Fundamentos de Programación Funcional y Concurrente

Complejidad de la concurrencia

Juan Francisco Díaz Frias

Profesor Titular (1993-hoy)
juanfco.diaz@correounivalle.edu.co
Edif. B13 - 4009
Universidad del Valle

Octubre 2023

Plan

1. ¿Cuán rápidos son los programas paralelos?
   - Complejidad de los programas secuenciales
   - Complejidad de los programas paralelos
2. Evaluación comparativa (*benchmarking*) de programas paralelos
   - Pruebas y Evaluación Comparativa
   - Evaluación comparativa usando *ScalaMeter*

Plan

1. ¿Cuán rápidos son los programas paralelos?
   - Complejidad de los programas secuenciales
   - Complejidad de los programas paralelos
2. Evaluación comparativa (*benchmarking*) de programas paralelos
   - Pruebas y Evaluación Comparativa
   - Evaluación comparativa usando *ScalaMeter*

¿Cuán rápidos son los programas paralelos?

- El desempeño es la motivación esencial para programar en paralelo. ¿Cómo se estima?
  - Medidas empíricas
  - Análisis asintótico
- El análisis asintótico es importante para entender de qué forma escalan los algoritmos cuando:
  - El tamaño de la entrada crece
  - Los recursos de hardware paralelo disponible crecen

  Se realiza, típicamente, con el peor caso y no con los casos promedio.

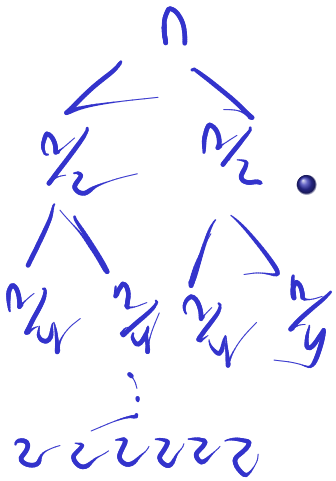
Plan

1. ¿Cuán rápidos son los programas paralelos?
   - Complejidad de los programas secuenciales
   - Complejidad de los programas paralelos
2. Evaluación comparativa (*benchmarking*) de programas paralelos
   - Pruebas y Evaluación Comparativa
   - Evaluación comparativa usando *ScalaMeter*

Análisis asintótico de los programas secuenciales

- El análisis asintótico caracteriza el comportamiento de los programas secuenciales acotando (superiormente) el número de operaciones que un programa realiza en función del tamaño de la entrada.
  - Sea $f(n)$, el número de operaciones que realiza un programa secuencial para insertar en el puesto adecuado un elemento en *una lista ordenada* de n elementos. Usando las técnicas apropiadas (en un curso de análisis de algoritmos, las aprenderán) podemos determinar que $\exists c, n_0 : f(n) \leq cn, \forall n \geq n_0$, lo que se denota:

$$f(n) = \mathcal{O}(n)$$



- Sea $f(n)$, el número de operaciones que realiza un programa secuencial para insertar en el puesto adecuado un elemento en *un árbol binario balanceado* de n elementos. Usando las técnicas apropiadas (en un curso de análisis de algoritmos, las aprenderán) podemos determinar que $\exists c, n_0 : f(n) \leq c\ln(n), \forall n \geq n_0$, lo que se denota:

$$f(n) = \mathcal{O}(\ln(n))$$

Análisis asintótico de *sumSegment*

Recordemos el código de *sumSegment* para calcular la $norma - p$

```
0  def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Double = {
1    var i= s; var sum: Double = 0
2    while (i < t) {
3      sum= sum + power(a(i), p)
4      i= i + 1
5    }
6    sum
7  }
```

Sea $W(s,t)$ el número de operaciones que realiza $sumSegment(a,p,s,t)$.

$$W(s,t) = \mathcal{O}(t-s)$$

- $t - s$ iteraciones
- un trabajo de costo constante en cada iteración

Análisis asintótico de *segmentRec* secuencial

Recordemos el código de *segmentRec* secuencial para calcular la $norma - p$

```
0  def segmentRec(a: Array[Int], p: Double, s: Int, t: Int):Double = {
1    if (t - s < limite)
2      sumSegment(a, p, s, t) // Segmento chico, suma secuencial
3    else {
4      val m = s + (t - s)/2
5      val (sum1, sum2) = (segmentRec(a, p, s, m),
6                          segmentRec(a, p, m, t))
7      sum1 + sum2 } }
```

Sea $W(s,t)$ el número de operaciones que realiza $segmentRec(a,p,s,t)$ secuencial.

$$W(s,t) = \begin{cases} \mathcal{O}(t-s) & \text{si} t-s < limite \\ W(s,m) + W(m,t) + k & \text{sino}, m = s + (t-s)/2 \end{cases}$$

Al usar métodos formales para resolver esta ecuación de recurrencia, se obtiene:

$$W(s,t) = \mathcal{O}(t-s)$$

Análisis asintótico de *segmentRec* paralelo

Recordemos el código de *segmentRec* paralelo para calcular la $norma - p$

```
0  def segmentRec(a: Array[Int], p: Double, s: Int, t: Int):Double = {
1    if (t - s < limite)
2      sumSegment(a, p, s, t) // Segmento chico, suma secuencial
3    else {
4      val m = s + (t - s)/2
5      val (sum1, sum2) = parallel(segmentRec(a, p, s, m),
6                                  segmentRec(a, p, m, t))
7      sum1 + sum2 } }
```

Sea $D(s,t)$ el número de operaciones que realiza $segmentRec(a,p,s,t)$ paralelo.

$$D(s,t) = \begin{cases} \mathcal{O}(t-s) & \text{si} t - s < limite \\ max(D(s,m), D(m,t)) + k & \text{sino,} m = s + (t-s)/2 \end{cases}$$

Al usar métodos formales para resolver esta ecuación de recurrencia, se obtiene:

$$D(s,t) = \mathcal{O}(\ln(t-s))$$

suponiendo que el paralelismo es ilimitado.

Plan

1. ¿Cuán rápidos son los programas paralelos?
   - Complejidad de los programas secuenciales
   - Complejidad de los programas paralelos
2. Evaluación comparativa (*benchmarking*) de programas paralelos
   - Pruebas y Evaluación Comparativa
   - Evaluación comparativa usando *ScalaMeter*

Trabajo (*Work*) y Profundidad (*Deep*)

- Calcular la complejidad asintótica de programas paralelos depende de los recursos paralelos disponibles.
- Se usarán dos medidas para un programa paralelo:
  - El trabajo $W(e)$: número de operaciones que se toma calcular e si no hay paralelismo
  - La profundidad $D(e)$: tamaño de la secuencia máxima de operaciones secuenciales, suponiendo un paralelismo ilimitado.
- Reglas clave:
  - $W(parallel(e_1, e_2)) = W(e_1) + W(e_2) + c_1$
  - $D(parallel(e_1, e_2)) = max(D(e_1), D(e_2)) + c_2$
  - $W(f(e_1, \ldots, e_n)) = W(e_1) + \ldots + W(e_n) + W(f)(v_1, \ldots, v_n)$
  - $D(f(e_1, \ldots, e_n)) = D(e_1) + \ldots + D(e_n) + D(f)(v_1, \ldots, v_n)$

Calculando límites de tiempo para programas paralelos

- Suponga que conocemos $W(e)$ y $D(e)$ y que nuestra plataforma ejecuta hasta P hilos en paralelo.
- Sin importar P, el programa se demora por lo menos $D(e)$ debido a las dependencias.
- Sin importar $D(e)$, el programa se demora por lo menos $W(e)/P$ porque todas las operaciones se deben hacer, y con P hilos eso sería lo más rápido.
- Es razonable entonces, estimar el tiempo de ejecución de el programa paralelo así:

$$D(e) + W(e)/P$$

Nótese que:

- Si P es constante, a medida que el tamaño de la entrada crece, la complejidad asintótica tiende a $W(e)$
- Aunque se tuvieran infinitos recursos ($P \leftarrow \infty$) , la complejidad asintótica tiende a $D(e)$ (nunca es cero)

Caso concreto: *segmentRec* en paralelo

- Para *segmentRec* se tiene:
  - $W(s,t) = \mathcal{O}(t-s)$
  - $D(s,t) = \mathcal{O}(\ln(t-s))$
- El tiempo de ejecución de el programa paralelo es:

$$\mathcal{O}(\ln(t-s)) + \mathcal{O}(t-s)/P$$

  Nótese que:
  - Si P es constante, a medida que el tamaño de la entrada crece, la complejidad asintótica tiende a $\mathcal{O}(t-s)$
  - Si P crece, la complejidad asintótica tiende a $\mathcal{O}(\ln(t-s))$

Paralelismo y ley de Amhdal

- Suponga que tenemos dos partes de una computación secuencial:
  - La parte 1 toma una fracción f del tiempo de computación (digamos, 40 %)
  - La parte 2 toma la fracción restante $1 - f$ del tiempo de computación (digamos, 60 %) y se puede acelerar
- Si se logra hacer la parte 2 P veces más rápido, la aceleración sería:

$$\frac{1}{(f + \frac{1-f}{P})}$$

Si $P = 100$ y $f = 0{,}4$, la aceleración es 2,46.
Aún si se logra hacer la parte 2 infinitamente más rápida, la aceleración máxima que se puede obtener es 2,5

Plan

1. ¿Cuán rápidos son los programas paralelos?
   - Complejidad de los programas secuenciales
   - Complejidad de los programas paralelos
2. Evaluación comparativa (*benchmarking*) de programas paralelos
   - Pruebas y Evaluación Comparativa
   - Evaluación comparativa usando *ScalaMeter*

Pruebas y Evaluación Comparativa

- Pruebas: aseguran que una parte del programa se comporta de acuerdo a lo esperado

$$reverse(List(1, 2, 3)) == List(3, 2, 1)$$

- Evaluación comparativa: calcula métricas de desempeño para partes del programa (tiempo de ejecución, memoria utilizada, uso del disco, latencia, ...). Su resultado es una variable aleatoria.

$$tiempoDeEjecucion(reverse(List(1, 2, 3)))$$

- Utilidad diferente:
  - Pruebas: su salida es típicamente binaria: 1, el programa pasó correctamente la prueba; 0, el programa tiene un error en ese caso.
  - Evaluación comparativa: su salida es típicamente un valor continuo, y cada que se prueba su resultado puede ser (ligeramente) diferente

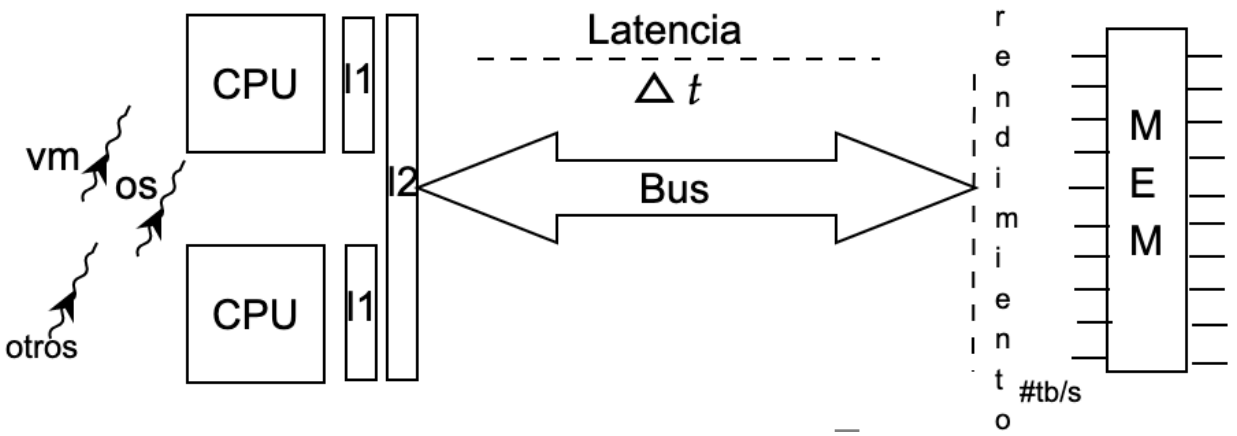
Evaluación comparativa de programas paralelos

- ¿Por qué hacer evaluación comparativa de programas paralelos?
  Porque los beneficios en términos de desempeño, son la principal razón por la que se hacen programas paralelos.
  Se necesita asegurar que el esfuerzo se ve reflejado, por ejemplo, en mejoras ostensibles del tiempo de ejecución.
- La evaluación comparativa de programas paralelos es mucho más importante que la de los programas secuenciales, porque el solo hecho de paralelizar es un cuello de botella.

Factores de desempeño

El desempeño, especialmente el tiempo de ejecución, depende de muchos factores:

- Velocidad del procesador
- Número de procesadores
- Latencia y rendimiento del acceso a la memoria
- Comportamiento de la memoria *caché*
- Comportamiento en tiempo de ejecución (e.g. recolector de basura, compilación JIT, planificación de hilos, . . .)



Metodologías para medir el desempeño

Medir el desempeño es difícil, pues típicamente la métrica se comporta como una variable aleatoria. ¿Qué podemos hacer?

- Múltiples repeticiones
- Tratamiento estadístico (media, varianza, . . .)
- Eliminar casos atípicos
- Medir en *estado estable* (calentamiento o *warm-up*)
- Prevenir anomalías (GC, JIT, optimizaciones agresivas)

Plan

1. ¿Cuán rápidos son los programas paralelos?
   - Complejidad de los programas secuenciales
   - Complejidad de los programas paralelos
2. Evaluación comparativa (*benchmarking*) de programas paralelos
   - Pruebas y Evaluación Comparativa
   - Evaluación comparativa usando *ScalaMeter*

Scalameter

ScalaMeter es un marco para la evaluación comparativa y pruebas de rendimiento usando análisis de regresión para JVM.

- Pruebas de rendimiento usando análisis de regresión: compara el desempeño de la ejecución actual del programa contra el desempeño en ejecuciones previas.
- Evaluación comparativa (*benchmarking*): mide el desempeño de la ejecución actual del programa

En este curso nos enfocaremos en *benchmarking*

Usando *ScalaMeter*

- Primero que todo, hay que añadir *ScalaMeter* como una dependencia (en *build.sbt*):

```
0  libraryDependencies += "com.storm-enroute" %% "scalameter-core" % "0.21"
```

- Luego, se puede importar el paquete y usarlo (en este ejemplo para convertir un rango en una matriz):

```
0  import org.scalameter._
1  val time = measure {
2    (0 until 1000000).toArray
3  }
4  measure {(0 until 1000000).toArray}
5  measure {(0 until 1000000).toArray}
6  measure {(0 until 1000000).toArray}
7  measure {(0 until 1000000).toArray}
8  measure {(0 until 1000000).toArray}
9  measure {(0 until 1000000).toArray}
10 measure {(0 until 1000000).toArray}
11 measure {(0 until 1000000).toArray}
12 measure {(0 until 1000000).toArray}
```

Demo . . .

Este es el resultado de correr varias veces la medición:

```
0  scala> import org.scalameter._
1  import org.scalameter._
2  scala> measure {(0 until 1000000).toArray}
3  val res0: org.scalameter.Quantity[Double] = 40.14966 ms
4  val res1: org.scalameter.Quantity[Double] = 9.392579 ms
5  val res2: org.scalameter.Quantity[Double] = 8.081661 ms
6  val res3: org.scalameter.Quantity[Double] = 9.661883 ms
7  val res4: org.scalameter.Quantity[Double] = 10.109058 ms
8  val res5: org.scalameter.Quantity[Double] = 9.395074 ms
9  val res6: org.scalameter.Quantity[Double] = 9.284056 ms
10 val res7: org.scalameter.Quantity[Double] = 10.446015 ms
11 val res8: org.scalameter.Quantity[Double] = 10.889441 ms
12 val res9: org.scalameter.Quantity[Double] = 9.671305 ms
13 val res10: org.scalameter.Quantity[Double] = 7.910284 ms
14 val res11: org.scalameter.Quantity[Double] = 8.538095 ms
15 val res12: org.scalameter.Quantity[Double] = 9.82974 ms
16 val res13: org.scalameter.Quantity[Double] = 27.050807 ms
17 val res14: org.scalameter.Quantity[Double] = 6.97913 ms
18 val res15: org.scalameter.Quantity[Double] = 6.482545 ms
```

- Nótese que los tiempos de dos ejecuciones consecutivas no son iguales
- Se necesita un tiempo antes de que el programa alcance su máximo desempeño (*warm-up*)
- Hay procesos que influyen: GC, Optimizaciones de compilación dinámicas, políticas del planificador de hilos, . . .

Calentamiento de la *JVM*

- El ejercicio muestra tiempos diferentes entre dos ejecuciones sucesivas del mismo programa.
- Cuando un programa se lanza en la *JVM*, transcurre un tiempo (*warm-up*) antes que el programa logre su máximo desempeño:
  - Primero, el programa es interpretado
  - Luego, algunas partes del programa se compilan a código de máquina
  - Más tarde, la *JVM* puede decidir aplicar optimizaciones adicionales
  - Finalmente, el programa alcanza un estado estable

Calentadores de *ScalaMeter*

- En general, se desea medir el desempeño en estado estable
- Para ellos, los objetos calentadores (*warmers*) de *ScalaMeter* ejecutan la medición deseada cuando detectan que el sistema ya está en estado estable:

```
0  scala> measure {(0 until 1000000).toArray}
1  val res83: org.scalameter.Quantity[Double] = 9.015408 ms
2
3  scala> withWarmer(new Warmer.Default) measure {(0 until 1000000).toArray}
4  val res84: org.scalameter.Quantity[Double] = 7.550117 ms
```

Nótese que estado estable la medición es mejor (verlo en vivo).

Configuración de *ScalaMeter*

- *ScalaMeter* tiene una cláusula de configuración que permite especificar varios parámetros como el número máximo y mínimo de ejecuciones de calentamiento:

```
0  val time = config(
1    KeyValue(Key.exec.minWarmupRuns -> 20),
2    KeyValue(Key.exec.maxWarmupRuns -> 60),
3    KeyValue(Key.verbose -> true)
4  ) with Warmer(new Warmer.Default) measure { (0 until 1000000).toArray}
```

- *scalaMeter* puede ser usado para medir otras cosas:
  - *Measurer.Default*: tiempo de ejecución
  - *IgnoringGC*: tiempo de ejecución sin pausas de recolección de basura
  - *OutlierElimination*: elimina casos atípicos
  - *MemoryFootprint*: huella de memoria de un objeto
  - *GarbageCollectionCycles*: número total de pausas de GC

```
0  scala> withMeasurer (new Measurer.MemoryFootprint) measure {(0 until 1000000).toArray}
1  val res58: org.scalameter.Quantity[Double] = 4194.264 kB
```