Fundamentos de Programación Funcional y Concurrente

Abstracciones para la concurrencia

Juan Francisco Díaz Frias

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

Universidad del Valle

Octubre 2023

Plan

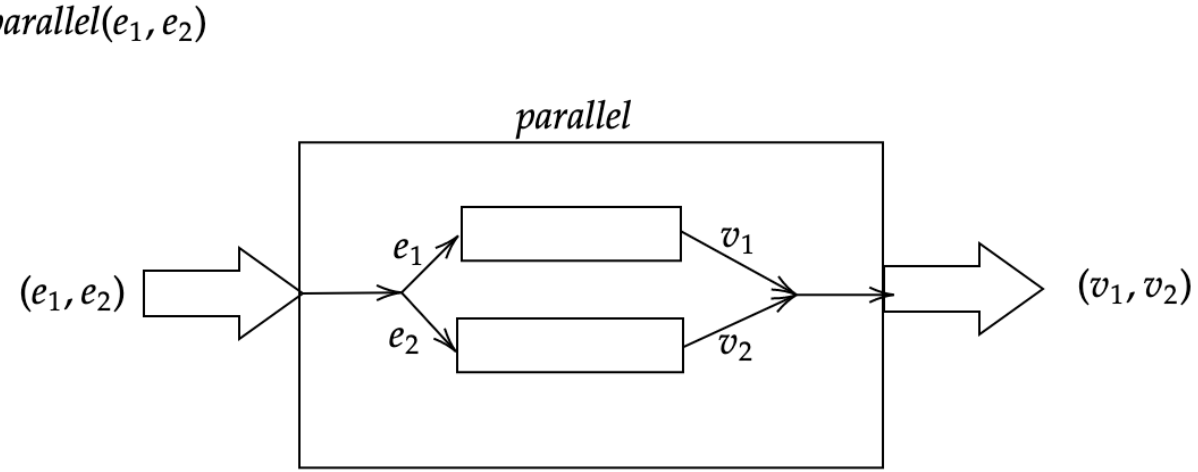
Plan

1. La abstracción *parallel*
   - Corriendo computaciones en paralelo
   - El cálculo de π en paralelo
2. La abstracción *task*
   - Tareas de primera clase
   - El cálculo de la $norma - p$ en paralelo

Contents

La abstracción *parallel*

- ¿Cuál podría ser una forma sencilla de indicar, explícitamente, que dos computaciones deben realizarse en paralelo? : la abstracción *parallel*
- *parallel*(e_1, e_2) toma dos expresiones e_1 y e_2 las computa en paralelo, y devuelve una pareja con los dos resultados.



Ejemplo: calculando la $norma - p$

- Dado un vector representado por un arreglo de enteros, se requiere calcular la $norma - p$.
- La $norma - p$ es una generalización de la noción de longitud de la geometría
  - La $norma - 2$ de un vector de n dimensiones $(a_0, a_1, \ldots, a_{n-1})$ es $(a_0^2 + a_1^2 + \ldots + a_{n-1}^2)^{\frac{1}{2}} = (\sum_{i=0}^{n-1} a_i^2)^{\frac{1}{2}}$
  - La $norma - p$ de un vector de n dimensiones $(a_0, a_1, \ldots, a_{n-1})$ es $(a_0^p + a_1^p + \ldots + a_{n-1}^p)^{\frac{1}{p}} = (\sum_{i=0}^{n-1} a_i^p)^{\frac{1}{p}}$
- Nótese que para calcular $\sum_{i=0}^{n-1} a_i^p$ es suficiente con tener una manera de calcular la suma para un segmento del arreglo entre s y t con $0 \leq s \leq t \leq n$:

$$\sum_{i=s}^{t-1} a_i^p$$

Calculando las sumas de potencias para un segmento

- Considere el siguiente programa en Scala para calcular

$$\sum_{i=s}^{t-1} a_i^p$$

```
0  def power(x: Double, p: Double): Double = math.exp(p * math.log(abs(x)))
1
2  // Calculando la p-norm secuencialmente
3
4  def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Double = {
5    var i= s; var sum: Double = 0
6    while (i < t) {
7      sum= sum + power(a(i), p)
8      i= i + 1
9    }
10   sum
11 }
```

- Calcular la $norma - p$, $(\sum_{i=0}^{n-1} a_i^p)^{\frac{1}{p}}$, es cuestión sencillamente de invocar *sumSegment*:

```
0  def pNorm(a: Array[Int], p: Double): Double =
1    power(sumSegment(a, p, 0, a.length), 1/p)
```

Dividiendo el problema en dos subproblemas

- Observe que para calcular la $norma - p$, $(\sum_{i=0}^{n-1} a_i^p)^{\frac{1}{p}}$, la sumatoria se puede partir en 2:

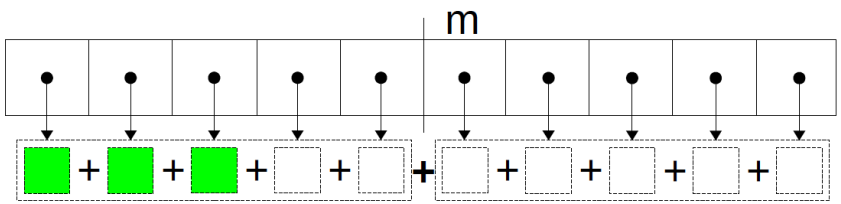
$$(\sum_{i=0}^{n-1} a_i^p)^{\frac{1}{p}} = ((\sum_{i=0}^{m-1} a_i^p) + (\sum_{i=m}^{n-1} a_i^p))^{\frac{1}{p}}, 0 \leq m < n$$

- La función resultante es:

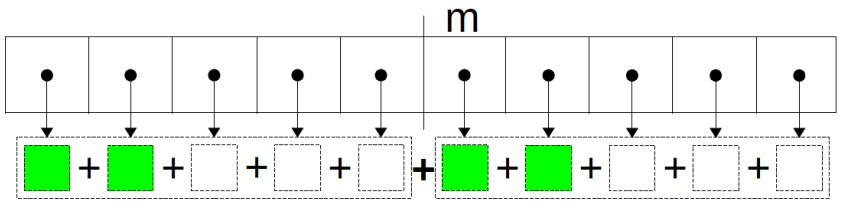
```
0  def pNormTwoPart(a: Array[Int], p: Double): Double = {
1    val m = a.length / 2
2    val (sum1, sum2) = (sumSegment(a, p, 0, m),
3      sumSegment(a, p, m, a.length))
4    power(sum1 + sum2, 1/p) }
```

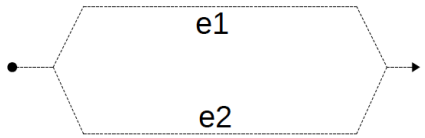
Comparando la ejecución de las dos versiones

```
val (sum1, sum2) = (sumSegment(a, p, 0, m),
                    sumSegment(a, p, m, a.length))
```




```
val (sum1, sum2) = parallel(sumSegment(a, p, 0, m),
                            sumSegment(a, p, m, a.length))
```

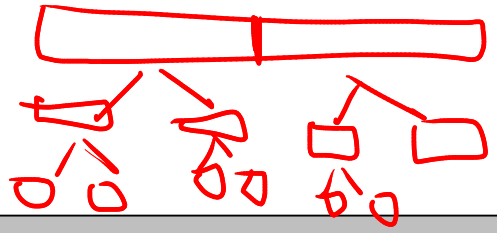




Tomado de transparencias de Victor Kunkac, EPFL

Nótese que la versión paralela puede tardar algún tiempo en configurar la ejecución paralela, pero después de eso podría progresar en el procesamiento de elementos del arreglo dos veces más rápido que la versión secuencial.

¿Se puede hacer con un número ilimitado de hilos?



```
0  // Calculando la norma-P para un numero no limitado de hilos
1  val limite = 2
2
3  // Lanza tantas sumas en paralelo como sea necesario
4  def segmentRec(a: Array[Int], p: Double, s: Int, t: Int):Double = {
5    if (t - s < limite)
6      sumSegment(a, p, s, t) // Segmento chico, suma secuencial
7    else {
8      val m = s + (t - s)/2
9      val (sum1, sum2) = parallel(segmentRec(a, p, s, m),
10       segmentRec(a, p, m, t))
11     sum1 + sum2 } }
12
13 def pNormRec(a: Array[Int], p: Double): Double =
14   power(segmentRec(a, p, 0, a.length), 1/p)
```

Tipo o *signature* de *parallel*

¿Cómo está definida *parallel* internamente?

$$def\ \ parallel[A, B](tareaA : => A, tareaB : => B) : (A, B) = \{\ldots\}$$

- $parallel(e_1, e_2)$ devuelve lo mismo que (e_1, e_2)
- Ventaja: $parallel(e_1, e_2)$ pude ser más rápido que (e_1, e_2)
- Nótese que el tipo de los argumentos de *parallel* se pasan por nombre. ¿Por qué?
- Considere la siguiente definición de $parallel1$:

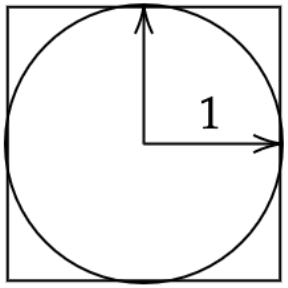
$$def\ \ parallel1[A, B](tareaA : A, tareaB : B) : (A, B) = \{\ldots\}$$

¿Hay diferencia entre ejecutar $parallel(e_1, e_2)$ y $parallel1(e_1, e_2)$?

¿Qué se pone en juego en el sistema cuando usamos *parallel*?

- El paralelismo eficiente requiere soporte de:
  - El lenguaje de programación y las librerías
  - La máquina virtual (en este caso, *JVM*)
  - El sistema operativo
  - El hardware
- La implementación de *parallel* utiliza los hilos de la *JVM*:
  - Los hilos de la *JVM* se asocian a hilos del sistema operativo.
  - El sistema operativo planifica la ejecución de los hilos en los múltiples núcleos.
- Si los recursos son suficientes, el programa en paralelo puede correr más rápido.
- Un programa pensado para correr en paralelo, puede correr incluso cuando sólo hay un procesador con un núcleo disponible (aunque sin aceleración)

El método Montecarlo para estimar π



- Considere un círculo de radio 1 y un cuadrado de lado 2 con el mismo centro.
- Sea λ la proporción del área del círculo con respecto al área del cuadrado:

$$\lambda = \frac{\pi \times 1^2}{2^2} = \frac{\pi}{4}$$

- Se puede estimar λ haciendo un muestreo aleatorio de puntos dentro del cuadrante superior derecho del cuadrado y mirar qué porcentaje de ellos cae dentro del círculo.
  Multiplicar este porcentaje por 4 es una estimación de π

Estimación secuencial de π

```
0  import scala.util.Random
1
2  def mcCount(iter: Int): Int = {
3    val randomX = new Random
4    val randomY = new Random
5    var hits = 0
6    for (i <- 0 until iter) {
7      val x = randomX.nextDouble() // in [0,1]
8      val y = randomY.nextDouble() // in [0,1]
9      if (x*x + y*y < 1) hits= hits + 1
10   }
11   hits
12 }
13
14 def monteCarloPiSeq(iter: Int): Double = 4.0 * mcCount(iter) / iter
15 monteCarloPiSeq(500000)
```

Estimación paralela de π

```
0  import scala.util.Random
1
2  def mcCount(iter: Int): Int = {
3    val randomX = new Random
4    val randomY = new Random
5    var hits = 0
6    for (i <- 0 until iter) {
7      val x = randomX.nextDouble() // in [0,1]
8      val y = randomY.nextDouble() // in [0,1]
9      if (x*x + y*y < 1) hits= hits + 1
10   }
11   hits
12 }
13
14 def monteCarloPiPar(iter: Int): Double = {
15   val ((pi1, pi2), (pi3, pi4)) =
16     parallel(
17       parallel(mcCount(iter/4), mcCount(iter/4)),
18       parallel(mcCount(iter/4), mcCount(iter - 3*(iter/4))))
19   4.0 * (pi1 + pi2 + pi3 + pi4) / iter
20 }
21 monteCarloPiPar(500000)
```

Contents

task: una abstracción más flexible para expresar paralelismo

- $val\ (v_1, v_2) = parallel(e_1, e_2)$
  se puede escribir alternativamente usando la abstracción *task*:
  $val\ t_1 = task(e_1)$
  $val\ t_2 = task(e_2)$
  $val\ v_1 = t_1.join$
  $val\ v_2 = t_2.join$
- $t = task(e)$ lanza una computación de e en paralelo con el hilo actual (*in the background*)
  - t es una tarea que realiza el cálculo de e
  - La computación actual continúa en paralelo con t
  - Para obtener el resultado de t se usa $t.join$
  - $t.join$ suspende el hilo actual, y espera hasta que el resultado sea calculado
  - Posteriores invocaciones a $t.join$ devuelven inmediatamente el resultado ya calculado
- En términos prácticos: $task(e).join == e$

Revisitando el cálculo de la *norma* − *p* en paralelo

Usando la abstracción *task*, el cálculo de la *norma* − *p* se escribe así:

```
0  // Calculando la p-norm en paralelo, con la abstraccion task
1
2  def pNormFourPartParTask(a: Array[Int], p: Double): Double = {
3    val m1 = a.length/4; val m2 = a.length/2; val m3 = 3*a.length/4
4    val t1 = task (sumSegment(a, p, 0, m1))
5    val t2 = task (sumSegment(a, p, m1, m2))
6    val t3 = task (sumSegment(a, p, m2, m3))
7    val t4 = task (sumSegment(a, p, m3, a.length))
8    val sum=t1.join() + t2.join() + t3.join() + t4.join()
9    power(sum, 1/p)
10   }
```

Definiendo *parallel* en términos de *task*

- ¿Cómo se definiría *parallel* en términos de *task*?:

$$def\ parallel[A, B](a : => A, b : => B) : (A, B) = \{\ldots\}$$

¿Cuál de las siguientes dos versiones es correcta? ¿Cuál no? ¿por qué?

- Versión 1:

```
0    def parallel[A, B](a: => A, n: => B): (A, B) = {
1      val tb= task(b)
2      val ta = a
3      (ta, tb.join())
4    }
```

- Versión 2:

```
0    def parallel[A, B](a: => A, n: => B): (A, B) = {
1      val tb= task(b).join
2      val ta = a
3      (ta, tb.join())
4    }
```