INFORME TALLER 3

Docente

DELGADO SAAVEDRA CARLOS ANDRES

Estudiantes

Juan David García Arroyave 2359450

Juan José Hincapie Tascón 2359493

Sebastián Zacipa Martínez 2359695

Universidad del Valle-Sede Tuluá

2024

Informe de procesos

```scala
type Matriz = Vector[Vector[Int]]

def matrizAlAzar(long:Int, vals: Int) : Matriz = {

  val v = Vector.fill(long, long){scala.util.Random.nextInt(vals)}
  v
}
```

En esta sección se tiene inicializa un tipo de Dato llamando Matriz que consiste en Vector que tiene como elementos otros vectores

Para la primera función tenemos parámetros como long que indica las dimensiones de la matriz cuadrada y vals que indica el limite de los valores que pueden ingresar, se utiliza Random para generar matrices aleatorias

Pila de llamadas para el valor matrizAlAzar(2,2) tendremos:

Vector.fill(2,2) Genera una matriz 2x2

(long, long){scala.util.Random.nextInt(2) Llenará la matriz con valores menores que 2 (0,1)

```scala
def vectorAlAzar(long:Int, vals: Int) : Vector[Int] = {
  val v = Vector.fill(long){scala.util.Random.nextInt(vals)}
  v
}
```

Crea un vector de enteros de longitud long con valores aleatorios entre 0 y vals

Pila de llamadas para los valores 2, 2

Vector.fill(2)

Scala.util.ramdom.nextInt(2)

Genera un vector con elementos entre 0 y 2

```scala
24    def prodPunto(v1: Vector[Int], v2: Vector[Int]) : Int = {
25      (v1 zip v2).map{case (i,j) => i*j}.sum
26
27    }
```

EL producto punto a través del método zip combina los valores de dos vectores y multiplica entre las diferentes parejas de valores, para a continuación sumar los productos

Pila de llamadas para los valores (1,2,3) y (4,5, 6)

Llamado de la función zip

(v1 zip v2) lo que genera (1,4) (2,5) (3,6)

Llamado función .map multiplica cada tupla (1*4), (2 * 5), (3 * 6)

Vector(4,10,18)

Aplicar el método .sum genera un valor entero 4, 10, 8

Esta función retorna una función transpuesta , ubica en la posición i, j el valor de la matriz j, i

Pila de llamadas para los valores vector = ((1,2) (3,4))

```
def transpuesta(m: Matriz): Matriz = {
  val l = m.length
  Vector.tabulate(l,l){(i,j) => m(j)(i)}
}
```

Genera un vector 2.2

Vector.tabulate(2,2){(0,0) => m(0)(0)} en la posición 0,0 se ubica el element de la matriz m en la posición 0,0

Vector.tabulate(2,2){(0,1) => m(1)(0)} en la posición 0,1 se ubica el element de la matriz m en la posición 1,0

Vector.tabulate(2,2){(1,0) => m(0)(1)} en la posición 1,0 se ubica el element de la matriz m en la posición 0,1

Vector.tabulate(2,2){(1,1) => m(1)(1)} en la posición 1,1 se ubica el element de la matriz m en la posición 1,1

```scala
def multMatriz(m1: Matriz, m2:Matriz): Matriz = {
  val m2transpuesta = transpuesta(m2)
  val dimension = m1.length

  Vector.tabulate(dimension, dimension){(i,j) => prodPunto(m1(i), m2transpuesta(j))}
}
```

Función que realiza multiplicación de manera secuencial

Pila de llamadas para matrices [[1,2], [3,4]] y [[1,2], [3,4]]

- val m2transpuesta = transpuesta([[1,2], [3,4]]) salida [[1,3], [2,4]
- val dimension = m1.length para este caso tenemos 2, para una matriz cuadrado 2x2
- Vector.tabulate(2, 2){(0,0) => prodPunto(m1([1,2],), m2transpuesta([1,3]))} con el prodpunto Obtenemos parejas (1 * 1) y (2 * 3) y la suma de estos productos 1 * 1 + 2 * 3 = 1 + 6 = 7
- prodPunto([1, 2], [2, 4]) 1 * 2 + 2 * 4 = 2 + 8 = 10
- prodPunto([3, 4], [1, 3]) 3 * 1 + 4 * 3 = 3 + 12 = 15
- prodPunto([3, 4], [2, 4]) 3 * 2 + 4 * 4 = 6 + 16 = 22
- Resultado [[7, 10],
- [15, 22]]

```scala
def multMatrizPar(A:Matriz, B:Matriz): Matriz = {
  val Btranspuesta = transpuesta(B)
  val dimension = A.length
  val Task = for {
    i <- 0 until dimension
    j <- 0 until dimension
  } yield task {(i,j,prodPunto(A(i), Btranspuesta(j)))}
  val resultado = Task.map(_.join()).toVector
  Vector.tabulate(dimension, dimension){(i,j) => resultado.find(res => res._1 == i && res._2 == j).map(_._3).getOrE

}
```

Función que realiza la multiplicación de dos matrices de paralela

Pila de llamadas para [[1,2], [3,4]] y [[1,2], [3,4]]

- Realizar la transpuesta de la matriz B [[1,3] [2,4]]
- Calcular la dimensión de la matriz A.lenght, esto es igual a 2

- Generar todas las tareas necesarias con un for, dividir las tareas en este caso:
  I desde 0 hasta 2 y j desde hasta 0
- Se generan cuatro tareas; tarea 1: (i = 0, j= 0) calcula prodPunto([1, 2], [1,3]) 1*1 + 2 * 3 = 7. Tarea 2: (i = 0, j=1) calcula prodPunto ([1,2] , [2,4]) 1*2 + 2*4 = 10. Tarea 3 (i = 1, j= 0) calcula prodPunto ([3,4], [1,3]) 3*1 + 4*3 = 15. Tarea 4 calcula prodPunto ([3,4],[2,4]) 3*2 + 4 * 4 = 22
- Con el Task.map(.join) se unen los resultados de cada Tarea
- Vector.tabulate(dimension, dimension){(i,j) => resultado.find(res => res._1 == i && res._2 == j).map(_._3).getOrElse(0)} esta parte toma la posición y ubica el tercer el elemento en esa posición de la nueva matriz

```
def subMatriz(m: Matriz, i:Int, j:Int, l:Int): Matriz = {
  Vector.tabulate(l,l){(x,y) => m(i+x)(j+y)}
}
```

- Esta función se encarga de tomar una matriz mas pequeña a partir de los valores i y j
- Pila de llamadas para subMatriz([1,2,3], [4,5,6], [7,8,9], 1, 1, 2)
- Genera una matriz 2x2 como indica el parámetro l vector.tabulate(2,2)
- En la posición (0,0) se colocara el elemento 0+1 y 0+1 lo cual nos dice que en esta posición ubicación tendrá el valor de 5
- En la posición (0,1) se colocara el elemento 0+1 y 1+1 lo cual nos dice que en esta posición ubicación tendrá el valor de 6
- En la posición (1,0) se colocara el elemento 1+1 y 0+1 lo cual nos dice que en esta posición ubicación tendrá el valor de 8
- En la posición (1,1) se colocara el elemento 1+1 y 1+1 lo cual nos dice que en esta posición ubicación tendrá el valor de 9
- Llamado final: [[5,6], [8,9]]

```
def sumMatriz(m1: Matriz, m2: Matriz): Matriz = {
  val dimension = m1.length
  Vector.tabulate(dimension, dimension){(i,j) => m1(i)(j) + m2(i)(j)}
}
```

Esta función realiza la suma de dos matrices suma cada pareja y los ubica en una nueva matriz

Pila de llamadas para [[1,2],[3,4]] y [[1,2],[3,4]]

- Primer llamado: Identificar las dimensiones de la matriz
- Vector.tabulate(2,2) se ubica en la posición (0,0) el resultado de la suma del primer elemento de cada matriz 1+1
- Segundo llamado se ubica en la posición (0,1) el resultado de la suma del segundo elemento de cada matriz 2+2
- Tercer llamado se ubica en la posición (1,0) el resultado de la suma del tercer elemento de cada matriz 3 + 3
- Cuarto llamado se ubica en la posición (1,1) el resultado de la suma del cuarto elemento de cada matriz 4+4. Esto genera una matriz con el resultado de cada suma [[2,4],[6,8]]

```
def multMatrizRec(m1: Matriz, m2: Matriz): Matriz = {
  val dimension = m1.length
  if (dimension == 1) {
    Vector(Vector(prodPunto(m1(0), m2(0))))
  } else {
    val mitad = dimension / 2
    val a11 = subMatriz(m1, 0, 0, mitad)
    val a12 = subMatriz(m1, 0, mitad, mitad)
    val a21 = subMatriz(m1, mitad, 0, mitad)
    val a22 = subMatriz(m1, mitad, mitad, mitad)

    val b11 = subMatriz(m2, 0, 0, mitad)
    val b12 = subMatriz(m2, 0, mitad, mitad)
    val b21 = subMatriz(m2, mitad, 0, mitad)
    val b22 = subMatriz(m2, mitad, mitad, mitad)

    val c11 = sumMatriz(multMatrizRec(a11, b11), multMatrizRec(a12, b21))
    val c12 = sumMatriz(multMatrizRec(a11, b12), multMatrizRec(a12, b22))
    val c21 = sumMatriz(multMatrizRec(a21, b11), multMatrizRec(a22, b21))
    val c22 = sumMatriz(multMatrizRec(a21, b12), multMatrizRec(a22, b22))

    Vector.tabulate(dimension, dimension){(i,j) =>
      if (i < mitad && j < mitad) c11(i)(j)
      else if (i < mitad && j >= mitad) c12(i)(j - mitad)
      else if (i >= mitad && j < mitad) c21(i - mitad)(j)
      else c22(i - mitad)(j - mitad)
    }
  }
}
```

Función recursiva para dividir una matriz en una matriz de menor tamaño

Pila de llamadas para los valores [[1,2],[3,4]] y [[5,6],[7,8]]

- Primer llamado se toma la dimensión de la matriz 2
- Divide la matriz en 2 obtenemos 1
- Se divide la matriz en cuatro submatrices

- a11 submatriz([[1,2],[3,4]], 0, 0, 1] se toma los elementos de la posición 0,0, dado que la dimensión de la submatriz es 1, solo obtenemos el primer valor
- a12 submatriz([[1,2],[3,4]], 0, 1, 1] se toma los elementos de la posición (0,1), dado que la dimensión de la submatriz es 1, solo obtenemos el segundo valor 2
- a13 submatriz([[1,2],[3,4]], 1, 0, 1] se toma los elementos de la posición (1,0), dado que la dimensión de la submatriz es 1, solo obtenemos el tercer valor 3
- a14 submatriz([[1,2],[3,4]], 1, 1, 1] se toma los elementos de la posición (1,1), dado que la dimensión de la submatriz es 1, solo obtenemos el tercer valor 4

- b11 submatriz([[1,2],[3,4]], 0, 0, 1] se toma los elementos de la posición 0,0, dado que la dimensión de la submatriz es 1, solo obtenemos el primer valor 5
- b12 submatriz([[1,2],[3,4]], 0, 1, 1] se toma los elementos de la posición (0,1), dado que la dimensión de la submatriz es 1, solo obtenemos el segundo valor 6
- b13 submatriz([[1,2],[3,4]], 1, 0, 1] se toma los elementos de la posición (1,0), dado que la dimensión de la submatriz es 1, solo obtenemos el tercer valor 7
- b14 submatriz([[1,2],[3,4]], 1, 1, 1] se toma los elementos de la posición (1,1), dado que la dimensión de la submatriz es 1, solo obtenemos el tercer valor 8
- Se realiza el último paso
- C11 = 1*5 + 2*7 = 19
- C12 = 1 * 6 + 2*8 = 22
- C13 = 3 * 5 + 4*7 = 43
- C14 = 3 * 6 + 4*8 = 50

Donde todo se agrupa en una nueva matriz

```
def multMatrizRecPar(m1: Matriz, m2: Matriz): Matriz = {
  val dimension = m1.length
  if (dimension == 1) {
    Vector(Vector(prodPunto(m1(0), m2(0))))
  } else {
    val mitad = dimension / 2
    val a11 = subMatriz(m1, 0, 0, mitad)
    val a12 = subMatriz(m1, 0, mitad, mitad)
    val a21 = subMatriz(m1, mitad, 0, mitad)
    val a22 = subMatriz(m1, mitad, mitad, mitad)

    val b11 = subMatriz(m2, 0, 0, mitad)
    val b12 = subMatriz(m2, 0, mitad, mitad)
    val b21 = subMatriz(m2, mitad, 0, mitad)
    val b22 = subMatriz(m2, mitad, mitad, mitad)

    val taskc11 = task{sumMatriz(multMatrizRecPar(a11, b11), multMatrizRecPar(a12, b21))}
    val taskc12 = task{sumMatriz(multMatrizRecPar(a11, b12), multMatrizRecPar(a12, b22))}
    val taskc21 = task{sumMatriz(multMatrizRecPar(a21, b11), multMatrizRecPar(a22, b21))}
    val taskc22 = task{sumMatriz(multMatrizRecPar(a21, b12), multMatrizRecPar(a22, b22))}

    val c11 = taskc11.join()
    val c12 = taskc12.join()
    val c21 = taskc21.join()
    val c22 = taskc22.join()


    Vector.tabulate(dimension, dimension){(i,j) =>
        if (i < mitad && j < mitad) c11(i)(j)
        else if (i < mitad && j >= mitad) c12(i)(j - mitad)
        else if (i >= mitad && j < mitad) c21(i - mitad)(j)
        else c22(i - mitad)(j - mitad)
    }
  }
}
```

Pila de llamadas para los valores [[1,2],[3,4]] y [[5,6],[7,8]]

- Primer llamado se toma la dimensión de la matriz 2
- Divide la matriz en 2 obtenemos 1
- Se divide la matriz en cuatro submatrices
- a11 submatriz([[1,2],[3,4]], 0, 0, 1] se toma los elementos de la posición 0,0, dado que la dimensión de la submatriz es 1, solo obtenemos el primer valor
- a12 submatriz([[1,2],[3,4]], 0, 1, 1] se toma los elementos de la posición (0,1), dado que la dimensión de la submatriz es 1, solo obtenemos el segundo valor 2
- a13 submatriz([[1,2],[3,4]], 1, 0, 1] se toma los elementos de la posición (1,0), dado que la dimensión de la submatriz es 1, solo obtenemos el tercer valor 3

- a14 submatriz([[1,2],[3,4]], 1, 1, 1] se toma los elementos de la posición (1,1), dado que la dimensión de la submatriz es 1, solo obtenemos el tercer valor 4

- b11 submatriz([[1,2],[3,4]], 0, 0, 1] se toma los elementos de la posición 0,0, dado que la dimensión de la submatriz es 1, solo obtenemos el primer valor 5
- b12 submatriz([[1,2],[3,4]], 0, 1, 1] se toma los elementos de la posición (0,1), dado que la dimensión de la submatriz es 1, solo obtenemos el segundo valor 6
- b13 submatriz([[1,2],[3,4]], 1, 0, 1] se toma los elementos de la posición (1,0), dado que la dimensión de la submatriz es 1, solo obtenemos el tercer valor 7
- b14 submatriz([[1,2],[3,4]], 1, 1, 1] se toma los elementos de la posición (1,1), dado que la dimensión de la submatriz es 1, solo obtenemos el tercer valor 8

- taskc11(C11 = 1*5 + 2*7 = 19)
- taskc12(C12 = 1 * 6 + 2*8 = 22)
- taskc13(C13 = 3 * 5 + 4*7 = 43)
- taskc14(C14 = 3 * 6 + 4*8 = 50
- se une cada una de las Tareas, para generar una nueva matriz con los nuevos valores

```
def restaMatriz(m1: Matriz, m2: Matriz): Matriz = {
  val dimension = m1.length
  Vector.tabulate(dimension, dimension){(i,j) => m1(i)(j) - m2(i)(j)}
}
```

Esta función realiza la suma de dos matrices suma cada pareja y los ubica en una nueva matriz

Pila de llamadas para [[1,2],[3,5]] y [[1,2],[3,4]]

- Primer llamado: Identificar las dimensiones de la matriz
- Vector.tabulate(2,2) se ubica en la posición (0,0) el resultado de la suma del primer elemento de cada matriz 1-1
- Segundo llamado se ubica en la posición (0,1) el resultado de la suma del segundo elemento de cada matriz 2-2

- Tercer llamado se ubica en la posición (1,0) el resultado de la suma del tercer elemento de cada matriz 3 - 3
- Cuarto llamado se ubica en la posición (1,1) el resultado de la suma del cuarto elemento de cada matriz 5-4. Esto genera una matriz con el resultado de cada suma [[0,0],[0,1]]

```
def multStrassen(m1: Matriz, m2: Matriz): Matriz = {
  val dimension = m1.length
  if (dimension == 1) {
    Vector(Vector(prodPunto(m1(0), m2(0))))
  } else {
    val mitad = dimension / 2
    val a11 = subMatriz(m1, 0, 0, mitad)
    val a12 = subMatriz(m1, 0, mitad, mitad)
    val a21 = subMatriz(m1, mitad, 0, mitad)
    val a22 = subMatriz(m1, mitad, mitad, mitad)

    val b11 = subMatriz(m2, 0, 0, mitad)
    val b12 = subMatriz(m2, 0, mitad, mitad)
    val b21 = subMatriz(m2, mitad, 0, mitad)
    val b22 = subMatriz(m2, mitad, mitad, mitad)

    val m1_ = multStrassen(sumMatriz(a11, a22), sumMatriz(b11, b22))
    val m2_ = multStrassen(sumMatriz(a21, a22), b11)
    val m3_ = multStrassen(a11, restaMatriz(b12, b22))
    val m4_ = multStrassen(a22, restaMatriz(b21, b11))
    val m5_ = multStrassen(sumMatriz(a11, a12), b22)
    val m6_ = multStrassen(restaMatriz(a21, a11), sumMatriz(b11, b12))
    val m7_ = multStrassen(restaMatriz(a12, a22), sumMatriz(b21, b22))


    val c11 = sumMatriz(restaMatriz(sumMatriz(m1_, m4_), m5_), m7_)
    val c12 = sumMatriz(m3_, m5_)
    val c21 = sumMatriz(m2_, m4_)
    val c22 = sumMatriz(sumMatriz(restaMatriz(m1_, m2_), m3_), m6_)

    Vector.tabulate(dimension, dimension){(i,j) =>
      if (i < mitad && j < mitad) c11(i)(j)
      else if (i < mitad && j >= mitad) c12(i)(j - mitad)
      else if (i >= mitad && j < mitad) c21(i - mitad)(j)
      else c22(i - mitad)(j - mitad)
    }
  }
}
```

Para m1:

A11 = 1, A12 = 2, A21 = 3, A22 = 4

Para m2:

B11 = 5, B12 = 6, B21 = 7, B22 = 8

Paso 2: Calcular M1 a M7

1. M1 = (A11 + A22) · (B11 + B22) = (1 + 4) · (5 + 8) = 5 · 13 = 65
2. M2 = (A21 + A22) · B11 = (3 + 4) · 5 = 7 · 5 = 35
3. M3 = A11 · (B12 - B22) = 1 · (6 - 8) = 1 · -2 = -2
4. M4 = A22 · (B21 - B11) = 4 · (7 - 5) = 4 · 2 = 8
5. M5 = (A11 + A12) · B22 = (1 + 2) · 8 = 3 · 8 = 24
6. M6 = (A21 - A11) · (B11 + B12) = (3 - 1) · (5 + 6) = 2 · 11 = 22
7. M7 = (A12 - A22) · (B21 + B22) = (2 - 4) · (7 + 8) = -2 · 15 = -30

Paso 3: Calcular bloques Cij

1. C11 = M1 + M4 - M5 + M7 = 65 + 8 - 24 - 30 = 19
2. C12 = M3 + M5 = -2 + 24 = 22
3. C21 = M2 + M4 = 35 + 8 = 43
4. C22 = M1 - M2 + M3 + M6 = 65 - 35 - 2 + 22 = 50

Paso 4: Resultado final

El resultado de multiplicar las matrices es:

C = [[19, 22], [43, 50]]

```scala
def multStrassenPar(m1: Matriz, m2: Matriz): Matriz = {
  val dimension = m1.length
  if (dimension == 1) {
    Vector(Vector(prodPunto(m1(0), m2(0))))
  } else {
    val mitad = dimension / 2
    val a11 = subMatriz(m1, 0, 0, mitad)
    val a12 = subMatriz(m1, 0, mitad, mitad)
    val a21 = subMatriz(m1, mitad, 0, mitad)
    val a22 = subMatriz(m1, mitad, mitad, mitad)

    val b11 = subMatriz(m2, 0, 0, mitad)
    val b12 = subMatriz(m2, 0, mitad, mitad)
    val b21 = subMatriz(m2, mitad, 0, mitad)
    val b22 = subMatriz(m2, mitad, mitad, mitad)

    val m1_ = task{multStrassenPar(sumMatriz(a11, a22), sumMatriz(b11, b22))}
    val m2_ = task{multStrassenPar(sumMatriz(a21, a22), b11)}
    val m3_ = task{multStrassenPar(a11, restaMatriz(b12, b22))}
    val m4_ = task{multStrassenPar(a22, restaMatriz(b21, b11))}
    val m5_ = task{multStrassenPar(sumMatriz(a11, a12), b22)}
    val m6_ = task{multStrassenPar(restaMatriz(a21, a11), sumMatriz(b11, b12))}
    val m7_ = task{multStrassenPar(restaMatriz(a12, a22), sumMatriz(b21, b22))}

    val c11 = sumMatriz(restaMatriz(sumMatriz(m1_.join(), m4_.join()), m5_.join()), m7_.join())
    val c12 = sumMatriz(m3_.join(), m5_.join())
    val c21 = sumMatriz(m2_.join(), m4_.join())
    val c22 = sumMatriz(sumMatriz(restaMatriz(m1_.join(), m2_.join()), m3_.join()), m6_.join())

    Vector.tabulate(dimension, dimension){(i,j) =>
      if (i < mitad && j < mitad) c11(i)(j)
      else if (i < mitad && j >= mitad) c12(i)(j - mitad)
      else if (i >= mitad && j < mitad) c21(i - mitad)(j)
      else c22(i - mitad)(j - mitad)
    }
  }
}
```

Paso 1: Dividir en submatrices

Para m1:
A11 = 1, A12 = 2, A21 = 3, A22 = 4

Para m2:
B11 = 5, B12 = 6, B21 = 7, B22 = 8

Paso 2: Calcular M1 a M7 en paralelo

Cada una de estas operaciones corresponde a una tarea que posteriormente a través del método se unirán los resultados

1. M1 = (A11 + A22) · (B11 + B22) = (1 + 4) · (5 + 8) = 5 · 13 = 65
2. M2 = (A21 + A22) · B11 = (3 + 4) · 5 = 7 · 5 = 35
3. M3 = A11 · (B12 - B22) = 1 · (6 - 8) = 1 · -2 = -2
4. M4 = A22 · (B21 - B11) = 4 · (7 - 5) = 4 · 2 = 8
5. M5 = (A11 + A12) · B22 = (1 + 2) · 8 = 3 · 8 = 24
6. M6 = (A21 - A11) · (B11 + B12) = (3 - 1) · (5 + 6) = 2 · 11 = 22
7. M7 = (A12 - A22) · (B21 + B22) = (2 - 4) · (7 + 8) = -2 · 15 = -30

Paso 3: Calcular bloques Cij

1. C11 = M1 + M4 - M5 + M7 = 65 + 8 - 24 - 30 = 19
2. C12 = M3 + M5 = -2 + 24 = 22
3. C21 = M2 + M4 = 35 + 8 = 43
4. C22 = M1 - M2 + M3 + M6 = 65 - 35 - 2 + 22 = 50

Paso 4: Resultado final

El resultado de multiplicar las matrices es:

C = [[19, 22], [43, 50]]

```
def prodPuntoParD(v1:ParVector[Int],v2: ParVector[Int]):Int ={
  (v1 zip v2).map{case (i,j) => i*j}.sum
}
```

Calcula la multiplicación en paralelo y suma de los valores parciales

- Primer llamado ParVector((1, 4), (2, 5), (3, 6))
- Tarea 1: Para el par(1,4) calcula 1 * 4 = 4
- Tarea 2: Para el par(2,5) calcula 2 * 5 = 10
- Tarea 3: Para el par(3,6) calcula 6 * 3 = 18
- La suma de los valores 4+10
- Suma de los valores 14 +18 = 32

  Función para tomar los tiempos de cada algoritmo en su forma secuencial y paralela

```
def compararAlgoritmos(algoritmoSec: (Matriz, Matriz) => Matriz, algoritmoPar: (Matriz, Matriz) => Matriz)(m1: Matriz, m2: Matriz): (Double, Double, Double) = {
  // Medir el tiempo de la versión secuencial
  val tiempoSec = medirTiempo(algoritmoSec(m1, m2))

  // Medir el tiempo de la versión paralela
  val tiempoPar = medirTiempo(algoritmoPar(m1, m2))

  // Convertir los tiempos a milisegundos
  val tiempoSecMs = tiempoSec / 1e6
  val tiempoParMs = tiempoPar / 1e6

  // Calcular la aceleración
  val aceleracion = tiempoSecMs / tiempoParMs

  (tiempoSecMs, tiempoParMs, aceleracion)
}

// Función para medir el tiempo de ejecución
def medirTiempo[A](f: => A): Long = {
  val start = System.nanoTime()
  f
  val end = System.nanoTime()
  (end - start)
}
```

Informe de paralelización

El objetivo fue paralelizar operaciones de multiplicación de matrices utilizando tareas (task) de Scala. Se abordaron tres métodos de multiplicación de matrices.

Se utilizó la función task para crear tareas independientes para las suboperaciones.

```scala
def benchmarkTiempo(): Unit = {
// Definir el tamaño de las matrices
val size = math.pow(2, 2).toInt
val m1 = matrizAlAzar(size, 2) // Generar matriz aleatoria de tamaño `size x size`
val m2 = matrizAlAzar(size, 2)

println(s"\nDimensión de la matriz: $size x $size")

// Comparar la versión secuencial y paralela para cada algoritmo
val resultadoRec = compararAlgoritmos(multMatriz, multMatrizPar)(m1, m2)
println(f"Secuencial: ${resultadoRec._1}%.2f ms, Paralelo: ${resultadoRec._2}%.2f ms, Aceleración: ${resultadoRec._3}%.2f")

val resultadoStrassen = compararAlgoritmos(multStrassen, multStrassenPar)(m1, m2)
println(f"Strassen Secuencial: ${resultadoStrassen._1}%.2f ms, Paralelo: ${resultadoStrassen._2}%.2f ms, Aceleración: ${resultadoStrassen._3}%.2f")

val resultadoMulrec = compararAlgoritmos(multMatrizRec, multMatrizRecPar)(m1, m2)
println(f"Secuencial Recursivo: ${resultadoMulrec._1}%.2f ms, Paralelo Recursivo: ${resultadoMulrec._2}%.2f ms, Aceleración: ${resultadoMulrec._3}%.2f")
}
}
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS

```
Dimensión de la matriz: 4 x 4
Secuencial: 28,08 ms, Paralelo: 50,09 ms, Aceleración: 0,56
Strassen Secuencial: 32,41 ms, Paralelo: 17,69 ms, Aceleración: 1,83
Secuencial Recursivo: 4,72 ms, Paralelo Recursivo: 10,16 ms, Aceleración: 0,46
```

```
Secuencial: 83,28 ms, Paralelo: 91,66 ms, Aceleración: 0,91
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
Strassen Secuencial: 101,89 ms, Paralelo: 60,99 ms, Aceleración: 1,67
 Secuencial rec: 32,74 ms, Paralelo rec: 60,66 ms, Aceleración: 0,54
```

```
Dimensión de la matriz: 16 x 16
Secuencial: 44,76 ms, Paralelo: 63,98 ms, Aceleración: 0,70
Strassen Secuencial: 88,43 ms, Paralelo: 83,47 ms, Aceleración: 1,06
 Secuencial rec: 42,66 ms, Paralelo rec: 42,39 ms, Aceleración: 1,01
```

```
Dimensión de la matriz: 32 x 32
Secuencial: 48,80 ms, Paralelo: 127,68 ms, Aceleración: 0,38
Strassen Secuencial: 282,93 ms, Paralelo: 110,60 ms, Aceleración: 2,56
 Secuencial rec: 75,06 ms, Paralelo rec: 72,97 ms, Aceleración: 1,03
```

```
Secuencial: 183,04 ms, Paralelo: 758,09 ms, Aceleración: 0,24
Secuencial: 183,04 ms, Paralelo: 758,09 ms, Aceleración: 0,24
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
Strassen Secuencial: 809,10 ms, Paralelo: 343,70 ms, Aceleración: 2,35
 Secuencial rec: 714,64 ms, Paralelo rec: 458,11 ms, Aceleración: 1,56
```

Prueba con multMatriz y multMatrizPar

Tamaño	Secuencial	Paralelo	Aceleración
4x4	28,88 ms	50,09ms	0,56
8x8	83,28 ms	91,66 ms	0.91
16x16	44,76 ms	63,98 ms	0.70
32x32	48,80 ms	127,68 ms	0,38
64x64	183,04 ms	758,09 ms	0.24

- Pequeñas matrices (4x4): El tiempo paralelo (50,09 ms) es más lento que el tiempo secuencial (28,88 ms). Esto ocurre porque la sobrecarga de manejar múltiples hilos es mayor que la ganancia de paralelización.
- Matrices medianas (8x8, 16x16): Los resultados muestran algo de mejora, pero la aceleración no es muy alta. La paralelización tiene algo de beneficio, pero no es suficientemente eficiente para estos tamaños de matrices.
- Matrices grandes (32x32, 64x64): La aceleración es baja, especialmente en matrices más grandes, donde la paralelización no mejora significativamente.

Prueba con multStraseen y multStrassenPar

Tamaño	Secuencial	Paralelo	Aceleración
4x4	32,41 ms	17,69	1,83

8x8	101,89 ms	60,99 ms	1,67
16x16	88,43 ms	83,47 ms	1,06
32x32	282,93 ms	110,60 ms	2,56
64x64	809,10 ms	343,70 ms	2,35

- pequeñas matrices (4x4): La aceleración es bastante buena (1.83), lo que significa que el algoritmo de Strassen tiene un buen potencial para paralelizarse, ya que reduce el número de operaciones.
- Matrices medianas (8x8, 16x16): La aceleración sigue siendo bastante buena (1.67 en 8x8 y 1.06 en 16x16), lo que muestra que el algoritmo sigue beneficiándose de la paralelización.
- Matrices grandes (32x32, 64x64): La aceleración es todavía decente (2.56 y 2.35), aunque no sigue aumentando mucho

Prueba con multMatrizRec y mulMatrizRecPar

Tamaño	Secuencial	Paralelo	Aceleración
4x4	4,72 ms	10,16 ms	0,46
8x8	32,74 ms	60,66 ms	0,54
16x16	42,66 ms	42,39 ms	1,01
32x32	75,06 ms	72,97 ms	1,03
64x64	714,64 ms	458,11 ms	1,56

- Matrices pequeñas (4x4, 8x8): En estas matrices, la paralelización no mejora mucho, ya que el tiempo secuencial es muy corto y la sobrecarga de gestionar múltiples hilos es más significativa.
- Matrices medianas (16x16, 32x32): En tamaños más grandes, la paralelización empieza a tener algo de sentido, y la aceleración se aproxima a 1 (en el caso de la matriz 16x16), lo que significa que se logra casi el mismo tiempo que la versión secuencial, pero con una ligera mejora.
- Matrices grandes (64x64): En este caso, la paralelización da una mejora significativa (1.56 de aceleración), lo que muestra que los algoritmos recursivos pueden beneficiarse más de la paralelización en tamaños grandes.

Informe de corrección

Función mulMatriz

Sea multMatriz(m1,m2) una función que multiplica dos matrices de tamaño nxn, demostremos que para matrices de n x n, la multiplicación funciona de manera adecuada.

Caso base n = 1

Si m y m2 son matrices de tamaño 1x1, entonces la multiplicación consiste en calcular el producto de los elementos de las matrices.

Caso de inducción n = k + 1

Supongamos que para matrices de tamaño, k x k, la multiplicación es correcta. Ahora debemos demostrar que la función también es correcta para matrices de tamaño (k + 1) * (k +1) dividimos ambas matrices en submatrices k * k

Calculamos el producto de las submatrices y sumamos los resultados.

Función mulMatrizRecc

Caso base: Matrices 1 x1, la multiplicación recursiva devuelve directamente el producto de los elementos.

Caso inductivo: Matrices de tamaño k +1 * k+1

Para matrices de tamaño k+1 * k +1, el algoritmo divide las matrices en submatrices k/2 y k/2 y realiza multiplicaciones recursivas para cada una de las submatrices. Posteriormente, las submatrices resultantes se suman adecuadamente para obtener la matriz final

Función mulStrassen

Caso base: Matrices 1 x1 , el producto únicamente es un producto

Caso inductivo: Para matrices de tamaño mayor, el algoritmo divide las matrices en submatrices de tamaño k/2 x k/2 y calcula 7 productos de matrices

Conclusiones

El taller nos permite reconocer las ventajas de la paralelización y se puede evidenciar a través de la implementación de métodos que calculan el tiempo de respuesta de una función en paralelo y secuencial.

Nos familiarizamos con los conceptos de la paralelización, percibimos diferencias entre funciones secuenciales y paralelas, y demás conceptos propios de la programación funcional.

la paralelización puede ser una estrategia efectiva para acelerar la multiplicación de matrices, pero es crucial seleccionar **el algoritmo adecuado y considerar el tamaño de las matrices.**