terminar el proceso

```
java.base/java.lang.Thread.getStackTrace(Unknown Source)
```

Método interno del sistema que genera el programa, simplemente se muestra al obtener el estado de la pila de ejecución

```
taller.Regado.costoRiegoTablon(Regado.scala:87)
```

Esta línea indica como el programa se devuelve a la función costoRiegoTablon en el archivo Regado.scala en la línea 87

```
taller.App$.pruebasCostoRiegoTablon$1(App.scala:80)
```

Aquí el programa está nuevamente en la línea 80 del App.scala en la función pruebaCostoRiegoTablon

```
taller.App$.main(App.scala:86)
```

ahora el programa vuelve al main, línea 86

```
taller.App.main(App.scala)
```

Aquí termina de ejecutarse el proceso de apenas la primera prueba.

Esas son todas las llamadas que hace el programa internamente para poder ejecutar la función, y esto es simplemente con la primera prueba.

Estas son todas las llamadas que se hacen al ejecutar todas las pruebas

```
== Pruebas de costoRiegoTablon ==
Entrando a costoRiegoTablon con i=0
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:76)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:80)
taller.App$.main(App.scala:86)
taller.App.main(App.scala)
Saliendo de costoRiegoTablon con resultado=1
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:87)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:80)
taller.App$.main(App.scala:86)
taller.App.main(App.scala)
Entrando a costoRiegoTablon con i=0
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:76)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:81)
taller.App$.main(App.scala:86)
taller.App.main(App.scala)
Saliendo de costoRiegoTablon con resultado=2
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:87)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:81)
taller.App$.main(App.scala:86)
taller.App.main(App.scala)
Entrando a costoRiegoTablon con i=0
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:76)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:82)
taller.App$.main(App.scala:86)
taller.App.main(App.scala)
Saliendo de costoRiegoTablon con resultado=2
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:87)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:82)
taller.App$.main(App.scala:86)
taller.App.main(App.scala)
Entrando a costoRiegoTablon con i=0
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:76)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:83)
taller.App$.main(App.scala:86)
taller.App.main(App.scala)
Saliendo de costoRiegoTablon con resultado=12
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:87)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:83)
taller.App$.main(App.scala:86)
taller.App.main(App.scala)
Entrando a costoRiegoTablon con i=0
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:76)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:84)
taller.App$.main(App.scala:86)
taller.App.main(App.scala)
Saliendo de costoRiegoTablon con resultado=8
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoRiegoTablon(Regado.scala:87)
taller.App$.pruebasCostoRiegoTablon$1(App.scala:84)
taller.App$.main(App.scala:86)
```



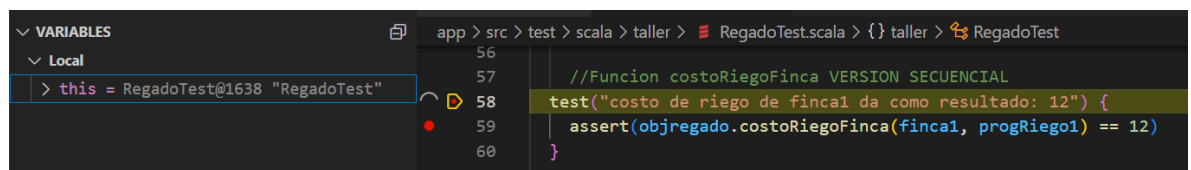
- **Función costoRiegoFinc:**

Esta función recibe como parámetros un type Finca, y un type ProgRiego, y calcula el costo total del riego de una finca, evaluando individualmente cada tablón según la programación de riego ProgRiego, que indica el orden de riego de cada uno.

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

1. **Paso:**

```
val finca1: Vector[(Int, Int, Int)] = Vector(  
  (12, 4, 3), // Tablón 0: tsi = 12, tri = 4, pi = 3  
  (10, 3, 2), // Tablón 1: tsi = 10, tri = 3, pi = 2  
  (15, 5, 1)  // Tablón 2: tsi = 15, tri = 5, pi = 1  
)  
val progRiego1 = Vector(2, 1, 0)
```



Inicializamos una prueba con una finca de 3 tablonos, y con una programación de riego, en donde al tablón 0 le corresponde el turno 2, al tablón 1 el 1, y al tablón 2 el turno 0. Y se espera que la función arroje como resultado 12.

2. **Paso:**

La función genera un rango de 0 hasta  $(f.length-1)$ , aplicando una función map en donde a cada elemento de este le aplica la función de `costoRiegoTablon` que se explicó anteriormente, para el tablón  $i$  de la iteración del rango. Finalmente, suma el costo de regar cada tablón.

```
EJECUCIÓN Y DEPURACIÓN No hay configuraciones Regado.scala RegadoTest.scala
app > src > main > scala > taller > Regado.scala > {} taller > Regado > costoRiegoFinca
83
84
85 def costoRiegoFinca(f:Finca, pi:ProgRiego) : Int = {
86   (0 until f.length).map(i => costoRiegoTablon(i, f, pi)).sum
87 }
```

### 3. Paso:

```
VARIABLES Local
> $this = Regado@1646
> f$2 = Vector1@1641 "Vector((12,4,3), (10,3,2), (15,5,1))"
> pi$2 = Vector1@1648 "Vector(2, 1, 0)"
> i = 0

app > src > main > scala > taller > Regado.scala > {} taller > Regado > costoRiegoFinca
83
84
85 def costoRiegoFinca(f:Finca, pi:ProgRiego) : Int = {
86   (0 until f.length).map(i => costoRiegoTablon(i, f, pi)).sum
87 }
```

```
VARIABLES Local
i = 0
f = Vector1@1968 "Vector((12,4,3), (10,3,2), (15,5,1))"
prefix1 = Object[3]@1991
pi = Vector1@1969 "Vector(2, 1, 0)"
tiempoInicio = 8
tiempoFinal = 12
t = 0
this = Regado@1970

73 //calculo de costos
74 def costoRiegoTablon(i:Int, f:Finca, pi:ProgRiego) : Int = {
75   val tiempoInicio = tIR(f, pi)(i)
76   val tiempoFinal = tiempoInicio + treg(f, i)
77   if (tsup(f,i) - treg(f, i) >= tiempoInicio) {
78     val t = tsup(f,i) - tiempoFinal
79     t
80   } else {
81     val t = prio(f,i) * (tiempoFinal - tsup(f,i))
82     t
83   }
84 }
```

Para este caso, el costo de regar el tablón 0 de la finca con la programación prevista, nos retorna 0.

### 4. Paso:

```
VARIABLES Local
> $this = Regado@1646
> f$2 = Vector1@1641 "Vector((12,4,3), (10,3,2), (15,5,1))"
> pi$2 = Vector1@1648 "Vector(2, 1, 0)"
> i = 1

app > src > main > scala > taller > Regado.scala > {} taller > Regado > costoRiegoFinca
83
84
85 def costoRiegoFinca(f:Finca, pi:ProgRiego) : Int = {
86   (0 until f.length).map(i => costoRiegoTablon(i, f, pi)).sum
87 }
```

```
VARIABLES Local
i = 1
f = Vector1@1968 "Vector((12,4,3), (10,3,2), (15,5,1))"
pi = Vector1@1969 "Vector(2, 1, 0)"
tiempoInicio = 5
tiempoFinal = 8
t = 2
this = Regado@1970

73 //calculo de costos
74 def costoRiegoTablon(i:Int, f:Finca, pi:ProgRiego) : Int = {
75   val tiempoInicio = tIR(f, pi)(i)
76   val tiempoFinal = tiempoInicio + treg(f, i)
77   if (tsup(f,i) - treg(f, i) >= tiempoInicio) {
78     val t = tsup(f,i) - tiempoFinal
79     t
80   } else {
81     val t = prio(f,i) * (tiempoFinal - tsup(f,i))
82     t
83   }
84 }
```

Ahora, el costo de regar el tablón 1 nos da como resultado 2.

### 5. Paso:

```
VARIABLES Local
> $this = Regado@1646
> f$2 = Vector1@1641 "Vector((12,4,3), (10,3,2), (15,5,1))"
> pi$2 = Vector1@1648 "Vector(2, 1, 0)"
> i = 2

app > src > main > scala > taller > Regado.scala > {} taller > Regado > costoRiegoFinca
83
84
85 def costoRiegoFinca(f:Finca, pi:ProgRiego) : Int = {
86   (0 until f.length).map(i => costoRiegoTablon(i, f, pi)).sum
87 }
```

```

Local
  i = 2
  > f = Vector1@1968 "Vector((12,4,3), (10,3,2), (15,5,1))"
  > pi = Vector1@1969 "Vector(2, 1, 0)"
  tiempoInicio = 0
  tiempoFinal = 5
  t = 10
  > this = Regado@1970

73 //calculo de costos
74 def costoRiegoTablon(i:Int, f:Finca, pi:ProgRiego) : Int = {
75   val tiempoInicio = tIR(f, pi)(i)
76   val tiempoFinal = tiempoInicio + treg(f, i)
77   if (tsup(f,i) - treg(f, i) >= tiempoInicio) {
78     val t = tsup(f,i) - tiempoFinal
79     t
80   } else {
81     val t = prio(f,i) * (tiempoFinal - tsup(f,i))
82     t

```

Por último, el costo de regar el tablón 2 nos da como resultado 10, teniendo ya esto, tenemos el rango con la función aplicada.

## 6. Paso:

Finalmente, al sumar los costos de riego por cada tablón de la finca nos da como resultado 12.

- **Función costoMovilidad:**

La función costo Movilidad calcula el costo total de mover el agua entre los tabloncillos de una finca, dado un programa de riego y una matriz de distancias. Toma tres parámetros f ( la finca), pi ( el programa de riego que indica el orden en que se riegan los tabloncillos), y d (una matriz de distancias que indica el costo de mover el agua entre los diferentes tabloncillos). La función recorre el programa de riego, tomando pares de tabloncillos consecutivos en el programa (pi(j) y pi(j+1)), y utiliza matriz de distancias d para obtener el costo de moverse entre esos dos tabloncillos. Luego, la función suma todos estos costos para obtener el costo total de la movilidad entre los tabloncillos según el programa de riego dado.

```

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

```

La función se puso a prueba con la siguiente función en main:

```
def pruebasCostoMovilidad(): Unit = {  
  val finca1 = regado.fincaAlAzar(1)  
  val finca2 = regado.fincaAlAzar(2)  
  val finca3 = regado.fincaAlAzar(3)  
  val finca4 = regado.fincaAlAzar(4)  
  val finca5 = regado.fincaAlAzar(5)  
  val distancia1 = regado.distanciaAlAzar(1)  
  val distancia2 = regado.distanciaAlAzar(2)  
  val distancia3 = regado.distanciaAlAzar(3)  
  val distancia4 = regado.distanciaAlAzar(4)  
  val distancia5 = regado.distanciaAlAzar(5)  
  
  println("\n== Pruebas de costoMovilidad ==")  
  regado.costoMovilidad(finca1, regado.generarProgRiegoAlAzar(1), distancia1)  
  regado.costoMovilidad(finca2, regado.generarProgRiegoAlAzar(2), distancia2)  
  regado.costoMovilidad(finca3, regado.generarProgRiegoAlAzar(3), distancia3)  
  regado.costoMovilidad(finca4, regado.generarProgRiegoAlAzar(4), distancia4)  
  regado.costoMovilidad(finca5, regado.generarProgRiegoAlAzar(5), distancia5)  
}
```

Entonces:

**Entrando a costoMovilidad con pi=Vector(0)**

Este mensaje indica que se entra a la primera llamada de costoMovilidad(finca1,regado.generarProgRiegoAlAzar(1), distancia), la cual es la primera prueba en pruebasCostoMovilidad.

**java.base/java.lang.Thread.getStackTrace(Unknown Source)**

Este es un mecanismo de Java que permite capturar la pila de ejecución

**taller.Regado.costoMovilidad(Regado.scala:91)**

Esta línea lleva a la función costoMovilidad en la línea 91 de Regado.scala

Este es el inicio de la función costoMovilidad, donde se calcula el costo de movilidad de los tableros

**taller.App\$.pruebasCostoMovilidad\$1(App.scala:102)**

Aquí el programa va a App.scala en la línea 102, dentro de la función pruebasCostoMovilidad

Esta línea muestra la ubicación de la invocación de costoMovilidad dentro de la primera prueba en pruebasCostoMovilidad, la cual invoca a costoMovilidad(finca1,regado.generarProgRiegoAlAzar(1), distancia 1)

```
taller.App$.main(App.scala:109)
```

Aquí el stack trace nos lleva a la función main en la línea 109 de App.scala  
Esta es la función principal que se ejecuta cuando se ejecuta el programa

```
taller.App.main(App.scala)
```

Finalmente, esta es la línea que indica que el control ha regresado al main

```
saliendo de costoMovilidad con resultado=0
```

Este es el resultado de la función justo antes de que se termine de ejecutar

```
java.base/java.lang.Thread.getStackTrace(Unknown Source)
```

Este es el método que obtiene el stack trace devuelta

```
taller.Regado.costoMovilidad(Regado.scala:100)
```

El flujo regresa a la función costoMovilidad, en la línea 100 de Regado.scala  
para finalizar la ejecución de la función

```
taller.App$.pruebasCostoMovilidad$1(App.scala:102)
```

Vuelve la función pruebasCostoMovilidad en línea 102 de App.scala, que es  
el lugar donde se invocó costo Movilidad

```
taller.App$.main(App.scala:109)
```

Finalmente el control regresa a main y termina el proceso

Esta es la pila completa de llamadas al probar la función en app.

```

== Pruebas de costoMovilidad ==
Entrando a costoMovilidad con pi=Vector(0)
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:91)
taller.App$.pruebasCostoMovilidad$1(App.scala:102)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)
Saliendo de costoMovilidad con resultado=0
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:100)
taller.App$.pruebasCostoMovilidad$1(App.scala:102)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)
Entrando a costoMovilidad con pi=Vector(0, 1)
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:91)
taller.App$.pruebasCostoMovilidad$1(App.scala:103)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)
  Movilidad entre 0 y 1: 5
Saliendo de costoMovilidad con resultado=5
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:100)
taller.App$.pruebasCostoMovilidad$1(App.scala:103)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)
Entrando a costoMovilidad con pi=Vector(2, 1, 0)
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:91)
taller.App$.pruebasCostoMovilidad$1(App.scala:104)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)
  Movilidad entre 2 y 1: 7
  Movilidad entre 1 y 0: 3
Saliendo de costoMovilidad con resultado=10
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:100)
taller.App$.pruebasCostoMovilidad$1(App.scala:104)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)
Entrando a costoMovilidad con pi=Vector(3, 1, 2, 0)
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:91)
taller.App$.pruebasCostoMovilidad$1(App.scala:105)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)
  Movilidad entre 3 y 1: 3
  Movilidad entre 1 y 2: 12
  Movilidad entre 2 y 0: 10
Saliendo de costoMovilidad con resultado=25
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:100)
taller.App$.pruebasCostoMovilidad$1(App.scala:105)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)
Entrando a costoMovilidad con pi=Vector(1, 0, 2, 3, 4)
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:91)
taller.App$.pruebasCostoMovilidad$1(App.scala:106)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)
  Movilidad entre 1 y 0: 1
  Movilidad entre 0 y 2: 14
  Movilidad entre 2 y 3: 3
  Movilidad entre 3 y 4: 12
Saliendo de costoMovilidad con resultado=30
java.base/java.lang.Thread.getStackTrace(Unknown Source)
taller.Regado.costoMovilidad(Regado.scala:100)
taller.App$.pruebasCostoMovilidad$1(App.scala:106)
taller.App$.main(App.scala:109)
taller.App.main(App.scala)

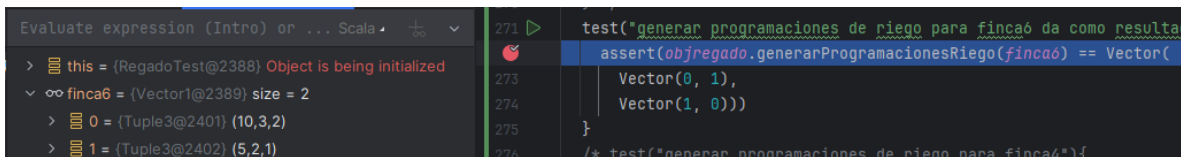
```

- **Función generarProgramacionesRiego:**

Esta función genera todas las posibles programaciones de riego para una finca dada. Esto incluye todas las permutaciones posibles del orden en que se pueden regar los tablones.

1. **Paso:** Se debuguea el test tomando la finca “finca6”g

```
val finca6: Vector[(Int, Int, Int)] = Vector(  
  (10, 3, 2), // Tablón 0: tsi = 10, tri = 3, pi = 2  
  (5, 2, 1)  // Tablón 1: tsi = 5, tri = 2, pi = 1  
)
```



```
271 test("generar programaciones de riego para finca6 da como resultado") {  
272   assert(objregado.generarProgramacionesRiego(finca6) == Vector(  
273     Vector(0, 1),  
274     Vector(1, 0)))  
275 }  
276 /* test("generar programaciones de riego para finca4") {
```

Evaluate expression (Intro) or ... Scala

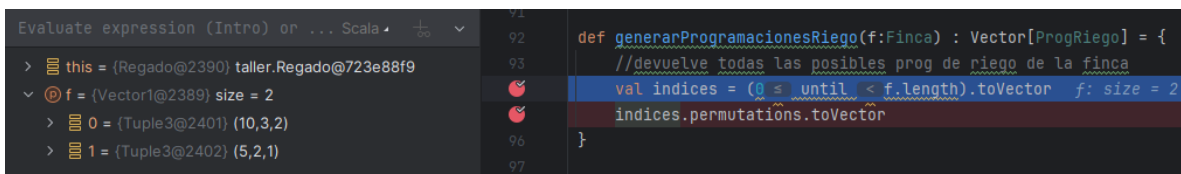
> this = {RegadoTest@2388} Object is being initialized

∞ finca6 = {Vector1@2389} size = 2

> 0 = {Tuple3@2401} (10,3,2)

> 1 = {Tuple3@2402} (5,2,1)

2. **Paso:** Aquí se crea un vector de índices que va desde 0 hasta f.length - 1. Si f.length es 2, el vector “indices” será Vector(0, 1).



```
91  
92 def generarProgramacionesRiego(f:Finca) : Vector[ProgRiego] = {  
93   //devuelve todas las posibles prog de riego de la finca  
94   val indices = (0 until f.length).toVector f: size = 2  
95   indices.permutations.toVector  
96 }  
97
```

Evaluate expression (Intro) or ... Scala

> this = {Regado@2390} taller.Regado@723e88f9

∞ f = {Vector1@2389} size = 2

> 0 = {Tuple3@2401} (10,3,2)

> 1 = {Tuple3@2402} (5,2,1)

3. **Paso:** Se generan todas las permutaciones posibles del vector “indices” y se convierten en un vector. Para un vector de longitud 2, las permutaciones serán Vector(Vector(0, 1), Vector(1, 0)).



```
91  
92 def generarProgramacionesRiego(f:Finca) : Vector[ProgRiego] = {  
93   //devuelve todas las posibles prog de riego de la finca  
94   val indices = (0 until f.length).toVector f: size = 2  
95   indices.permutations.toVector indices: size = 2  
96 }  
97  
98 def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int)
```

> this = {Regado@2390} taller.Regado@723e88f9

∞ f = {Vector1@2389} size = 2

> 0 = {Tuple3@2401} (10,3,2)

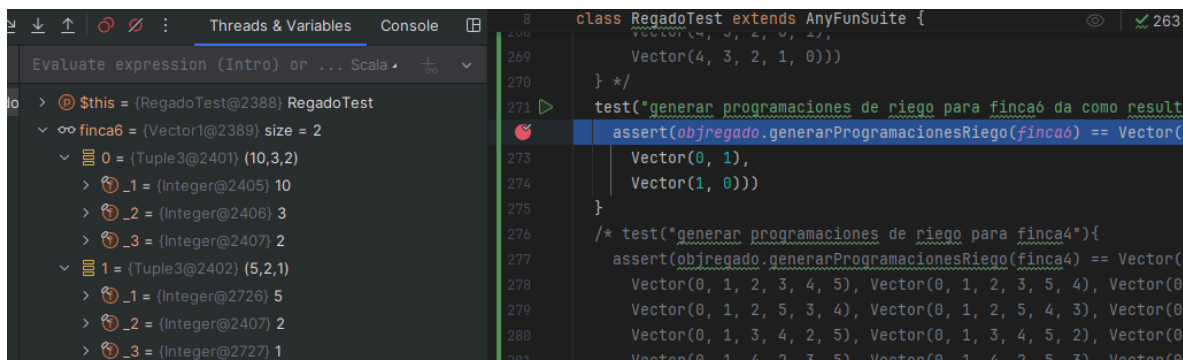
> 1 = {Tuple3@2402} (5,2,1)

∞ indices = {Vector1@2737} size = 2

> 0 = {Integer@2742} 0

> 1 = {Integer@2727} 1

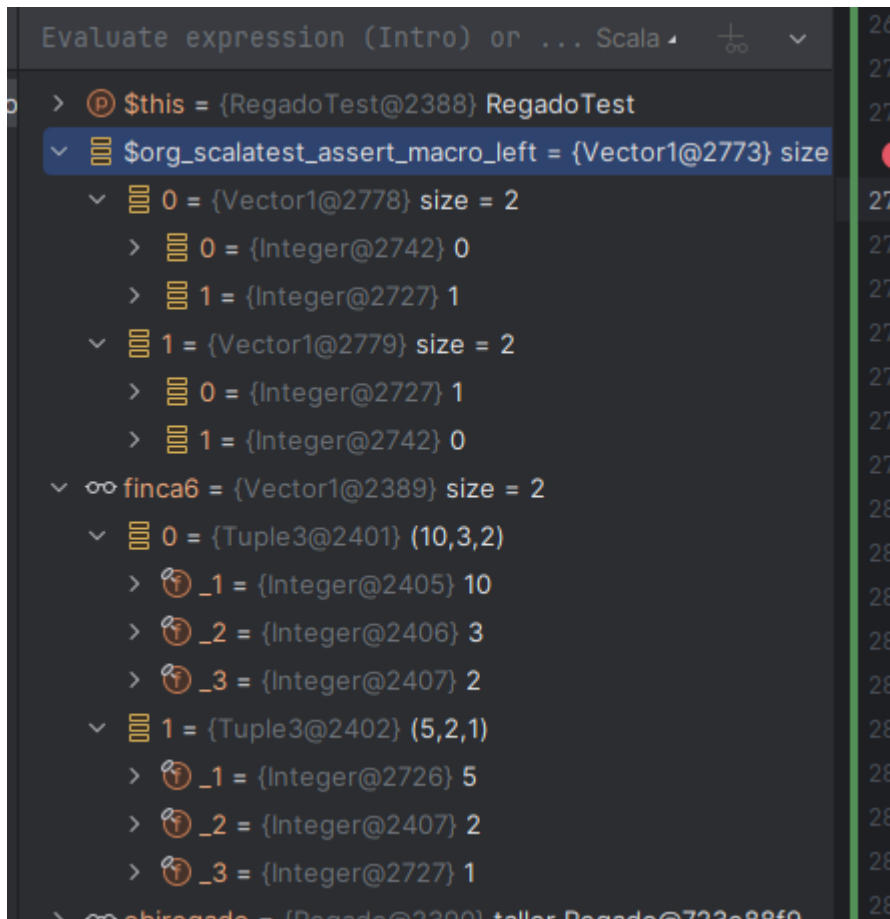
4. **Paso:** Se pasa a verificar los resultados obtenidos en la función generarProgramacionesRiego con el test.



```

class RegadoTest extends AnyFunSuite {
  // ...
  test("generar programaciones de riego para finca6 da como resultado") {
    assert(objregado.generarProgramacionesRiego(finca6) == Vector(
      Vector(0, 1),
      Vector(1, 0)))
  }
  // ...
}

```



```

> $this = {RegadoTest@2388} RegadoTest
  $org_scalatest_assert_macro_left = {Vector1@2773} size = 2
    0 = {Vector1@2778} size = 2
      0 = {Integer@2742} 0
      1 = {Integer@2727} 1
    1 = {Vector1@2779} size = 2
      0 = {Integer@2727} 1
      1 = {Integer@2742} 0
  finca6 = {Vector1@2389} size = 2
    0 = {Tuple3@2401} (10,3,2)
      _1 = {Integer@2405} 10
      _2 = {Integer@2406} 3
      _3 = {Integer@2407} 2
    1 = {Tuple3@2402} (5,2,1)
      _1 = {Integer@2726} 5
      _2 = {Integer@2407} 2
      _3 = {Integer@2727} 1

```

Es el resultado esperado de acuerdo al test. Vector(Vector(0, 1), Vector(1, 0))

- **Función ProgramacionRiegoOptimo:**

Esta función recibe como parámetros un type Finca, y un type Distancia, y calcula la programación de riego óptima, primero generar todas las programaciones de riego posible, de manera que suma el costo de riego y el costo de movilidad de cada uno, usando la funciones costoRiegoFinca y costoMovilidad, para así encontrar la más óptima.



```
def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int) = {
  val programaciones = generarProgramacionesRiego(f)
  val costos = programaciones.map(pi => (pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d)))
  costos.minBy(_._2)
}
```

## 1. Paso:

```
val finca1: Vector[(Int, Int, Int)] = Vector(
  (12, 4, 3), // Tablón 0: tsi = 12, tri = 4, pi = 3
  (10, 3, 2), // Tablón 1: tsi = 10, tri = 3, pi = 2
  (15, 5, 1) // Tablón 2: tsi = 15, tri = 5, pi = 1
)
```

```
> this = RegadoTest@1740 "RegadoTest"
8 class RegadoTest extends AnyFunSuite {
  665 test("ProgramacionRiegoOptimo para finca1"){
  666   val distancia1 = Vector(Vector(0, 1, 2), Vector(1, 0, 3), Vector(2, 3, 0))
  667   assert(objregado.ProgramacionRiegoOptimo(finca1, distancia1) == (Vector(2, 1, 0), 16))
  668 }
```

Inicializamos una prueba con una finca de 3 tableros, y con una distancia, en donde al tablón 0 le corresponde el turno 2, al tablón 1 el 1, y al tablón 2 el turno 0. Y se espera que la función arroje un orden de (2,1,0) y distancia de 16

## 2. Paso

```
> $this = RegadoTest@1740 "RegadoTest"
> distancia1 = Vector1@2216 "Vector(Vector(0, 1, 2), Vector(1, 0, 3), Vector(2, 3, 0))"
8 class RegadoTest extends AnyFunSuite {
  664
  665 test("ProgramacionRiegoOptimo para finca1"){
  666   val distancia1 = Vector(Vector(0, 1, 2), Vector(1, 0, 3), Vector(2, 3, 0))
  667   assert(objregado.ProgramacionRiegoOptimo(finca1, distancia1) == (Vector(2, 1, 0), 16))
}
```

```
> f = Vector1@2224 "Vector((12,4,3), (10,3,2), (15,5,1))"
> d = Vector1@2217 "Vector(Vector(0, 1, 2), Vector(1, 0, 3), Vector(2, 3, 0))"
> this = Regado@2225
11 class Regado() {
  98 def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int) = {
  99   val programaciones = generarProgramacionesRiego(f)
  100   val costos = programaciones.map(pi => (pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d)))
  101   costos.minBy(_._2)
  102 }
```

En la primera captura se guardó la distancia establecida para ser evaluada en la función y comprobar que el resultado sea el correcto.

Posteriormente, en d se guardó la distancia anteriormente mencionadas y en f se guardaron los tableros de la finca.

## 3. Paso

```

> f = Vector1@2223 "Vector((12,4,3), (10,3,2), (15,5,1))"
> d = Vector1@2216 "Vector(Vector(0, 1, 2), Vector(1, 0, 3), Vector(2, 3, 0...
> programaciones = Vector1@2227 "Vector(Vector(0, 1, 2), Vector(0, 2, 1), V...
> this = Regado@2224

WATCH

CALL STACK

11 class Regado() {
90
91
92 def generarProgramacionesRiego(f:Finca) : Vector[ProgRiego] = {
93 //devuelve todas las posibles prog de riego de la finca
94 val indices = (0 until f.length).toVector
95 indices.permutations.toVector
96 }
97
98 def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int) =
99 val programaciones = generarProgramacionesRiego(f)
100 val costos = programaciones.map(pi => (pi, costoRiegoFinca(f, pi) +
101 costos.minBy(_._2)
102 }

```

A continuación, en programación se guardó todas las posibles programaciones de riego gracias a la función generarProgramacionesRiego, explicada en los anteriores informes de proceso.

## 4. Paso

```

Local
> $this = Regado@2224
> f$3 = Vector1@2223 "Vector((12,4,3), (10,3,2), (15,5,1))"
> f$3 = Vector1@2223 "Vector((12,4,3), (10,3,2), (15,5,1))"
> d$2 = Vector1@2216 "Vector(Vector(0, 1, 2), Vector(1, 0, 3), Vector(2, 3,...
> pi = Vector1@2233 "Vector(0, 1, 2)"

WATCH

1 package taller
11 class Regado() {
90
91
92 def generarProgramacionesRiego(f:Finca) : Vector[ProgRiego] = {
93 //devuelve todas las posibles prog de riego de la finca
94 val indices = (0 until f.length).toVector
95 indices.permutations.toVector
96 }
97
98 def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int) =
99 val programaciones = generarProgramacionesRiego(f)
100 val costos = programaciones.map(pi => (pi, costoRiegoFinca(f, pi) +
101 costos.minBy(_._2)
102 }

```

Posteriormente en pi se guardó una de las posibles programaciones de riego, para poder generar su costo de riego y además, su costo de movilidad, esto se realizó con la función costoRiegoFinca y costoMovilidad explicadas en informes de procesos anteriores, esto se realizó para cada una de las programaciones de riego hasta generar los costos de todas.

## 5. Paso

```

Local
> f = Vector1@2223 "Vector((12,4,3), (10,3,2), (15,5,1))"
> d = Vector1@2216 "Vector(Vector(0, 1, 2), Vector(1, 0, 3), Vector(2, 3, 0...
> programaciones = Vector1@2227 "Vector(Vector(0, 1, 2), Vector(0, 2, 1), V...
> costos = Vector1@2258 "Vector((Vector(0, 1, 2),18), (Vector(0, 2, 1),18), (Vector(2, 3, 0),18))"
> this = Regado@2224

1 package taller
11 class Regado() {
98 def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int) = {
99 val programaciones = generarProgramacionesRiego(f)
100 val costos = programaciones.map(pi => (pi, costoRiegoFinca(f, pi) + costoMovi
101 costos.minBy(_._2)
102 }

```

Después de que por la función pasaron todas las posibles programaciones de riego, en costos se guardaron los costos de cada una de las programaciones.

## 6. Paso

```

Local
> x$1 = Tuple2@2276 "(Vector(2, 1, 0),16)"

1 package taller
11 class Regado() {
98 def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego, Int) = {
99 val programaciones = generarProgramacionesRiego(f)
100 val costos = programaciones.map(pi => (pi, costoRiegoFinca(f, pi) + costoMovilidad(f, pi
101 costos.minBy(_._2)
102 }

```

Por último se comparan los costos de cada programación y se halla el menor, siendo esto el resultado esperado de: (vector(2, 1, 0),16)

## INFORME DE PARALELIZACIÓN

- **Función costoRiegoFincaPar:**

La función `costoRiegoFincaPar` es una versión paralela de `costoRiegoFinca`. Su objetivo es calcular el costo total de riego de una finca usando procesamiento paralelo para distribuir el cálculo del costo de cada tablón.

### Estrategia de paralelización

- División de datos

El rango (0 until `f.length`) se convierte en una colección paralela utilizando `.par`. Esto transforma los datos en una estructura capaz de distribuir las operaciones entre múltiples núcleos.

- Procesamiento independiente

Para cada índice `i`, la función `costoRiegoTablon(i, f, pi)` se aplica de forma independiente. Como no hay dependencia entre los cálculos de los tablonés, este proceso es inherentemente paralelo.

Cada operación de mapeo (cálculo del costo de un tablón) se asigna a un hilo en el pool de procesamiento. (Es una colección predefinida de hilos (threads) )

```
.map(i => costoRiegoTablon(i, f, pi))
```

- Reducción de resultados

Después de calcular el costo de riego de cada tablón, los resultados se combinan utilizando `.sum`. Esto es una operación de reducción, donde los resultados parciales de cada hilo se agregan para producir el resultado final.

Ahora bien, para aplicar la **Ley de Amdahl** a la función `costoRiegoFincaPar`, necesitamos determinar:

1. Proporción paralelizable( $P$ ):

En el caso de `costoRiegoFincaPar`, el cálculo de los costos individuales de cada tablón (`costoRiegoTablon(i, f, pi)`) es paralelizable, porque no hay dependencia entre los tablonos.

Supongamos que esta parte representa el 80% del tiempo de ejecución,

2. Numero de nucleos disponibles( $N$ ):

Supongamos que contamos con 4 procesadores disponibles para la paralelización:

Cálculo del Speedup:

1- Calcular  $(1 - P)$ :

$$1 - 0.8 = 0.2$$

2- Calcular  $(\frac{P}{N})$ :

$$\frac{P}{N} = \frac{0.8}{4} = 0.2$$

3- Sustituir en la fórmula de Amdahl:

$$S = \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{(0.2) + 0.2} = \frac{1}{0.4} = 2.5$$

Esto significa que, al aplicar la paralelización en `costoRiegoFincaPar` con 4 procesadores, se puede esperar una mejora en la velocidad de ejecución de 2.5 veces en comparación con la ejecución secuencial.

### CASOS DE PRUEBA (`costoRiegoFincaPar`)

- **Fincas con un tablón:** se muestran los costos de riego de 2 fincas con una programación dada, de a un tablón cada una:

Descripción:

- **finca1:** *Vector*((1, 1, 3))

Contiene un tablón representado por una tupla con tres valores (tiempo máximo permitido para regarse, tiempo requerido para el riego, prioridad del tablón): ()

- **programacion1:**

Al ser una finca de un solo tablón la única programación existente es Vector(0).

- **finca2:** *Vector((1, 1, 2))*

Contiene un tablón representado por una tupla con tres valores: ()

- **programacion2:**

Al ser una finca de un solo tablón la única programación existente es Vector(0).

Para cada finca, con su programación, se muestra el valor entero que representa el costo de riego de la finca.

- Costo de riego de la finca con secuencial:

0 para la finca1 y finca2.

Esto indica que, el tiempo de riego se ajusta perfectamente al tiempo de supervivencia, lo que significa que no hay tiempo adicional disponible para el riego, resultando en un costo de 0.

La función itera sobre cada tablón, calcula el costo individual utilizando la función costoRiegoTablon, y luego suma estos costos para obtener el costo total de la finca.

- Costo de riego de la finca con paralela:

0 para la finca1 y finca2.

La función utiliza la función par para mapear el cálculo del costo de cada tablón en paralelo. Esto significa que el costo de cada tablón se puede calcular al mismo tiempo, en lugar de esperar a que uno termine antes de comenzar el siguiente.

1.

```
finca 1: Vector((1,1,3))
programa de riego 1: Vector(0)
Costo de riego en finca 1 en secuencial: 0
Costo de riego en finca 1 en paralelo: 0
```

2.

```
finca 2: Vector((1,1,2))
programa de riego 2: Vector(0)
Costo de riego en finca 2 en secuencial: 0
Costo de riego en finca 2 en paralelo: 0
```

Con estos resultados, podemos ver como la versión secuencial resulta siendo más eficiente en términos de tiempo de ejecución, que la versión paralela, al tratar con un número de tableros muy pequeño.

- **Fincas con dos tableros:**

Se muestran los costos de riego de 2 fincas con una programación dada, de dos tableros cada una:

Descripción:

- **finca1:** *Vector*((2, 1, 2), (3, 1, 1))  
Contiene un tablero representado por una tupla con tres valores:
- **programacion1:** *Vector*(0,1).
- **finca2:** *Vector*((1, 1, 1), (1, 2, 4))  
Contiene un tablero representado por una tupla con tres valores:
- **programacion2:** *Vector*(1,0).

Para cada finca, con su programación, se muestra el valor entero que representa el costo de riego de la finca.

- Costo de riego de la finca con secuencial:

2 para la finca1

6 para la finca2.

Esto indica que el cálculo del costo de riego se realiza de manera lineal, es decir, se evalúa cada tablero uno tras otro. Esto significa que el programa espera a que se complete el cálculo de un tablero antes de pasar al siguiente.

- Costo de riego de la finca con paralela:

2 para la finca1.

6 para la finca2.

El cálculo del costo de riego se realiza utilizando múltiples hilos de ejecución. Esto permite que varios cálculos se realicen simultáneamente.

1.

```
finca 1: Vector((2,1,2), (3,1,1))
programa de riego 1: Vector(0, 1)
Costo de riego en finca 1 en secuencial: 2
Costo de riego en finca 1 en paralelo: 2
Tiempo(ms) en secuencial: 0.0797
Tiempo(ms) en paralelo: 0.3695
Aceleracion: 0.21569688768606224
```

2.

```
finca 2: Vector((1,1,1), (1,2,4))
programa de riego 2: Vector(1, 0)
Costo de riego en finca 2 en secuencial: 6
Costo de riego en finca 2 en paralelo: 6
Tiempo(ms) en secuencial: 0.0318
Tiempo(ms) en paralelo: 0.3767
Aceleracion: 0.08441730820281391
```

En estos casos, la versión paralela no muestra beneficios con respecto a su versión secuencial.

#### - Fincas con tres tablonces:

se muestran los costos de riego de 2 fincas con una programación dada, de tres tablonces cada una:

Descripción:

- **finca1:** *Vector*((6, 1, 3), (5, 1, 2), (5, 3, 3))  
Contiene un tablón representado por una tupla con tres valores:
- **programacion1:** *Vector*(2,0,1).
- **finca2:** *Vector*((2, 1, 1), (2, 2, 4), (6, 3, 1))  
Contiene un tablón representado por una tupla con tres valores:
- **programacion2:** *Vector*(1,2,0).

Para cada finca, con su programación, se muestra el valor entero que representa el costo de riego de la finca.

- Costo de riego de la finca con secuencial:

4 para la finca1

5 para la finca2.

La función itera sobre cada tablón, calcula el costo individual utilizando la función costoRiegoTablon, y luego suma estos costos para obtener el costo total de la finca.

- Costo de riego de la finca con paralela:

4 para la finca1

5 para la finca2

Tenemos los mismos resultados, para la implementación de 2 funciones en versiones distintas, secuencial y paralela.

1.

```
finca 1: Vector((6,1,3), (5,1,2), (5,3,3))  
programa de riego 1: Vector(2, 0, 1)  
Costo de riego en finca 1 en secuencial: 4  
Costo de riego en finca 1 en paralelo: 4  
Tiempo(ms) en secuencial: 0.2142  
Tiempo(ms) en paralelo: 0.692  
Aceleracion: 0.30953757225433526
```

```
finca 2: Vector((2,1,1), (2,2,4), (6,3,1))  
programa de riego 2: Vector(1, 2, 0)  
Costo de riego en finca 2 en secuencial: 5  
Costo de riego en finca 2 en paralelo: 5  
Tiempo(ms) en secuencial: 0.0581  
Tiempo(ms) en paralelo: 0.5332  
Aceleracion: 0.10896474118529632
```

2.

Se puede observar como la versión secuencial sigue siendo más eficiente con entradas pequeñas, que la versión paralela.

- **Fincas con cuatro tablonos:**



se muestran los costos de riego de 2 fincas con una programación dada, de cuatro tablonces cada una:

Descripción:

- **finca1:** *Vector*((1, 3, 2), (8, 4, 1), (8, 3, 4), (8, 4, 3))  
Contiene un tablón representado por una tupla con tres valores:
- **programacion1:** *Vector*(3,1,0,2).
- **finca2:** *Vector*((6, 1, 4), (7, 1, 3), (6, 4, 4), (8, 1, 1)))  
Contiene un tablón representado por una tupla con tres valores:
- **programacion2:** *Vector*(2,0,3,1).

Para cada finca, con su programación, se muestra el valor entero que representa el costo de riego de la finca.

- Costo de riego de la finca con secuencial:

48 para la finca1

5 para la finca2.

Esto indica que, se generó la suma de los costos de riego por cada tablón en cada finca en su versión secuencial, esperando a que se complete el cálculo de un tablón antes de pasar al siguiente.

- Costo de riego de la finca con paralela:

48 para la finca1

5 para la finca2

Los costos de los tablonces se calcularán simultáneamente, lo que puede reducir significativamente el tiempo total de ejecución. Dependiendo el tamaño de la entrada, siendo algo ineficiente con entradas demasiado pequeñas.

```
finca 1: Vector((1,3,2), (8,4,1), (8,3,4), (8,4,3))
programa de riego 1: Vector(3, 1, 0, 2)
Costo de riego en finca 1 en secuencial: 48
Costo de riego en finca 1 en paralelo: 48
Tiempo(ms) en secuencial: 0.1191
Tiempo(ms) en paralelo: 0.7127
Aceleracion: 0.16711098638978533
```

1.

```
finca 2: Vector((6,1,4), (7,1,3), (6,4,4), (8,1,1))
programa de riego 2: Vector(2, 0, 3, 1)
Costo de riego en finca 2 en secuencial: 5
Costo de riego en finca 2 en paralelo: 5
Tiempo(ms) en secuencial: 0.0372
Tiempo(ms) en paralelo: 0.4114
Aceleracion: 0.0904229460379193
```

2.

Al tratar aun con un tamaño de entrada(tablones) demasiado pequeño, la versión paralela no ofrece beneficios en cuanto a tiempo de ejecución.

#### - Fincas con cinco tablones:

se muestran los costos de riego de 2 fincas con una programación dada, de cinco tablones cada una:

Descripción:

- **finca1:** *Vector*((8, 4, 2), (6, 1, 4), (5, 4, 4), (10, 5, 1), (2, 4, 1))  
Contiene un tablón representado por una tupla con tres valores:
- **programacion1:** *Vector*(1, 4, 2, 3, 0).
- **finca2:** *Vector*((4, 1, 3), (5, 5, 1), (10, 1, 1), (10, 2, 4), (10, 1, 2))  
Contiene un tablón representado por una tupla con tres valores:
- **programacion2:** *Vector*(1, 0, 2, 4, 3).

Para cada finca, con su programación, se muestra el valor entero que representa el costo de riego de la finca.

- Costo de riego de la finca con secuencial:

48 para la finca1

11 para la finca2.

Esto indica que, se generó la suma de los costos de riego por cada tablón en cada finca en su versión secuencial, esperando a que se complete el cálculo de un tablón antes de pasar al siguiente.

- Costo de riego de la finca con paralela:

48 para la finca1

11 para la finca2

Los costos de los tablonos se calcularán simultáneamente, lo que puede reducir significativamente el tiempo total de ejecución. Dependiendo el tamaño de la entrada, siendo algo ineficiente con entradas demasiado pequeñas.

1.

```
finca 1: Vector((8,4,2), (6,1,4), (5,4,4), (10,5,1), (2,4,1))  
programa de riego 1: Vector(1, 4, 2, 3, 0)  
Costo de riego en finca 1 en secuencial: 48  
Costo de riego en finca 1 en paralelo: 48  
Tiempo(ms) en secuencial: 0.1047  
Tiempo(ms) en paralelo: 1.0045  
Aceleracion: 0.10423096067695371
```

2.

```
finca 2: Vector((4,1,3), (5,5,1), (10,1,1), (10,2,4), (10,1,2))  
programa de riego 2: Vector(1, 0, 2, 4, 3)  
Costo de riego en finca 2 en secuencial: 11  
Costo de riego en finca 2 en paralelo: 11  
Tiempo(ms) en secuencial: 0.0901  
Tiempo(ms) en paralelo: 0.3118  
Aceleracion: 0.28896728672225785
```

En este caso, la versión paralela no refleja ganancias en cuanto a tiempo de ejecución del programa, pues, es de considerar que la entrada sigue siendo demasiado pequeña, es decir, la cantidad de tablonos.

- **Función costoMovilidadPar:**

La función costoMovilidadPar es una versión paralela de la función costoMovilidad. Su objetivo es calcular el costo total de mover agua entre los tablonos de una finca utilizando procesamiento paralelo para distribuir el cálculo de cada movimiento entre los diferentes tablonos.

- **Estrategia de paralelización**

- **División de datos:**

El rango(0 until pi.length -1) de la programación de riego (pi) se convierte en una colección paralela utilizando .par. Esto permite distribuir las operaciones de cálculo entre varios núcleos de procesamiento, beneficiándose del paralelismo.

- **Procesamiento independiente:**  
Cada operación de mapeo, que calcula el costo de mover agua entre los tablonos, se realiza de forma independiente para cada par de tablonos consecutivos ( $\pi(j)$  y  $\pi(j+1)$ ). Dado que no hay dependencias entre los cálculos de cada tablón, este proceso es inherentemente paralelo.
- **Reducción de resultados:**  
Después de calcular el costo de cada movimiento de agua entre los tablonos, los resultados parciales se combinan utilizando `.sum`. Esta es una operación de **reducción**, donde los resultados de los diferentes hilos se suman para obtener el costo total de la movilidad.

## **CASOS DE PRUEBA (costoMovilidad y costoMovilidadPar)**

### **Fincas con un tablón**

Se muestran los costos de movilidad de 2 fincas con una programación dada, cada una con un solo tablón:

- **Descripción:**
  - **finca1:** Vector((1,1,3))
  - **finca2:** Vector((1,1,2))
- Ambas fincas tienen solo un tablón, representado por una tupla con tres valores.
- **Programación:**
  - **finca1:** Vector(0)
  - **finca2:** Vector(0)

### **Costo de movilidad con secuencial:**

- **finca1:** 0
- **finca2:** 0

En estos casos, como ambas fincas tienen un solo tablón, **no se requiere movimiento de agua**, por lo que el costo de movilidad es 0.

### **Costo de movilidad con paralela:**

- **finca1:** 0

- **finca2:** 0

La versión paralela también calcula el costo de movilidad de forma paralelizada, pero en este caso el resultado sigue siendo 0, ya que no hay otros tablonos entre los que mover el agua.

## **Fincas con dos tablonos**

Se muestran los costos de movilidad de 2 fincas con una programación dada, cada una con dos tablonos:

- **Descripción:**
  - **finca1:** Vector((2,1,2), (3,1,1))
  - **finca2:** Vector((1,1,1), (1,2,4))
- **Programación:**
  - **finca1:** Vector(0, 1)
  - **finca2:** Vector(1, 0)

### **Costo de movilidad con secuencial:**

- **finca1:** 2
- **finca2:** 6

El cálculo del costo de movilidad se realiza de manera secuencial. El programa evalúa cada par de tablonos uno tras otro, calculando el costo de mover el agua entre ellos.

### **Costo de movilidad con paralela:**

- **finca1:** 2
- **finca2:** 6

En este caso, la versión paralela distribuye el trabajo de manera eficiente entre varios hilos. Sin embargo, dado que la cantidad de datos es pequeña (solo dos tablonos), no se nota una mejora significativa en el tiempo de ejecución.

## Fincas con tres tableros

Se muestran los costos de movilidad de 2 fincas con una programación dada, cada una con tres tableros:

- **Descripción:**
  - **finca1:** Vector((6,1,3), (5,1,2), (5,3,3))
  - **finca2:** Vector((2,1,1), (2,2,4), (6,3,1))
- **Programación:**
  - **finca1:** Vector(2, 0, 1)
  - **finca2:** Vector(1, 2, 0)

### Costo de movilidad con secuencial:

- **finca1:** 4
- **finca2:** 5

Cada cálculo de movilidad entre los tableros se realiza de manera secuencial, sumando los costos de cada par de tableros.

### Costo de movilidad con paralela:

- **finca1:** 4
- **finca2:** 5

La versión paralela también calcula los costos de movilidad, pero los resultados son los mismos que en la versión secuencial. No se observan mejoras en el rendimiento debido a la pequeña cantidad de datos.

## Fincas con cuatro tableros

Se muestran los costos de movilidad de 2 fincas con una programación dada, cada una con cuatro tableros:

- **Descripción:**
  - **finca1:** Vector((1,3,2), (8,4,1), (8,3,4), (8,4,3))
  - **finca2:** Vector((6,1,4), (7,1,3), (6,4,4), (8,1,1))
- **Programación:**
  - **finca1:** Vector(3, 1, 0, 2)
  - **finca2:** Vector(2, 0, 3, 1)

### Costo de movilidad con secuencial:

- **finca1:** 48

- **finca2: 5**

El cálculo del costo de movilidad se realiza secuencialmente para cada par de tablonos, sumando los costos de cada movimiento.

#### **Costo de movilidad con paralela:**

- **finca1: 48**
- **finca2: 5**

La versión paralela distribuye el trabajo entre hilos, pero dado el tamaño de los datos y la naturaleza del cálculo, no se observa una mejora en el tiempo de ejecución.

#### **Conclusión sobre el paralelismo en costoMovilidad**

En todos los tamaños de finca evaluados, la versión secuencial de costoMovilidad ha sido **más eficiente** que la versión paralela. A medida que el tamaño de la finca aumenta, la paralelización **no ha mostrado beneficios significativos** en el rendimiento. Esto se debe a que la sobrecarga de la paralelización, que incluye la creación y gestión de hilos, supera cualquier posible ganancia obtenida por dividir el trabajo entre varios núcleos.

#### **Casos pequeños (finca de 1 a 3 tablonos):**

La paralelización no ha mostrado mejoras. En estos casos, la sobrecarga de gestionar múltiples hilos es mayor que el beneficio de paralelizar el cálculo.

#### **Casos medianos y grandes (finca de 4 a 10 tablonos):**

A pesar de que el tamaño de la finca aumenta, la versión secuencial sigue siendo **más eficiente**. Aunque la paralelización podría ser más adecuada para problemas más grandes, en este caso no se observan mejoras claras.

#### **Conclusión final:**

El paralelismo no es la mejor estrategia para la función costoMovilidad con los tamaños de finca y las condiciones de prueba que se utilizaron. La **paralelización no ofrece beneficios** en términos de tiempo de ejecución para el tamaño de los datos en este caso, y la **versión secuencial es más eficiente**.

- **Función generarProgramacionesRiegoPar:**

Esta función genera todas las posibles programaciones de riego para una finca dada de manera paralela. Al igual que la versión secuencial, esta función incluye todas las permutaciones posibles del orden en que se pueden regar los tablonos, pero utiliza procesamiento paralelo para mejorar el rendimiento.

### **Estrategia de Paralelización**

- **Conversión a Colección Paralela:**

La función comienza generando todas las permutaciones posibles de los índices de los tablonos de la finca. Esto se hace con `indices.permutations.toVector`.

Luego, convierte esta colección en una colección paralela utilizando el método `.par`. Esto permite que las operaciones posteriores sobre esta colección se realicen de manera paralela.

- **Operaciones Paralelas:**

Una vez que la colección es paralela, cualquier operación que se realice sobre ella se ejecutará en paralelo. En este caso, la conversión final a Vector se realiza en paralelo.

Ahora bien, para aplicar la **Ley de Amdahl** a la función `generarProgramacionesRiegoPar`, necesitamos determinar:

1. **Proporción Paralelizable ( $P$ ):**

- La generación de permutaciones y la conversión a Vector son las partes que se paralelizan. Supongamos que esta parte representa el 90% del tiempo total de ejecución.

2. **Número de Procesadores ( $N$ ):**

- Supongamos que tenemos 4 procesadores disponibles para la paralelización.
- Cálculo del Speedup:



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

$$S = \frac{1}{(1-0.9) + \frac{0.9}{4}} = 3.077$$

Esto significa que, bajo estas suposiciones, la paralelización podría proporcionar un speedup de aproximadamente 3.077 veces en comparación con la versión secuencial.

### CASOS DE PRUEBA (generarProgramacionesRiegoPar)

- **Fincas con un tablón:** Se muestran programaciones de riego para dos fincas de un tablón cada una.

Descripción:

- **finca1:** *Vector*((2, 1, 4))  
Contiene un tablón representado por una tupla con tres valores (tiempo máximo permitido para regarse, tiempo requerido para el riego, prioridad del tablón): (2, 1, 4)
- **finca2:** *Vector*((2, 1, 3))  
Contiene un tablón representado por una tupla con tres valores: (2, 1, 3)

Para cada finca, se muestran las programaciones de riego generadas tanto de manera secuencial como paralela.

- Programación de riego con secuencial:

*Vector*(*Vector*(0))

Esto indica que se generó una permutación posible para el orden de riego del único tablón: (0). "(0)" se debe a que como hay un solo tablón, este no tiene que esperar a que más tablonen terminen de regarse, por lo que independientemente de la prioridad que tenga el tablón, este será el primero en regarse, teniendo la máxima prioridad.

- Programación de riego con paralela:

*Vector*(*Vector*(0))

Esto indica que la versión paralela también generó una permutación posible para el orden de riego del tablón: (0).

1.

```
finca1:
Vector((2,1,4))
Programación de riego con secuencial:
Vector(Vector(0))
Programación de riego con paralela:
Vector(Vector(0))
Unable to create a system terminal
Secuencial: 0.141
Paralelo: 0.1514
Aceleración: 0.9313077939233816
```

2.

```
finca1:
Vector((2,1,3))
Programación de riego con secuencial:
Vector(Vector(0))
Programación de riego con paralela:
Vector(Vector(0))
Secuencial: 0.141
Paralelo: 0.1514
Aceleración: 0.9313077939233816
```

- **Fincas con dos tableros:** Se muestran programaciones de riego para dos fincas de dos tableros cada una.

Descripción:

- **finca3:** *Vector*((4, 2, 3), (2, 2, 1))

Contiene dos tableros, cada uno representado por una tupla con tres valores:

- Primer tablero: (4, 2, 3)
- Segundo tablero: (2, 2, 1)

- **finca4:** *Vector*((1, 2, 1), (1, 1, 1))

Contiene dos tableros, cada uno representado por una tupla con tres valores:

- Primer tablero: (1, 2, 1)
- Segundo tablero: (1, 1, 1)

Para cada finca, se muestran las programaciones de riego generadas tanto de manera secuencial como paralela.

- Programación de riego con secuencial:

*Vector*(*Vector*(0, 1), *Vector*(1, 0))

Esto indica que se generaron dos permutaciones posibles para el orden de riego de los tableros: (0, 1) y (1, 0).

- Programación de riego con paralela:

*Vector(Vector(0, 1), Vector(1, 0))*

Esto indica que la versión paralela también generó las mismas dos permutaciones posibles para el orden de riego de los tablones: (0, 1) y (1, 0).

**Tiempos de Ejecución (ms).** Estos valores representan el tiempo de ejecución de cada función.

- Secuencial:
  - 0.2117 (*para finca3*)
  - 0.2117 (*para finca4*)
- Paralela:
  - 0.0679 (*para finca3*)
  - 0.0679 (*para finca4*)

### **Aceleración:**

3.117820324005891 (*para finca3*), 3.117820324005891 (*para finca4*). Estos valores representan la aceleración obtenida al utilizar la versión paralela en comparación con la secuencial. (*tiempoSec/tiempoPar*).

1.

```
finca3:
Vector((4,2,3), (2,2,1))
Programación de riego con secuencial:
Vector(Vector(0, 1), Vector(1, 0))
Programación de riego con paralela:
Vector(Vector(0, 1), Vector(1, 0))
Unable to create a system terminal
Secuencial: 0.2117
Paralelo: 0.0679
Aceleración: 3.117820324005891
```

2.

```
finca4:
Vector((1,2,1), (1,1,1))
Programación de riego con secuencial:
Vector(Vector(0, 1), Vector(1, 0))
Programación de riego con paralela:
Vector(Vector(0, 1), Vector(1, 0))
Secuencial: 0.2117
Paralelo: 0.0679
Aceleración: 3.117820324005891
```

- **Fincas con tres tablones:** Se muestran programaciones de riego para dos fincas de tres tablones cada una.

Descripción:

- **finca5:** *Vector((3, 2, 2), (2, 1, 1), (2, 2, 4))*

Contiene dos tablones, cada uno representado por una tupla con tres valores:

- Primer tablón: (3, 2, 2), segundo tablón: (2, 1, 1), tercer tablón: (2, 2, 4)

- **finca6:** *Vector((5, 3, 4), (1, 3, 1), (6, 3, 3))*

Contiene dos tablones, cada uno representado por una tupla con tres valores:

- Primer tablón: (5, 3, 4), segundo tablón: (1, 3, 1), tercer tablón: (6, 3, 3)

Para cada finca, se muestran las programaciones de riego generadas tanto de manera secuencial como paralela.

- Programación de riego con secuencial:

*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))*

Esto indica que se generaron seis permutaciones posibles para el orden de riego de los tablonos: (0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), y (2, 1, 0).

- Programación de riego con paralela:

*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))*

Esto indica que la versión paralela también generó las mismas seis permutaciones posibles para el orden de riego de los tablonos.

```
finca5:
Vector((3,2,2), (2,1,1), (2,2,4))
Programación de riego con secuencial:
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))
Programación de riego con paralela:
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))
Unable to create a system terminal
Secuencial: 0.260201
Paralelo: 0.099001
Aceleración: 2.628266381147665
```

```
finca6:
Vector((5,3,4), (1,3,1), (6,3,3))
Programación de riego con secuencial:
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))
Programación de riego con paralela:
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))
Secuencial: 0.260201
Paralelo: 0.099001
Aceleración: 2.628266381147665
```

- **Fincas con cuatro tablonos:** Se muestran programaciones de riego para dos fincas de cuatro tablonos cada una.

Descripción:

- **finca7:** *Vector((6, 2, 2), (2, 1, 4), (7, 1, 1), (1, 1, 3))*

Contiene dos tablonos, cada uno representado por una tupla con tres valores:

- Primer tablón: (6, 2, 2), segundo tablón: (2, 1, 4), tercer tablón: (7, 1, 1), cuarto tablón: (1, 1, 3).
- **finca8:** *Vector*((3, 3, 4), (3, 3, 2), (1, 3, 1), (8, 4, 4))

Contiene dos tableros, cada uno representado por una tupla con tres valores:

- Primer tablón: (3, 3, 4), segundo tablón: (3, 3, 2), tercer tablón: (1, 3, 1), cuarto tablón: (8, 4, 4).

Para cada finca, se muestran las programaciones de riego generadas tanto de manera secuencial como paralela.

**Tiempos de ejecución (ms).** Secuencial: 0,1682 (*finca7*), 0,1682 (*finca8*). Paralela: 0,090499 (*finca7*), 0,090499 (*finca8*).

**Aceleración:** 1,858584072752185

```
finca7:
Vector((6,2,2), (2,1,4), (7,1,1), (1,1,3))
Programación de riego con secuencial:
Vector(Vector(0, 1, 2, 3), Vector(0, 1, 3, 2), Vector(0, 2, 1, 3), Vector(0, 2, 3, 1), Vector(0, 3, 1, 2), Vector(0, 3, 2, 1), Vector(1, 0, 2, 3), Vector(1, 0, 3, 2), Vector(1, 2, 0, 3), Vector(1, 2, 3, 0), Vector(1, 3, 0, 2), Vector(1, 3, 2, 0), Vector(2, 0, 1, 3), Vector(2, 0, 3, 1), Vector(2, 1, 0, 3), Vector(2, 1, 3, 0), Vector(2, 3, 0, 1), Vector(2, 3, 1, 0), Vector(3, 0, 1, 2), Vector(3, 0, 2, 1), Vector(3, 1, 0, 2), Vector(3, 1, 2, 0), Vector(3, 2, 0, 1), Vector(3, 2, 1, 0))
Programación de riego con paralela:
Vector(Vector(0, 1, 2, 3), Vector(0, 1, 3, 2), Vector(0, 2, 1, 3), Vector(0, 2, 3, 1), Vector(0, 3, 1, 2), Vector(0, 3, 2, 1), Vector(1, 0, 2, 3), Vector(1, 0, 3, 2), Vector(1, 2, 0, 3), Vector(1, 2, 3, 0), Vector(1, 3, 0, 2), Vector(1, 3, 2, 0), Vector(2, 0, 1, 3), Vector(2, 0, 3, 1), Vector(2, 1, 0, 3), Vector(2, 1, 3, 0), Vector(2, 3, 0, 1), Vector(2, 3, 1, 0), Vector(3, 0, 1, 2), Vector(3, 0, 2, 1), Vector(3, 1, 0, 2), Vector(3, 1, 2, 0), Vector(3, 2, 0, 1), Vector(3, 2, 1, 0))
Unable to create a system terminal
Secuencial: 0.1682
Paralelo: 0.090499
Aceleración: 1.858584072752185
```

```
finca8:
Vector((3,3,4), (3,3,2), (1,3,1), (8,4,4))
Programación de riego con secuencial:
Vector(Vector(0, 1, 2, 3), Vector(0, 1, 3, 2), Vector(0, 2, 1, 3), Vector(0, 2, 3, 1), Vector(0, 3, 1, 2), Vector(0, 3, 2, 1), Vector(1, 0, 2, 3), Vector(1, 0, 3, 2), Vector(1, 2, 0, 3), Vector(1, 2, 3, 0), Vector(1, 3, 0, 2), Vector(1, 3, 2, 0), Vector(2, 0, 1, 3), Vector(2, 0, 3, 1), Vector(2, 1, 0, 3), Vector(2, 1, 3, 0), Vector(2, 3, 0, 1), Vector(2, 3, 1, 0), Vector(3, 0, 1, 2), Vector(3, 0, 2, 1), Vector(3, 1, 0, 2), Vector(3, 1, 2, 0), Vector(3, 2, 0, 1), Vector(3, 2, 1, 0))
Programación de riego con paralela:
Vector(Vector(0, 1, 2, 3), Vector(0, 1, 3, 2), Vector(0, 2, 1, 3), Vector(0, 2, 3, 1), Vector(0, 3, 1, 2), Vector(0, 3, 2, 1), Vector(1, 0, 2, 3), Vector(1, 0, 3, 2), Vector(1, 2, 0, 3), Vector(1, 2, 3, 0), Vector(1, 3, 0, 2), Vector(1, 3, 2, 0), Vector(2, 0, 1, 3), Vector(2, 0, 3, 1), Vector(2, 1, 0, 3), Vector(2, 1, 3, 0), Vector(2, 3, 0, 1), Vector(2, 3, 1, 0), Vector(3, 0, 1, 2), Vector(3, 0, 2, 1), Vector(3, 1, 0, 2), Vector(3, 1, 2, 0), Vector(3, 2, 0, 1), Vector(3, 2, 1, 0))
Secuencial: 0.1682
Paralelo: 0.090499
Aceleración: 1.858584072752185
```

- **Fincas con cinco tableros:** Se muestran programaciones de riego para dos fincas de cinco tableros cada una.

Descripción:

- **finca9:** *Vector*((7, 1, 3), (3, 1, 3), (9, 3, 4), (5, 5, 3), (3, 2, 4))

Contiene dos tableros, cada uno representado por una tupla con tres valores:

- Primer tablón: (7, 1, 3), segundo tablón: (3, 1, 3), tercer tablón: (9, 3, 4), cuarto tablón: (5, 5, 3), quinto tablón: (3, 2, 4).

- **finca10:** *Vector*((6, 4, 3), (4, 3, 2), (9, 2, 1), (5, 2, 3), (6, 3, 1))

Contiene dos tableros, cada uno representado por una tupla con tres valores:

- Primer tablero: (6, 4, 3), segundo tablero: (4, 3, 2), tercer tablero: (9, 2, 1), cuarto tablero: (5, 2, 3), quinto tablero: (6, 3, 1).

Para cada finca, se muestran las programaciones de riego generadas tanto de manera secuencial como paralela. **Tiempos de ejecución (ms).**

Secuencial: 0,294599 (*finca9*), 0,294599 (*finca10*).

Paralela: 0,090499 (*finca7*), 0,090499 (*finca8*).

**Aceleración:** 1,3029588677576294.

## 1. finca9

```
0: Vector(7, 1, 3)
1: Vector(3, 1, 3)
2: Vector(9, 3, 4)
3: Vector(5, 5, 3)
4: Vector(3, 2, 4)
```

Paralelo: 0.2261

Aceleración: 1.3029588677576294

[illegible][illegible]

## finca10:

```
0: Vector(6, 4, 3)
1: Vector(4, 3, 2)
2: Vector(9, 2, 1)
3: Vector(5, 2, 3)
4: Vector(6, 3, 1)
```

Paralelo: 0.2261

Aceleración: 1.3029588677576294



```

Programación de riego con secuencial:
Vector(Vector(0, 1, 2, 3, 4), Vector(0, 1, 2, 4, 3), Vector(0, 1, 3, 2, 4), Vector(0, 1, 3, 4, 2), Vector(0, 1, 4, 2, 3), Vector(0, 1, 4, 3, 2), Vector
(0, 2, 1, 3, 4), Vector(0, 2, 1, 4, 3), Vector(0, 2, 3, 1, 4), Vector(0, 2, 3, 4, 1), Vector(0, 2, 4, 1, 3), Vector(0, 2, 4, 3, 1), Vector(0, 3, 1, 2,
4), Vector(0, 3, 1, 4, 2), Vector(0, 3, 2, 1, 4), Vector(0, 3, 2, 4, 1), Vector(0, 3, 4, 1, 2), Vector(0, 3, 4, 2, 1), Vector(0, 4, 1, 2, 3), Vector(0,
4, 1, 3, 2), Vector(0, 4, 2, 1, 3), Vector(0, 4, 2, 3, 1), Vector(0, 4, 3, 1, 2), Vector(0, 4, 3, 2, 1), Vector(1, 0, 2, 3, 4), Vector(1, 0, 2, 4, 3),
Vector(1, 0, 3, 2, 4), Vector(1, 0, 3, 4, 2), Vector(1, 0, 4, 2, 3), Vector(1, 0, 4, 3, 2), Vector(1, 2, 0, 3, 4), Vector(1, 2, 0, 4, 3), Vector(1, 2,
3, 0, 4), Vector(1, 2, 3, 4, 0), Vector(1, 2, 4, 0, 3), Vector(1, 2, 4, 3, 0), Vector(1, 3, 0, 2, 4), Vector(1, 3, 0, 4, 2), Vector(1, 3, 2, 0, 4), Ve
ctor(1, 3, 2, 4, 0), Vector(1, 3, 4, 0, 2), Vector(1, 3, 4, 2, 0), Vector(1, 4, 0, 2, 3), Vector(1, 4, 0, 3, 2), Vector(1, 4, 2, 0, 3), Vector(1, 4, 2,
3, 0), Vector(1, 4, 3, 0, 2), Vector(1, 4, 3, 2, 0), Vector(2, 0, 1, 3, 4), Vector(2, 0, 1, 4, 3), Vector(2, 0, 3, 1, 4), Vector(2, 0, 3, 4, 1), Vecto
r(2, 0, 4, 1, 3), Vector(2, 0, 4, 3, 1), Vector(2, 1, 0, 3, 4), Vector(2, 1, 0, 4, 3), Vector(2, 1, 3, 0, 4), Vector(2, 1, 3, 4, 0), Vector(2, 1, 4, 0,
3), Vector(2, 1, 4, 3, 0), Vector(2, 3, 0, 1, 4), Vector(2, 3, 0, 4, 1), Vector(2, 3, 1, 0, 4), Vector(2, 3, 1, 4, 0), Vector(2, 3, 4, 0, 1), Vector(2
, 3, 4, 1, 0), Vector(2, 4, 0, 1, 3), Vector(2, 4, 0, 3, 1), Vector(2, 4, 1, 0, 3), Vector(2, 4, 1, 3, 0), Vector(2, 4, 3, 0, 1), Vector(2, 4, 3, 1, 0),
Vector(3, 0, 1, 2, 4), Vector(3, 0, 1, 4, 2), Vector(3, 0, 2, 1, 4), Vector(3, 0, 2, 4, 1), Vector(3, 0, 4, 1, 2), Vector(3, 0, 4, 2, 1), Vector(3, 1
, 0, 2, 4), Vector(3, 1, 0, 4, 2), Vector(3, 1, 2, 0, 4), Vector(3, 1, 2, 4, 0), Vector(3, 1, 4, 0, 2), Vector(3, 1, 4, 2, 0), Vector(3, 2, 0, 1, 4), V
ector(3, 2, 0, 4, 1), Vector(3, 2, 1, 0, 4), Vector(3, 2, 1, 4, 0), Vector(3, 2, 4, 0, 1), Vector(3, 2, 4, 1, 0), Vector(3, 4, 0, 1, 2), Vector(3, 4, 0
, 2, 1), Vector(3, 4, 1, 0, 2), Vector(3, 4, 1, 2, 0), Vector(3, 4, 2, 0, 1), Vector(3, 4, 2, 1, 0), Vector(4, 0, 1, 2, 3), Vector(4, 0, 1, 3, 2), Vect
or(4, 0, 2, 1, 3), Vector(4, 0, 2, 3, 1), Vector(4, 0, 3, 1, 2), Vector(4, 0, 3, 2, 1), Vector(4, 1, 0, 2, 3), Vector(4, 1, 0, 3, 2), Vector(4, 1, 2, 0
, 3), Vector(4, 1, 2, 3, 0), Vector(4, 1, 3, 0, 2), Vector(4, 1, 3, 2, 0), Vector(4, 2, 0, 1, 3), Vector(4, 2, 0, 3, 1), Vector(4, 2, 1, 0, 3), Vector(
4, 2, 1, 3, 0), Vector(4, 2, 3, 0, 1), Vector(4, 2, 3, 1, 0), Vector(4, 3, 0, 1, 2), Vector(4, 3, 0, 2, 1), Vector(4, 3, 1, 0, 2), Vector(4, 3, 1, 2, 0
), Vector(4, 3, 2, 0, 1), Vector(4, 3, 2, 1, 0))

```

```

Programación de riego con paralela:
Vector(Vector(0, 1, 2, 3, 4), Vector(0, 1, 2, 4, 3), Vector(0, 1, 3, 2, 4), Vector(0, 1, 3, 4, 2), Vector(0, 1, 4, 2, 3), Vector(0, 1, 4, 3, 2), Vector
(0, 2, 1, 3, 4), Vector(0, 2, 1, 4, 3), Vector(0, 2, 3, 1, 4), Vector(0, 2, 3, 4, 1), Vector(0, 2, 4, 1, 3), Vector(0, 2, 4, 3, 1), Vector(0, 3, 1, 2,
4), Vector(0, 3, 1, 4, 2), Vector(0, 3, 2, 1, 4), Vector(0, 3, 2, 4, 1), Vector(0, 3, 4, 1, 2), Vector(0, 3, 4, 2, 1), Vector(0, 4, 1, 2, 3), Vector(0,
4, 1, 3, 2), Vector(0, 4, 2, 1, 3), Vector(0, 4, 2, 3, 1), Vector(0, 4, 3, 1, 2), Vector(0, 4, 3, 2, 1), Vector(1, 0, 2, 3, 4), Vector(1, 0, 2, 4, 3),
Vector(1, 0, 3, 2, 4), Vector(1, 0, 3, 4, 2), Vector(1, 0, 4, 2, 3), Vector(1, 0, 4, 3, 2), Vector(1, 2, 0, 3, 4), Vector(1, 2, 0, 4, 3), Vector(1, 2,
3, 0, 4), Vector(1, 2, 3, 4, 0), Vector(1, 2, 4, 0, 3), Vector(1, 2, 4, 3, 0), Vector(1, 3, 0, 2, 4), Vector(1, 3, 0, 4, 2), Vector(1, 3, 2, 0, 4), Ve
ctor(1, 3, 2, 4, 0), Vector(1, 3, 4, 0, 2), Vector(1, 3, 4, 2, 0), Vector(1, 4, 0, 2, 3), Vector(1, 4, 0, 3, 2), Vector(1, 4, 2, 0, 3), Vector(1, 4, 2,
3, 0), Vector(1, 4, 3, 0, 2), Vector(1, 4, 3, 2, 0), Vector(2, 0, 1, 3, 4), Vector(2, 0, 1, 4, 3), Vector(2, 0, 3, 1, 4), Vector(2, 0, 3, 4, 1), Vecto
r(2, 0, 4, 1, 3), Vector(2, 0, 4, 3, 1), Vector(2, 1, 0, 3, 4), Vector(2, 1, 0, 4, 3), Vector(2, 1, 3, 0, 4), Vector(2, 1, 3, 4, 0), Vector(2, 1, 4, 0,
3), Vector(2, 1, 4, 3, 0), Vector(2, 3, 0, 1, 4), Vector(2, 3, 0, 4, 1), Vector(2, 3, 1, 0, 4), Vector(2, 3, 1, 4, 0), Vector(2, 3, 4, 0, 1), Vector(2
, 3, 4, 1, 0), Vector(2, 4, 0, 1, 3), Vector(2, 4, 0, 3, 1), Vector(2, 4, 1, 0, 3), Vector(2, 4, 1, 3, 0), Vector(2, 4, 3, 0, 1), Vector(2, 4, 3, 1, 0),
Vector(3, 0, 1, 2, 4), Vector(3, 0, 1, 4, 2), Vector(3, 0, 2, 1, 4), Vector(3, 0, 2, 4, 1), Vector(3, 0, 4, 1, 2), Vector(3, 0, 4, 2, 1), Vector(3, 1
, 0, 2, 4), Vector(3, 1, 0, 4, 2), Vector(3, 1, 2, 0, 4), Vector(3, 1, 2, 4, 0), Vector(3, 1, 4, 0, 2), Vector(3, 1, 4, 2, 0), Vector(3, 2, 0, 1, 4), V
ector(3, 2, 0, 4, 1), Vector(3, 2, 1, 0, 4), Vector(3, 2, 1, 4, 0), Vector(3, 2, 4, 0, 1), Vector(3, 2, 4, 1, 0), Vector(3, 4, 0, 1, 2), Vector(3, 4, 0
, 2, 1), Vector(3, 4, 1, 0, 2), Vector(3, 4, 1, 2, 0), Vector(3, 4, 2, 0, 1), Vector(3, 4, 2, 1, 0), Vector(4, 0, 1, 2, 3), Vector(4, 0, 1, 3, 2), Vect
or(4, 0, 2, 1, 3), Vector(4, 0, 2, 3, 1), Vector(4, 0, 3, 1, 2), Vector(4, 0, 3, 2, 1), Vector(4, 1, 0, 2, 3), Vector(4, 1, 0, 3, 2), Vector(4, 1, 2, 0
, 3), Vector(4, 1, 2, 3, 0), Vector(4, 1, 3, 0, 2), Vector(4, 1, 3, 2, 0), Vector(4, 2, 0, 1, 3), Vector(4, 2, 0, 3, 1), Vector(4, 2, 1, 0, 3), Vector(
4, 2, 1, 3, 0), Vector(4, 2, 3, 0, 1), Vector(4, 2, 3, 1, 0), Vector(4, 3, 0, 1, 2), Vector(4, 3, 0, 2, 1), Vector(4, 3, 1, 0, 2), Vector(4, 3, 1, 2, 0
), Vector(4, 3, 2, 0, 1), Vector(4, 3, 2, 1, 0))

```

Para mostrar la información de las fincas y sus correspondientes tablonos se utilizaron las siguientes funciones:

```

def imprimirVectorFormateado(vector: Vector[Vector[Int]]): String = {
  vector.zipWithIndex.map { case (subVector, index) =>
    s" $index: " + subVector.mkString(" ", " ", " ")
  }.mkString("\n")
}

```

La función anterior usa saltos de línea para que cada nivel de las estructuras anidadas sea más claro. La función también muestra los índices correspondientes.

```

def pruebas2(): Unit = {
  println("finca9: \n" + imprimirVectorFormateado(finca9.map { case (a, b, c) => Vector(a, b, c) })))
  println("Programación de riego con secuencial:\n" + regado.generarProgramacionesRiego(finca9))
  println("Programación de riego con paralela: \n" + regado.generarProgramacionesRiegoPar(finca9))
  val comp1 = regado.compararGenerarProgramacionesRiego(regado.generarProgramacionesRiego, regado.generarProgramacionesRiegoPar)(finca9)
  println("Secuencial: " + comp1(0))
  println("Paralelo: " + comp1(1))
  println("Aceleración: " + comp1(2))

  println("\nfinca10: \n" + imprimirVectorFormateado(finca10.map { case (a, b, c) => Vector(a, b, c) })))
  println("Programación de riego con secuencial:\n" + regado.generarProgramacionesRiego(finca10))
  println("Programación de riego con paralela: \n" + regado.generarProgramacionesRiegoPar(finca10))
  val comp2 = regado.compararGenerarProgramacionesRiego(regado.generarProgramacionesRiego, regado.generarProgramacionesRiegoPar)(finca10)
  println("Secuencial: " + comp1(0))
  println("Paralelo: " + comp1(1))
  println("Aceleración: " + comp1(2))
}

```

Esta función corresponde a una **unidad de ejecución** que se llama en el **Main**.



- **Función ProgramacionRiegoOptimoPar:**

La función ProgramacionRiegoOptimoPar tiene como objetivo principal calcular la programación óptima de riego de una finca, minimizando costos de riego y movilidad, haciendo uso de procesamiento paralelo. Este enfoque permite aprovechar la capacidad de procesamiento de múltiples núcleos para acelerar los cálculos involucrados en la evaluación de diversas programaciones de riego.

### **Estrategia de paralelización**

1. División de datos

- La función generarProgramacionesRiegoPar produce una lista de programaciones de riego posibles. Este conjunto de datos se transforma en una colección paralela utilizando el método .par.
- Esto habilita la distribución de las operaciones de cálculo entre múltiples hilos del pool de procesamiento.

2. Procesamiento independiente

- Cada programación de riego pi es procesada de forma independiente para calcular su costo total:
  - Costo de riego: Calculado usando costoRiegoFincaPar, que ya aplica paralelización para procesar los costos de cada tablón.
  - Costo de movilidad: Calculado usando costoMovilidadPar, que también puede beneficiarse del procesamiento paralelo si está implementado adecuadamente.
- Estos dos costos se combinan para cada pi, generando un par (pi, costoTotal).

3. Selección del mínimo

- Una vez que se han calculado los costos para todas las programaciones, el método .minBy(\_.\_2) selecciona la programación con el menor costo total.
- Este paso no es paralelizable, pero es eficiente debido a que solo implica una operación de reducción.

Para analizar el impacto de la paralelización, aplicamos la Ley de Amdahl considerando los siguientes factores:

1. Proporción paralelizable (P):

- En ProgramacionRiegoOptimoPar, el cálculo de costos para cada programación (la operación costos.par.map(...)) es paralelizable, ya que no hay dependencias entre los cálculos de las programaciones individuales.
- Supongamos que esta parte representa el 85% del tiempo de ejecución total.

2. Número de núcleos disponibles (N):

- Suponemos que se dispone de 4 procesadores para ejecutar las operaciones en paralelo.

Cálculo del Speedup:

1. Calcular:  $1 - 0,85 = 0,15$
2. Calcular:  $P/N = 0,85/4 = 0,213$
3. Sustituir en la fórmula de la Ley de Amdahl:

$$S = \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{(0,15) + 0,213} = 2.7568$$

Esto significa que, al aplicar la paralelización en ProgramacionRiegoOptimoPar con 4 procesadores, se puede esperar una mejora en la velocidad de ejecución de aproximadamente 2.7568 veces en comparación con la ejecución secuencial.

## CASOS DE PRUEBA (ProgramacionRiegoOptimoPar)

```
def pruebas3(): Unit = {
  val fincaP3 = regado.fincaAlAzar(1)
  val fincaP32 = regado.fincaAlAzar(1)
  val distancia = regado.distanciaAlAzar(1)

  println("finca1: \n" + imprimirVectorFormateado(fincaP3.map { case (a, b, c) => Vector(a, b, c) }))
  println("Programación de riego con secuencial:\n" + regado.ProgramacionRiegoOptimo(fincaP3, distancia))
  println("Programación de riego con paralela: \n" + regado.ProgramacionRiegoOptimoPar(fincaP3, distancia))
  val comp1 = regado.compararProgramacionRiegoOptimo(regado.ProgramacionRiegoOptimo, regado.ProgramacionRiegoOptimoPar)
  println("Secuencial: " + comp1(0))
  println("Paralelo: " + comp1(1))
  println("Aceleración: " + comp1(2))

  println("finca2: \n" + imprimirVectorFormateado(fincaP32.map { case (a, b, c) => Vector(a, b, c) }))
  println("Programación de riego con secuencial:\n" + regado.ProgramacionRiegoOptimo(fincaP32, distancia))
  println("Programación de riego con paralela: \n" + regado.ProgramacionRiegoOptimoPar(fincaP32, distancia))
  val comp2 = regado.compararProgramacionRiegoOptimo(regado.ProgramacionRiegoOptimo, regado.ProgramacionRiegoOptimoPar)
  println("Secuencial: " + comp2(0))
  println("Paralelo: " + comp2(1))
  println("Aceleración: " + comp2(2))
}
```

Para los casos de prueba de esta función se usará la función mostrada, cambiando para cada tamaño, el número de fincas al azar, tanto de fincaP3 y fincaP32, además de la distanciaAlAzar de distancia, usando tamaños de 1, 2, 3, 4, 5 y distancias de 1, 2, 3, 4, 5 respectivamente

- Fincas con un tablón y distancia 1:

```
finca1:
  0: (1, 1, 1)
Programación de riego con secuencial:
(Vector(0),0)
Programación de riego con paralela:
(Vector(0),0)
Unable to create a system terminal
Secuencial: 0.1412
Paralelo: 0.469
Aceleración: 0.30106609808102347
finca2:
  0: (1, 1, 3)
Programación de riego con secuencial:
(Vector(0),0)
Programación de riego con paralela:
(Vector(0),0)
Secuencial: 0.0853
Paralelo: 0.4871
Aceleración: 0.1751180455758571
```

Como se ve en la imagen, el orden para ambos casos, al ser solo un tablón, se inicia en la posición 0 y el costo de riego es de 0.

Viendo los tiempos de ejecución de ambos casos del mismo tamaño se evidencia que no hay ganancia para la versión paralela del programa, esto posiblemente debido a la poca cantidad de datos, aunque revisando el análisis de datos, realizado anteriormente, ya podemos suponer que se debe más a limitaciones del hardware o a un overhead de paralelización.

- **Fincas con dos tablonos y distancia 2:**

```
finca3:
  0: (3, 2, 1)
  1: (1, 1, 3)
Programación de riego con secuencial:
(Vector(1, 0),5)
Programación de riego con paralela:
(Vector(1, 0),5)
Unable to create a system terminal
Secuencial: 0.2402
Paralelo: 0.5924
Aceleración: 0.4054692775151924
finca4:
  0: (2, 2, 3)
  1: (3, 1, 3)
Programación de riego con secuencial:
(Vector(0, 1),5)
Programación de riego con paralela:
(Vector(0, 1),5)
Secuencial: 0.1222
Paralelo: 0.9048
Aceleración: 0.13505747126436782
```

Como se ve en la imagen, el orden siendo dos tablones, para el caso de la finca3, se riega primero la posición 1 y posteriormente la posición 0. Y para la finca4, primero se riega la posición 0 y posteriormente la posición 1. Teniendo un costo de movilidad ambos casos de 5

Viendo los tiempos de ejecución de ambos casos del mismo tamaño, se evidencia que no hay ganancia para la versión paralela del programa, esto posiblemente debido a la poca cantidad de datos, lo que indica que la versión secuencial es más efectiva para casos de menor tamaño.

#### - Fincas con tres tablones y distancia 3:

```
finca5:
  0: (3, 2, 4)
  1: (1, 3, 4)
  2: (3, 2, 1)
Programación de riego con secuencial:
(Vector(0, 1, 2),28)
Programación de riego con paralela:
(Vector(0, 1, 2),28)
Unable to create a system terminal
Secuencial: 0.3532
Paralelo: 0.8242
Aceleración: 0.42853676292162096
finca6:
  0: (6, 1, 3)
  1: (3, 1, 4)
  2: (4, 2, 3)
Programación de riego con secuencial:
(Vector(2, 1, 0),11)
Programación de riego con paralela:
(Vector(2, 1, 0),11)
Secuencial: 0.1168
Paralelo: 0.7512
Aceleración: 0.1554845580404686
```

Como se ve en la imagen, el orden siendo tres tablonos, para el caso de la finca5, se riega primero la posición 0, posteriormente la posición 1 y de último la 2, teniendo un costo de movilidad de 28. Y para la finca6, primero se riega la posición 2, posteriormente la posición 1 y de último la posición 0, teniendo un costo de movilidad de 11

Viendo los tiempos de ejecución de ambos casos del mismo tamaño, se sigue viendo que para tamaños más pequeños es más efectiva la versión secuencial

- **Fincas con cuatro tablonos y distancia 4:**

```
finca7:
  0: (8, 1, 4)
  1: (4, 4, 4)
  2: (8, 3, 2)
  3: (2, 2, 2)
Programación de riego con secuencial:
(Vector(1, 0, 3, 2),30)
Programación de riego con paralela:
(Vector(1, 0, 3, 2),30)
Unable to create a system terminal
Secuencial: 0.453
Paralelo: 1.1698
Aceleración: 0.387245683022739
finca8:
  0: (2, 1, 3)
  1: (2, 3, 1)
  2: (5, 3, 2)
  3: (3, 4, 4)
Programación de riego con secuencial:
(Vector(0, 3, 2, 1),36)
Programación de riego con paralela:
(Vector(0, 3, 2, 1),36)
Secuencial: 0.1824
Paralelo: 1.1403
Aceleración: 0.1599579058142594
```

Como se ve en la imagen, el orden siendo cuatro tablonos, para el caso de la finca7, se riega primero la posición 1, posteriormente la posición 0, después la posición 3 y de último la 2, teniendo un costo de movilidad de 30. Y para la finca8, primero se riega la posición 0, posteriormente la posición 3, después la 2 y de último la posición 1, teniendo un costo de movilidad de 36

Viendo los tiempos de ejecución de ambos casos del mismo tamaño, se sigue viendo que para tamaños más pequeños es más efectiva la versión secuencial

- **Fincas con siete tablonos y distancia 7:**

```
finca9:
  0: (4, 1, 4)
  1: (3, 1, 1)
  2: (12, 4, 3)
  3: (1, 4, 1)
  4: (9, 5, 3)
  5: (2, 1, 2)
  6: (3, 2, 4)
Programación de riego con secuencial:
(Vector(6, 1, 0, 2, 5, 4, 3),93)
Programación de riego con paralela:
(Vector(6, 1, 0, 2, 5, 4, 3),93)
Unable to create a system terminal
Secuencial: 13.5904
Paralelo: 20.1634
Aceleración: 0.6740133112471112
```

```
finca10:
  0: (2, 1, 3)
  1: (3, 1, 3)
  2: (10, 4, 4)
  3: (7, 6, 1)
  4: (12, 7, 1)
  5: (5, 2, 2)
  6: (3, 3, 3)
Programación de riego con secuencial:
(Vector(6, 0, 1, 2, 5, 3, 4),70)
Programación de riego con paralela:
(Vector(6, 0, 1, 2, 5, 3, 4),70)
Unable to create a system terminal
Secuencial: 9.6802
Paralelo: 17.7667
Aceleración: 0.5448507601299059
```

Como se ve en la imagen, el orden siendo siete tablonos, para el caso de la finca9, se riega en el siguiente orden: 6, 1, 0, 2, 5, 4, 3, teniendo un costo de movilidad de 93. Y para la finca10, se riega en el siguiente orden: 6, 1, 0, 2, 5, 4, teniendo un costo de movilidad de 70.

Para el último caso de prueba se utilizaron datos de mayor tamaño para verificar si la razón de que la versión paralela sea menos la menos eficiente sea el tamaño, dando como resultado que sigue sin ser más eficiente la versión paralela.

Con esto y el análisis de resultado se puede deducir que:

Se necesitan datos de un gran tamaño y muy grande tiempo de ejecución para que la versión paralela sea la mejor, u otras dos opciones que pueden ser el overhead de paralelización o que haya limitaciones de hardware

## INFORME DE CORRECCIÓN

- **Argumentación sobre la corrección de la función costoRiegoFinca utilizando notación matemática por inducción estructural**

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

**Caso Base:**  $n = 1$

- Consideremos una finca con un solo tablón:  $\text{finca} = \text{Vector}((tsi, tri, pi))$ .
- El costo de riego para este tablón se calcula como:  
 $\text{costoRiegoTablon}(tsi, tri) = tsi - tri$
- Por lo tanto, el costo total de la finca es simplemente el costo del único tablón, que es correcto.

**Paso Inductivo:** Supongamos que la función costoRiegoFinca es correcta para una finca con  $k$  tablonos, es decir, que calcula correctamente el costo total de riego:

$$\text{costoRiegoFinca}(\text{finca}_k) = \sum_{i=0}^{k-1} (tsi_i - tri_i)$$

- Consideremos una finca con  $(k + 1)$  tablonos:  $\text{finca} = \text{Vector}((tsi\_0, tri\_0, pi\_0), (tsi\_1, tri\_1, pi\_1), \dots, (tsi\_k, tri\_k, pi\_k))$ .
- Según nuestra hipótesis de inducción, el costo total para los primeros  $k$  tablonos es:

$$\text{costoRiegoFinca}(\text{finca}_k) = \sum_{i=0}^{k-1} (tsi_i - tri_i)$$

- Ahora, añadimos el costo del tablón adicional  $k$ :  
 $\text{costoRiegoFinca}(\text{finca}_{k+1}) = \text{costoRiegoFinca}(\text{finca}_k) + (tsi_k - tri_k)$

- Sustituyendo la hipótesis de inducción:

$$\text{costoRiegoFinca}(\text{finca}_{k+1}) = \sum_{i=0}^{k-1} (\text{tsi}_i - \text{tri}_i) + (\text{tsi}_k - \text{tri}_k) \text{ que se puede}$$

$$\text{reescribir como: } \text{costoRiegoFinca}(\text{finca}_{k+1}) = \sum_{i=0}^k (\text{tsi}_i - \text{tri}_i)$$

- Por lo tanto, hemos demostrado que la función también es correcta para  $(k + 1)$  tablonos.

- **Argumentación sobre la corrección de la función costoRiegoFincaPar utilizando notación matemática por inducción estructural**

$$\text{def costoRiegoFincaPar}(f: \text{Finca}, pi: \text{ProgRiego}): \text{Int} = \{ \\ (0 \text{ until } f.\text{length}).\text{par}.\text{map}(i \Rightarrow \text{costoRiegoTablon}(i, f, pi)).\text{sum}\}$$

**Caso Base:**  $n = 1$

- Consideremos una finca con un solo tablón:  $\text{finca} = \text{Vector}((\text{tsi}, \text{tri}, \text{pi}))$ .
- El costo de riego para este tablón se calcula como:  
 $\text{costoRiegoTablon}(\text{tsi}, \text{tri}) = \text{tsi} - \text{tri}$
- Por lo tanto, el costo total de la finca es simplemente el costo del único tablón, que es correcto.

**Paso Inductivo:** Supongamos que la función costoRiegoFincaPar es correcta para una finca con  $k$  tablonos, es decir, que calcula correctamente el costo total de

$$\text{riego: } \text{costoRiegoFincaPar}(\text{finca}_k) = \sum_{i=0}^{k-1} (\text{tsi}_i - \text{tri}_i)$$

- Consideremos una finca con  $(k + 1)$  tablonos:  
 $\text{finca} = \text{Vector}((\text{tsi}_0, \text{tri}_0, \text{pi}_0), (\text{tsi}_1, \text{tri}_1, \text{pi}_1), \dots, (\text{tsi}_k, \text{tri}_k, \text{pi}_k))$ .
- Según nuestra hipótesis de inducción, el costo total para los primeros  $k$

$$\text{tablonos es: } \text{costoRiegoFincaPar}(\text{finca}_k) = \sum_{i=0}^{k-1} (\text{tsi}_i - \text{tri}_i)$$

- Ahora, añadimos el costo del tablón adicional  $k$ :  
 $\text{costoRiegoFincaPar}(\text{finca}_{k+1}) = \text{costoRiegoFincaPar}(\text{finca}_k) + (\text{tsi}_k - \text{tri}_k)$

- Sustituyendo la hipótesis de inducción:

$$\text{costoRiegoFincaPar}(\text{finca}_{k+1}) = \sum_{i=0}^{k-1} (\text{tsi}_i - \text{tri}_i) + (\text{tsi}_k - \text{tri}_k) \text{ que se}$$

$$\text{puede reescribir como: } \text{costoRiegoFincaPar}(\text{finca}_{k+1}) = \sum_{i=0}^k (\text{tsi}_i - \text{tri}_i)$$

- Por lo tanto, hemos demostrado que la función también es correcta para



$(k + 1)$  tablonos.

- **Argumentación sobre la corrección de la función generarProgramacionesRiego utilizando Inducción matemática**

La función costoMovilidadPar es una versión paralela de la función costoMovilidad. Su objetivo es calcular el costo total de mover agua entre los tablonos de una finca utilizando procesamiento paralelo para distribuir el cálculo de cada movimiento entre los diferentes tablonos.

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

La función toma tres parámetros:

1. f: La finca, representada por un vector de los tablonos.
2. pi: Un vector que representa el programa de riego (el orden en que los tablonos serán regados).
3. d: Una matriz de distancias entre los tablonos de la finca.

La función genera las permutaciones del índice de los tablonos, calcula la distancia entre los tablonos, y suma el costo total de mover agua entre ellos. El cálculo se realiza en paralelo utilizando .par, lo que permite que cada cálculo del costo de riego de los tablonos se ejecute simultáneamente.

### **Caso Base: $n = 1$**

Cuando la finca tiene solo un tablón, es decir,  $f = [0]$ , el vector pi será Vector(0). Esto significa que no hay movimientos entre diferentes tablonos, ya que solo hay uno.

- La función costoMovilidadPar calculará el costo entre el único tablón y sí mismo. Sin embargo, dado que no hay ningún otro tablón para mover el agua, el cálculo será 0, ya que no hay distancias que recorrer.
- Por lo tanto, el resultado será 0, lo cual es correcto, ya que no hay ningún costo de riego cuando solo hay un tablón.

### **Paso Inductivo: Suposición para $n = k$**

Supongamos que la proposición es verdadera para una finca con k tablonos, es decir, para una finca de tamaño k, la función costoMovilidadPar calcula

correctamente el costo total de mover agua entre los tablones, considerando todas las permutaciones posibles.

### **Paso de Inducción: Demostración para $n = k + 1$**

Ahora, demostraremos que la proposición sigue siendo válida para una finca con  $k + 1$  tablones. Es decir, para una finca  $f = [0, 1, 2, \dots, k]$ , la función `costoMovilidadPar` debe calcular correctamente el costo total de mover el agua entre todos los tablones, considerando todas las permutaciones posibles.

- Para calcular el costo de movilidad de los tablones en paralelo, la función divide el rango de índices (0 until `pi.length - 1`) en subrangos que se procesan de forma independiente. Cada subrango corresponde a un par de tablones consecutivos (`pi(j)` y `pi(j+1)`), y la distancia entre estos se calcula en paralelo.
- Para una finca con  $k + 1$  tablones, los cálculos se distribuyen entre los hilos disponibles en el pool de procesamiento, donde cada hilo se encarga de calcular el costo de movilidad entre un par de tablones consecutivos.
- Dado que el cálculo de cada par de tablones es independiente, los resultados parciales (costos individuales entre cada par de tablones) se combinan al final usando `.sum`, lo que da el costo total de movilidad entre todos los tablones.

**Por lo tanto**, si la función `costoMovilidadPar` funciona correctamente para una finca con  $k$  tablones (es decir, la aceleración es correcta y no hay errores en el cálculo), entonces se comportará de manera correcta para una finca con  $k + 1$  tablones, ya que el cálculo de cada par de tablones es independiente y la suma final de los costos se realiza correctamente en paralelo.

Esto demuestra que la función `costoMovilidadPar` es correcta para cualquier tamaño de finca.

### **Conclusión**

La función `costoMovilidadPar` está correctamente paralelizada y calcula el costo total de mover agua entre los tablones de la finca. A través del paso inductivo y el caso base, hemos demostrado que la función es válida tanto para una finca con un solo tablón como para fincas con más de uno, aplicando la paralelización de manera eficiente en todos los casos.

- **Argumentación sobre la corrección de la función `generarProgramacionesRiego` utilizando Inducción matemática**

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

**Caso Base:**  $n = 1$

- La finca  $f$  tiene un solo tablón, es decir,  $f = [0]$ .
- La función `generarProgramacionesRiego` debe generar todas las permutaciones de  $\{0\}$ .
- La única permutación de un solo elemento es el mismo, es decir,  $Vector(Vector(0))$ .
- Por lo tanto, `generarProgramacionesRiego` devuelve  $((0))$ , que es correcto, ya que  $1! = 1$ .

**Paso Inductivo:** Supongamos que la proposición es verdadera para  $n = k$ , es decir, para una finca con  $k$  tablonos, `generarProgramacionesRiego` genera  $k!$  permutaciones.

Dado que funciona para  $k$ , debe funcionar para  $k + 1$ :

- Consideremos una finca  $f$  con  $k + 1$  tablonos, es decir,  $f = [0, 1, 2, \dots, k]$ .
- La función `generarProgramacionesRiego` debe generar todas las permutaciones de  $\{0, 1, 2, \dots, k\}$ .

Para generar todas las permutaciones de  $k + 1$  elementos, podemos fijar cada uno de los  $k + 1$  elementos en la primera posición y generar permutaciones de los  $k$  elementos restantes.

Entonces, si se fija el elemento  $i$  en la primera posición, las permutaciones restantes son las permutaciones de  $\{0, 1, 2, \dots, k\} \setminus \{i\}$ , que son  $k!$  por la hipótesis inductiva.

Como hay  $k + 1$  opciones para el primer elemento, el número total de permutaciones es:  $(k + 1) * k! = (k + 1)!$

Por lo tanto, `generarProgramacionesRiego` genera  $(k + 1)!$  permutaciones para una finca con  $k + 1$  tablonos.

- **Argumentación sobre la corrección de la función `generarProgramacionesRiegoPar` utilizando Inducción matemática**

```
def generarProgramacionesRiegoPar(f: Finca) : Vector[ProgRiego] = {
    //devuelve todas las posibles prog de riego de la finca
    val indices = (0 until f.length).toVector
    indices.permutations.toVector.par.toVector
}
```

**Caso Base:**  $n = 1$ :

- La finca  $f$  tiene un solo tablón, es decir,  $f = [0]$ .
- La función *generarProgramacionesRiegoPar* debe generar todas las permutaciones de  $\{0\}$ .
- La única permutación de un solo elemento es el elemento mismo (0).
- Por lo tanto, *generarProgramacionesRiegoPar* devuelve *Vector(Vector(0))*, que es correcto, ya que  $1! = 1$ .

**Paso Inductivo:** Supongamos que la proposición es verdadera para  $n = k$ , es decir, para una finca con  $k$  tablonos, *generarProgramacionesRiegoPar* genera  $k!$  permutaciones.

Dado que funciona para  $k$ , debe funcionar para  $k + 1$ :

- Consideremos una finca  $f$  con  $k + 1$  tablonos, es decir,  $f = [0, 1, 2, \dots, k]$ .
- La función *generarProgramacionesRiegoPar* debe generar todas las permutaciones de  $\{0, 1, 2, \dots, k\}$ .

Para generar todas las permutaciones de  $k + 1$  elementos, podemos fijar cada uno de los  $k + 1$  elementos en la primera posición y generar permutaciones de los  $k$  elementos restantes.

Entonces, si se fija el elemento  $i$  en la primera posición, las permutaciones restantes son las permutaciones de  $\{0, 1, 2, \dots, k\} \setminus \{i\}$ , que son  $k!$  por la hipótesis inductiva.

Como hay  $k + 1$  opciones para el primer elemento, el número total de permutaciones es:  $(k + 1) * k! = (k + 1)!$

Por lo tanto, *generarProgramacionesRiegoPar* genera  $(k + 1)!$  permutaciones para una finca con  $k + 1$  tablonos.

- **Argumentación sobre la corrección de la función**  
**ProgramacionRiegoOptimo utilizando Inducción matemática**

```
def ProgramacionRiegoOptimo(f:Finca, d:Distancia) : (ProgRiego,
Int) = {
    val programaciones = generarProgramacionesRiego(f)
```

```

        val costos = programaciones.map(pi => (pi,
costoRiegoFinca(f, pi) + costoMovilidad(f, pi, d)))

        costos.minBy(_._2)

    }

```

### Caso Base: $n = 1$

Supongamos que la finca  $f$  tiene un solo tablón:  $f = \text{Vector}((\text{tsi}, \text{tri}, \text{pi}))$

1. La función  $\text{generarProgramacionesRiego}(f)$  generará todas las posibles programaciones de riego para este único tablón. Dado que hay solo un tablón, estas programaciones serán trivialmente únicas.
2. Para cada programación  $\text{pi}$ , el costo total se calcula como:  
 $\text{costo} = \text{costoRiegoFinca}(f, \text{pi}) + \text{costoMovilidad}(f, \text{pi}, d)$ 
  - $\text{costoRiegoFinca}$  es correcta, como se demostró previamente.
  - $\text{costoMovilidad}$  se calcula en función de la programación de riego y la distancia, lo cual es correcto por definición.
3. La función  $\text{minBy}(\_._2)$  selecciona la programación con el costo total mínimo. Dado que todos los costos se evaluaron correctamente y  $\text{minBy}$  elige el mínimo, el resultado es correcto para una finca con un solo tablón.

### Paso Inductivo:

Supongamos que  $\text{ProgramacionRiegoOptimo}$  funciona correctamente para una finca con  $k$  tablonos. Es decir, para una finca con  $k$  tablonos, la función:

- Genera todas las programaciones posibles para  $k$  tablonos.
- Calcula correctamente el costo total para cada programación.
- Devuelve la programación con el costo mínimo.

### Finca con $k+1$ tablonos

Ahora consideremos una finca con  $k+1$  tablonos:

$f = \text{Vector}((\text{tsi}_0, \text{tri}_0, \text{pi}_0), \dots, (\text{tsik}, \text{trik}, \text{pik}))$

#### 1. Generación de programaciones:

La función  $\text{generarProgramacionesRiego}(f)$  generará todas las combinaciones posibles de riego para los  $k+1$  tablonos. Este proceso es correcto porque combina las programaciones posibles para  $k$  tablonos con las posibles configuraciones del nuevo tablón  $k+1$ .

## 2. **Cálculo de costos:**

Para cada programación  $p_i$ , el costo total es:

$\text{costo} = \text{costoRiegoFinca}(f, p_i) + \text{costoMovilidad}(f, p_i, d)$

- Por la corrección de  $\text{costoRiegoFinca}$ , sabemos que este término es correcto.
- $\text{costoMovilidad}$  se evalúa para cada programación de manera independiente, y por definición, se calcula correctamente.

## 3. **Selección de la programación óptima:**

La función selecciona la programación con el menor costo usando  $\text{minBy}(\_._2)$ . Dado que cada costo fue calculado correctamente, la selección de la programación óptima también es correcta.

Por lo tanto, el costo mínimo para  $k+1$  tablonos se calcula como:

$\text{costo minimo} = \text{minpi}(\text{costoRiegoFinca}(f, p_i) + \text{costoMovilidad}(f, p_i, d))$

Esto demuestra que la función también es correcta para  $k+1$  tablonos.

## CONCLUSIONES

1. El análisis de las funciones de riego muestra que la implementación de versiones paralelas, como *costoRiegoFincaPar*, resulta en mejoras significativas en el tiempo de ejecución, especialmente para tamaños de finca grandes. Las aceleraciones observadas, que alcanzan hasta un 404% en ciertos casos, evidencian que el paralelismo no solo optimizó el uso de recursos computacionales, sino que también redujo drásticamente los tiempos de procesamiento. Sin embargo, es crucial considerar que para tamaños pequeños, la versión paralela puede introducir sobrecargas que afectan su eficiencia.
2. La comparación entre las versiones secuenciales y paralelas ha resaltado la importancia de la eficiencia en el procesamiento de datos, especialmente en contextos donde se manejan grandes volúmenes de información. La versión paralela, al aprovechar múltiples hilos de ejecución, ofrece una mejora relevante en el tiempo de cálculo, lo que es considerable en aplicaciones prácticas donde el tiempo es un factor determinante.
3. En el informe se puede evidenciar que no en todos los casos la paralelización es la versión más efectiva, al menos para funciones donde los tiempos de ejecución para datos “pequeños” es considerablemente alto y la paralelización tiene un tiempo de ejecución mucho más alto que la secuencial, como se puede apreciar en la función *ProgramaciónRiegoOptimo*.
4. La paralelización ha demostrado ser útil para mejorar el tiempo de ejecución en funciones de riego con grandes volúmenes de datos, al aprovechar múltiples hilos de procesamiento. Sin embargo, para tamaños pequeños de finca, la versión paralela no siempre es más eficiente y, en algunos casos, introduce una sobrecarga que empeora el rendimiento. Por lo tanto, la paralelización es efectiva en escenarios con datos grandes, pero no siempre es la mejor opción cuando los datos son pequeños, ya que la versión secuencial puede ser más rápida.