

Informe Taller 3

Jaime Andrés Noreña 2359523
Dilan Muricio Lemos 2359416
Juan Jose Restrepo 2359517
Diego Fernando Lenis 2359540

September 2024

1 Informe de Procesos

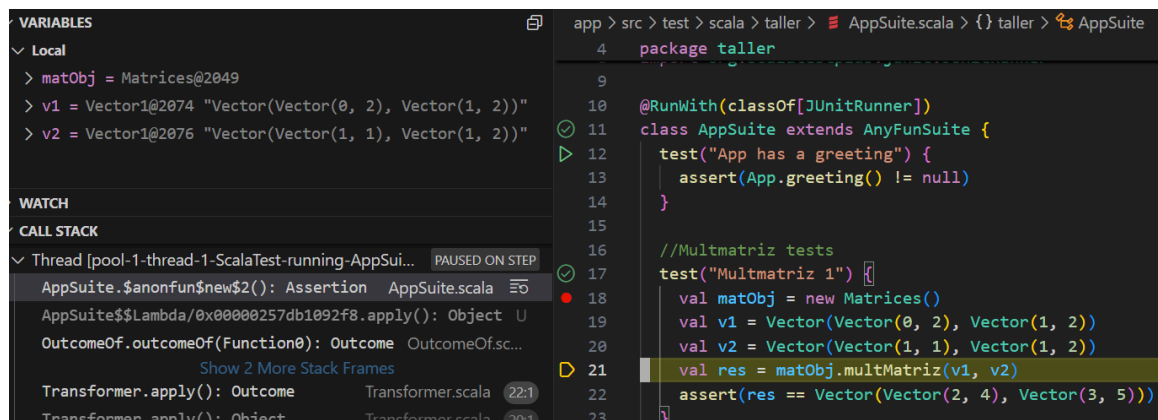
Para cada función implementada en el código entregado se le realizará un ejemplo sencillo para mostrar su comportamiento a lo largo de la ejecución del código y comprobar su correcto funcionamiento.

1.1 Funcion multMatriz

```
def multMatriz(m1: Matriz, m2: Matriz): Matriz = {  
  val length = m1.length  
  val transp_m2 = transpuesta(m2)  
  assert(m1.length == m2.length)  
  Vector.tabulate(length, length) { (i, j) => prodPunto(m1(i), transp_m2(j))}  
}
```

La función `multMatriz`, realiza la multiplicación de dos matrices cuadradas `m1` y `m2`. Primero calcula la transpuesta de `m2` y verifica que ambas matrices tengan dimensiones compatibles. Luego, utiliza `Vector.tabulate` para construir la matriz resultante, donde cada elemento se obtiene como el producto punto entre las filas de `m1` y las columnas de `m2` (usando la transpuesta).

Ejemplo



The screenshot shows an IDE with a Scala project. On the left, the 'VARIABLES' pane shows local variables: `matObj` (Matrices@2049), `v1` (Vector(Vector(0, 2), Vector(1, 2))), and `v2` (Vector(Vector(1, 1), Vector(1, 2))). The 'CALL STACK' pane shows the execution path through `AppSuite.scala` and `Transformer.scala`. On the right, the code editor shows the `AppSuite` class with a test for `multMatriz`. The test creates a `Matrices` object, initializes `v1` and `v2`, and calls `multMatriz(v1, v2)`, asserting that the result is `Vector(Vector(2, 4), Vector(3, 5))`.

Podemos ver como se llama a la funcion con m1 y m2 como parametros

```

1 package taller
2 class Matrices {
3   def transpuesta(m: Matriz): Matriz = {
4     ...
5   }
6
7   def multMatriz(m1: Matriz, m2: Matriz): Matriz = {
8     val length = m1.length
9     val transp_m2 = transpuesta(m2)
10    assert(m1.length == m2.length)
11    Vector.tabulate(length, length) { (i, j) => prodPunto(m1(i), transp_m2(j)) }
12  }
13
14  def multMatrizPar(m1: Matriz, m2: Matriz): Matriz = {
15    ...
16  }
17 }

```

Se asigna a `transp_m2` el valor de la transpuesta de `m2` paso seguido se guarda en una matriz $n \times n$ el resultado del producto punto de las filas de `m1` y `transp_m2` usando los índices i, j

```

4 package taller
5 class AppSuite extends AnyFunSuite {
6   //Multmatriz tests
7   test("Multmatriz 1") {
8     val matObj = new Matrices()
9     val v1 = Vector(Vector(0, 2), Vector(1, 2))
10    val v2 = Vector(Vector(1, 1), Vector(1, 2))
11    val res = matObj.multMatriz(v1, v2)
12    assert(res == Vector(Vector(2, 4), Vector(3, 5)))
13  }
14 }

```

1.2 Funcion multMatrizPar

```

def multMatrizPar(m1: Matriz, m2: Matriz): Matriz = {
  val length = m1.length
  val transp_m2 = transpuesta(m2)
  assert(m1.length == m2.length)
  val joins = Vector.tabulate(length, length) {(i, j) => task(prodPunto(m1(i), transp_m2(j)))}
  Vector.tabulate(length, length) {(i, j) => joins(i)(j).join()}
}

```

La función `multMatrizPar`, escrita en Scala, realiza la multiplicación de dos matrices cuadradas (`m1` y `m2`) de manera paralela para mejorar el rendimiento. Primero, calcula la transpuesta de `m2` y verifica que ambas matrices sean compatibles para la multiplicación. Luego, utiliza `Vector.tabulate` para crear una matriz de tareas, donde cada tarea calcula de manera independiente el producto punto de una fila de `m1` con una columna de `m2` (usando su transpuesta). Finalmente, espera a que todas las tareas terminen con el método `join` para construir la matriz resultante. Esto permite ejecutar los cálculos en paralelo, aprovechando múltiples núcleos del procesador.

```

4 package taller
5 class AppSuite extends AnyFunSuite {
6   // tests MultmatrizPar
7   test("MultmatrizPar 1") {
8     val matObj = new Matrices()
9     val v1 = Vector(Vector(0, 2), Vector(1, 2))
10    val v2 = Vector(Vector(1, 1), Vector(1, 2))
11    val res = matObj.multMatrizPar(v1, v2)
12    assert(res == Vector(Vector(2, 4), Vector(3, 5)))
13  }
14
15  test("MultmatrizPar 2") {
16    ...
17  }
18 }

```

Podemos ver como se llama a la funcion con m1 y m2 como parametros

```

> m1 = Vector1@2089 "Vector(Vector(0, 2), Vector(1, 2))"
> m2 = Vector1@2091 "Vector(Vector(1, 1), Vector(1, 2))"
  length = 2
> transp_m2 = Vector1@2098 "Vector(Vector(1, 1), Vector(1, 2))"
> this = Matrices@2088

WATCH
CALL STACK
Thread [pool-1-thread-1-ScalaTest-running-AppSuite] PAUSED ON STEP
Matrices.multMatrizPar(Vector,Vector): Vector  Matrice...

```

```

6 class Matrices {
42
43
44 def multMatrizPar(m1: Matriz, m2: Matriz): Matriz = {
45   val length = m1.length
46   val transp_m2 = transpuesta(m2)
47   assert(m1.length == m2.length)
48   // val tab: Vector[(Int, Int)] = Vector.tabulate(length*length) {(i: Int) => (i
49   val joins = Vector.tabulate(length, length) {(i, j) => task(prodPunto(m1(i), tr
50   Vector.tabulate(length, length) {(i, j) => joins(i)(j).join()}
51 }

```

Se asigna a `transp_m2` el valor de la transpuesta de `m2` paso seguido se guarda en una matriz $n \times n$ tareas concurrentes equivalentes al resultado del producto punto de las filas de `m1` y `transp_m2` usando los indices $\{i, j \in [0, n)\}$. Al final se construye otra matriz $n \times n$ con los resultados de cada tarea concurrente.

```

Local
> -> multMatrizPar() = Vector1@2133 "Vector(Vector(2, 4), V..."
> matObj = Matrices@2088
> v1 = Vector1@2089 "Vector(Vector(0, 2), Vector(1, 2))"
> v2 = Vector1@2091 "Vector(Vector(1, 1), Vector(1, 2))"

WATCH
CALL STACK
Thread [pool-1-thread-1-ScalaTest-running-AppSuite] PAUSED ON STEP
AppSuite.$anonfun$new$7(): Assertion AppSuite.scala 63:1

```

```

4 package taller
11 class AppSuite extends AnyFunSuite {
58
59 // tests MultmatrizPar
60 test("MultmatrizPar 1") {
61   val matObj = new Matrices()
62   val v1 = Vector(Vector(0, 2), Vector(1, 2))
63   val v2 = Vector(Vector(1, 1), Vector(1, 2))
64   val res = matObj.multMatrizPar(v1, v2)
65   assert(res == Vector(Vector(2, 4), Vector(3, 5)))
66 }
67
68 test("MultmatrizPar 2") {

```

1.3 Funcion subMatriz

```

def subMatriz(m: Matriz, i: Int, j: Int, l: Int) : Matriz = {
  assert(i+l-1 < m.length && j+l-1 < m.length)
  Vector.tabulate(l,l) {(x, y) => m(i+x)(j+y)}
}

```

La función `subMatriz` toma cuatro parámetros: `m`, una matriz representada como un tipo `Matriz`; `i` y `j`, que son índices de tipo `Int`; y `l`, que indica el tamaño de la submatriz a extraer. La función devuelve una nueva `Matriz` que corresponde a una submatriz cuadrada de tamaño $l \times l$ extraída a partir de la posición (i, j) de la matriz original `m`.

El primer paso de la función es una verificación mediante `assert`, que asegura que los límites de la submatriz no excedan los límites de `m`. A continuación, se utiliza `Vector.tabulate`, que genera una matriz nueva de tamaño $l \times l$ aplicando una función a cada posición (x, y) para mapear los elementos correspondientes desde la matriz `m` desplazados por los índices `i` y `j`.

The screenshot shows an IDE with the following components:

- VARIABLES:**
 - Local:
 - `m = Vector1@2111 "Vector(Vector(2, 7, 9, 1), Vector(9, 7, ...))"`
 - `i = 2`
 - `j = 2`
 - `l = 2`
 - `this = Matrices@2112`
 - WATCH:
 - CALL STACK:
 - Thread [pool-1-thread-1-ScalaTest-running-A...]: PAUSED ON BREAKPOINT
 - Matrices.subMatriz(Vector,Int,Int,Int): Vector Matric...
 - AppSuite.\$anonfun\$new\$12(): Assertion AppSuite.scala
- Code Editor:**

```

app > src > main > scala > taller > Matrices.scala > {} taller > Mat
1 package taller
6 class Matrices {
52 def subMatriz(m: Matriz, i: Int, j: Int, l: Int): Matriz = {
53   assert(i+1-1 < m.length && j+1-1 < m.length)
54   Vector.tabulate(1,1) {(x, y) => m(i+x)(j+y)}
55 }
56
57 def sumMatriz(m1: Matriz, m2: Matriz) :Matriz = {
58   assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
59   val length : Int = m1.length
60   Vector.tabulate(length, length) {(i, j) => m1(i)(j) + m2(i)(j)}
61 }
62
63 def isPowerOfTwo(n: Int) : Boolean = {

```

En la función `subMatriz` podemos ver como recibe como argumentos la matriz de la cual saldrá la nueva matriz, los índices desde donde se empezará a cortar dicha matriz i, j y el tamaño de dicha matriz l . Después se verifica que al hacer el corte no se sobrepase el tamaño de la matriz. Por último se retornará una nueva matriz 1×1 empezando en el índice i, j siendo estos sumados a los iteradores x, y de la función `Vector.tabulate`.

1.4 Funcion sumMatriz

```

def sumMatriz(m1: Matriz, m2: Matriz) :Matriz = {
  assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
  val length : Int = m1.length
  Vector.tabulate(length, length) {(i, j) => m1(i)(j) + m2(i)(j)}
}

```

La función `sumMatriz` toma como parámetros dos matrices, `m1` y `m2`, ambas del tipo `Matriz`, y devuelve una nueva `Matriz` que es el resultado de sumar las dos matrices elemento por elemento.

Primero, realiza una verificación mediante `assert` para asegurar que las longitudes de `m1` y `m2` sean potencias de dos (`isPowerOfTwo`) y que ambas matrices tengan la misma longitud. Luego, almacena el tamaño de las matrices en la variable `length` de tipo `Int`. Finalmente, utiliza `Vector.tabulate` para crear una nueva matriz de tamaño `length x length`, donde cada elemento en la posición (i, j) es la suma de los elementos correspondientes de `m1` y `m2`.

The screenshot shows an IDE with the following components:

- VARIABLES:**
 - Local:
 - `m1 = Vector1@2116 "Vector(Vector(3, 3), Vector(4, 2))"`
 - `m2 = Vector1@2117 "Vector(Vector(8, 1), Vector(5, 5))"`
 - `this = Matrices@2118`
 - WATCH:
 - CALL STACK:
 - Thread [pool-1-thread-1-ScalaTest-running-A...]: PAUSED ON BREAKPOINT
 - Matrices.sumMatriz(Vector,Vector): Vector Matrices.scala
- Code Editor:**

```

app > src > main > scala > taller > Matrices.scala > {} taller > Matrices > sumMatriz
1 package taller
6 class Matrices {
56
57
58 def sumMatriz(m1: Matriz, m2: Matriz) :Matriz = {
59   assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
60   val length : Int = m1.length
61   Vector.tabulate(length, length) {(i, j) => m1(i)(j) + m2(i)(j)}
62 }
63
64 def isPowerOfTwo(n: Int) : Boolean = {
  @tailrec

```

Vemos como se verifica que las dos matrices tengan tamaño 2^k y que también sean del mismo tamaño, paso seguido se retorna una matriz del mismo tamaño de las dos primeras siendo cada elemento la suma de los dos elementos correspondientes en las matrices anteriores.

1.5 Funcion restaMatriz

```

def restaMatriz(m1: Matriz , m2: Matriz ) : Matriz ={

```

```

    assert(m1.length == m2.length)
    val length = m1.length
    Vector.tabulate(length, length) {(i, j) => m1(i)(j) - m2(i)(j)}
  }

```

La función `restMatriz` toma como parámetros dos matrices, `m1` y `m2`, ambas del tipo `Matriz`, y devuelve una nueva `Matriz` que es el resultado de restar `m2` de `m1` elemento por elemento.

Primero, realiza una verificación mediante `assert` para asegurarse de que ambas matrices tengan la misma longitud. Luego, almacena la longitud de las matrices en la variable `length`. Finalmente, utiliza `Vector.tabulate` para crear una nueva matriz de tamaño `length` x `length`, donde cada elemento en la posición `(i, j)` es la diferencia entre los elementos correspondientes de `m1` y `m2`.

```

app > src > main > scala > taller > Matrices.scala > {} taller > Matrices > restMatriz
1 package taller
6 class Matrices {
120 def restMatriz(m1: Matriz, m2: Matriz) : Matriz = {
121   assert(m1.length == m2.length)
122   val length = m1.length
123   Vector.tabulate(length, length) {(i, j) => m1(i)(j) - m2(i)(j)}
124 }
125
126 def multStrassen(m1: Matriz, m2: Matriz): Matriz = {
127   assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length))
128   val length = m1.length
129   val subMlength = length/2

```

Vemos como se verifica que las dos matrices tengan el mismo tamaño, paso seguido se retorna una matriz del mismo tamaño de las dos primeras siendo cada elemento la resta de los dos elementos de `m1` con los elementos de `m2`.

1.6 Funcion multMatrizRec

```

def multMatrizRec(m1: Matriz, m2: Matriz) : Matriz = {
  assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
  val length = m1.length
  if (length <= 2) multMatriz(m1, m2)
  else {
    val subM1 = length/2
    val subMats = Vector.tabulate(2, 2)((i, j) =>
      sumMatriz(multMatrizRec(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1),
        multMatrizRec(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
      ))
    val result: Vector[Vector[Int]] = Vector(
      subMats(0)(0)(0) ++ subMats(0)(1)(0),
      subMats(0)(0)(1) ++ subMats(0)(1)(1),
      subMats(1)(0)(0) ++ subMats(1)(1)(0),
      subMats(1)(0)(1) ++ subMats(1)(1)(1))
    result
  }
}

```

La función `multMatrizRec` realiza la multiplicación recursiva de dos matrices cuadradas `m1` y `m2`, ambas del tipo `Matriz`. Emplea un enfoque de división y conquista, dividiendo las matrices en submatrices más pequeñas hasta alcanzar un tamaño manejable, donde se realiza la multiplicación.

directamente. El resultado es una nueva *Matriz* que representa el producto de las matrices de entrada.

```

1 package taller
2 class Matrices {
3
4   def multMatrizRec(m1: Matriz, m2: Matriz) : Matriz = {
5     assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
6     val length = m1.length
7     if (length <= 2) multMatriz(m1, m2)
8     else {
9       val subM1 = length/2
10      val subMats = Vector.tabulate(2, 2)((i, j) =>
11        sumMatriz(multMatrizRec(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1)),
12          multMatrizRec(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
13        ))
14      val result: Vector[Vector[Int]] = Vector(
15        subMats(0)(0) ++ subMats(0)(1)(0),
16        subMats(0)(0)(1) ++ subMats(0)(1)(1),
17        subMats(1)(0) ++ subMats(1)(1)(0),
18        subMats(1)(0)(1) ++ subMats(1)(1)(1)
19      )
20      result
21    }
22  }
23 }

```

Vemos como la funcion verifica que las dos funciones input sean de igual tamaño y que este tambien sea 2^k . Paso seguido se verifica el tamaño de las funciones, si es menor o igual a 2 se realizara la multiplicación con la función *multMatriz*.

```

1 package taller
2 class Matrices {
3
4   def multMatrizRec(m1: Matriz, m2: Matriz) : Matriz = {
5     assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
6     val length = m1.length
7     if (length <= 2) multMatriz(m1, m2)
8     else {
9       val subM1 = length/2
10      val subMats = Vector.tabulate(2, 2)((i, j) =>
11        sumMatriz(multMatrizRec(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1)),
12          multMatrizRec(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
13        ))
14      val result: Vector[Vector[Int]] = Vector(
15        subMats(0)(0) ++ subMats(0)(1)(0),
16        subMats(0)(0)(1) ++ subMats(0)(1)(1),
17        subMats(1)(0) ++ subMats(1)(1)(0),
18        subMats(1)(0)(1) ++ subMats(1)(1)(1)
19      )
20      result
21    }
22  }
23 }

```

Podemos ver como va entrar en el if y retornar el resultado de esta funcion.

```

1 package taller
2 class Matrices {
3
4   def multMatrizRec(m1: Matriz, m2: Matriz) : Matriz = {
5     assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
6     val length = m1.length
7     if (length <= 2) multMatriz(m1, m2)
8     else {
9       val subM1 = length/2
10      val subMats = Vector.tabulate(2, 2)((i, j) =>
11        sumMatriz(multMatrizRec(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1)),
12          multMatrizRec(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
13        ))
14      val result: Vector[Vector[Int]] = Vector(
15        subMats(0)(0) ++ subMats(0)(1)(0),
16        subMats(0)(0)(1) ++ subMats(0)(1)(1),
17        subMats(1)(0) ++ subMats(1)(1)(0),
18        subMats(1)(0)(1) ++ subMats(1)(1)(1)
19      )
20      result
21    }
22  }
23 }

```

Aqui podemos ver otro ejemplo con matrices de tamaño 4 o 2^2 donde no se va a entrar al if sino que procedera a dividir las matrices en 4 submatrices siendo resueltas cada una por separado y al final unidas en una sola matriz

```

app > src > main > scala > taller > Matrices.scala > {} taller > Matrices > multMatrizRec
1 package taller
2 class Matrices {
3   def isPowerOfTwo(n: Int) : Boolean = {
4     def iPOT(n: Int): Boolean = {
5       else if (n%2 != 0) false
6       else iPOT(n/2)
7     }
8     iPOT(n)
9   }
10
11   def multMatrizRec(m1: Matriz, m2: Matriz) : Matriz = {
12     assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
13     val length = m1.length
14     if (length <= 2) multMatriz(m1, m2)
15     else {
16       val subM1 = length/2
17       val subMats = Vector.tabulate(2, 2)((i, j) =>
18         sumMatriz(multMatrizRec(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1)),
19           multMatrizRec(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
20       )
21       val result: Vector[Vector[Int]] = Vector(
22         subMats(0)(0)(0) ++ subMats(0)(1)(0),
23         subMats(0)(0)(1) ++ subMats(0)(1)(1),
24         subMats(1)(0)(0) ++ subMats(1)(1)(0),
25         subMats(1)(0)(1) ++ subMats(1)(1)(1)
26       )
27       result
28     }
29   }
30
31   def multMatrizRecPar(m1: Matriz, m2: Matriz, maxProf: Int, prof: Int) : Matriz = {
32     assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
33     val length = m1.length
34     if (length <= 2) multMatriz(m1, m2)
35     else {
36       val subM1 = length/2
37       if (prof < maxProf) {
38         val subMatTasks = Vector.tabulate(2, 2)((i, j) =>
39           task(sumMatriz(
40             multMatrizRecPar(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1)),
41             multMatrizRecPar(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
42           )))
43       val subMats = Vector.tabulate(2,2) {(i, j) => subMatTasks(i)(j).join()}
44       val result: Vector[Vector[Int]] = Vector(
45         subMats(0)(0)(0) ++ subMats(0)(1)(0),
46         subMats(0)(0)(1) ++ subMats(0)(1)(1),
47         subMats(1)(0)(0) ++ subMats(1)(1)(0),
48         subMats(1)(0)(1) ++ subMats(1)(1)(1)
49       )
50       result
51     }
52   }
53 }

```

Aquí podemos ver un subllamado de las matrices anteriores.

```

app > src > main > scala > taller > Matrices.scala > {} taller > Matrices > multMatrizRec
1 package taller
2 class Matrices {
3   def isPowerOfTwo(n: Int) : Boolean = {
4     def iPOT(n: Int): Boolean = {
5       else if (n%2 != 0) false
6       else iPOT(n/2)
7     }
8     iPOT(n)
9   }
10
11   def multMatrizRec(m1: Matriz, m2: Matriz) : Matriz = {
12     assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
13     val length = m1.length
14     if (length <= 2) multMatriz(m1, m2)
15     else {
16       val subM1 = length/2
17       val subMats = Vector.tabulate(2, 2)((i, j) =>
18         sumMatriz(multMatrizRec(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1)),
19           multMatrizRec(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
20       )
21       val result: Vector[Vector[Int]] = Vector(
22         subMats(0)(0)(0) ++ subMats(0)(1)(0),
23         subMats(0)(0)(1) ++ subMats(0)(1)(1),
24         subMats(1)(0)(0) ++ subMats(1)(1)(0),
25         subMats(1)(0)(1) ++ subMats(1)(1)(1)
26       )
27       result
28     }
29   }
30
31   def multMatrizRecPar(m1: Matriz, m2: Matriz, maxProf: Int, prof: Int) : Matriz = {
32     assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
33     val length = m1.length
34     if (length <= 2) multMatriz(m1, m2)
35     else {
36       val subM1 = length/2
37       if (prof < maxProf) {
38         val subMatTasks = Vector.tabulate(2, 2)((i, j) =>
39           task(sumMatriz(
40             multMatrizRecPar(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1)),
41             multMatrizRecPar(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
42           )))
43       val subMats = Vector.tabulate(2,2) {(i, j) => subMatTasks(i)(j).join()}
44       val result: Vector[Vector[Int]] = Vector(
45         subMats(0)(0)(0) ++ subMats(0)(1)(0),
46         subMats(0)(0)(1) ++ subMats(0)(1)(1),
47         subMats(1)(0)(0) ++ subMats(1)(1)(0),
48         subMats(1)(0)(1) ++ subMats(1)(1)(1)
49       )
50       result
51     }
52   }
53 }

```

Vemos como al final se crea una matriz resultado con la matriz de matrices que nos devolvio la funcion Vector.tabulate

1.7 Funcion multMatrizRecPar

```

def multMatrizRecPar(m1: Matriz, m2: Matriz, maxProf: Int, prof: Int) : Matriz = {
  assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
  val length = m1.length
  if (length <= 2) multMatriz(m1, m2)
  else {
    val subM1 = length/2
    if (prof < maxProf) {
      val subMatTasks = Vector.tabulate(2, 2)((i, j) =>
        task(sumMatriz(
          multMatrizRecPar(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1)),
          multMatrizRecPar(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
        )))
    }
    val subMats = Vector.tabulate(2,2) {(i, j) => subMatTasks(i)(j).join()}
    val result: Vector[Vector[Int]] = Vector(
      subMats(0)(0)(0) ++ subMats(0)(1)(0),
      subMats(0)(0)(1) ++ subMats(0)(1)(1),
      subMats(1)(0)(0) ++ subMats(1)(1)(0),
      subMats(1)(0)(1) ++ subMats(1)(1)(1)
    )
    result
  }
}

```

```

        subMats(0)(0)(0) ++ subMats(0)(1)(0),
        subMats(0)(0)(1) ++ subMats(0)(1)(1),
        subMats(1)(0)(0) ++ subMats(1)(1)(0),
        subMats(1)(0)(1) ++ subMats(1)(1)(1))

    result
  } else{
    multMatrizRec(m1, m2)
  }
}
}
}

```

La función `multMatrizRecPar` realiza la multiplicación recursiva de dos matrices cuadradas `m1` y `m2`, ambas del tipo `Matriz`, utilizando paralelismo para optimizar el rendimiento según la profundidad de la recursión. Recibe cuatro parámetros: `m1` y `m2`, las matrices a multiplicar; `maxProf`, la profundidad máxima para aplicar paralelismo; y `prof`, la profundidad actual. La función comienza verificando que ambas matrices sean cuadradas, tengan el mismo tamaño y que sus dimensiones sean potencias de dos mediante un `assert`. Luego, almacena la longitud de las matrices en la variable `length`. Si `length` es menor o igual a 2, llama a la función `multMatriz` para realizar la multiplicación directa. Si el tamaño es mayor a 2, calcula la mitad del tamaño y lo almacena en la variable `subM1`. Si la profundidad actual `prof` es menor que `maxProf`, crea tareas paralelas usando `task` para multiplicar y sumar submatrices de forma concurrente, almacenando las tareas en `subMatTasks`. Posteriormente, sincroniza y recoge los resultados con `join`, almacenándolos en `subMats`. La matriz final `result` se construye concatenando las submatrices con el operador `++`. Si la profundidad actual alcanza o supera `maxProf`, la función llama a `multMatrizRec` para continuar la multiplicación de manera recursiva sin paralelismo. Finalmente, devuelve `result`, que representa el producto de las matrices `m1` y `m2`, calculado utilizando paralelismo hasta la profundidad permitida.

```

> m1 = Vector1@2229 "Vector(Vector(9, 1, 3, 5), Vector(7, 6, 5), Vector(9, 1, 3, 5))"
> m2 = Vector1@2230 "Vector(Vector(7, 6, 5), Vector(9, 1, 3, 5), Vector(9, 1, 3, 5))"
> maxProf = 1
> prof = 0
> this = Matrices@2231

class Matrices {
  def multMatrizRecPar(m1: Matriz, m2: Matriz, maxProf: Int, prof: Int): Matriz = {
    assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
    val length = m1.length
    if (length <= 2) multMatriz(m1, m2)
    else {
      val subM1 = length/2
      if (prof < maxProf) {
        val subMatTasks = Vector.tabulate(2, 2)((i, j) =>
          task(sumMatriz(
            multMatrizRecPar(subMatriz(m1, 1*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1), maxProf, prof+1),
            multMatrizRecPar(subMatriz(m1, 1*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1), maxProf, prof+1)
          )))
        val subMats = Vector.tabulate(2,2)((i, j) => subMatTasks(i)(j).join())
        val result: Vector[Vector[Int]] = Vector(
          subMats(0)(0)(0) ++ subMats(0)(1)(0),
          subMats(0)(0)(1) ++ subMats(0)(1)(1),
          subMats(1)(0)(0) ++ subMats(1)(1)(0),
          subMats(1)(0)(1) ++ subMats(1)(1)(1))
        result
      } else{
        multMatriz(m1, m2) // Por alguna razon el multMatriz es 40 veces mas rapido que multMatrizRec asi que usamos m
      }
    }
  }
}

```

Esta función es parecida a `multMatrizRec` solo que esta vez debemos definir dos argumentos más para tener en cuenta la profundidad a la que paralelizamos ya que el número de ramas paralelas diferentes que se crean por nivel es de 8^k . La función empieza verificando que las matrices sean de tamaño 2^k y que también sean de igual tamaño.


```

1 package taller
2 class Matrices {
3
4   def multMatrizRecPar(m1: Matriz, m2: Matriz, maxProf: Int, prof: Int) : Matriz = {
5     assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
6     val length = m1.length
7     if (length <= 2) multMatriz(m1, m2)
8     else {
9       val subM1 = length/2
10      if (prof < maxProf) {
11        val subMatTasks = Vector.tabulate(2, 2)((i, j) =>
12          task(sumMatriz(
13            multMatrizRecPar(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1), maxProf, prof+1),
14            multMatrizRecPar(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1), maxProf, prof+1)
15          )))
16        val subMats = Vector.tabulate(2,2) ((i, j) => subMatTasks(i)(j).join())
17        val result: Vector[Vector[Int]] = Vector(
18          subMats(0)(0) ++ subMats(0)(1)(0),
19          subMats(0)(0)(1) ++ subMats(0)(1)(1),
20          subMats(1)(0) ++ subMats(1)(1)(0),
21          subMats(1)(0)(1) ++ subMats(1)(1)(1))
22        result
23      } else {
24        multMatriz(m1, m2) // Por alguna razon el multMatriz es 40 veces mas rapido que multMatrizRec asi que usamos multMatriz
25      } // igualmente multMatrizRecPar es el doble de rapida que multMatrizRec cuando usamos multMatriz
26    } // En realidad para calcular las matrices de tamaño 40x40
27  }
28 }

```

Se verifica el tamaño de las matrices, si es menor o igual que dos se utiliza la función `multMatriz`, si es mayor se utiliza el algoritmo recursivo.

```

1 package taller
2 class Matrices {
3
4   def multMatrizRec(m1: Matriz, m2: Matriz) : Matriz = {
5     result
6   }
7
8   def multMatrizRecPar(m1: Matriz, m2: Matriz, maxProf: Int, prof: Int) : Matriz = {
9     assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
10    val length = m1.length
11    if (length <= 2) multMatriz(m1, m2)
12    else {
13      val subM1 = length/2
14      if (prof < maxProf) {
15        val subMatTasks = Vector.tabulate(2, 2)((i, j) =>
16          task(sumMatriz(
17            multMatrizRecPar(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1), maxProf, prof+1),
18            multMatrizRecPar(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1), maxProf, prof+1)
19          )))
20        val subMats = Vector.tabulate(2,2) ((i, j) => subMatTasks(i)(j).join())
21        val result: Vector[Vector[Int]] = Vector(
22          subMats(0)(0) ++ subMats(0)(1)(0),
23          subMats(0)(0)(1) ++ subMats(0)(1)(1),
24          subMats(1)(0) ++ subMats(1)(1)(0),
25          subMats(1)(0)(1) ++ subMats(1)(1)(1))
26        result
27      } else {
28        multMatriz(m1, m2)
29      }
30    }
31  }
32 }

```

Paso seguido decidimos si deberíamos profundizar mas en la paralelización, sino se solucionara llamando a la función `multMatriz`, esto se decide teniendo en cuenta que la profundidad actual no sea mas grande que la profundidad máxima. la resolución del problema por cada llamado se divide en otros 8 llamados paralelos y recursivos a la misma función. Cuando todos los llamados retornan el resultado es procesado para devolver una matriz de igual tamaño a las matrices ingresadas para multiplicar, siendo esta el resultado de la multiplicación.

1.8 Funcion multStrassen

```

def strassensAlg(m1: Matriz, m2: Matriz):Matriz = {
  assert(m1.length == m2.length)

```

```

val p1 = (m1(0)(0)) * (m2(0)(1) - m2(1)(1))
val p2 = (m1(0)(0) + m1(0)(1)) * (m2(1)(1))
val p3 = (m1(1)(0) + m1(1)(1)) * (m2(0)(0))
val p4 = (m1(1)(1)) * (m2(1)(0) - m2(0)(0))
val p5 = (m1(0)(0) + m1(1)(1)) * (m2(0)(0) + m2(1)(1))
val p6 = (m1(0)(1) - m1(1)(1)) * (m2(1)(0) + m2(1)(1))
val p7 = (m1(0)(0) - m1(1)(0)) * (m2(0)(0) + m2(0)(1))

val c11 = p5 + p4 - p2 + p6
val c12 = p1 + p2
val c21 = p3 + p4
val c22 = p5 + p1 - p3 - p7
val result: Vector[Vector[Int]] = Vector(
  Vector(c11, c12),
  Vector(c21, c22))

result
}

def multStrassen(m1: Matriz, m2: Matriz) : Matriz = {
  assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
  val length = m1.length
  if (length == 1) multMatriz(m1, m2)
  else if (length == 2) strassensAlg(m1, m2)
  else {
    val subM1 = length/2
    val subMats = Vector.tabulate(2, 2)((i, j) =>
      sumMatriz(
        multStrassen(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1)),
        multStrassen(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, subM1))
      )
    )

    val result: Vector[Vector[Int]] = Vector(
      subMats(0)(0)(0) ++ subMats(0)(1)(0),
      subMats(0)(0)(1) ++ subMats(0)(1)(1),
      subMats(1)(0)(0) ++ subMats(1)(1)(0),
      subMats(1)(0)(1) ++ subMats(1)(1)(1)
    )
    result
  }
}
}

```

1. strassensAlg(m1: Matriz, m2: Matriz): Matriz

Esta función implementa el algoritmo de Strassen para multiplicar dos matrices 2×2 .

- **Entradas:** Dos matrices m_1 y m_2 de tamaño 2×2 .

- **Proceso:** Se calculan 7 productos intermedios (p_1, p_2, \dots, p_7) y luego se obtiene la matriz resultante $C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$.

Los productos intermedios se calculan como:

$$\begin{aligned} p_1 &= m_{1,11} \cdot (m_{2,12} - m_{2,22}) \\ p_2 &= (m_{1,11} + m_{1,12}) \cdot m_{2,22} \\ p_3 &= (m_{1,21} + m_{1,22}) \cdot m_{2,11} \\ p_4 &= m_{1,22} \cdot (m_{2,21} - m_{2,11}) \\ p_5 &= (m_{1,11} + m_{1,22}) \cdot (m_{2,11} + m_{2,22}) \\ p_6 &= (m_{1,12} - m_{1,22}) \cdot (m_{2,21} + m_{2,22}) \\ p_7 &= (m_{1,11} - m_{1,21}) \cdot (m_{2,11} + m_{2,12}) \end{aligned}$$

Luego, se calculan los elementos de la matriz resultante:

$$\begin{aligned} c_{11} &= p_5 + p_4 - p_2 + p_6 \\ c_{12} &= p_1 + p_2 \\ c_{21} &= p_3 + p_4 \\ c_{22} &= p_5 + p_1 - p_3 - p_7 \end{aligned}$$

Finalmente, la matriz resultante se construye como:

$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

2. multStrassen(m1: Matriz, m2: Matriz): Matriz

Esta función maneja la multiplicación recursiva de matrices de tamaño general utilizando el algoritmo de Strassen. Si las matrices son de tamaño 1×1 o 2×2 , aplica directamente las funciones base. Si son más grandes, las divide en submatrices y recurre en las submatrices.

- **Entradas:** Dos matrices m_1 y m_2 de tamaño $n \times n$ que deben ser potencias de 2.
- **Proceso:**
 1. Se valida que las matrices tengan tamaños compatibles y que sean potencias de 2.
 2. Si las matrices son 1×1 o 2×2 , se aplica directamente el algoritmo de Strassen.
 3. Para matrices de tamaño mayor, se dividen en submatrices y se aplican recursivamente las multiplicaciones de Strassen a las submatrices. Finalmente, las submatrices se combinan para obtener la matriz resultante.

1.9 Funcion multStrassenPar

```
def multStrassenPar(m1: Matriz, m2: Matriz, maxProf: Int, prof: Int) : Matriz = {
  assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
  val length = m1.length
  if (length == 1) multMatriz(m1, m2)
  else if (length == 2) strassensAlg(m1, m2)
```

```

else {
  if (prof < maxProf) {
    val subM1 = length/2
    val subMatsTasks = Vector.tabulate(2, 2)((i, j) =>
      task(sumMatriz(
        multStrassenPar(subMatriz(m1, i*subM1, 0, subM1), subMatriz(m2, 0, j*subM1, subM1), pr
        multStrassenPar(subMatriz(m1, i*subM1, subM1, subM1), subMatriz(m2, subM1, j*subM1, su
      ))
    )

    val subMats = Vector.tabulate(2, 2)((i, j) => subMatsTasks(i)(j).join())

    val result: Vector[Vector[Int]] = Vector(
      subMats(0)(0)(0) ++ subMats(0)(1)(0),
      subMats(0)(0)(1) ++ subMats(0)(1)(1),
      subMats(1)(0)(0) ++ subMats(1)(1)(0),
      subMats(1)(0)(1) ++ subMats(1)(1)(1))
    result
  } else {
    multStrassen(m1, m2)
  }
}
}

```

La función `multStrassenPar` es una versión paralelizada del algoritmo de Strassen para la multiplicación de matrices. Esta implementación utiliza programación concurrente para dividir el trabajo de multiplicación de submatrices y aprovechar múltiples núcleos o procesadores, mejorando así la eficiencia en sistemas paralelos.

Firma de la Función

La función recibe los siguientes parámetros:

- `m1, m2`: Las matrices de entrada a multiplicar.
- `maxProf`: El nivel máximo de profundidad en la recursión. Si se alcanza este nivel, no se paraleliza más y se utiliza la multiplicación secuencial.
- `prof`: El nivel actual de profundidad en la recursión.

```
def multStrassenPar(m1: Matriz, m2: Matriz, maxProf: Int, prof: Int): Matriz
```

Flujo de la Función

1. Validación de Tamaño de las Matrices

La función comienza validando que las matrices sean cuadradas y que tengan un tamaño que sea una potencia de 2:

```
assert(isPowerOfTwo(m1.length) && isPowerOfTwo(m2.length) && m1.length == m2.length)
```

Esta verificación asegura que se está trabajando con matrices de tamaño $2^k \times 2^k$, lo cual es necesario para la eficiencia del algoritmo de Strassen.

2. Caso Base

Si las matrices tienen tamaño 1×1 o 2×2 , la función realiza la multiplicación directamente, ya sea utilizando la multiplicación convencional o el algoritmo de Strassen de manera secuencial:

```
if (length == 1) multMatriz(m1, m2)
else if (length == 2) strassensAlg(m1, m2)
```

3. Recursión con Paralelización

Si las matrices son más grandes, la función entra en recursión. Si la profundidad actual (**prof**) es menor que la máxima permitida (**maxProf**), se paraleliza el cálculo utilizando tareas concurrentes.

Primero, se divide el tamaño de la matriz a la mitad (**subM1 = length / 2**) y luego se generan tareas para calcular las submatrices:

```
if (prof < maxProf) {
    val subM1 = length / 2

    val subMatsTasks = Vector.tabulate(2, 2)((i, j) => task(sumMatriz(
        multStrassenPar(subMatriz(m1, i * subM1, 0, subM1), subMatriz(m2, 0, j * subM1, subM1), prof + 1,
        multStrassenPar(subMatriz(m1, i * subM1, subM1, subM1), subMatriz(m2, subM1, j * subM1, subM1), p
    )))
}
```

4. Esperar los Resultados de las Tareas

Una vez que las tareas son generadas, se espera a que todas terminen mediante el método **join**, que garantiza que los resultados de las submatrices estén listos antes de combinarlos:

```
val subMats = Vector.tabulate(2, 2)((i, j) => subMatsTasks(i)(j).join())
```

5. Combinación de las Submatrices

Después de obtener los resultados de las submatrices, estas se combinan para formar la matriz resultante:

```
val result: Vector[Vector[Int]] = Vector(
    subMats(0)(0)(0) ++ subMats(0)(1)(0),
    subMats(0)(0)(1) ++ subMats(0)(1)(1),
    subMats(1)(0)(0) ++ subMats(1)(1)(0),
    subMats(1)(0)(1) ++ subMats(1)(1)(1))
result
```

6. Recursión Secuencial cuando se Alcanzó la Profundidad Máxima

Si la profundidad actual ha alcanzado el valor máximo permitido (`prof >= maxProf`), la función realiza la multiplicación de matrices utilizando el algoritmo de Strassen convencional sin paralelización:

```
else { multStrassen(m1, m2) }
```

Resumen del Flujo

El flujo de ejecución de la función es el siguiente:

1. Si las matrices tienen tamaño 1×1 o 2×2 , se multiplican directamente.
2. Si el tamaño es mayor y la profundidad es menor que `maxProf`, se divide la matriz en submatrices y se calculan en paralelo utilizando tareas.
3. Si la profundidad ha alcanzado `maxProf`, se realiza la multiplicación secuencial utilizando el algoritmo de Strassen estándar.
4. Los resultados de las submatrices se combinan para formar la matriz final.

Objetivo de la Paralelización

La paralelización de la multiplicación de submatrices tiene como objetivo aprovechar los recursos de múltiples núcleos o procesadores en sistemas modernos, lo que mejora significativamente el rendimiento para matrices grandes.

Consideraciones

- La paralelización mejora el rendimiento cuando se trabaja con matrices grandes en sistemas multicore.
- El parámetro `maxProf` es utilizado para evitar que la recursión se paralelice demasiado, lo que podría generar una sobrecarga por la creación de tareas.

2 Informe de Parlelizacion

2.1 Comparacion mulMatrizRec y multMatrizRecPar

Al ejecutar la siguiente funcion:

```
def testmultMatrix(): Unit = {  
  val bench = new Benchmark()  
  val ObjMatrices = new Matrices()  
  val results = for {i <- 1 to 9  
    m1 = ObjMatrices.MatrizAlAzar(math.pow(2, i).toInt, 10)  
    m2 = ObjMatrices.MatrizAlAzar(math.pow(2, i).toInt, 10)  
  } yield (bench.compararAlgoritmos(ObjMatrices.multMatrizRec,  
    ObjMatrices.multMatrizRecPar)(m1,m2),  
    math.pow(2,i).toInt)  
  for (r <- results) println(r)  
}
```

Obtuvimos los siguientes resultados:

```
> Task :app:run
((0.1171 ms,0.0534 ms,2.192883895131086),2)
((0.1756 ms,0.2511 ms,0.6993229788928714),4)
((0.8873 ms,0.3991 ms,2.2232523177148584),8)
((1.241 ms,0.7157 ms,1.7339667458432306),16)
((4.8219 ms,4.4114 ms,1.0930543591603572),32)
((42.0875 ms,21.1928 ms,1.9859339020799518),64)
((335.8922 ms,170.488 ms,1.9701808924968327),128)
((2688.3611 ms,1443.1507 ms,1.8628415590970508),256)
((21475.0 ms,11036.238 ms,1.9458623491084555),512)

BUILD SUCCESSFUL in 6m 58s
2 actionable tasks: 2 executed
Watched directory hierarchies: [C:\dev\scala\Taller3PFC]
NTERMINAL 17672:cmd.exe
```

Del cual podemos concluir que el uso de la función `multMatrizRecPar` es en general 2 veces mas rapida que su homologa no paralela `mulMatrizRec` exepto para el tamaño $n = 4$

2.2 Comparacion prodPunto y prodPuntoParD

Al ejecutar la funcion:

```
def testProdPunto(): Unit = {
  val bench = new Benchmark()
  val ObjMatrices = new Matrices()
  val results = for {n <- 1 to 10001 by 1000}
    yield (bench.compararProdPunto(n), n)
  for (r <- results) println(r)
}
```

Obtenemos los siguientes resultados

```
Task Application
Unable to create a system terminal
((0.058799 ms,0.579201 ms,0.10151743522542261),1)
((0.3434 ms,2.132301 ms,0.16104668149571752),1001)
((0.5494 ms,1.530401 ms,0.35899087886116127),2001)
((0.263901 ms,1.7973 ms,0.14683191453847438),3001)
((0.284199 ms,1.659101 ms,0.17129698553614275),4001)
((0.358101 ms,1.829299 ms,0.19575859386573763),5001)
((0.3022 ms,1.8463 ms,0.16367870876888913),6001)
((0.3277 ms,2.154301 ms,0.15211430528974365),7001)
((0.3133 ms,2.149701 ms,0.14574119842713013),8001)
((0.3915 ms,2.451 ms,0.15973072215422277),9001)
((0.6049 ms,2.370399 ms,0.2551891052940876),10001)

BUILD SUCCESSFUL in 6s
2 actionable tasks: 2 executed
C:\dev\scala\Taller3PFC>

TERMINAL 17672:cmd.exe
"app\src\main\scala\taller\App.scala" 34L, 1076B written
```

De lo cual podemos concluir que para este caso la paralización nunca tuvo ningún efecto positivo en el efecto tiempo de ejecución. La version secuencial tuvo mejor rendimiento.

3 Informe de Corrección

3.1 multMatriz

La corrección matemática de la función `multMatriz` se demuestra en una exposición detallada del producto matricial en el álgebra lineal, definiendo formalmente la transformación

$$C = A \times B,$$

donde cada elemento C_{ij} se calcula mediante el producto punto entre la fila i de A y la columna j de B . Esto se expresa matemáticamente como:

$$C_{ij} = \sum_{k=0}^{n-1} (A_{ik} \times B_{kj}),$$

para todo i, j en el rango de índices.

La implementación captura esta definición mediante una estrategia trifásica:

1. Transposición de la matriz para facilitar el cálculo de columnas como vectores.

2. Utilización del producto punto para computar cada elemento.
3. Restricción de dimensionalidad para garantizar compatibilidad algebraica.

El método preserva propiedades algebraicas fundamentales como:

- El cierre en matrices cuadradas.
- La estructura de grupo bajo multiplicación.
- La conservación de la dimensionalidad original.

De este modo, transforma una operación compleja en una construcción declarativa que respeta los principios matemáticos del álgebra lineal con precisión y elegancia.

3.2 multMatrizPar

La función `multMatrizPar` representa una implementación paralela del algoritmo de multiplicación matricial, extendiendo la versión secuencial anterior mediante la introducción de concurrencia en el cálculo de los elementos de la matriz resultante.

La estrategia matemática mantiene la definición formal del producto matricial

$$C_{ij} = \sum_{k=0}^{n-1} (A_{ik} \times B_{kj}),$$

pero introduce la paralelización a través de la función `task` que permite ejecutar el cálculo del producto punto de manera concurrente para cada elemento.

La transformación preserva las propiedades algebraicas fundamentales: cierre en matrices cuadradas, compatibilidad dimensional y conservación de la estructura del producto matricial. El método `joins(i)(j).join()` garantiza la sincronización y recuperación de los resultados calculados concurrentemente, añadiendo una capa de complejidad computacional que mantiene la corrección matemática mientras mejora potencialmente el rendimiento mediante la ejecución paralela de operaciones independientes.

3.3 subMatriz

La función `subMatriz` implementa correctamente la extracción de una submatriz desde una matriz m , comenzando en la posición (i, j) . La función primero valida que las coordenadas de inicio más la dimensión de la submatriz no excedan los límites de la matriz original mediante una aserción.

Matemáticamente, la función genera una nueva matriz donde cada elemento (x, y) se mapea desde la matriz original utilizando desplazamientos $(i + x, j + y)$, lo que garantiza una extracción precisa y preserva la estructura de la submatriz original.

La implementación utiliza `Vector.tabulate()` para construir eficientemente la submatriz sin modificar la matriz original. La función es correcta y eficiente, proporcionando una forma concisa de obtener una submatriz cuadrada de dimensión l , comenzando en cualquier posición válida dentro de la matriz de entrada.

3.4 sumMatriz

La corrección matemática de la función `sumMatriz` se fundamenta en una demostración rigurosa del álgebra lineal, donde se define formalmente el conjunto $\mathbb{R}^{n \times n}$ de matrices cuadradas con entradas reales, restringido a dimensiones que son potencias de 2. La operación de suma matricial

$$: \mathbb{R}^{n \times n} + \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^{n \times n}$$

se caracteriza por la construcción elemento a elemento, donde

$$C_{ij} = A_{ij} + B_{ij},$$

para todo i, j en el rango de índices.

La implementación captura meticulosamente las propiedades algebraicas fundamentales:

- Dominio restringido mediante `isPowerOfTwo()`.
- Preservación de la estructura cuadrada.
- Garantía de dimensiones idénticas.
- Cierre bajo la operación.
- Cumplimiento de propiedades como:
 - Conmutatividad: $A + B = B + A$.
 - Asociatividad: $(A + B) + C = A + (B + C)$.

El método `Vector.tabulate` materializa matemáticamente esta definición, asegurando una transformación precisa que respeta las estructuras algebraicas originales de las matrices de entrada. Por lo tanto, el algoritmo es correcto bajo las precondiciones especificadas.

3.5 restMatriz

Sea $\forall M_1, M_2 \in \mathbb{Z}^{n \times n}$ dos matrices cuadradas de igual dimensión, la función `restMatriz` implementa una operación de sustracción matricial definida como

$$M_1 - M_2 = \{(m_{1ij} - m_{2ij}) \mid \forall i, j \in \{0, \dots, n-1\}\}.$$

La implementación garantiza corrección matemática mediante validación de dimensiones iguales a través de `assert(m1.length == m2.length)`, y genera una nueva matriz aplicando sustracción elemento a elemento utilizando `Vector.tabulate()`, preservando la estructura algebraica de la resta matricial.

Formalmente, la función satisface las propiedades de sustracción matricial: dimensionalidad invariante, elemento-a-elemento, y generación de matriz resultante sin modificar las matrices originales.

3.6 multMatrizRec

La función `multMatrizRec` implementa la multiplicación recursiva de matrices, utilizando una estrategia de dividir y conquistar. A continuación, se presenta una demostración matemática de su funcionamiento.

Definición de la multiplicación de matrices

Dada dos matrices A y B de dimensión $n \times n$, el producto matricial $C = A \times B$ se define como:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \times B_{kj}$$

Este producto se calcula para todos los elementos C_{ij} , lo que requiere $O(n^3)$ operaciones en el caso tradicional.

Dividir y conquistar

La idea principal de la multiplicación recursiva de matrices es dividir las matrices A y B en submatrices más pequeñas, calcular sus productos y luego combinar los resultados.

Supongamos que las matrices A y B son de tamaño $2^n \times 2^n$ (la función usa un `assert` para asegurar que las matrices son de tamaño $2^k \times 2^k$, donde k es un número entero).

Dividimos las matrices A y B en submatrices de tamaño $2^{n-1} \times 2^{n-1}$ de la siguiente forma:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

El producto $C = A \times B$ se puede calcular como:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

donde:

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

Implementación recursiva

El algoritmo recursivo `multMatrizRec` sigue la misma estructura matemática. La función divide las matrices en submatrices más pequeñas hasta que llega a matrices de tamaño 2×2 , que son multiplicadas directamente usando una función base `multMatriz`.

Paso 1: División de matrices

La matriz A se divide en cuatro submatrices de tamaño $n/2 \times n/2$:

$$A_{11}, A_{12}, A_{21}, A_{22}$$

De manera similar, la matriz B también se divide en cuatro submatrices $B_{11}, B_{12}, B_{21}, B_{22}$.

Paso 2: Recursión y multiplicación

La función recursiva se llama para calcular las submatrices de C . Por ejemplo:

- Para C_{11} , se calcula:

$$C_{11} = \text{multMatrizRec}(A_{11}, B_{11}) + \text{multMatrizRec}(A_{12}, B_{21})$$

- Para C_{12} , se calcula:

$$C_{12} = \text{multMatrizRec}(A_{11}, B_{12}) + \text{multMatrizRec}(A_{12}, B_{22})$$

Y así sucesivamente para C_{21} y C_{22} .

Paso 3: Combinación de resultados

Finalmente, las submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ se combinan para formar la matriz resultado C .

3.7 multMatrizRecPar

La función `multMatrizRecPar` implementa una versión paralelizada de la multiplicación recursiva de matrices. A continuación, se presenta una demostración matemática de su funcionamiento, con énfasis en cómo se paraleliza el proceso de multiplicación utilizando tareas concurrentes.

Dividir y conquistar con paralelización

Al igual que en la multiplicación recursiva tradicional, en este caso, las matrices A y B se dividen en submatrices de tamaño $n/2 \times n/2$. Sin embargo, la diferencia clave es la paralelización del cálculo de los productos de las submatrices, que se logra mediante tareas concurrentes.

El proceso se puede dividir en los siguientes pasos:

Paso 1: División de matrices

Las matrices A y B se dividen en submatrices de tamaño $n/2 \times n/2$ de la siguiente forma:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

El producto $C = A \times B$ se calcula mediante los siguientes componentes:

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

Paso 2: Paralelización de las submatrices

El algoritmo `multMatrizRecPar` emplea un parámetro de control de paralelización `prof` (profundidad) y `maxProf` (profundidad máxima). Mientras la profundidad actual (`prof`) sea menor que `maxProf`, el algoritmo crea tareas paralelas para calcular los productos de las submatrices. Esto se logra utilizando la función `task` para crear tareas concurrentes que ejecutan las multiplicaciones de las submatrices.

Por ejemplo, para C_{11} , se paraleliza el cálculo de la suma de los productos:

$$\begin{aligned}C_{11} &= \text{task}(\text{multMatrizRecPar}(A_{11}, B_{11}) + \text{multMatrizRecPar}(A_{12}, B_{21})) \\C_{12} &= \text{task}(\text{multMatrizRecPar}(A_{11}, B_{12}) + \text{multMatrizRecPar}(A_{12}, B_{22}))\end{aligned}$$

Esta paralelización ocurre de manera recursiva para todas las submatrices, hasta que la profundidad de la recursión alcanza `maxProf`. Si `prof` alcanza el valor de `maxProf`, la multiplicación recursiva se realiza de manera secuencial mediante la función `multMatrizRec`.

Paso 3: Sincronización de las tareas

Una vez que las tareas paralelizadas han sido lanzadas, se utiliza el método `join()` para sincronizar los resultados de cada tarea. Esto asegura que el cálculo de cada submatriz se haya completado antes de combinar los resultados.

Por ejemplo, la matriz resultante se obtiene combinando las submatrices calculadas de la siguiente manera:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

donde las submatrices C_{ij} son obtenidas de la siguiente forma:

$$\begin{aligned}C_{11} &= \text{subMatTasks}(0, 0).\text{join}() + \text{subMatTasks}(0, 1).\text{join}() \\C_{12} &= \text{subMatTasks}(0, 0).\text{join}() + \text{subMatTasks}(0, 1).\text{join}()\end{aligned}$$

y de manera similar para C_{21} y C_{22} .

Paso 4: Recursión y salida

Si el tamaño de la submatriz es lo suficientemente pequeño (típicamente cuando es 2×2), la multiplicación se realiza de forma secuencial mediante la llamada a la función `multMatriz`.

4 Conclusiones

En el desarrollo de este trabajo, hemos profundizado significativamente en los principios de la programación funcional en Scala, explorando técnicas avanzadas de manipulación de vectores y matrices. Las implementaciones realizadas, como `multMatriz`, `multMatrizPar`, y `multMatrizRecPar`, nos permitieron comprender cómo los conceptos funcionales como inmutabilidad, recursividad y funciones de orden superior pueden aplicarse eficientemente en operaciones matemáticas complejas. Especialmente notable fue el uso de `Vector.tabulate()` para crear matrices de manera declarativa, evitando mutaciones y generando código más conciso y predecible. La introducción del paralelismo en funciones como `multMatrizPar` y `multMatrizRecPar` nos reveló las capacidades de Scala para aprovechar

la concurrencia y mejorar el rendimiento computacional. Mediante el uso de la función `task` y `join()`, logramos dividir operaciones matriciales en tareas independientes ejecutables simultáneamente, reduciendo potencialmente el tiempo de cómputo en operaciones de gran escala. Este enfoque no solo optimiza el rendimiento, sino que también nos enseña a pensar en algoritmos desde una perspectiva de computación distribuida, fundamental en el desarrollo de software moderno de alto rendimiento.