Programación de Riego – Informe final

Curso: Fundamentos De Programación Funcional Y Concurrente
Fecha: [08/12/2025]
Integrantes: Karen Vanessa Gómez, Brayan Fidel Carmona, Juan Stevan Cruz,

Samuel Ocampo Villegas

1. Introducción

Se modela una finca como un conjunto de tablones con tiempo de supervivencia (ts), tiempo de riego (tr) y prioridad (p). Una programación de riego es una permutación de los tablones. El objetivo es minimizar el costo total = costo de riego + costo de movilidad. Se implementa en Scala con programación funcional, versiones secuencial y paralela, pruebas unitarias y un benchmark comparativo (seq vs par).

---

2. Definiciones y tipos

- Tablon = (Int, Int, Int) → (ts, tr, p)
- Finca = Vector[Tablon]
- Distancia = Vector[Vector[Int]]
- ProgRiego = Vector[Int] (permutación de índices)
- TiempoInicioRiego = Vector[Int]

Funciones clave:

- tsup, treg, prio
- tIR
- costoRiegoTablon, costoRiegoFinca
- costoMovilidad
- generarProgramacionesRiego
- ProgramacionRiegoOptimo

- Versiones
  paralelas: costoRiegoFincaPar, costoMovilidadPar, generarProgramacion
  esRiegoPar, ProgramacionRiegoOptimoPar

3. Proceso (abstracción → código funcional)

3.1 tIR (tiempos de inicio)

Acumula tr en el orden de la permutación pi:

$$tIR(f,\pi)[\pi(k)] = \sum_{j=0}^{k-1} tr(\pi(j))$$

Implementado con recursión de cola sobre la lista pi, sin mutabilidad.

3.2 Costos

Para cada tablón i:

- start = tIR(f, pi)[i], finish = start + tr(i)
- Si $ts(i) - tr(i) \geq start$: $max(0, ts(i) - finish)$
- Si no: $p(i) \cdot (finish - ts(i))$

Costo de movilidad: suma de distancias de pares consecutivos en pi.

3.3 Generación y óptimo

- Generar todas las permutaciones (factorial) y evaluar costo total
  → minBy.
- Paralelo: usar .par en los mapas/sumas; misma semántica.

4. Corrección

4.1 Invariantes

- pi es permutación de 0..n-1 (sin repetición/omisión).
- tIR asigna a cada tablón el acumulado correcto de tr previos.
- Cada tablón cae en exactamente una rama de costo (en margen o penalizado).
- Pureza: sin estado mutable ni efectos laterales.

4.2 Terminación

- tIR: recursión de cola, consume la lista finita pi.
- Costos: map/sum sobre colecciones finitas.
- Óptimo: dominio finito de permutaciones (factorial). Para benchmarking se usa muestreo (K permutaciones) para evitar OOM.
- Paralelo: mismo dominio finito, termina.

4.3 Esbozo de prueba

- tIR: inducción sobre la lista pi; base lista vacía, paso asigna cabeza y avanza acumulado.
- costoRiegoTablon: usa start/finish correctos; aplica la condición de margen o atraso.
- costoMovilidad: suma de arcos consecutivos = definición.
- Óptimo: explora todas las permutaciones y toma el mínimo → coincide con la definición de óptimo global. Paralelo preserva el mismo conjunto y criterio.

---

5. Casos de prueba documentados (≥5 por función)

tIR

1. tr=(2), pi=[0] → [0]
2. tr=(2,3), pi=[1,0] → tIR(1)=0, tIR(0)=3
3. tr=(1,1,1), pi=[0,1,2] → [0,1,2]
4. tr=(1,4,2), pi=[2,0,1] → [2→0, 0→2, 1→3]
5. tr=(2,2,2,2), pi=[3,2,1,0] → [3→0, 2→2, 1→4, 0→6]

costoRiegoTablon (usa tIR)

1. ts=10,tr=2,p=1,start=0 → finish=2 ≤ ts → 8
2. ts=5,tr=2,p=3,start=3 → finish=5=ts → 0
3. ts=6,tr=4,p=2,start=5 → finish=9>ts → 6
4. ts=4,tr=1,p=5,start=0 → finish=1 ≤ ts → 3

5. ts=7,tr=3,p=4,start=2 → finish=5 ≤ ts → 2

costoRiegoFinca

1. Un tablón → ese costo
2. Dos tablones (8,0) → 8
3. Tres (3,6,2) → 11
4. Cuatro todos en margen → suma de sobrantes
5. Mixto (margen + atrasos) → suma con prioridad

costoMovilidad

1. |pi|=1 → 0
2. pi=[0,1], d=5 → 5
3. pi=[1,2,0], d(1,2)=3, d(2,0)=4 → 7
4. pi=[2,1,3,0] → suma de arcos consecutivos
5. pi=[0,3,1,2] → d(0,3)+d(3,1)+d(1,2)

Óptimo (seq/par)

1. n=1 → única permutación = óptimo
2. n=2 → mínimo entre [0,1] y [1,0]
3. n=3 simétrica → seq = par
4. Caso F1 (rúbrica): Prog2 mejor (38 vs 45)
5. Caso F2 (rúbrica): Prog1 mejor (36 vs 41)

---

6. Paralelización

- Tipo: paralelización de datos con .par y CollectionConverters._.
- Funciones paralelas: costoRiegoFincaPar, costoMovilidadPar, evaluación de costos sobre el conjunto muestreado de programaciones.
- Semántica: misma que secuencial, sin estado compartido.

Ley de Amdahl

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

Para n y K pequeños, el overhead puede dominar → desaceleración.

---

7. Benchmark (muestra K=200 permutaciones, 3 corridas, n pequeño)

Tamaño	Perms	Secuencial (ms)	Paralelo (ms)	Aceleración (%)
8	200	12.71	44.25	-71.27
10	200	3.04	12.08	-74.86
12	200	2.78	5.78	-51.83

Interpretación: Para cargas pequeñas, el overhead de paralelizar supera el trabajo útil (consistente con Amdahl). Para ver speedup, se necesitaría más trabajo por tarea (mayor K o n), cuidando de no disparar factorial ni OOM.

---

8. Conclusiones

- Se cumple la especificación: tiempos de inicio, costos, óptimo global y versiones paralelas con misma semántica.
- Terminación garantizada en dominios finitos; para benchmarking se muestrea el espacio de permutaciones.
- Pureza funcional: sin mutabilidad ni efectos laterales; recursión de cola donde aplica.
- Paralelización correcta pero con overhead dominante en casos pequeños; ajustable elevando la carga de trabajo.
- Benchmarks, pruebas y documentación alineados con la rúbrica.

---

9. Cómo ejecutar

- Pruebas: ./gradlew test
- Benchmark: ./gradlew test --tests taller.BenchmarkRiego (produce la tabla anterior en consola).