

TALLER 3 - PARALELISMO DE TAREAS Y DATOS

Juan Damián Cuervo Buitrago – 2349413

Vanessa Alexandra Durán Mona – 2359394

Juan Sebastián Rodas Ramírez – 2359681

Ingeniería de Sistemas, Universidad del
Valle

750013C: Fundamentos de
Programación Funcional y Concurrente

Carlos Andrés Delgado Saavedra

29 de Noviembre de 2024

INFORME DE PROCESOS

- **función prodPunto:**

Esta función calcula el producto punto entre 2 vectores, donde selecciona el elemento de cada vector en mismo índices para multiplicarlos, y después, sumarlos todos.

- **función subMatriz:**

Esta función recibe una matriz, una posición fila y una posición columna que será (i,j), que será desde donde la submatriz comenzará, hasta la longitud l dada. (Su funcionamiento ya se encuentra dentro de la función multStrassen)

- **función sumMatriz:**

Esta función recibe 2 matrices. Con “Vector.tabulate” genero un vector, con dimensiones l x l, que será la misma dimensión de las matrices dadas, esta nos permite trabajar con los índices del mismo vector, y empezar a sumar los valores de las matrices dadas en los mismos índices i,j. (Su funcionamiento ya se encuentra dentro de la función multStrassen)

- **función restaMatriz:**

Esta función recibe 2 matrices, con “Vector.tabulate” generó un vector, con dimensiones l x l, que será la misma dimensión de las matrices dadas, esta nos permite trabajar con los índices del mismo vector, y empezar a restar los valores de las matrices dadas en los mismos índices i,j. (Su funcionamiento ya se encuentra dentro de la función multStrassen)

- **Función multMatrizEstandarSec:**

Esta función realiza la multiplicación de dos matrices m1 y m2 utilizando una técnica optimizada para el acceso a las columnas de m2. Se trasmuta la matriz m2 con el fin de optimizar el acceso a sus columnas. La multiplicación se realiza calculando el producto punto entre cada fila de m1 y cada columna de m2. El resultado es una nueva matriz donde cada elemento es el producto punto de la fila correspondiente de m1 y la columna correspondiente de m2.

1. **Paso :** El test verifica si el resultado de la multiplicación es correcto.

```
val matriz1 = Vector(  
  Vector(1, 2, 3, 4),  
  Vector(5, 6, 7, 8),  
  Vector(9, 10, 11, 12),  
  Vector(13, 14, 15, 16))  
  
val matriz2 = Vector(  
  Vector(16, 15, 14, 13),  
  Vector(12, 11, 10, 9),  
  Vector(8, 7, 6, 5),  
  Vector(4, 3, 2, 1))  
  
400  //Tests para la funcion multMatrizEstandar  
401  test("La multiplicación de matriz1 y matriz2") {  
402    assert(objMatriz.multMatrizEstandarSec(matriz1, matriz2) == Vector(  
403      Vector(80, 70, 60, 50),  
404      Vector(240, 214, 188, 162),  
405      Vector(400, 358, 316, 274),  
406      Vector(560, 502, 444, 386)))  
407  }
```

2. Paso: Se saca la traspuesta de la segunda matriz

```
> m1 = Vector1@1818 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Ve...
< m2 = Vector1@1822 "Vector(Vector(16, 15, 14, 13), Vector(12, 11, 10, 9), Vector(8, 7, 6, 5)...
< prefix1 = Object[4]@2390
  > 0 = Vector1@2391 "Vector(16, 15, 14, 13)"
  > 1 = Vector1@2392 "Vector(12, 11, 10, 9)"
  > 2 = Vector1@2393 "Vector(8, 7, 6, 5)"
  > 3 = Vector1@2394 "Vector(4, 3, 2, 1)"
< m2T = Vector1@2358 "Vector(Vector(16, 12, 8, 4), Vector(15, 11, 7, 3), Vector(14, 10, 6, 2)...
< prefix1 = Object[4]@2362
  > 0 = Vector1@2363 "Vector(16, 12, 8, 4)"
  > 1 = Vector1@2364 "Vector(15, 11, 7, 3)"
  > 2 = Vector1@2365 "Vector(14, 10, 6, 2)"
  > 3 = Vector1@2366 "Vector(13, 9, 5, 1)"
> this = MatrizOps@1830
```

3. Paso: Se utiliza el método map para iterar sobre cada fila de la matriz m1.

Para cada fila row de m1, se aplica otro map sobre las filas de la matriz transpuesta m2T. Para cada fila row de m1 y cada columna col de m2T (que en realidad es una fila de m2), se calcula el producto punto entre row y col. El producto punto es la suma de los productos de los elementos correspondientes de row y col. El resultado de estos cálculos es una nueva matriz donde cada elemento C_{ij} de la matriz resultante C es el producto punto de la fila i de m1 y la columna j de m2. Este proceso se repite para todos los elementos. En este caso, la siguiente es la primera iteración que obtiene el elemento C_{11} :

```
238  def multMatrizEstandarSec(m1: Matriz, m2: Matriz): Matriz = {
239    val m2T = transpuesta(m2) // Transponemos m2 para optimizar
240    m1.map(row => m2T.map(col => prodPunto(row, col))) //Se ejecuta
241  }

> $this = MatrizOps@2301
> row$1 = Vector1@2306 "Vector(1, 2, 3, 4)"
> col = Vector1@2312 "Vector(16, 12, 8, 4)"

< Local
  ->$anonfun$multMatrizEstandarSec$2() = 80
  i = 80
```

4. Paso: Segunda iteración que obtiene el elemento C_{12} (se actualiza la pila):

```

    < Local
    | > $this = MatrizOps@2301
    | > row$1 = Vector1@2306 "Vector(1, 2, 3, 4)"
    | > col = Vector1@2330 "Vector(15, 11, 7, 3)"

```

```

    < Local
    | ->$anonfun$multMatrizEstandarSec$2() = 70
    | i = 70

```

5. **Paso:** Tercera iteración que obtiene el elemento C_{13} (se actualiza la pila):

```

    < Local
    | > $this = MatrizOps@2301
    | > row$1 = Vector1@2306 "Vector(1, 2, 3, 4)"
    | > col = Vector1@2338 "Vector(14, 10, 6, 2)"

```

```

    < Local
    | ->$anonfun$multMatrizEstandarSec$2() = 60
    | i = 60

```

6. **Paso:** iteración que obtiene el elemento C_{14} :

```

    < Local
    | > $this = MatrizOps@2301
    | > row$1 = Vector1@2306 "Vector(1, 2, 3, 4)"
    | > col = Vector1@2346 "Vector(13, 9, 5, 1)"

```

```

    < Local
    | ->$anonfun$multMatrizEstandarSec$2() = 50
    | i = 50

```

7. **Paso:** iteración que obtiene el elemento C_{21} (se pasa a la segunda fila de m1):

```

    < Local
    | > $this = MatrizOps@2301
    | > m2T$1 = Vector1@2300 "Vector(Vector(16, 12, 8, 4), Vector(15, 11, 7, 3), Vector(14, 10, 6, 2), Vect...
    | > row = Vector1@2352 "Vector(5, 6, 7, 8)"

```

```

    < Local
    | > $this = MatrizOps@2301
    | > row$1 = Vector1@2352 "Vector(5, 6, 7, 8)"
    | > col = Vector1@2312 "Vector(16, 12, 8, 4)"

```

```

    < Local
    | ->$anonfun$multMatrizEstandarSec$2() = 240
    | i = 240

```

8. **Paso:** iteración que obtiene el elemento C_{22} :

```

    < Local
    > $this = MatrizOps@2301
    > row$1 = Vector1@2352 "Vector(5, 6, 7, 8)"
    > col = Vector1@2330 "Vector(15, 11, 7, 3)"

```

```

    < Local
    ->$anonfun$multMatrizEstandarSec$2() = 214
    i = 214

```

9. Paso: iteración que obtiene el elemento C_{23} :

```

    < Local
    > $this = MatrizOps@2301
    > row$1 = Vector1@2352 "Vector(5, 6, 7, 8)"
    > col = Vector1@2338 "Vector(14, 10, 6, 2)"

```

```

    < Local
    ->$anonfun$multMatrizEstandarSec$2() = 188
    i = 188

```

10. Paso: iteración que obtiene el elemento C_{24} :

```

    < Local
    > $this = MatrizOps@2301
    > row$1 = Vector1@2352 "Vector(5, 6, 7, 8)"
    > col = Vector1@2346 "Vector(13, 9, 5, 1)"

```

```

    < Local
    ->$anonfun$multMatrizEstandarSec$2() = 162
    i = 162

```

se pasa a la tercera fila de m1.

```

    < Local
    > $this = MatrizOps@2301
    > m2T$1 = Vector1@2300 "Vector(Vector(16, 12, 8, 4), Vector(15, 11, 7, 3), Vector(14, 10, 6, 2), Vect.
    > row = Vector1@2380 "Vector(9, 10, 11, 12)"

```

11. Paso: iteración que obtiene el elemento C_{31} :

```

    < Local
    > $this = MatrizOps@2301
    > row$1 = Vector1@2380 "Vector(9, 10, 11, 12)"
    > col = Vector1@2312 "Vector(16, 12, 8, 4)"

```

```

    < Local
    ->$anonfun$multMatrizEstandarSec$2() = 400
    i = 400

```

12. Paso: iteración que obtiene el elemento C_{32} :

```
✓ Local
> $this = MatrizOps@2300
> row$1 = Vector1@2363 "Vector(9, 10, 11, 12)"
> col = Vector1@2329 "Vector(15, 11, 7, 3)"

✓ Local
->$anonfun$multMatrizEstandarSec$2() = 358
i = 358
```

13. Paso: iteración que obtiene el elemento C_{33} :

```
✓ Local
> $this = MatrizOps@2300
> row$1 = Vector1@2363 "Vector(9, 10, 11, 12)"
> col = Vector1@2337 "Vector(14, 10, 6, 2)"

✓ Local
->$anonfun$multMatrizEstandarSec$2() = 316
i = 316
```

14. Paso: iteración que obtiene el elemento C_{34} :

```
✓ Local
> $this = MatrizOps@2300
> row$1 = Vector1@2363 "Vector(9, 10, 11, 12)"
> col = Vector1@2340 "Vector(13, 9, 5, 1)"

✓ Local
->$anonfun$multMatrizEstandarSec$2() = 274
i = 274
```

se pasa a la cuarta fila de m1

```
✓ Local
> $this = MatrizOps@2300
> m2T$1 = Vector1@2303 "Vector(Vector(16, 12, 8, 4), Vector(15, 11, 7, 3), Vector(14, 10, 6, 2), Vect...
> row = Vector1@2392 "Vector(13, 14, 15, 16)"
```

15. Paso: iteración que obtiene el elemento C_{41}

```
✓ Local
> $this = MatrizOps@2300
> row$1 = Vector1@2392 "Vector(13, 14, 15, 16)"
> col = Vector1@2312 "Vector(16, 12, 8, 4)"

✓ Local
->$anonfun$multMatrizEstandarSec$2() = 560
i = 560
```

16. **Paso:** iteración que obtiene el elemento C_{42}

```
✓ Local
> $this = MatrizOps@2300
> row$1 = Vector1@2392 "Vector(13, 14, 15, 16)"
> col = Vector1@2329 "Vector(15, 11, 7, 3)"

✓ Local
->$anonfun$multMatrizEstandarSec$2() = 502
i = 502
HTM VALUE: 0.0000000000000002
```

17. **Paso:** iteración que obtiene el elemento C_{43}

```
✓ Local
> $this = MatrizOps@2300
> row$1 = Vector1@2392 "Vector(13, 14, 15, 16)"
> col = Vector1@2337 "Vector(14, 10, 6, 2)"

✓ Local
->$anonfun$multMatrizEstandarSec$2() = 444
i = 444
```

18. **Paso:** iteración que obtiene el elemento C_{44}

```
✓ Local
> $this = MatrizOps@2300
> row$1 = Vector1@2392 "Vector(13, 14, 15, 16)"
> col = Vector1@2340 "Vector(13, 9, 5, 1)"

✓ Local
->$anonfun$multMatrizEstandarSec$2() = 386
i = 386
```

19. **Paso:** Se pasa a comparar si el resultado obtenido por la función es el resultado esperado:

```
✓ Local
> $this = MatrizTest@1815 "MatrizTest"
> $org_scalatest_assert_macro_left = Vector1@2418 "Vector(Vector(80, 70, 60, 50), Vector(240, 214, 188, 162), Vector(400, 358, 316, 274), ...)"
> $org_scalatest_assert_macro_right = Vector1@2419 "Vector(Vector(80, 70, 60, 50), Vector(240, 214, 188, 162), Vector(400, 358, 316, 274), ..."

✓ Local
->$anonfun$new$51() = Succeeded@2423 "Succeeded"
> f = MatrizTest$$Lambda$0x0000014caa11dc70@2424 "taller.MatrizTest$$Lambda$0x0000014caa11dc70@15e563df"
> this = OutcomeOf$@2425
```

- **Función mulMatrizRec:**

Esta función recursiva recibe 2 matrices y retorna otra matriz del mismo tipo, en este caso se usarán matrices cuadradas y su tamaño potencia de 2.

En esta función se usa la multiplicación de matrices para multiplicar dos matrices.

1. Paso:

The screenshot shows a Scala code editor with a test file named `MatrizTest.scala`. The code defines a class `MatrizTest` that extends `AnyFunSuite`. It contains a single test method `test` that checks if multiplying two matrices results in the expected vector. The test fails at line 158.

```
app > src > test > scala > taller > MatrizTest.scala > {} taller > MatrizTest
  1 package taller
  9 class MatrizTest extends AnyFunSuite {
D 158   test("Multiplicar la matriz11 por la matriz12 dará como resultado Vector(Vector(8,5), Vector(20,1))
  ● 159     assert(objMatriz.multMatrizRec(matriz11, matriz12) == Vector(Vector(8,5), Vector(20,13)))
  160 }
```

Se toman 2 matrices de dimensión 2x2, y se compara si el valor es el esperado

2. Paso

The screenshot shows a Scala code editor with a file named `Matrizscala.M`. It contains a class `MatrizOps` with two methods: `restaMatriz` and `multMatrizRec`. The `multMatrizRec` method is highlighted. A breakpoint is set at line 71, which corresponds to the base case where the length of the matrices is 1. The code uses `Vector.tabulate` to create a vector of the same dimensions and then subtracts the values of the matrices `m1` and `m2`.

```
RUN AND DEBUG No Configurations ... app > src > main > scala > taller > Matrizscala > {} taller > MatrizOps > Matrizscala.M
  1 package taller
  2 class MatrizOps(){
  3   def restaMatriz(m1:Matriz, m2:Matriz):Matriz ={
  4     //Resta dos matrices
  5     Vector.tabulate(m1.length,m1.length)((i,j) => m1(i,j) - m2(i,j))
  6     //se crea un vector de mismas dimensiones, donde
  7     //y se restan los valores de las matrices m1 y m2
  8   }
  9
 10  def multMatrizRec(m1: Matriz, m2: Matriz): Matriz = [
 11    val n = m1.length
 12    if (n == 1) { You, last week • Agregue las
 13      // Caso base: matrices de tamaño 1x1
 14      Vector(Vector(m1(0)(0) * m2(0)(0)))
 15    } else [
 16      val mid = n / 2
 17      // Dividimos las matrices en 4 submatrices
D 18      val (a11, a12, a21, a22) = ( You, last
 19        subMatriz(m1, 0, 0, mid),
 20        subMatriz(m1, 0, mid, mid),
 21        subMatriz(m1, mid, 0, mid),
 22        subMatriz(m1, mid, mid, mid)
 23      )
 24    ]
 25  }
```

Se evalúa si se cumple el caso base, en el que la longitud de las matrices debe ser igual a 1.

3. Paso

The screenshot shows the same `Matrizscala.M` file. The code has been modified to handle the case where the matrices are not 1x1. It uses a recursive approach to divide the matrices into four submatrices of size $\frac{n}{2} \times \frac{n}{2}$ and then multiplies them using the `multMatrizRec` method. A breakpoint is set at line 78, which corresponds to the division of the matrices into four submatrices.

```
app > src > main > scala > taller > Matrizscala > {} taller > MatrizOps > Matrizscala.M
  1 package taller
  2 class MatrizOps(){
  3   def multMatrizRec(m1: Matriz, m2: Matriz): Matriz = [
  4     val n = m1.length
  5     if (n == 1) {
  6       // Caso base: matrices de tamaño 1x1
  7       Vector(Vector(m1(0)(0) * m2(0)(0)))
  8     } else [
  9       val mid = n / 2
 10       // Dividimos las matrices en 4 submatrices
D 11       val (a11, a12, a21, a22) = ( You, last
 12         subMatriz(m1, 0, 0, mid),
 13         subMatriz(m1, 0, mid, mid),
 14         subMatriz(m1, mid, 0, mid),
 15         subMatriz(m1, mid, mid, mid)
 16       )
 17     ]
 18   }
 19 }
```

Entra en la condición de `else` al no cumplir que la longitud de las matrices sea 1. Entra en esta condición, donde divide la matriz original en 4. Siendo m la mitad de la dimensión de las matrices, para así hallar las submatrices.

4. Paso

```
✓ VARIABLES
  ✓ Local
    > m1 = Vector1@2081 "Vector(Vector(1, 2), Vector(3, 4))"
    > m2 = Vector1@2082 "Vector(Vector(4, 3), Vector(2, 1))"
    n = 2
    mid = 1
    > a11 = Vector1@2099 "Vector(Vector(1))"
    > a12 = Vector1@2100 "Vector(Vector(2))"
    > a21 = Vector1@2101 "Vector(Vector(3))"
    > a22 = Vector1@2102 "Vector(Vector(4))"
    > b11 = Vector1@2115 "Vector(Vector(4))"
    > b12 = Vector1@2116 "Vector(Vector(3))"
    > b21 = Vector1@2117 "Vector(Vector(2))"
    > b22 = Vector1@2118 "Vector(Vector(1))"
    > this = MatrizOps@2083
```

Se realiza la división de matrices para la matriz 1 (siendo estas las submatrices a11, a12, a21 y a22) y para la matriz 2 (siendo estas las b11, b12, b21, b22)

5. Paso

```
✓ VARIABLES
  ✓ Local
    > m1 = Vector1@2081 "Vector(Vector(1, 2), Vector(3, 4))"
    > m2 = Vector1@2082 "Vector(Vector(4, 3), Vector(2, 1))"
    n = 2
    mid = 1
    > a11 = Vector1@2099 "Vector(Vector(1))"
    > a12 = Vector1@2100 "Vector(Vector(2))"
    > a21 = Vector1@2101 "Vector(Vector(3))"
    > a22 = Vector1@2102 "Vector(Vector(4))"
    > b11 = Vector1@2115 "Vector(Vector(4))"
    > b12 = Vector1@2116 "Vector(Vector(3))"
    > b21 = Vector1@2117 "Vector(Vector(2))"
    > b22 = Vector1@2118 "Vector(Vector(1))"
    > this = MatrizOps@2083
      app > src > main > scala > taller > Matriz.scala > {} taller > MatrizOps > multMatrizRec
      1 package taller
      12 class MatrizOps(){
      69   def multMatrizRec(m1: Matriz, m2: Matriz): Matriz = {
      85     subMatriz(m2, 0, 0, mid),
      86     subMatriz(m2, 0, mid, mid),
      87     subMatriz(m2, mid, 0, mid),
      88     subMatriz(m2, mid, mid, mid)
      89   }
      90   // Calculamos las submatrices de la matriz resultante
      91   val c11 = sumMatriz(multMatrizRec(a11, b11), multMatrizRec(a12, b21))
      92   val c12 = sumMatriz(multMatrizRec(a11, b12), multMatrizRec(a12, b22))
      93   val c21 = sumMatriz(multMatrizRec(a21, b11), multMatrizRec(a22, b21))
      94   val c22 = sumMatriz(multMatrizRec(a21, b12), multMatrizRec(a22, b22))
      95   // Combinamos las submatrices para formar la matriz completa
      96   val top = c11.zip(c12).map { case (left, right) => left ++ right }
      97   val bottom = c21.zip(c22).map { case (left, right) => left ++ right }
      98   top ++ bottom
      99
      100
```

En este paso se llamará de manera recursiva a la función para que haga la multiplicación de las submatrices (siendo estas submatrices de un solo valor), respectivas al correcto orden de la multiplicación de matrices

6. Paso

Se ejecuta la función con los valores de a_{11} y b_{11} como estaba indicado y evalúa si la longitud de las matrices es igual a 1.

7. Paso

The screenshot shows a Scala IDE interface with the following details:

- VARIABLES** sidebar:
 - Local**:
 - `m1 = Vector1@2099 "Vector(Vector(1))"`
 - `m2 = Vector1@2115 "Vector(Vector(4))"`
 - `n = 1`
 - `this = MatrizOps@2083`- Code Editor**:

```
app > src > main > scala > taller > Matriz.scala > {} taller > MatrizOps > MatrizOps.scala
```

Content of `MatrizOps.scala`:

```
1 package taller
2 class MatrizOps(){
3     def restaMatriz(m1:Matriz, m2:Matriz):Matriz = {
4         //Resta dos matrices
5         Vector.tabulate(m1.length,m1.length)((i,j) => m1(i,j) - m2(i,j))
6         //se crea un vector de mismas dimensiones, donde
7         //y se restan los valores de las matrices m1 y m2
8     }
9
10    def multMatrizRec(m1: Matriz, m2: Matriz): Matriz = {
11        val n = m1.length
12        if (n == 1) {
13            // Caso base: matrices de tamaño 1x1
14            Vector(Vector(m1(0)(0) * m2(0)(0)))
15        } else {
16            val mid = n / 2
17            val sub1 = multMatrizRec(m1(0 until mid, 0 until mid), m2(0 until mid, 0 until mid))
18            val sub2 = multMatrizRec(m1(mid until n, 0 until mid), m2(0 until mid, mid until n))
19            val sub3 = multMatrizRec(m1(0 until mid, mid until n), m2(mid until n, 0 until mid))
20            val sub4 = multMatrizRec(m1(mid until n, mid until n), m2(mid until n, mid until n))
21            val result = new Matriz(mid * 2, mid * 2)
22            for (i <- 0 until mid; j <- 0 until mid) {
23                result(i, j) = sub1(i, j) + sub2(i, j) + sub3(i, j) + sub4(i, j)
24            }
25            result
26        }
27    }
28}
```

The code editor highlights the line `Vector(Vector(m1(0)(0) * m2(0)(0)))` in yellow, indicating it is currently selected or being edited.

Al ser la longitud igual a 1 ingresa en la condición y multiplica el primer elemento de la matriz m1 (en este caso único) por el primer elemento de la matriz 2 (en este caso único)

8. Paso

The screenshot shows a code editor with two panes. The left pane displays a variable list under 'VARIABLES' and a local variable 'm1' with its value: `> m1 = Vector1@2081 "Vector(Vector(1, 2), Vector(3, 4))"`. The right pane shows a Scala file with the following code:

```
app > src > main > scala > taller > Matrizscala > () taller > MatrizOps > multMatrixRec
  1 package taller
  2 class MatrizOps(){
  3   def multMatrixRec(m1: Matriz, m2: Matriz): Matriz = {
  4     // Calculamos las submatrices de la matriz resultante
  5     val c11 = sumMatriz(multMatrixRec(a11, b11), multMatrixRec(a12, b21))
  6     val c12 = sumMatriz(multMatrixRec(a11, b12), multMatrixRec(a12, b22))
  7     val c21 = sumMatriz(multMatrixRec(a21, b11), multMatrixRec(a22, b21))
  8     val c22 = sumMatriz(multMatrixRec(a21, b12), multMatrixRec(a22, b22))
  9
 10    // Combinamos las submatrices para formar la matriz completa
 11    val top = c11.zip(c12).map { case (left, right) => left ++ right }
 12    val bottom = c21.zip(c22).map { case (left, right) => left ++ right }
 13    top ++ bottom
 14  }
 15
 16  def multMatrixRecPar(m1: Matriz, m2: Matriz): Matriz = {
 17    val n = m1.length
 18
 19    val
```

Realizo el mismo procedimiento para las otras sumas y llamadas recursivas de la función, dando como resultado las submatrices asignadas a c11, c12, c21 y c22, y juntará estas matrices.

9. Paso

```

1 package taller
2 class MatrizOps(){}
69 def multMatrizRec(m1: Matriz, m2: Matriz): Matriz = {
91     // Calculamos las submatrices de la matriz resultante
92     val c11 = sumMatriz(multMatrizRec(a11, b11), multMatrizRec(a12, b21))
93     val c12 = sumMatriz(multMatrizRec(a11, b12), multMatrizRec(a12, b22))
94     val c21 = sumMatriz(multMatrizRec(a21, b11), multMatrizRec(a22, b21))
95     val c22 = sumMatriz(multMatrizRec(a21, b12), multMatrizRec(a22, b22))
96
97     // Combinamos las submatrices para formar la matriz completa
98     val top = c11.zip(c12).map { case (left, right) => left ++ right }
99     val bottom = c21.zip(c22).map { case (left, right) => left ++ right }
100    top ++ bottom
101
102 }

```

Primero se juntaron las submatrices c11 y c12, creando así la primera parte de la matriz

10. Paso

```

1 package taller
2 class MatrizOps(){}
69 def multMatrizRec(m1: Matriz, m2: Matriz): Matriz = {
91     // Calculamos las submatrices de la matriz resultante
92     val c11 = sumMatriz(multMatrizRec(a11, b11), multMatrizRec(a12, b21))
93     val c12 = sumMatriz(multMatrizRec(a11, b12), multMatrizRec(a12, b22))
94     val c21 = sumMatriz(multMatrizRec(a21, b11), multMatrizRec(a22, b21))
95     val c22 = sumMatriz(multMatrizRec(a21, b12), multMatrizRec(a22, b22))
96
97     // Combinamos las submatrices para formar la matriz completa
98     val top = c11.zip(c12).map { case (left, right) => left ++ right }
99     val bottom = c21.zip(c22).map { case (left, right) => left ++ right }
100    top ++ bottom
101
102 }

```

Y después se juntaron las submatrices c21 y c22 para crear la segunda parte de la matriz.

Obteniéndose así la siguiente matriz:

```

8   5
20  13

```

- función multStrassen:**

Esta función secuencial recibe 2 matrices, es decir un type Matriz = Vector[Vector[Int]], y retorna otra matriz del mismo tipo. Para esta función se utilizarán matrices cuadradas y su tamaño será potencia de 2.

En esta función se lleva a cabo el algoritmo de Strassen para multiplicar 2 matrices.

1. primer paso:

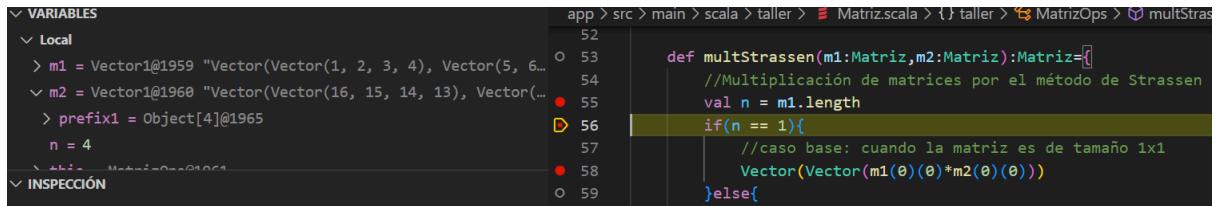
```

14 val matriz1 = Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))
15 val matriz2 = Vector(Vector(16, 15, 14, 13), Vector(12, 11, 10, 9), Vector(8, 7, 6, 5), Vector(4, 3, 2, 1))
16
17 test("Multiplicar la matriz 1 por la matriz 2 dará como resultado: " +
18     "Vector(Vector(80, 70, 60, 50), Vector(240, 214, 188, 162), Vector(400, 358, 316, 274), Vector(560, 502, 444, 386))") {
19     assert[objMatriz.multStrassen(matriz1, matriz2) == Vector(
20         Vector(80, 70, 60, 50), Vector(240, 214, 188, 162), Vector(400, 358, 316, 274), Vector(560, 502, 444, 386))]
21
22 }

```

Se toman 2 matrices de dimensión 4x4, y se compara si el valor es el esperado.

2. paso:

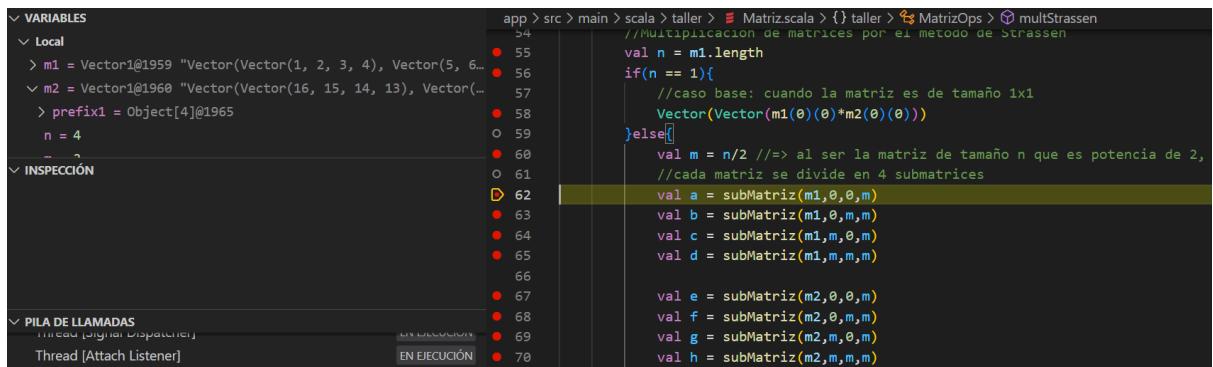


```

app > src > main > scala > taller > Matrizscala > {} taller > MatrizOps > multStrassen
  52
  53   def multStrassen(m1:Matriz,m2:Matriz):Matriz= [
  54     //Multiplicación de matrices por el método de Strassen
  55     val n = m1.length
  56     if(n == 1){
  57       //caso base: cuando la matriz es de tamaño 1x1
  58       Vector(Vector(m1(0)(0)*m2(0)(0)))
  59     }else{
  
```

En este caso se evalúa si se cumple el caso base, que es en el que la longitud de las matrices es 1. Aún no es el caso.

3. Paso:

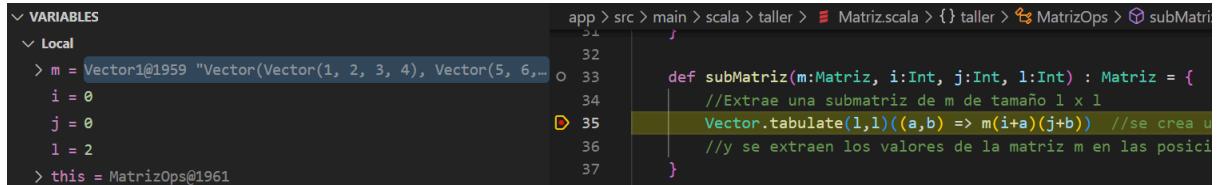


```

app > src > main > scala > taller > Matrizscala > {} taller > MatrizOps > multStrassen
  54   //Multiplicación de matrices por el método de Strassen
  55   val n = m1.length
  56   if(n == 1){
  57     //caso base: cuando la matriz es de tamaño 1x1
  58     Vector(Vector(m1(0)(0)*m2(0)(0)))
  59   }else{
  60     val m = n/2 //=> al ser la matriz de tamaño n que es potencia de 2,
  61     //cada matriz se divide en 4 submatrices
  62     val a = subMatriz(m1,0,0,m)
  63     val b = subMatriz(m1,0,m,m)
  64     val c = subMatriz(m1,m,0,m)
  65     val d = subMatriz(m1,m,m,m)
  66
  67     val e = subMatriz(m2,0,0,m)
  68     val f = subMatriz(m2,0,m,m)
  69     val g = subMatriz(m2,m,0,m)
  70     val h = subMatriz(m2,m,m,m)
  
```

Entraría en esta condición, donde genera las cuatro submatrices por cada matriz. Es decir, divide la matriz original en 4. m es la mitad de la dimensión de las matrices, para así hallar las submatrices.

4. Paso



```

app > src > main > scala > taller > Matrizscala > {} taller > MatrizOps > subMatriz
  31   }
  32
  33   def subMatriz(m:Matriz, i:Int, j:Int, l:Int) : Matriz = {
  34     //Extrae una submatriz de m de tamaño l x l
  35     Vector.tabulate(l,l)((a,b) => m(i+a)(j+b)) //se crea un vector de l filas y l columnas
  36     //y se extraen los valores de la matriz m en las posiciones indicadas
  37   }
  
```

Se inicia con la primera matriz, con parámetros i, j fila columna igual a 0, que será desde donde iniciará la primera submatriz, y l será la dimensión de la misma, a y b representan los índices del vector a retornar, y se trabajan con estos para asegurar tomar desde los índices indicados de la matriz original.

```

▽ Local
    m$2 = Vector1@1959 "Vector(Vector(1, 2, 3, 4), Vector(5, ...)
    prefix1 = Object[4]@1964
        0 = Vector1@1985 "Vector(1, 2, 3, 4)"
            > 0 = Integer@1994 "1"
            > 1 = Integer@1995 "2"
            > 2 = Integer@1996 "3"
            > 3 = Integer@1997 "4"
        1 = Vector1@1986 "Vector(5, 6, 7, 8)"
        2 = Vector1@1987 "Vector(9, 10, 11, 12)"
        3 = Vector1@1988 "Vector(13, 14, 15, 16)"
    i$1 = 0
    j$1 = 0
    a = 0
    b = 1
    32
    33
    34
    35
    36
    37
    38
    39
    40
    41
    42
    43
    44
    45
    46
    47
    48
    49
    50
    51
    52
    53
    54
    55
    56
    57
    58
    59
    60
    61
    62
    63
    64
    65
    66
    67
    68
    69
    70
    71
    72
    73
    74
    75
    76
    77
    78
    79
    80
    81
    82
    83
    84
    85
    86
    87
    88
    89
    90
    91
    92
    93
    94
    95
    96
    97
    98
    99
    100
    101
    102
    103
    104
    105
    106
    107
    108
    109
    110
    111
    112
    113
    114
    115
    116
    117
    118
    119
    120
    121
    122
    123
    124
    125
    126
    127
    128
    129
    130
    131
    132
    133
    134
    135
    136
    137
    138
    139
    140
    141
    142
    143
    144
    145
    146
    147
    148
    149
    150
    151
    152
    153
    154
    155
    156
    157
    158
    159
    160
    161
    162
    163
    164
    165
    166
    167
    168
    169
    170
    171
    172
    173
    174
    175
    176
    177
    178
    179
    180
    181
    182
    183
    184
    185
    186
    187
    188
    189
    190
    191
    192
    193
    194
    195
    196
    197
    198
    199
    200
    201
    202
    203
    204
    205
    206
    207
    208
    209
    210
    211
    212
    213
    214
    215
    216
    217
    218
    219
    220
    221
    222
    223
    224
    225
    226
    227
    228
    229
    230
    231
    232
    233
    234
    235
    236
    237
    238
    239
    240
    241
    242
    243
    244
    245
    246
    247
    248
    249
    250
    251
    252
    253
    254
    255
    256
    257
    258
    259
    260
    261
    262
    263
    264
    265
    266
    267
    268
    269
    270
    271
    272
    273
    274
    275
    276
    277
    278
    279
    280
    281
    282
    283
    284
    285
    286
    287
    288
    289
    290
    291
    292
    293
    294
    295
    296
    297
    298
    299
    300
    301
    302
    303
    304
    305
    306
    307
    308
    309
    310
    311
    312
    313
    314
    315
    316
    317
    318
    319
    320
    321
    322
    323
    324
    325
    326
    327
    328
    329
    330
    331
    332
    333
    334
    335
    336
    337
    338
    339
    340
    341
    342
    343
    344
    345
    346
    347
    348
    349
    350
    351
    352
    353
    354
    355
    356
    357
    358
    359
    360
    361
    362
    363
    364
    365
    366
    367
    368
    369
    370
    371
    372
    373
    374
    375
    376
    377
    378
    379
    380
    381
    382
    383
    384
    385
    386
    387
    388
    389
    390
    391
    392
    393
    394
    395
    396
    397
    398
    399
    400
    401
    402
    403
    404
    405
    406
    407
    408
    409
    410
    411
    412
    413
    414
    415
    416
    417
    418
    419
    420
    421
    422
    423
    424
    425
    426
    427
    428
    429
    430
    431
    432
    433
    434
    435
    436
    437
    438
    439
    440
    441
    442
    443
    444
    445
    446
    447
    448
    449
    450
    451
    452
    453
    454
    455
    456
    457
    458
    459
    460
    461
    462
    463
    464
    465
    466
    467
    468
    469
    470
    471
    472
    473
    474
    475
    476
    477
    478
    479
    480
    481
    482
    483
    484
    485
    486
    487
    488
    489
    490
    491
    492
    493
    494
    495
    496
    497
    498
    499
    500
    501
    502
    503
    504
    505
    506
    507
    508
    509
    510
    511
    512
    513
    514
    515
    516
    517
    518

```

```

✓ b = Vector1@2044 "Vector(Vector(3, 4), Vector(7, 8))"
  ✓ prefix1 = Object[2]@2049
    ✓ 0 = Vector1@2050 "Vector(3, 4)"
      ✓ prefix1 = Object[2]@2054
        > 0 = Integer@1996 "3"
        > 1 = Integer@1997 "4"
    ✓ 1 = Vector1@2051 "Vector(7, 8)"
      ✓ prefix1 = Object[2]@2055
        > 0 = Integer@2056 "7"
        > 1 = Integer@2057 "8"
  > this = MatrizOps@1961

```

6. Paso

<pre> ✓ Local > m1 = Vector1@1959 "Vector(Vector(1, 2, 3, 4), Vector(5, 6... o 52 > m2 = Vector1@1960 "Vector(Vector(16, 15, 14, 13), Vector(... 53 n = 4 m = 2 > a = Vector1@2022 "Vector(Vector(1, 2), Vector(5, 6))" 54 > b = Vector1@2044 "Vector(Vector(3, 4), Vector(7, 8))" 55 > this = MatrizOps@1961 56 </pre>	<pre> def multStrassen(m1:Matriz,m2:Matriz):Matriz //Multiplicación de matrices por el método de Strassen val n = m1.length if(n == 1){ //caso base: cuando la matriz es de 1x1 Vector(Vector(m1(0)(0)*m2(0)(0))) }else{ val m = n/2 //=> al ser la matriz de 4x4 se divide en 4 submatrices de 2x2 val a = subMatriz(m1,0,0,m) val b = subMatriz(m1,0,m,m) val c = subMatriz(m1,m,0,m) 57 val d = subMatriz(m1,m,m,m) 58 } </pre>
--	---

Ahora, para la tercera submatriz, ya deberá comenzar desde la mitad de las filas, y desde la columna 0, con dimensión de la mitad del tamaño de la matriz.

Se repite de nuevo el proceso, y obtenemos:

```

✓ c = Vector1@2065 "Vector(Vector(9, 10), Vector(13, 14))"
  ✓ prefix1 = Object[2]@2071
    ✓ 0 = Vector1@2072 "Vector(9, 10)"
      ✓ prefix1 = Object[2]@2076
        > 0 = Integer@2077 "9"
        > 1 = Integer@2078 "10"
    ✓ 1 = Vector1@2073 "Vector(13, 14)"
      ✓ prefix1 = Object[2]@2081
        > 0 = Integer@2082 "13"
        > 1 = Integer@2083 "14"
  > this = MatrizOps@1961

```

Ahora, se calcula la cuarta submatriz, y esta ya comenzará desde la mitad de filas y la mitad de columnas, de misma longitud que las anteriores. Obtenemos como resultado la matriz:

```
✓ d = Vector1@2091 "Vector(Vector(11, 12), Vector(15, 16))"  
  ✓ prefix1 = Object[2]@2098  
    ✓ 0 = Vector1@2099 "Vector(11, 12)"  
      > prefix1 = Object[2]@2103  
    ✓ 1 = Vector1@2100 "Vector(15, 16)"  
      ✓ prefix1 = Object[2]@2108  
        > 0 = Integer@2109 "15"  
        > 1 = Integer@2110 "16"  
> this = MatrixOps@1961
```

7. Paso

En este caso, se repite exactamente el mismo procedimiento, solo que con la segunda matriz.

Ahora, es útil saber que a,b,c,d serán las 4 submatrices de la matriz 1, y e,f,g,h serán las de la matriz 2.

Finalmente, las submatrices de la segunda matriz serán:

```
> e = Vector1@2015 "Vector(Vector(16, 15), Vector(12, 11))"  
> f = Vector1@2016 "Vector(Vector(14, 13), Vector(10, 9))"  
> g = Vector1@2029 "Vector(Vector(8, 7), Vector(4, 3))"  
> h = Vector1@2044 "Vector(Vector(6, 5), Vector(2, 1))"  
> this = MatrizOps@1958
```

```

> a = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> b = Vector1@1986 "Vector(Vector(3, 4), Vector(7, 8))"
> c = Vector1@1992 "Vector(Vector(9, 10), Vector(13, 14))"
> d = Vector1@2002 "Vector(Vector(11, 12), Vector(15, 16))"
> e = Vector1@2015 "Vector(Vector(16, 15), Vector(12, 11))"
> f = Vector1@2016 "Vector(Vector(14, 13), Vector(10, 9))"
> g = Vector1@2029 "Vector(Vector(8, 7), Vector(4, 3))"
> h = Vector1@2044 "Vector(Vector(6, 5), Vector(2, 1))"
> this = MatrizOps@1958

```

Estas son las submatrices resultantes totales.

8. Paso:

```

< Local
> m1 = Vector1@1956 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, ...)"
> m2 = Vector1@1957 "Vector(Vector(16, 15, 14, 13), Vector(12, 11, ..."
n = 4
m = 2
> a = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> b = Vector1@1986 "Vector(Vector(3, 4), Vector(7, 8))" 38
o 39
40
● 41
42
43
44
45
o 46
47
D 48
49
50
51
      ● 68
      ● 69
      ● 70
      ● 71
      ● 72
      ● 73
      ● 74
      ● 75
      val f = subMatriz(m2,0,m,m)
      val g = subMatriz(m2,m,0,m)
      val h = subMatriz(m2,m,m,m)
      // se calcula los productos de las submatrices
      val p1 = multStrassen(a,restaMatriz(f,h))
      val p2 = multStrassen(sumMatriz(a,b),h)
      val p3 = multStrassen(sumMatriz(c,d),g)

```

En este caso, se genera una llamada recursiva a la función, pasándole como parámetro la matriz a, y la resta entre f y h, según el mismo algoritmo de Strassen.

```

< Local
> m1 = Vector1@2016 "Vector(Vector(14, 13), Vector(10, 9))"
> m2 = Vector1@2044 "Vector(Vector(6, 5), Vector(2, 1))"
> this = MatrizOps@1958 38
o 39
40
● 41
42
43
44
45
o 46
47
D 48
49
50
51
      def sumMatriz(m1:Matriz, m2:Matriz) : Matriz = {
        //Suma dos matrices
        Vector.tabulate(m1.length,m1.length)((i,j) => m1(i)(j) + m2(i)(j))
        //se crea un vector de mismas dimensiones, donde i,j son sus indices
        //y se suman los valores de las matrices m1 y m2 en las posiciones i
      }
      def restaMatriz(m1:Matriz, m2:Matriz):Matriz ={
        //Resta dos matrices
        Vector.tabulate(m1.length,m1.length)((i,j) => m1(i)(j) - m2(i)(j))
        //se crea un vector de mismas dimensiones, donde i,j son sus indices
        //y se restan los valores de las matrices m1 y m2 en las posiciones i
      }

```

Se realiza la resta entre las dos matrices, esta función usa el tabulate, para igualmente generar una matriz con mismas dimensiones que las dadas, a partir de los índices de la misma, donde se acceden a mismas posiciones en las 2 matrices para hacer la resta. Al finalizar los cálculos la matriz que retorna es:

```

< VARIABLES
< Local
> m1 = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2067 "Vector(Vector(8, 8), Vector(8, 8))"
n = 2
m = 1
> this = MatrizOps@1958 38
o 39
40
● 41
42
43
44
45
o 46
47
D 48
49
50
51
      ● 58
      ● 59
      ● 60
      ● 61
      ● 62
      ● 63
      ● 64
      ● 65
      app > src > main > scala > taller > Matriz.scala > {} taller > MatrizOps >
      Vector(Vector(m1(0)(0)*m2(0)(0)))
      }else{
        val m = n/2 //=> al ser la matriz de tama
        //cada matriz se divide en 4 submatrices
        val a = subMatriz(m1,0,0,m)
        val b = subMatriz(m1,0,m,m)
        val c = subMatriz(m1,m,0,m)
        val d = subMatriz(m1,m,m,m)
      }
> m2 = Vector1@2067 "Vector(Vector(8, 8), Vector(8, 8))" 52

```

9. Paso:

Como seguimos en el llamado recursivo, se invoca la función ahora con la matriz a, y la resultante de la resta.

Ahora, se repite el mismo proceso anterior, para el cálculo de las 8 submatrices totales (4 de la matriz 1 y 4 de la matriz 2). Al final, las submatrices ahora obtenidas serán:

```
> m1 = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2067 "Vector(Vector(8, 8), Vector(8, 8))"
  n = 2
  m = 1
> a = Vector1@2078 "Vector(Vector(1))"
> b = Vector1@2083 "Vector(Vector(2))"
> c = Vector1@2089 "Vector(Vector(5))"
> d = Vector1@2097 "Vector(Vector(6))"
> e = Vector1@2098 "Vector(Vector(8))"
> f = Vector1@2108 "Vector(Vector(8))"
> g = Vector1@2109 "Vector(Vector(8))"
> h = Vector1@2121 "Vector(Vector(8))"
> this = MatrizOps@1958
```

10. Paso

The screenshot shows a debugger interface with two panes. The left pane, titled 'Local', lists variables: m1, m2, n, m, and a. The right pane shows the execution of code lines 73 through 76. Line 73 is highlighted in yellow. The code involves matrix operations: sumMatriz, subMatriz, multStrassen, and sumMatriz again. A note in the code states: // se calcula los productos de las submatrices. The code uses Vector1 and MatrizOps objects.

```
val g = sumMatriz(m2,m,o,m)
val h = subMatriz(m2,m,m,m)

// se calcula los productos de las submatrices
val p1 = multStrassen(a,restaMatriz(f,h))
val p2 = multStrassen(sumMatriz(a,b),h)
val p3 = multStrassen(sumMatriz(c,d),e)
val p4 = multStrassen(sumMatriz(d,c),g)
```

Ahora, nuevamente se hace un llamado recursivo con la matriz ahora obtenida dentro de la primera recursión, a, y la resta de f y h.

This screenshot continues the debugger session from the previous one. It shows the execution of the multStrassen function. The code defines the function and handles the base case where n == 1, returning a Vector of size 1x1 containing the product of the single elements of m1 and m2.

```
def multStrassen(m1:Matriz,m2:Matriz):Matriz = {
    //Multiplicación de matrices por el método de Strassen
    val n = m1.length
    if(n == 1){
        //caso base: cuando la matriz es de tamaño 1x1
        Vector(Vector(m1(0)(0)*m2(0)(0)))
    }
}
```

Como vemos, la matriz nos ha quedado de tamaño indivisible, es decir, cumple el caso base, y es allí cuando según el algoritmo, debemos multiplicar los elementos de ambas matrices. Lo cual nos retornará 0.

11. Paso

```

// Local
> m1 = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2067 "Vector(Vector(8, 8), Vector(8, 8))"
  n = 2
  m = 1
> a = Vector1@2078 "Vector(Vector(1))"
> b = Vector1@2083 "Vector(Vector(2))"
> c = Vector1@2089 "Vector(Vector(5))"
> d = Vector1@2097 "Vector(Vector(6))"
> e = Vector1@2098 "Vector(Vector(8))"
> f = Vector1@2108 "Vector(Vector(8))"
> g = Vector1@2109 "Vector(Vector(8))"
> h = Vector1@2121 "Vector(Vector(8))"
> p1 = Vector1@2143 "Vector(Vector(0))"
> this = MatrizOps@1958

```

```

  62   val a = subMatriz(m1,0,0,m)
  63   val b = subMatriz(m1,0,m,m)
  64   val c = subMatriz(m1,m,0,m)
  65   val d = subMatriz(m1,m,m,m)
  66
  67   val e = subMatriz(m2,0,0,m)
  68   val f = subMatriz(m2,0,m,m)
  69   val g = subMatriz(m2,m,0,m)
  70   val h = subMatriz(m2,m,m,m)
  71
  72   // se calcula los productos de las submatrices
  73   val p1 = multStrassen(a,restaMatriz(f,h))
D 74   val p2 = multStrassen(sumMatriz(a,b),h)
  75   val p3 = multStrassen(sumMatriz(c,d),e)
  76   val p4 = multStrassen(d,restaMatriz(g,e))
  77   val p5 = multStrassen(sumMatriz(a,d),sumMatriz(e,h))
  78   val p6 = multStrassen(restaMatriz(b,d),sumMatriz(g,h))
  79   val p7 = multStrassen(restaMatriz(a,c),sumMatriz(e,f))
  80
  81
  82

```

Como vemos, p1 ya valdrá una matriz de dimensión 1 con valor 0, ahora, continuamos con hallar el valor p2, haciendo otro llamado recursivo, pero con la suma de la matriz a y b, y la matriz h, en este nivel de recursión, todas las submatrices ya son de tamaño 1.

```

// Local
> m1 = Vector1@2161 "Vector(Vector(3))"
> m2 = Vector1@2121 "Vector(Vector(8))"
  n = 1
> this = MatrizOps@1958

```

```

  52
  53   def multStrassen(m1:Matriz,m2:Matriz):Matriz={
  54     //Multiplicación de matrices por el método de Strassen
  55     val n = m1.length
  56
  57     if(n == 1){
  58       //caso base: cuando la matriz es de tamaño 1x1
D 59       Vector(Vector(m1(0)(0)*m2(0)(0)))
  60     }else{
  61
  62
  63
  64
  65
  66
  67
  68
  69
  70
  71
  72
  73
  74
  75
  76
  77
  78
  79
  80
  81
  82

```

Ahora, me retorna una matriz de tamaño 1 con valor 24.

12. Paso

```

// Local
> m1 = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2067 "Vector(Vector(8, 8), Vector(8, 8))"
  n = 2
  m = 1
> a = Vector1@2078 "Vector(Vector(1))"
> b = Vector1@2083 "Vector(Vector(2))"
> c = Vector1@2089 "Vector(Vector(5))"
> d = Vector1@2097 "Vector(Vector(6))"
> e = Vector1@2098 "Vector(Vector(8))"

```

```

  71
  72   // se calcula los productos de las submatrices
  73   val p1 = multStrassen(a,restaMatriz(f,h))
  74   val p2 = multStrassen(sumMatriz(a,b),h)
D 75   val p3 = multStrassen(sumMatriz(c,d),e)
  76   val p4 = multStrassen(d,restaMatriz(g,e))
  77   val p5 = multStrassen(sumMatriz(a,d),sumMatriz(e,h))
  78   val p6 = multStrassen(restaMatriz(b,d),sumMatriz(g,h))
  79   val p7 = multStrassen(restaMatriz(a,c),sumMatriz(e,f))
  80
  81   //se calculan las submatrices de la matriz resultante
  82   val c11 = sumMatriz(restaMatriz(sumMatriz(p5,p4),p2),p6)

```

Se hará nuevamente un llamado recursivo, con la suma de las matrices c y d, y la matriz e.

Finalmente, al repetir el procedimiento para el resto de los productos de las matrices se obtiene:

```

< Local
> m1 = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2067 "Vector(Vector(8, 8), Vector(8, 8))"
n = 2
m = 1
> a = Vector1@2078 "Vector(Vector(1))"
> b = Vector1@2083 "Vector(Vector(2))"
> c = Vector1@2089 "Vector(Vector(5))"
> d = Vector1@2097 "Vector(Vector(6))"
> e = Vector1@2098 "Vector(Vector(8))"
> f = Vector1@2108 "Vector(Vector(8))"
> g = Vector1@2109 "Vector(Vector(8))"
> h = Vector1@2121 "Vector(Vector(8))"
> p1 = Vector1@2143 "Vector(Vector(0))"
> p2 = Vector1@2168 "Vector(Vector(24))"
> p3 = Vector1@2197 "Vector(Vector(88))"
> p4 = Vector1@2198 "Vector(Vector(0))"
> p5 = Vector1@2223 "Vector(Vector(112))"
> p6 = Vector1@2255 "Vector(Vector(-64))"
> p7 = Vector1@2286 "Vector(Vector(-64))"
> this = MatrizOps@1958

```

13. Paso

```

< Local
> m1 = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2067 "Vector(Vector(8, 8), Vector(8, 8))"
n = 2
m = 1
81
82 //se calculan las submatrices de la matriz resultante
83 val c11 = sumMatriz(restaMatriz(sumMatriz(p5,p4),p2),p6)
84 val c12 = sumMatriz(p1,p2)
85 val c21 = sumMatriz(p3,p4)
86 val c22 = restaMatriz(restaMatriz(sumMatriz(p1,p5),p3),p7)
87
88
89
90
91
92

```

En este paso, vamos a calcular las submatrices de la matriz resultante, haciendo los cálculos correspondientes con los productos ya obtenidos previamente.

Al hacer todos estos cálculos, se obtiene:

```

> c11 = Vector1@2314 "Vector(Vector(24))"
> c12 = Vector1@2337 "Vector(Vector(24))"
> c21 = Vector1@2357 "Vector(Vector(88))"
> c22 = Vector1@2388 "Vector(Vector(88))"
> this = MatrizOps@1958

```

14. Paso

```

< Local
> m1 = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2067 "Vector(Vector(8, 8), Vector(8, 8))"
n = 2
m = 1
88
89 val c1 = c11.zip(c12).map({case (i,j) => i ++ j})
90 val c2 = c21.zip(c22).map({case (i,j) => i ++ j})
91 c1 ++ c2
92

```

Para este paso, empezaremos a concatenar fila por fila los elementos de c11 con c12. y las filas de c21 con c22. Finalmente, concatenamos estas 2 matrices, quedándonos una matriz cuadrada.

```

> c1 = Vector1@2417 "Vector(Vector(24, 24))"
> c2 = Vector1@2443 "Vector(Vector(88, 88))"

```

Finalmente, obtenemos el p1 para la matriz original, es decir, ya se cumplio con el primer llamado recursivo, para hallar solo p1, el cual nos quedó una matriz de 2x2.

```

> m1 = Vector1@2467 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
> m2 = Vector1@2468 "Vector(Vector(16, 15, 14, 13), Vector(12, 11, 10, 9), Vector(8, 7, 6, 5), Vector(4, 3, 2, 1))"
n = 4
m = 2
> a = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))"
> b = Vector1@2469 "Vector(Vector(3, 4), Vector(7, 8))"
> c = Vector1@2470 "Vector(Vector(9, 10), Vector(13, 14))"
> d = Vector1@2471 "Vector(Vector(11, 12), Vector(15, 16))"
> e = Vector1@2472 "Vector(Vector(16, 15), Vector(12, 11))"
> f = Vector1@2016 "Vector(Vector(14, 13), Vector(10, 9))"
> g = Vector1@2473 "Vector(Vector(8, 7), Vector(4, 3))"
> h = Vector1@2044 "Vector(Vector(6, 5), Vector(2, 1))"
> p1 = Vector1@2474 "Vector(Vector(24, 24), Vector(88, 88))"
> this = MatrizOps@1958

```

Para ahorrarnos repetir el mismo procedimiento, vamos a dar solo el cálculo final de todas las llamadas recursivas para hallar los productos, quedandonos:

```

> p1 = Vector1@2474 "Vector(Vector(24, 24), Vector(88, 88))"
> p2 = Vector1@2590 "Vector(Vector(36, 26), Vector(100, 74))"
> p3 = Vector1@2765 "Vector(Vector(584, 542), Vector(808, 750))"
> p4 = Vector1@2938 "Vector(Vector(-184, -184), Vector(-248, -248))"
> p5 = Vector1@3092 "Vector(Vector(460, 408), Vector(748, 664))"
> p6 = Vector1@3215 "Vector(Vector(-160, -128), Vector(-160, -128))"
> p7 = Vector1@3343 "Vector(Vector(-416, -384), Vector(-416, -384))"
> this = MatrizOps@1958

```

15. Paso

✓ Local > m1 = Vector1@2467 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))" > m2 = Vector1@2468 "Vector(Vector(16, 15, 14, 13), Vector(12, 11, 10, 9), Vector(8, 7, 6, 5), Vector(4, 3, 2, 1))" n = 4 m = 2 > a = Vector1@1981 "Vector(Vector(1, 2), Vector(5, 6))" > b = Vector1@2469 "Vector(Vector(3, 4), Vector(7, 8))" > c = Vector1@2470 "Vector(Vector(9, 10), Vector(13, 14))" > d = Vector1@2471 "Vector(Vector(11, 12), Vector(15, 16))" > e = Vector1@2472 "Vector(Vector(16, 15), Vector(12, 11))" > f = Vector1@2016 "Vector(Vector(14, 13), Vector(10, 9))" > g = Vector1@2473 "Vector(Vector(8, 7), Vector(4, 3))" > h = Vector1@2044 "Vector(Vector(6, 5), Vector(2, 1))" > p1 = Vector1@2474 "Vector(Vector(24, 24), Vector(88, 88))" > this = MatrizOps@1958	● 79 80 81 D 82 ● 83 ● 84 ● 85 86	val p7 = multStrassen(restaMatriz(a,c),sumMatriz(e,f)) //se calculan las submatrices de la matriz resultante val c11 = sumMatriz(restaMatriz(sumMatriz(p5,p4),p2),p6) val c12 = sumMatriz(p1,p2) val c21 = sumMatriz(p3,p4) val c22 = restaMatriz(restaMatriz(sumMatriz(p1,p5),p3),p7)
--	---	---

Ya obtenidos los productos, podemos hallar las submatrices de la matriz resultante, nuevamente, para ahorrarnos los llamados que ya hemos visto, se concluyen los siguientes resultados:

```

> c11 = Vector1@3391 "Vector(Vector(80, 70), Vector(240, 214))"
> c12 = Vector1@3418 "Vector(Vector(60, 50), Vector(188, 162))"
> c21 = Vector1@3419 "Vector(Vector(400, 358), Vector(560, 502))"
> c22 = Vector1@3472 "Vector(Vector(316, 274), Vector(444, 386))"
> this = MatrizOps@1958

```

16. Paso

```
val c22 = restamatrix2(restamatrix2(summatrix2(p1,p3),p5),p7)
//se concatenan las submatrices para formar la matriz resultante
val c1 = c11.zip(c12).map({case (i,j) => i + j})
val c2 = c21.zip(c22).map({case (i,j) => i + j})
c1 ++ c2
```

Ahora, se concatenan estas submatrices para obtener la matriz resultante, es decir, el resultado obtenido de la multiplicación de m1 y m2.

la matriz que se obtiene es:

80	70	60	50
240	214	188	162
400	358	316	274
560	502	444	386

INFORME DE PARALELIZACIÓN

- **función multStrassenPar**

La estrategia que se llevó a cabo para paralelizar esta función, fue con la abstracción de parallel.

Primero se extrajeron las 4 submatrices de la primera matriz dada, después las 4 de la segunda matriz dada, y por último, se halló las 4 submatrices de la matriz resultante usando parallel para paralelizar las multiplicaciones necesarias, es decir, se usó un parallel de 4. Por último se unieron estas 4 submatrices para formar la matriz resultante.

La Ley de Amdahl establece que el rendimiento de un sistema se ve limitado por la parte que no se puede mejorar. En el caso de la multiplicación de matrices utilizando el algoritmo de Strassen, podemos identificar dos componentes:

1. Parte Paralelizable (P): Esta es la parte del algoritmo que se puede ejecutar en paralelo. En multStrassenPar, las multiplicaciones de las submatrices y las sumas de las matrices resultantes se pueden realizar en paralelo. Esto representa una fracción significativa del tiempo total de ejecución.
2. Parte No Paralelizable (1 - P): Esta es la parte del algoritmo que debe ejecutarse secuencialmente. En el caso de Strassen, esto incluye la creación de submatrices y la combinación final de los resultados.

Cálculo de Ganancias

Para calcular las ganancias de rendimiento utilizando la Ley de Amdahl, se puede aplicar la fórmula:

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

donde:

- S es la mejora en el rendimiento.
- P es la fracción del tiempo que se puede parallelizar.
- N es el número de procesadores o núcleos utilizados.

Ejemplo de Cálculo

Suponiendo que:

- El 80% del tiempo de ejecución se puede parallelizar ($P=0.8$)
- Se utilizan 4 núcleos para la ejecución paralela ($N=4$).

Sustituyendo en la fórmula:

$$S = \frac{1}{(1-0.8) + \frac{0.8}{4}} = 2.5$$

Esto significa que, en este escenario, el rendimiento del sistema podría mejorar hasta 2.5 veces en comparación con la ejecución secuencial.

Esta función puede lograr una aceleración significativa, pero la ganancia depende del tamaño de las matrices a tratar.

- **12 CASOS PRUEBA**

Los tiempos de ejecución fueron tomados en la medida **ms**.

Para la toma de medida del tiempo, se usó el **whitWarmer**(calentador), para evitar tomas incorrectas, en el que el sistema no está en su punto estable aún.

- **Matrices de 2x2**

```

def pruebas(): Unit = {
    println("Multiplicacion Strassen 2x2")
    println("Matriz 1: ")
    m1.foreach(row => println(row))
    println("Matriz 2: ")
    m2.foreach(row => println(row))
    println("Resultado: ")
    (mat1.multStrassen(m1, m2)).foreach(row => println(row))
    val comp1 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m1, m2)
    println("Tiempo de ejecución de Strassen: " + comp1(0))
    println("Tiempo de ejecución de Strassen Paralelo: " + comp1(1))

    println("multiplicacion Strassen 2x2")
    println("Matriz 1: ")
    m2.foreach(row => println(row))
    println("Matriz 2: ")
    m1.foreach(row => println(row))
    println("Resultado: ")
    (mat1.multStrassen(m2, m1)).foreach(row => println(row))
    val comp2 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m2, m1)
    println("Tiempo de ejecución de Strassen: " + comp2(0))
    println("Tiempo de ejecución de Strassen Paralelo: " + comp2(1))
}

```

1)

Para todos los casos, los resultados siempre serán correctos, ya que se ha evaluado todos estos, y además, se ha comprobado en la sección anterior la función secuencial, que es la misma que se lleva a cabo en esta versión.

En este caso, la ganancia obtenida no es tan significativa, al ser matrices de un tamaño mínimo, estos algoritmos se desempeñan mejor si la matriz es más grande. La aceleración en esta prueba es mínima.

```

Multiplicacion Strassen 2x2
Matriz 1:
Vector(2, 4)
Vector(4, 3)
Matriz 2:
Vector(2, 2)
Vector(4, 3)
Resultado:
Vector(20, 16)
Vector(20, 17)
Unable to create a system terminal
Tiempo de ejecución de Strassen: 0.1654
Tiempo de ejecución de Strassen Paralelo: 0.1069

```

2)

Cabe recalcar que, se usaron 7 tamaños distintos, y para realizar 12 pruebas distintas se multiplicaron opuestamente las mismas, ya que la multiplicación de matrices no es conmutativa. Para este caso, el tiempo de ejecución del algoritmo secuencial resulta siendo mejor que el paralelo, a pesar de que las matrices son del mismo tamaño tratado antes, los tiempos fluctúan.

```

multiplicacion Strassen 2x2
Matriz 1:
Vector(2, 2)
Vector(4, 3)
Matriz 2:
Vector(2, 4)
Vector(4, 3)
Resultado:
Vector(12, 14)
Vector(20, 25)
Tiempo de ejecución de Strassen: 0.0384
Tiempo de ejecución de Strassen Paralelo: 0.1312

```

- Matrices 4x4

```
def pruebas(): Unit = {  
  
    println("Multiplicacion Strassen 4x4")  
    println("Matriz 1: ")  
    m3.foreach(row => println(row))  
    println("Matriz 2: ")  
    m4.foreach(row => println(row))  
    println("Resultado: ")  
    (mat1.multStrassen(m3, m4)).foreach(row => println(row))  
    val comp3 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m3, m4)  
    println("Tiempo de ejecución de Strassen: " + comp3(0))  
    println("Tiempo de ejecución de Strassen Paralelo: " + comp3(1))  
  
    println("Multiplicacion Strassen 4x4")  
    println("Matriz 1: ")  
    m4.foreach(row => println(row))  
    println("Matriz 2: ")  
    m3.foreach(row => println(row))  
    println("Resultado: ")  
    (mat1.multStrassen(m4, m3)).foreach(row => println(row))  
    val comp4 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m4, m3)  
    println("Tiempo de ejecución de Strassen: " + comp4(0))  
    println("Tiempo de ejecución de Strassen Paralelo: " + comp4(1))  
}
```

1)

En este caso, si vemos una mejora en los tiempos de ejecución, el algoritmo paralelizado se desempeña un poco mejor que el secuencial. La aceleración del segundo algoritmo con respecto al primero es de 2.44.

```
Multiplicacion Strassen 4x4  
Matriz 1:  
Vector(2, 1, 1, 2)  
Vector(2, 0, 0, 1)  
Vector(2, 1, 2, 1)  
Vector(1, 1, 0, 0)  
Matriz 2:  
Vector(1, 2, 1, 0)  
Vector(0, 2, 1, 2)  
Vector(1, 1, 1, 0)  
Vector(0, 1, 2, 1)  
Resultado:  
Vector(3, 9, 8, 4)  
Vector(2, 5, 4, 1)  
Vector(4, 9, 7, 3)  
Vector(1, 4, 2, 2)  
Unable to create a system terminal  
Tiempo de ejecución de Strassen: 0.4362  
Tiempo de ejecución de Strassen Paralelo: 0.1784
```

2)

Ahora, en este caso, el tiempo de ejecución del algoritmo paralelo se mantuvo en su rango, mientras que el secuencial se redujo significativamente.

-Matrices 8x8

```
Tiempo de ejecución de Strassen Paralelo: 0.1796  
Multiplicacion Strassen 4x4  
Matriz 1:  
Vector(1, 2, 1, 0)  
Vector(0, 2, 1, 2)  
Vector(1, 1, 1, 0)  
Vector(0, 1, 2, 1)  
Matriz 2:  
Vector(2, 1, 1, 2)  
Vector(2, 0, 0, 1)  
Vector(2, 1, 2, 1)  
Vector(1, 1, 0, 0)  
Resultado:  
Vector(8, 2, 3, 5)  
Vector(8, 3, 2, 3)  
Vector(6, 2, 3, 4)  
Vector(7, 3, 4, 3)  
Tiempo de ejecución de Strassen: 0.1331  
Tiempo de ejecución de Strassen Paralelo: 0.1796
```

```

def pruebas(): Unit = {
    println("Multiplicacion Strassen 8x8")
    println("Matriz 1: ")
    m5.foreach(row => println(row))
    println("Matriz 2: ")
    m6.foreach(row => println(row))
    println("Resultado: ")
    (mat1.multStrassen(m5, m6)).foreach(row => println(row))
    val comp5 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m5, m6)
    println("Tiempo de ejecución de Strassen: " + comp5(0))
    println("Tiempo de ejecución de Strassen Paralelo: " + comp5(1))

    println("Multiplicacion Strassen 8x8")
    println("Matriz 1: ")
    m6.foreach(row => println(row))
    println("Matriz 2: ")
    m5.foreach(row => println(row))
    println("Resultado: ")
    (mat1.multStrassen(m6, m5)).foreach(row => println(row))
    val comp6 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m6, m5)
    println("Tiempo de ejecución de Strassen: " + comp6(0))
    println("Tiempo de ejecución de Strassen Paralelo: " + comp6(1))
}

```

1)

```

Multiplicacion Strassen 8x8
Matriz 1:
Vector(1, 2, 3, 2, 0, 1, 1, 3)
Vector(2, 3, 2, 3, 0, 1, 0, 2)
Vector(1, 1, 3, 2, 3, 2, 2, 2)
Vector(3, 2, 1, 1, 2, 0, 3, 0)
Vector(0, 0, 3, 0, 1, 1, 2, 0)
Vector(0, 3, 0, 3, 3, 2, 3, 1)
Vector(1, 2, 0, 1, 1, 2, 1, 2)
Vector(0, 0, 3, 0, 1, 0, 1, 3)
Matriz 2:
Vector(3, 2, 1, 0, 1, 2, 3, 2)
Vector(2, 1, 0, 2, 2, 2, 1, 1)
Vector(2, 2, 1, 0, 2, 3, 1, 2)
Vector(1, 1, 1, 1, 2, 2, 1, 1)
Vector(0, 1, 0, 3, 0, 0, 0, 2)
Vector(0, 0, 2, 0, 2, 0, 2, 3)
Vector(1, 2, 1, 3, 2, 2, 1, 0)
Vector(2, 0, 0, 1, 1, 3, 0, 3)
Resultado:
Vector(22, 14, 9, 12, 22, 30, 13, 24)
Vector(23, 14, 9, 11, 22, 28, 16, 23)
Vector(19, 18, 12, 21, 23, 27, 15, 29)
Vector(19, 19, 8, 20, 17, 21, 16, 15)
Vector(8, 11, 7, 9, 12, 13, 7, 11)
Vector(14, 15, 10, 28, 23, 21, 13, 21)
Vector(13, 8, 7, 13, 15, 16, 11, 19)
Vector(13, 9, 4, 9, 11, 20, 4, 17)
Unable to create a system terminal
Tiempo de ejecución de Strassen: 0.5423
Tiempo de ejecución de Strassen Paralelo: 0.397

```

2)

```

Multiplicacion Strassen 8x8
Matriz 1:
Vector(3, 2, 1, 0, 1, 2, 3, 2)
Vector(2, 1, 0, 2, 2, 2, 1, 1)
Vector(2, 2, 1, 0, 2, 3, 1, 2)
Vector(1, 1, 1, 1, 2, 2, 1, 1)
Vector(0, 1, 0, 3, 0, 0, 0, 2)
Vector(0, 0, 2, 0, 2, 0, 2, 3)
Vector(1, 2, 1, 3, 2, 2, 1, 0)
Vector(2, 0, 0, 1, 1, 3, 0, 3)
Matriz 2:
Vector(1, 2, 3, 2, 0, 1, 1, 3)
Vector(2, 3, 2, 3, 0, 1, 0, 2)
Vector(1, 1, 3, 2, 3, 2, 2, 2)
Vector(3, 2, 1, 1, 2, 0, 3, 0)
Vector(0, 0, 3, 0, 1, 1, 2, 0)
Vector(0, 3, 0, 3, 3, 2, 3, 1)
Vector(1, 2, 0, 1, 1, 2, 1, 2)
Vector(0, 0, 3, 0, 1, 0, 1, 3)
Resultado:
Vector(11, 25, 25, 23, 15, 18, 18, 29)
Vector(11, 19, 19, 16, 14, 11, 20, 15)
Vector(8, 22, 25, 22, 17, 16, 20, 23)
Vector(8, 16, 18, 15, 15, 12, 18, 14)
Vector(11, 9, 11, 6, 8, 1, 11, 8)
Vector(4, 6, 21, 6, 13, 10, 13, 17)
Vector(16, 23, 19, 20, 18, 13, 23, 13)
Vector(5, 15, 19, 14, 15, 9, 19, 18)
Tiempo de ejecución de Strassen: 0.4788
Tiempo de ejecución de Strassen Paralelo: 0.2398

```

- 1) En este caso, el algoritmo paralelo también se desempeña mejor que el secuencial, con respecto a su tiempo de ejecución.
- 2) Hubo variaciones en ambos casos con respecto a sus tiempos de ejecución, sin embargo, el algoritmo paralelo sigue desempeñándose mejor, aunque no sea tan significativa la diferencia.

-Matrices 16x16

```
// 10 pruebas para la función Strassen
def pruebas(): Unit = [
    println("Multiplicacion Strassen 16x16")
    println("Matriz 1: ")
    m7.foreach(row => println(row))
    println("Matriz 2: ")
    m8.foreach(row => println(row))
    println("Resultado: ")
    (mat1.multStrassen(m7, m8)).foreach(row => println(row))
    val comp7 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m7, m8)
    println("Tiempo de ejecución de Strassen: " + comp7(0))
    println("Tiempo de ejecución de Strassen Paralelo: " + comp7(1))

    println("Multiplicacion Strassen 16x16")
    println("Matriz 1: ")
    m8.foreach(row => println(row))
    println("Matriz 2: ")
    m7.foreach(row => println(row))
    println("Resultado: ")
    (mat1.multStrassen(m8, m7)).foreach(row => println(row))
    val comp8 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m8, m7)
    println("Tiempo de ejecución de Strassen: " + comp8(0))
    println("Tiempo de ejecución de Strassen Paralelo: " + comp8(1))
```

1)

```
Multiplicacion Strassen 16x16
Matriz 1:
Vector(4, 4, 0, 4, 3, 2, 4, 2, 0, 2, 4, 0, 2, 0, 3, 2)
Vector(2, 4, 4, 3, 2, 1, 4, 4, 2, 2, 4, 1, 0, 1, 1, 2)
Vector(0, 0, 3, 0, 1, 0, 4, 3, 0, 0, 2, 0, 1, 0, 2, 2)
Vector(2, 4, 0, 2, 0, 0, 2, 0, 0, 4, 4, 2, 0, 0, 0, 2)
Vector(4, 4, 1, 0, 0, 2, 2, 1, 2, 0, 1, 3, 0, 4, 1, 2)
Vector(3, 0, 4, 2, 4, 4, 0, 2, 0, 2, 2, 1, 1, 1, 3, 2)
Vector(4, 3, 1, 2, 2, 1, 4, 0, 1, 1, 0, 1, 4, 0, 4, 3)
Vector(3, 4, 2, 4, 3, 3, 4, 0, 3, 0, 3, 2, 4, 2, 4, 2)
Vector(3, 0, 2, 1, 2, 1, 0, 0, 3, 3, 1, 2, 2, 1, 3, 2)
Vector(4, 0, 4, 3, 1, 3, 2, 2, 0, 4, 0, 1, 3, 4, 0, 1)
Vector(3, 2, 0, 0, 3, 1, 0, 2, 1, 3, 4, 2, 3, 0, 4, 3)
Vector(3, 0, 1, 4, 0, 1, 2, 2, 4, 0, 4, 0, 0, 3, 4, 0)
Vector(4, 2, 1, 4, 2, 3, 1, 0, 0, 4, 0, 3, 3, 0, 2, 2)
Vector(3, 3, 3, 2, 2, 0, 1, 3, 0, 0, 3, 1, 2, 0, 0)
Vector(4, 4, 4, 1, 0, 4, 4, 2, 4, 2, 4, 3, 2, 2, 1, 1)
Vector(1, 2, 2, 3, 1, 3, 0, 2, 4, 0, 4, 1, 0, 4, 4, 3)
```

```
Matriz 2:
Vector(0, 4, 0, 0, 1, 1, 2, 2, 0, 1, 4, 1, 3, 0, 0, 4)
Vector(4, 3, 4, 3, 2, 3, 1, 1, 0, 1, 3, 4, 0, 2, 2, 1)
Vector(0, 4, 4, 2, 1, 0, 2, 0, 3, 2, 4, 4, 4, 0, 1, 0)
Vector(2, 2, 2, 1, 1, 1, 4, 4, 0, 1, 2, 2, 3, 4, 3, 2)
Vector(1, 2, 4, 4, 2, 2, 4, 2, 3, 3, 0, 3, 2, 1, 0, 2)
Vector(3, 1, 2, 0, 2, 2, 4, 3, 3, 2, 4, 4, 0, 3, 1)
Vector(2, 2, 4, 1, 3, 4, 1, 2, 4, 2, 1, 4, 4, 2, 4, 3)
Vector(0, 0, 1, 1, 2, 2, 4, 0, 3, 1, 3, 4, 1, 0, 4)
Vector(1, 3, 1, 2, 0, 4, 2, 1, 2, 3, 4, 0, 0, 0, 0)
Vector(4, 3, 3, 4, 3, 0, 4, 0, 3, 4, 0, 1, 1, 1, 4)
Vector(3, 1, 3, 0, 0, 0, 2, 3, 2, 2, 0, 4, 0, 4, 3)
Vector(3, 0, 0, 4, 4, 0, 2, 2, 3, 2, 3, 2, 3, 0, 1)
Vector(2, 1, 0, 2, 1, 1, 0, 0, 2, 3, 4, 3, 3, 2, 3)
Vector(0, 4, 1, 4, 4, 0, 0, 4, 2, 4, 1, 0, 0, 0, 2)
Vector(2, 1, 3, 1, 2, 2, 1, 3, 1, 3, 3, 0, 0, 0, 1)
Vector(3, 4, 1, 1, 1, 1, 3, 4, 4, 0, 3, 2, 3, 0, 3)
```

Ahora, al tratar con matrices aún más grandes, si vemos cómo el algoritmo paralelo se desempeña aún mejor.

```
Resultado:
Vector(77, 75, 85, 51, 56, 60, 60, 83, 64, 72, 77, 90, 90, 51, 64, 91)
Vector(68, 84, 90, 62, 57, 59, 60, 79, 71, 79, 86, 90, 96, 45, 58, 81)
Vector(27, 35, 46, 23, 27, 31, 26, 34, 50, 43, 32, 56, 57, 21, 29, 43)
Vector(64, 52, 54, 42, 48, 26, 24, 54, 34, 44, 56, 40, 50, 36, 42, 58)
Vector(48, 70, 46, 52, 56, 42, 36, 45, 60, 54, 77, 62, 53, 28, 27, 53)
Vector(51, 68, 68, 51, 50, 34, 60, 69, 68, 71, 73, 79, 83, 28, 34, 65)
Vector(62, 72, 65, 50, 53, 56, 50, 55, 65, 64, 74, 89, 68, 49, 41, 70)
Vector(84, 93, 94, 72, 71, 73, 72, 75, 97, 88, 106, 112, 97, 59, 67, 82)
Vector(47, 63, 47, 52, 43, 32, 42, 48, 55, 60, 75, 51, 51, 27, 19, 53)
Vector(48, 82, 57, 62, 63, 31, 47, 67, 61, 72, 92, 76, 85, 38, 40, 78)
Vector(68, 60, 59, 55, 48, 39, 43, 64, 62, 72, 73, 67, 67, 36, 32, 77)
Vector(39, 61, 53, 34, 38, 45, 48, 54, 58, 52, 78, 48, 61, 22, 40, 57)
Vector(70, 68, 58, 59, 59, 37, 55, 74, 51, 66, 81, 76, 74, 52, 40, 72)
Vector(40, 63, 47, 55, 45, 38, 53, 46, 46, 52, 76, 62, 58, 33, 26, 41)
Vector(80, 95, 86, 69, 78, 66, 57, 78, 88, 90, 118, 98, 106, 42, 65, 83)
Vector(60, 77, 67, 54, 51, 49, 55, 65, 81, 74, 87, 71, 67, 31, 40, 59)
Unable to create a system terminal
Tiempo de ejecución de Strassen: 5.0067
Tiempo de ejecución de Strassen Paralelo: 0.8936
```

2)

```
Multiplicacion Strassen 16x16
Matriz 1:
Vector(0, 4, 0, 0, 1, 1, 2, 2, 0, 1, 4, 1, 3, 0, 0, 4)
Vector(4, 3, 4, 3, 2, 3, 1, 1, 0, 1, 3, 4, 0, 2, 2, 1)
Vector(0, 4, 4, 2, 1, 0, 2, 0, 3, 2, 4, 4, 4, 0, 1, 0)
Vector(2, 2, 2, 1, 1, 1, 4, 4, 0, 1, 2, 2, 3, 4, 3, 2)
Vector(1, 2, 4, 4, 2, 2, 4, 2, 3, 3, 0, 3, 2, 1, 0, 2)
Vector(3, 1, 2, 0, 2, 2, 2, 4, 3, 3, 2, 4, 4, 0, 3, 1)
Vector(2, 2, 4, 1, 3, 4, 1, 2, 4, 2, 1, 4, 4, 2, 4, 3)
Vector(0, 0, 0, 1, 1, 2, 2, 4, 0, 3, 1, 3, 4, 1, 0, 4)
Vector(1, 3, 1, 2, 0, 4, 2, 1, 2, 3, 4, 0, 0, 0, 0, 0)
Vector(4, 3, 3, 4, 3, 0, 0, 4, 0, 3, 4, 0, 1, 1, 1, 4)
Vector(3, 1, 3, 0, 0, 0, 2, 3, 2, 2, 0, 4, 0, 4, 3)
Vector(3, 0, 0, 4, 4, 0, 2, 2, 3, 2, 3, 2, 3, 0, 1)
Vector(2, 1, 0, 2, 1, 1, 0, 2, 3, 3, 4, 3, 3, 2, 3)
Vector(0, 4, 1, 4, 4, 0, 0, 4, 2, 4, 1, 0, 0, 0, 2)
Vector(2, 1, 3, 1, 2, 2, 2, 1, 3, 1, 3, 3, 0, 0, 0, 1)
Vector(3, 4, 1, 1, 1, 1, 3, 4, 4, 0, 3, 2, 3, 0, 3)
```

```
Matriz 2:
Vector(4, 4, 0, 4, 3, 2, 4, 2, 0, 2, 4, 0, 2, 0, 3, 2)
Vector(2, 4, 4, 3, 2, 1, 4, 4, 2, 2, 4, 1, 0, 1, 1, 2)
Vector(0, 0, 3, 0, 1, 0, 4, 3, 0, 0, 2, 0, 1, 0, 2, 2)
Vector(2, 4, 0, 2, 0, 0, 2, 0, 0, 4, 4, 2, 0, 0, 0, 2)
Vector(4, 4, 1, 0, 0, 2, 2, 1, 2, 0, 1, 3, 0, 4, 1, 2)
Vector(3, 0, 4, 2, 4, 0, 2, 0, 2, 2, 1, 1, 1, 3, 2)
Vector(4, 3, 1, 2, 2, 1, 4, 0, 1, 1, 0, 1, 4, 0, 4, 3)
Vector(3, 4, 2, 3, 3, 4, 0, 3, 0, 3, 2, 4, 2, 4, 2)
Vector(3, 0, 2, 1, 2, 1, 0, 0, 3, 3, 1, 2, 2, 1, 3, 2)
Vector(4, 0, 4, 3, 1, 3, 2, 2, 0, 4, 0, 1, 3, 4, 0, 1)
Vector(3, 2, 0, 0, 3, 1, 0, 2, 1, 3, 4, 2, 3, 0, 4, 3)
Vector(3, 0, 1, 4, 0, 1, 2, 2, 4, 0, 4, 0, 0, 3, 4, 0)
Vector(4, 2, 1, 4, 2, 3, 1, 0, 0, 4, 0, 3, 3, 0, 2, 2)
Vector(3, 3, 3, 2, 2, 0, 1, 3, 0, 0, 3, 1, 2, 0, 0)
Vector(4, 4, 4, 1, 0, 4, 2, 4, 2, 4, 3, 2, 2, 1, 1, 1)
Vector(1, 2, 2, 3, 1, 3, 0, 2, 4, 0, 1, 0, 4, 4, 3)
```

El cambio en el tiempo de ejecución del algoritmo secuencial fue drástico, hubo una fluctuación, sin embargo, el tiempo del algoritmo paralelo se mantiene en su rango.

```
Resultado:
Vector(64, 56, 43, 57, 45, 47, 41, 39, 42, 40, 61, 36, 41, 36, 66, 53)
Vector(92, 77, 65, 73, 54, 56, 80, 64, 51, 50, 99, 41, 41, 44, 76, 58)
Vector(85, 54, 57, 62, 44, 41, 66, 51, 45, 65, 72, 44, 50, 33, 71, 57)
Vector(103, 90, 68, 86, 59, 70, 81, 45, 64, 46, 75, 57, 67, 47, 81, 68)
Vector(95, 67, 68, 79, 50, 55, 78, 45, 50, 59, 71, 46, 54, 50, 78, 65)
Vector(116, 72, 70, 89, 61, 78, 82, 48, 63, 63, 81, 54, 73, 56, 94, 63)
Vector(124, 81, 96, 95, 66, 91, 84, 65, 80, 68, 97, 67, 63, 70, 99, 73)
Vector(79, 51, 49, 78, 44, 63, 44, 28, 48, 41, 53, 41, 51, 53, 71, 48)
Vector(67, 42, 51, 44, 52, 41, 42, 39, 21, 56, 55, 30, 44, 23, 54, 49)
Vector(93, 97, 60, 78, 56, 63, 81, 57, 51, 60, 101, 52, 54, 55, 75, 72)
Vector(78, 58, 57, 61, 45, 61, 60, 41, 47, 55, 67, 44, 55, 36, 65, 53)
Vector(104, 81, 40, 73, 49, 54, 55, 29, 49, 62, 65, 59, 55, 47, 68, 58)
Vector(92, 59, 55, 76, 43, 61, 43, 44, 58, 59, 73, 49, 43, 54, 65, 47)
Vector(69, 60, 44, 40, 33, 33, 42, 41, 44, 56, 70, 44, 27, 43, 50, 54)
Vector(69, 42, 42, 46, 43, 37, 50, 39, 39, 38, 62, 29, 39, 33, 70, 49)
Vector(99, 78, 75, 96, 58, 66, 68, 51, 65, 57, 74, 47, 53, 62, 75, 56)
Tiempo de ejecución de Strassen: 1.0364
Tiempo de ejecución de Strassen Paralelo: 0.914
```

- Matrices 32x32

```
def pruebas(): Unit = {
    println("Multiplicacion Strassen 32x32")
    println("Matriz 1: ")
    m9.foreach(row => println(row))
    println("Matriz 2: ")
    m10.foreach(row => println(row))
    println("Resultado: ")
    (mat1.multStrassen(m9, m10)).foreach(row => println(row))
    val comp9 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m9, m10)
    println("Tiempo de ejecución de Strassen: " + comp9(0))
    println("Tiempo de ejecución de Strassen Paralelo: " + comp9(1))

    println("Multiplicacion Strassen 32x32")
    println("Matriz 1: ")
    m10.foreach(row => println(row))
    println("Matriz 2: ")
    m9.foreach(row => println(row))
    println("Resultado: ")
    (mat1.multStrassen(m10, m9)).foreach(row => println(row))
    val comp10 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m10, m9)
    println("Tiempo de ejecución de Strassen: " + comp10(0))
    println("Tiempo de ejecución de Strassen Paralelo: " + comp10(1))
}
```

1)

```

Matrix 1:
Vector(1, 3, 0, 2, 1, 0, 3, 1, 2, 3, 3, 2, 1, 0, 0, 2, 1, 2, 1, 3, 3, 2, 1, 1, 0, 3, 1, 3, 0, 2, 3, 2, 1)
Vector(1, 2, 3, 2, 2, 0, 2, 0, 0, 3, 3, 2, 0, 0, 3, 3, 2, 1, 3, 0, 3, 2, 3, 0, 0, 1, 2, 2, 1, 3, 1, 1)
Vector(0, 2, 3, 0, 1, 1, 0, 2, 2, 3, 1, 1, 0, 2, 3, 1, 2, 1, 1, 2, 1, 2, 0, 2, 2, 3, 0, 2, 1, 3, 2, 3)
Vector(2, 1, 3, 3, 0, 2, 0, 2, 1, 2, 0, 0, 3, 1, 0, 2, 0, 1, 1, 3, 2, 3, 1, 3, 0, 0, 0, 2, 3, 1, 1, 1, 2)
Vector(3, 0, 2, 3, 0, 2, 2, 0, 1, 2, 2, 2, 3, 1, 1, 1, 1, 2, 0, 1, 1, 0, 3, 0, 3, 1, 1, 0, 1, 0, 3, 2)
Vector(2, 0, 3, 0, 0, 2, 1, 2, 2, 1, 2, 2, 0, 3, 1, 1, 3, 0, 2, 1, 3, 2, 1, 0, 0, 2, 1, 0, 3, 3, 3, 3)
Vector(1, 0, 1, 3, 0, 2, 2, 0, 3, 1, 0, 0, 1, 2, 1, 0, 2, 1, 1, 3, 1, 0, 1, 2, 3, 0, 2, 1, 0, 2, 1, 0)
Vector(1, 0, 3, 0, 2, 0, 3, 1, 1, 3, 1, 1, 0, 3, 2, 0, 3, 1, 1, 0, 2, 2, 0, 2, 3, 1, 3, 3, 3, 2, 1)
Vector(1, 0, 1, 2, 2, 3, 0, 3, 1, 3, 0, 1, 3, 2, 3, 0, 2, 0, 3, 2, 1, 0, 2, 2, 1, 2, 2, 0, 2, 2, 2)
Vector(2, 0, 3, 3, 0, 2, 2, 2, 2, 1, 0, 0, 3, 2, 1, 0, 2, 0, 3, 0, 2, 3, 1, 3, 3, 3, 3, 0, 2, 2, 2, 1, 1)
Vector(0, 0, 1, 2, 3, 2, 1, 0, 2, 2, 3, 0, 0, 1, 0, 2, 2, 3, 1, 0, 3, 2, 2, 0, 3, 2, 1, 0, 1, 2, 1, 0, 0)
Vector(0, 2, 0, 2, 3, 3, 3, 2, 0, 2, 3, 2, 2, 0, 2, 2, 1, 2, 2, 2, 1, 0, 0, 2, 1, 3, 0, 0, 0, 3, 0, 1, 1)
Vector(2, 2, 2, 2, 0, 2, 3, 3, 2, 0, 0, 0, 2, 2, 0, 3, 1, 2, 3, 2, 1, 3, 3, 0, 0, 0, 0, 2, 3, 0, 2, 1)
Vector(3, 3, 2, 0, 3, 3, 3, 1, 3, 3, 3, 0, 2, 3, 2, 1, 3, 2, 2, 3, 3, 2, 3, 1, 0, 0, 3, 0, 2, 3, 1)
Vector(0, 2, 2, 2, 1, 0, 1, 2, 3, 3, 1, 0, 3, 2, 3, 1, 0, 3, 1, 2, 0, 0, 3, 1, 0, 2, 0, 1, 3, 1, 0, 3)
Vector(3, 0, 0, 0, 1, 0, 0, 1, 0, 3, 3, 2, 3, 0, 0, 1, 0, 3, 1, 0, 1, 0, 2, 3, 3, 2, 0, 1, 1, 0, 1)
Vector(0, 2, 1, 0, 3, 1, 2, 2, 1, 3, 2, 3, 2, 1, 1, 2, 0, 2, 0, 3, 1, 2, 2, 1, 1, 0, 1, 0, 1, 1)
Vector(1, 0, 3, 0, 2, 1, 2, 2, 2, 0, 1, 0, 3, 0, 0, 3, 0, 0, 3, 2, 1, 1, 2, 3, 2, 1, 3, 3, 2, 3, 2)
Vector(2, 1, 1, 3, 2, 1, 1, 2, 0, 0, 1, 0, 3, 3, 1, 1, 3, 1, 2, 3, 2, 2, 2, 1, 1, 1, 0, 0, 0, 2, 2, 2)
Vector(2, 3, 3, 1, 0, 0, 2, 0, 1, 3, 3, 0, 3, 3, 3, 2, 2, 0, 1, 1, 2, 2, 3, 0, 2, 1, 2, 0, 0, 2, 3, 2)
Vector(3, 3, 3, 1, 1, 3, 2, 3, 2, 2, 3, 3, 1, 2, 2, 1, 3, 1, 1, 2, 2, 0, 1, 1, 2, 3, 3, 3, 2, 3, 0, 3)
Vector(2, 1, 2, 2, 3, 3, 2, 1, 0, 1, 0, 1, 3, 0, 3, 1, 1, 1, 0, 1, 0, 1, 1, 2, 0, 0, 3, 1, 3, 0, 1)
Vector(2, 1, 2, 3, 0, 0, 2, 2, 2, 0, 3, 1, 3, 3, 3, 2, 3, 2, 1, 3, 3, 2, 3, 2, 3, 0, 2, 0, 1, 2, 1, 1)
Vector(1, 2, 2, 1, 0, 1, 0, 3, 3, 2, 1, 0, 3, 2, 0, 2, 0, 1, 0, 3, 3, 0, 2, 2, 3, 0, 0, 2, 2, 2, 2, 2)
Vector(0, 3, 0, 1, 3, 3, 3, 3, 1, 0, 0, 1, 0, 3, 3, 3, 2, 0, 1, 0, 1, 0, 2, 0, 1, 2, 1, 3, 1, 3, 3)
Vector(1, 1, 2, 0, 0, 3, 1, 0, 2, 0, 1, 0, 2, 3, 2, 2, 2, 1, 0, 1, 3, 1, 1, 1, 1, 3, 3, 2, 2, 3, 1, 1)
Vector(3, 1, 1, 3, 2, 0, 3, 3, 1, 1, 1, 2, 2, 3, 0, 1, 1, 2, 1, 3, 1, 1, 2, 3, 3, 2, 2, 1, 0, 3, 0, 1)
Vector(0, 1, 1, 3, 2, 2, 3, 2, 0, 3, 2, 0, 3, 2, 0, 3, 1, 0, 1, 3, 2, 2, 2, 1, 3, 1, 1, 0, 0, 1, 3, 2)
Vector(2, 2, 2, 1, 0, 0, 2, 3, 1, 1, 1, 1, 3, 2, 2, 1, 2, 3, 0, 2, 1, 1, 1, 1, 0, 3, 1, 2, 2, 1, 1, 0)
Vector(2, 2, 3, 1, 1, 2, 2, 2, 3, 1, 1, 1, 0, 1, 2, 2, 1, 0, 2, 1, 2, 1, 3, 1, 3, 2, 2, 1, 1, 1, 0, 1)
Vector(2, 1, 2, 1, 2, 0, 3, 0, 2, 0, 2, 1, 1, 3, 0, 3, 3, 1, 2, 3, 1, 2, 0, 1, 3, 3, 3, 3, 3, 1, 0)
Vector(0, 2, 3, 2, 1, 3, 0, 1, 1, 2, 2, 0, 0, 1, 1, 1, 2, 0, 0, 3, 0, 3, 2, 1, 1, 2, 2, 0, 2, 2, 3, 1)

```

Matrix 2:

```

Vector(3, 3, 1, 1, 1, 1, 0, 2, 2, 3, 0, 0, 2, 3, 0, 1, 3, 3, 1, 2, 2, 3, 0, 3, 2, 1, 0, 0, 2, 1, 1, 3, 3)
Vector(3, 2, 3, 2, 2, 1, 3, 0, 2, 1, 0, 3, 0, 0, 2, 3, 3, 1, 0, 3, 2, 0, 3, 0, 2, 3, 1, 2, 3, 0, 0, 2)
Vector(1, 3, 1, 0, 1, 0, 0, 1, 2, 0, 0, 2, 0, 1, 1, 3, 0, 2, 3, 0, 0, 2, 2, 3, 0, 0, 1, 2, 1, 3, 2, 2)
Vector(3, 2, 0, 3, 3, 3, 0, 1, 3, 2, 0, 2, 0, 1, 3, 0, 2, 2, 2, 0, 1, 0, 2, 0, 0, 0, 3, 3, 0, 3, 1, 0)
Vector(3, 1, 2, 0, 1, 1, 1, 2, 3, 1, 3, 2, 2, 1, 0, 2, 3, 1, 1, 3, 0, 3, 1, 3, 2, 1, 1, 0, 0, 0, 2, 0)
Vector(0, 0, 0, 2, 1, 1, 2, 3, 1, 1, 2, 3, 1, 0, 0, 1, 1, 3, 0, 1, 2, 2, 3, 1, 1, 2, 1, 1, 0, 0, 0, 2)
Vector(1, 1, 2, 2, 1, 0, 2, 1, 1, 2, 0, 0, 0, 3, 2, 1, 3, 2, 2, 2, 1, 3, 3, 0, 2, 0, 3, 1, 3, 1, 3, 1)
Vector(2, 0, 1, 2, 1, 1, 0, 2, 3, 1, 0, 0, 2, 0, 0, 1, 3, 0, 0, 2, 1, 3, 2, 0, 1, 3, 0, 1, 1, 1, 3, 1)
Vector(3, 3, 0, 3, 3, 2, 2, 0, 1, 2, 0, 1, 0, 2, 0, 3, 0, 1, 0, 2, 0, 0, 2, 1, 3, 0, 0, 3, 2, 1, 0, 2)
Vector(3, 1, 1, 0, 3, 2, 2, 2, 0, 3, 1, 2, 2, 2, 3, 0, 2, 1, 3, 0, 1, 1, 3, 3, 0, 0, 1, 1, 2, 0, 1, 0)
Vector(0, 0, 2, 2, 1, 1, 0, 1, 2, 1, 3, 1, 3, 2, 1, 1, 0, 3, 3, 0, 2, 3, 3, 3, 2, 0, 3, 2, 1, 0, 1, 1)
Vector(0, 2, 1, 1, 0, 0, 2, 0, 3, 3, 0, 3, 0, 0, 1, 2, 3, 2, 2, 0, 3, 3, 1, 0, 0, 3, 2, 3, 3, 2)
Vector(1, 1, 1, 0, 1, 1, 2, 2, 3, 3, 0, 1, 1, 3, 3, 2, 2, 1, 3, 1, 2, 0, 2, 2, 3, 3, 3, 3, 2, 1, 3, 2)
Vector(1, 2, 3, 2, 1, 3, 0, 2, 1, 3, 3, 3, 1, 0, 2, 1, 0, 0, 1, 2, 2, 3, 1, 3, 3, 2, 1, 2, 3, 0)
Vector(0, 0, 0, 3, 2, 2, 2, 3, 1, 0, 2, 1, 3, 1, 3, 2, 3, 1, 1, 1, 3, 1, 2, 3, 0, 0, 1, 2, 2, 3, 3)
Vector(2, 1, 1, 1, 1, 2, 0, 3, 1, 2, 1, 0, 1, 2, 0, 2, 2, 2, 1, 3, 2, 2, 1, 0, 0, 1, 0, 2, 1, 0, 2, 2)
Vector(0, 0, 0, 2, 0, 2, 2, 0, 3, 0, 1, 3, 2, 1, 1, 0, 3, 2, 3, 2, 0, 2, 1, 0, 0, 1, 2, 0, 3, 0, 1, 1)
Vector(0, 0, 2, 1, 3, 3, 2, 2, 3, 1, 0, 2, 2, 1, 0, 3, 2, 0, 1, 0, 3, 1, 0, 0, 3, 1, 2, 2, 0, 0, 3, 3)
Vector(3, 1, 3, 1, 2, 2, 3, 3, 3, 2, 3, 0, 1, 1, 2, 2, 2, 0, 1, 3, 0, 1, 3, 3, 2, 1, 0, 1, 3, 1, 1)
Vector(3, 3, 1, 3, 2, 3, 1, 3, 3, 1, 3, 1, 0, 1, 1, 1, 2, 2, 0, 0, 2, 2, 3, 1, 1, 3, 1, 0, 0, 3, 0, 3)
Vector(2, 0, 3, 2, 3, 1, 2, 2, 2, 1, 2, 3, 2, 0, 1, 1, 3, 0, 3, 2, 2, 0, 3, 2, 3, 1, 2, 0, 1, 3, 0)
Vector(0, 1, 2, 0, 0, 1, 3, 1, 2, 3, 0, 3, 0, 0, 0, 3, 1, 0, 1, 0, 2, 2, 0, 3, 3, 1, 0, 1, 2, 3, 0, 3)
Vector(1, 1, 3, 3, 3, 1, 0, 0, 3, 2, 3, 2, 2, 2, 1, 3, 1, 0, 1, 0, 3, 3, 1, 0, 2, 2, 2, 1, 3, 3, 0, 1)
Vector(0, 3, 3, 1, 0, 0, 1, 2, 1, 3, 1, 2, 1, 1, 3, 1, 2, 2, 1, 2, 3, 1, 3, 0, 2, 1, 0, 1, 0, 3, 2, 2)
Vector(3, 2, 3, 3, 3, 0, 3, 3, 0, 3, 1, 0, 0, 0, 2, 1, 1, 0, 0, 3, 2, 1, 1, 3, 3, 0, 2, 0, 3, 0, 3, 3)
Vector(2, 0, 3, 3, 0, 3, 3, 3, 3, 1, 1, 2, 3, 2, 2, 1, 0, 1, 2, 1, 0, 2, 1, 2, 1, 0, 1, 0, 1, 0, 1, 1)
Vector(1, 0, 1, 1, 0, 0, 1, 2, 0, 2, 1, 2, 1, 2, 3, 0, 1, 3, 1, 2, 0, 1, 0, 2, 1, 2, 1, 2, 1, 1, 1, 1)
Vector(1, 3, 1, 0, 1, 2, 2, 3, 2, 1, 0, 0, 3, 1, 2, 1, 2, 0, 3, 2, 1, 1, 1, 1, 3, 3, 3, 1, 3, 3, 3)
Vector(3, 0, 3, 0, 0, 2, 1, 1, 0, 0, 0, 1, 3, 1, 0, 1, 3, 0, 0, 1, 3, 3, 2, 2, 1, 3, 3, 3, 0, 0, 0, 0, 1)
Vector(1, 1, 3, 1, 0, 0, 1, 1, 2, 3, 0, 0, 1, 1, 2, 3, 1, 0, 2, 0, 3, 0, 3, 0, 2, 0, 3, 2, 3, 2, 3, 0, 0)
Vector(2, 0, 3, 1, 1, 1, 1, 1, 3, 2, 3, 3, 1, 0, 2, 0, 2, 2, 1, 3, 1, 2, 2, 2, 2, 3, 3, 3, 0, 1, 1, 1)
Vector(1, 2, 0, 1, 3, 3, 2, 2, 0, 0, 3, 2, 2, 0, 2, 0, 2, 3, 2, 1, 3, 2, 0, 1, 3, 3, 2, 0, 2, 2, 2, 1, 1)

```

```

Resultado:
Vector(85, 64, 90, 78, 77, 79, 82, 82, 184, 58, 52, 82, 88, 68, 70, 68, 94, 83, 70, 75, 90, 71, 92, 73, 77, 86, 80, 94, 69, 73, 81, 74)
Vector(74, 53, 87, 68, 72, 66, 71, 77, 95, 91, 49, 89, 91, 68, 71, 68, 95, 87, 76, 63, 94, 73, 83, 81, 72, 71, 76, 86, 77, 76, 81, 66)
Vector(73, 64, 81, 72, 67, 75, 79, 88, 81, 80, 50, 79, 80, 54, 72, 66, 87, 61, 61, 66, 84, 60, 93, 76, 75, 69, 70, 78, 72, 81, 74)
Vector(72, 72, 66, 55, 65, 67, 58, 86, 83, 82, 36, 72, 68, 59, 69, 56, 85, 68, 59, 59, 80, 57, 78, 71, 65, 76, 71, 79, 59, 77, 78, 74)
Vector(68, 54, 66, 71, 74, 61, 56, 71, 80, 83, 45, 70, 72, 63, 67, 55, 77, 70, 65, 53, 77, 72, 62, 77, 64, 58, 84, 78, 62, 64, 73, 66)
Vector(70, 51, 83, 70, 57, 70, 65, 77, 85, 77, 51, 85, 93, 63, 52, 61, 85, 77, 58, 65, 81, 82, 79, 89, 63, 80, 82, 88, 65, 72, 74, 59)
Vector(61, 57, 55, 70, 60, 52, 66, 61, 70, 33, 55, 47, 51, 61, 51, 59, 51, 43, 44, 62, 43, 57, 55, 60, 57, 59, 58, 49, 67, 55, 55)
Vector(73, 60, 93, 62, 58, 61, 76, 86, 72, 89, 40, 80, 96, 66, 72, 58, 92, 72, 69, 69, 85, 75, 82, 93, 75, 81, 81, 85, 73, 71, 94, 69)
Vector(77, 64, 75, 83, 76, 79, 61, 102, 83, 87, 55, 71, 84, 69, 70, 65, 80, 73, 56, 67, 91, 66, 86, 77, 86, 79, 78, 86, 57, 71, 89, 75)
Vector(94, 63, 93, 90, 77, 79, 67, 87, 83, 103, 47, 79, 80, 73, 75, 59, 100, 68, 46, 61, 96, 67, 89, 72, 81, 79, 78, 83, 64, 67, 75, 72)
Vector(68, 38, 78, 65, 68, 64, 63, 68, 74, 78, 38, 71, 77, 60, 48, 54, 67, 56, 50, 54, 80, 53, 68, 68, 72, 55, 71, 69, 51, 41, 69, 53)
Vector(75, 42, 79, 72, 62, 71, 73, 84, 87, 82, 47, 74, 86, 62, 66, 68, 87, 68, 58, 60, 99, 64, 85, 67, 73, 75, 74, 77, 47, 60, 78, 63)
Vector(89, 71, 83, 78, 77, 78, 64, 75, 93, 80, 46, 88, 81, 63, 63, 59, 107, 74, 52, 64, 98, 79, 85, 69, 76, 87, 68, 76, 64, 79, 74, 72)
Vector(89, 86, 109, 97, 96, 86, 91, 111, 124, 115, 59, 108, 120, 76, 85, 94, 124, 98, 83, 88, 122, 101, 111, 104, 99, 99, 100, 106, 95, 106, 119, 105)
Vector(77, 59, 71, 71, 81, 84, 63, 77, 79, 74, 42, 72, 73, 66, 70, 69, 90, 54, 58, 42, 90, 61, 80, 62, 74, 71, 71, 86, 60, 68, 70, 67)
Vector(51, 48, 67, 55, 35, 46, 55, 68, 59, 80, 36, 56, 59, 55, 47, 41, 57, 57, 39, 49, 59, 49, 61, 78, 61, 46, 59, 48, 52, 51, 58, 55)
Vector(71, 48, 82, 67, 64, 57, 69, 70, 80, 89, 44, 83, 83, 64, 64, 54, 84, 70, 59, 63, 78, 68, 79, 72, 70, 72, 72, 69, 69, 68, 79, 54)
Vector(70, 52, 69, 63, 63, 62, 69, 80, 74, 73, 42, 62, 71, 45, 59, 65, 75, 50, 49, 54, 82, 56, 76, 72, 74, 63, 65, 73, 56, 61, 73, 75)
Vector(72, 56, 77, 74, 66, 73, 59, 79, 97, 85, 49, 82, 72, 63, 64, 51, 93, 73, 54, 55, 88, 62, 77, 68, 70, 82, 77, 73, 59, 71, 77, 61)
Vector(78, 64, 88, 87, 77, 72, 64, 80, 98, 91, 60, 99, 87, 61, 75, 56, 102, 93, 66, 72, 86, 85, 91, 85, 69, 72, 88, 90, 83, 78, 87, 72)
Vector(96, 82, 98, 100, 83, 88, 93, 110, 110, 105, 55, 95, 115, 86, 89, 92, 112, 95, 88, 87, 108, 97, 104, 100, 95, 102, 102, 103, 98, 97, 103, 94)
Vector(58, 56, 56, 53, 54, 50, 59, 73, 73, 74, 26, 54, 72, 52, 67, 63, 78, 51, 63, 48, 86, 47, 68, 58, 71, 66, 69, 66, 65, 68, 76, 67)
Vector(81, 75, 97, 99, 84, 80, 76, 98, 107, 104, 55, 92, 94, 79, 79, 74, 111, 93, 68, 77, 106, 88, 96, 92, 90, 86, 89, 99, 82, 86, 106, 94)
Vector(71, 56, 85, 72, 64, 67, 77, 73, 69, 82, 43, 82, 82, 60, 63, 58, 78, 65, 52, 66, 77, 58, 85, 87, 68, 59, 69, 82, 66, 58, 73, 60)
Vector(76, 48, 69, 73, 64, 75, 76, 79, 88, 73, 45, 79, 86, 56, 70, 71, 100, 68, 56, 78, 97, 64, 79, 57, 75, 87, 71, 93, 62, 58, 78, 72)
Vector(65, 53, 83, 76, 59, 65, 72, 90, 72, 79, 40, 66, 83, 70, 67, 72, 74, 56, 54, 69, 84, 59, 69, 78, 81, 80, 76, 80, 66, 65, 79, 70)
Vector(93, 74, 99, 84, 70, 78, 76, 97, 107, 114, 44, 86, 90, 75, 71, 60, 108, 81, 59, 80, 90, 88, 94, 86, 84, 90, 83, 87, 68, 78, 102, 87)
Vector(91, 65, 84, 87, 84, 74, 71, 88, 91, 102, 56, 82, 75, 71, 75, 64, 88, 80, 60, 71, 93, 68, 90, 82, 76, 84, 83, 94, 66, 72, 85, 68)
Vector(65, 52, 74, 67, 55, 70, 62, 77, 91, 74, 30, 64, 80, 66, 58, 67, 91, 55, 58, 58, 75, 68, 74, 61, 70, 79, 65, 76, 59, 62, 88, 74)
Vector(87, 68, 79, 89, 79, 66, 69, 83, 85, 91, 41, 76, 74, 67, 69, 73, 86, 73, 54, 70, 86, 68, 76, 77, 76, 68, 66, 78, 73, 79, 77, 73)
Vector(91, 65, 104, 80, 64, 79, 75, 92, 100, 101, 46, 84, 94, 88, 63, 75, 97, 85, 66, 75, 94, 81, 83, 86, 85, 101, 85, 92, 71, 81, 83, 73)
Vector(67, 49, 72, 68, 54, 62, 58, 68, 78, 73, 49, 81, 67, 50, 68, 53, 53, 74, 56, 88, 67, 58, 65, 72, 74, 57, 68, 53, 60)

```

Tiempo de ejecución de Strassen: 12.2208

Tiempo de ejecución de Strassen Paralelo: 7.7061

Ahora que si se tratan con matrices más grandes que las anteriores, se puede apreciar la diferencia en los tiempos de ejecución de cada algoritmo, siendo más eficiente la versión paralela.

2)

```

Matriz 1:
Vector(3, 3, 1, 1, 1, 0, 2, 2, 3, 0, 0, 2, 3, 0, 1, 3, 3, 1, 2, 2, 3, 0, 3, 2, 1, 0, 0, 2, 1, 1, 3)
Vector(3, 2, 3, 2, 2, 1, 3, 0, 2, 1, 0, 3, 0, 0, 2, 3, 3, 1, 0, 3, 2, 0, 3, 0, 2, 3, 1, 2, 3, 0, 0, 2)
Vector(1, 3, 1, 0, 1, 0, 0, 1, 2, 0, 0, 2, 0, 1, 1, 3, 0, 2, 3, 0, 0, 2, 2, 3, 0, 0, 1, 2, 1, 3, 2, 2)
Vector(3, 2, 0, 3, 3, 3, 0, 1, 3, 2, 0, 2, 0, 1, 3, 0, 2, 2, 2, 0, 1, 0, 2, 0, 0, 0, 3, 3, 0, 3, 1, 0)
Vector(3, 1, 2, 0, 1, 1, 1, 2, 3, 1, 3, 2, 2, 1, 0, 2, 3, 1, 1, 3, 0, 3, 1, 3, 2, 1, 1, 0, 0, 2, 0)
Vector(0, 0, 2, 1, 1, 2, 3, 1, 1, 1, 2, 3, 1, 2, 3, 0, 0, 1, 1, 3, 0, 1, 2, 2, 2, 3, 1, 1, 2, 1, 1)
Vector(1, 1, 2, 2, 1, 0, 2, 1, 1, 2, 0, 0, 3, 2, 1, 3, 2, 2, 2, 1, 3, 3, 0, 2, 0, 3, 1, 3, 3, 1)
Vector(2, 0, 1, 2, 1, 1, 0, 2, 3, 1, 0, 0, 2, 0, 0, 1, 3, 0, 0, 2, 1, 3, 2, 0, 1, 3, 0, 1, 1, 3, 1)
Vector(3, 3, 0, 3, 3, 2, 2, 0, 1, 2, 0, 1, 0, 2, 0, 3, 0, 1, 0, 2, 0, 0, 2, 1, 3, 0, 0, 3, 2, 1, 0, 2)
Vector(3, 1, 1, 0, 3, 2, 2, 2, 0, 3, 1, 2, 2, 2, 3, 0, 2, 1, 3, 0, 1, 1, 3, 3, 0, 0, 1, 1, 2, 0, 1, 0)
Vector(0, 0, 2, 2, 1, 1, 0, 1, 2, 1, 3, 1, 2, 1, 1, 0, 3, 3, 0, 2, 3, 3, 3, 2, 0, 3, 2, 1, 0, 1, 1)
Vector(0, 2, 1, 1, 0, 0, 2, 0, 3, 3, 0, 3, 0, 0, 1, 2, 3, 2, 2, 0, 3, 3, 1, 0, 0, 3, 2, 3, 3, 2)
Vector(1, 1, 0, 1, 1, 2, 2, 3, 0, 1, 1, 3, 3, 2, 2, 1, 3, 1, 2, 0, 2, 2, 3, 3, 3, 2, 1, 3, 2)
Vector(1, 2, 3, 2, 1, 3, 0, 3, 0, 2, 1, 3, 3, 3, 1, 0, 2, 1, 0, 0, 1, 2, 2, 3, 1, 3, 3, 2, 1, 2, 3, 0)
Vector(0, 0, 3, 2, 2, 2, 3, 1, 0, 2, 1, 3, 1, 2, 3, 1, 1, 1, 3, 1, 1, 2, 3, 0, 0, 1, 2, 2, 3, 3)
Vector(2, 1, 1, 1, 2, 0, 3, 1, 2, 1, 0, 1, 2, 0, 2, 2, 2, 1, 3, 2, 2, 1, 0, 0, 1, 0, 2, 1, 0, 2, 2)
Vector(0, 0, 2, 0, 2, 2, 0, 3, 0, 1, 2, 1, 0, 0, 2, 0, 3, 2, 0, 1, 0, 0, 1, 2, 0, 0, 3, 0, 1, 0, 1)
Vector(0, 0, 2, 1, 3, 3, 2, 2, 3, 1, 0, 2, 2, 1, 0, 0, 3, 1, 0, 1, 0, 0, 3, 1, 2, 2, 0, 0, 3, 3)
Vector(3, 1, 3, 1, 2, 2, 3, 3, 3, 2, 3, 0, 1, 2, 2, 2, 0, 1, 3, 2, 1, 1, 3, 1, 2, 1, 0, 1, 3, 1, 1)
Vector(3, 3, 1, 3, 2, 3, 1, 3, 1, 3, 0, 1, 1, 2, 2, 0, 0, 2, 2, 3, 1, 1, 3, 1, 0, 1, 0, 3, 0, 3)
Vector(2, 0, 3, 2, 3, 1, 2, 2, 2, 1, 1, 2, 3, 2, 0, 1, 1, 3, 0, 3, 2, 2, 0, 3, 1, 2, 0, 1, 3, 1, 0)
Vector(0, 1, 2, 0, 0, 1, 3, 1, 2, 3, 0, 3, 3, 0, 0, 0, 3, 1, 0, 2, 2, 0, 3, 3, 1, 0, 1, 2, 3, 0, 3)
Vector(1, 1, 3, 3, 3, 1, 0, 0, 3, 3, 2, 3, 2, 2, 2, 1, 3, 1, 1, 0, 3, 3, 1, 0, 2, 2, 2, 1, 3, 3, 0, 1)
Vector(0, 3, 3, 1, 0, 0, 1, 2, 1, 3, 1, 2, 1, 1, 3, 1, 2, 2, 1, 2, 3, 1, 3, 0, 2, 1, 0, 1, 0, 3, 2, 2)
Vector(3, 2, 3, 3, 0, 3, 3, 0, 1, 0, 0, 0, 2, 1, 1, 0, 0, 3, 2, 1, 1, 3, 3, 0, 2, 0, 3, 0, 3, 3)
Vector(2, 0, 3, 3, 0, 3, 3, 3, 1, 1, 2, 3, 2, 2, 1, 0, 1, 2, 1, 0, 0, 2, 1, 2, 1, 0, 1, 0, 1, 1, 1)
Vector(1, 0, 1, 1, 0, 0, 1, 2, 0, 2, 1, 2, 1, 2, 3, 0, 1, 3, 1, 2, 0, 1, 0, 2, 1, 2, 1, 2, 1, 1, 1)
Vector(1, 3, 1, 0, 1, 2, 2, 3, 2, 1, 0, 0, 3, 1, 2, 1, 2, 0, 3, 2, 1, 1, 3, 3, 1, 3, 3, 1, 3, 3, 3)
Vector(3, 0, 3, 0, 0, 2, 1, 1, 0, 0, 0, 1, 3, 1, 0, 1, 3, 0, 0, 1, 3, 3, 2, 2, 1, 3, 3, 0, 0, 0, 1)
Vector(1, 1, 3, 1, 0, 0, 1, 1, 2, 3, 0, 0, 1, 1, 2, 3, 1, 0, 2, 0, 3, 0, 3, 0, 2, 3, 2, 3, 2, 0, 0)
Vector(2, 0, 3, 1, 1, 1, 1, 3, 2, 3, 3, 1, 0, 2, 0, 2, 2, 1, 3, 1, 2, 2, 2, 3, 3, 0, 1, 1, 1, 1, 1)
Vector(1, 2, 0, 1, 3, 3, 2, 2, 0, 0, 3, 2, 2, 0, 2, 3, 2, 1, 3, 2, 0, 2, 0, 1, 3, 3, 2, 2, 2, 1, 1)

```

master* ↵ ⌘ 1 △ 0 ⌘ 0 ⌘ Live Share ⌘ Java: Ready

```

Matriz 2:
Vector(1, 3, 0, 2, 1, 0, 3, 1, 2, 3, 3, 2, 1, 0, 0, 2, 1, 2, 1, 3, 3, 2, 1, 1, 0, 3, 1, 3, 0, 2, 3, 2)
Vector(1, 2, 3, 2, 2, 0, 2, 0, 0, 3, 3, 2, 0, 0, 3, 3, 2, 1, 3, 0, 3, 2, 3, 0, 0, 1, 2, 2, 1, 3, 1, 1)
Vector(0, 2, 3, 0, 1, 1, 0, 2, 2, 3, 1, 1, 0, 2, 3, 1, 2, 1, 1, 2, 1, 0, 2, 2, 3, 0, 2, 1, 3, 2, 3)
Vector(2, 1, 3, 3, 0, 2, 0, 2, 1, 2, 0, 0, 3, 1, 0, 2, 0, 1, 1, 3, 2, 3, 1, 3, 0, 0, 2, 3, 1, 1, 1, 2)
Vector(3, 0, 2, 3, 0, 2, 0, 1, 2, 2, 1, 2, 2, 3, 1, 1, 1, 2, 0, 1, 0, 3, 0, 3, 1, 1, 0, 1, 0, 3, 2)
Vector(2, 0, 3, 0, 0, 2, 1, 2, 2, 1, 2, 2, 0, 3, 1, 1, 3, 0, 2, 1, 3, 2, 1, 0, 0, 2, 1, 0, 3, 3, 3)
Vector(1, 0, 1, 3, 0, 2, 2, 0, 3, 1, 0, 0, 1, 2, 1, 0, 2, 1, 1, 3, 1, 0, 1, 2, 3, 0, 2, 1, 0, 2, 1, 0)
Vector(1, 0, 3, 0, 2, 0, 3, 1, 1, 3, 1, 0, 3, 2, 0, 3, 1, 1, 0, 2, 2, 0, 2, 3, 1, 3, 3, 3, 2, 1)
Vector(1, 0, 1, 2, 2, 3, 0, 3, 3, 1, 0, 1, 3, 2, 3, 0, 2, 0, 3, 2, 1, 0, 2, 2, 2, 1, 2, 2, 0, 2, 2)
Vector(2, 2, 0, 3, 3, 0, 2, 2, 2, 2, 1, 0, 0, 3, 2, 1, 0, 2, 0, 3, 0, 2, 3, 1, 3, 3, 0, 2, 2, 1, 1)
Vector(0, 0, 1, 2, 3, 2, 1, 0, 2, 2, 3, 0, 0, 1, 0, 2, 2, 3, 1, 0, 3, 2, 2, 0, 3, 2, 0, 1, 2, 1, 0)
Vector(0, 2, 0, 2, 3, 3, 2, 0, 2, 3, 2, 2, 0, 2, 2, 1, 0, 0, 2, 1, 3, 0, 0, 0, 3, 0, 1, 0, 1, 1)
Vector(2, 2, 2, 2, 0, 2, 3, 3, 2, 0, 0, 2, 2, 0, 3, 1, 2, 3, 2, 1, 3, 3, 0, 0, 2, 3, 0, 2, 2, 1)
Vector(3, 3, 3, 2, 0, 3, 3, 3, 1, 3, 3, 3, 0, 2, 3, 2, 1, 3, 2, 3, 3, 2, 3, 1, 0, 0, 3, 0, 2, 3, 1)
Vector(0, 2, 2, 2, 1, 0, 1, 2, 3, 3, 1, 0, 3, 2, 3, 1, 0, 3, 1, 2, 0, 0, 3, 1, 0, 2, 0, 1, 3, 1, 0)
Vector(3, 0, 0, 0, 1, 0, 0, 1, 0, 3, 3, 2, 3, 0, 0, 1, 0, 3, 1, 0, 1, 0, 2, 3, 3, 2, 0, 1, 1, 0, 1)
Vector(0, 2, 1, 0, 3, 1, 2, 2, 1, 3, 2, 3, 2, 2, 1, 1, 2, 0, 2, 0, 2, 0, 3, 1, 2, 2, 1, 1, 0, 1, 1)
Vector(1, 0, 3, 0, 2, 2, 1, 2, 2, 0, 1, 0, 0, 3, 0, 0, 3, 0, 2, 0, 3, 1, 2, 0, 0, 3, 2, 1, 1, 2, 3, 2)
Vector(2, 1, 1, 3, 2, 1, 1, 2, 0, 0, 1, 0, 3, 3, 1, 1, 3, 1, 2, 3, 2, 2, 2, 1, 1, 1, 0, 0, 0, 2, 2, 2)
Vector(2, 3, 3, 1, 0, 0, 2, 0, 1, 3, 3, 0, 3, 2, 2, 0, 1, 1, 2, 2, 3, 0, 2, 1, 2, 0, 0, 2, 3, 3)
Vector(3, 3, 3, 1, 1, 2, 3, 2, 2, 3, 3, 1, 2, 2, 1, 3, 1, 1, 0, 2, 2, 0, 3, 1, 1, 2, 3, 3, 2, 1)
Vector(2, 1, 2, 2, 3, 3, 2, 1, 0, 1, 0, 3, 3, 3, 2, 0, 1, 1, 1, 0, 1, 1, 2, 0, 0, 3, 1, 3, 0, 1)
Vector(2, 1, 2, 3, 0, 0, 2, 2, 2, 0, 3, 1, 3, 3, 3, 2, 2, 1, 3, 2, 3, 3, 2, 0, 2, 0, 1, 2, 3, 3)
Vector(1, 2, 2, 1, 0, 1, 1, 0, 3, 3, 2, 1, 0, 3, 2, 0, 2, 0, 1, 0, 3, 3, 2, 0, 2, 2, 1, 2, 1, 1)
Vector(0, 3, 0, 1, 3, 3, 3, 3, 1, 0, 0, 1, 0, 3, 3, 2, 0, 1, 0, 1, 0, 2, 0, 1, 2, 1, 3, 1, 3, 3)
Vector(1, 0, 1, 2, 0, 3, 1, 0, 2, 0, 1, 0, 2, 3, 2, 2, 1, 0, 1, 3, 1, 1, 1, 0, 3, 2, 2, 2, 3, 1, 1)
Vector(3, 1, 1, 3, 2, 0, 3, 1, 1, 1, 2, 2, 3, 0, 1, 1, 2, 1, 3, 1, 2, 3, 3, 2, 2, 1, 0, 3, 0, 0)
Vector(0, 1, 1, 3, 2, 2, 3, 2, 2, 0, 3, 2, 0, 3, 1, 0, 1, 3, 2, 2, 2, 1, 3, 1, 1, 0, 0, 1, 3, 2)
Vector(2, 2, 2, 1, 0, 0, 2, 3, 1, 1, 1, 3, 2, 2, 1, 2, 3, 0, 2, 1, 1, 1, 0, 3, 1, 2, 2, 1, 1, 0)
Vector(2, 2, 3, 1, 2, 2, 2, 3, 1, 1, 1, 0, 1, 2, 2, 1, 0, 2, 1, 2, 1, 3, 1, 3, 2, 2, 1, 1, 0, 1)
Vector(2, 1, 2, 1, 2, 0, 3, 0, 2, 1, 1, 3, 3, 2, 1, 2, 3, 1, 2, 0, 1, 3, 3, 3, 3, 3, 1, 0, 0, 1)
Vector(0, 2, 3, 2, 1, 3, 0, 1, 1, 2, 0, 0, 3, 2, 0, 1, 1, 1, 0, 2, 0, 3, 2, 1, 1, 2, 0, 2, 2, 3, 1)

```

master* ↵ ⌘ 1 △ 0 ⌘ 0 ⌘ Live Share ⌘ Java: Ready

```

Resultado:
Vector(72, 85, 98, 75, 73, 71, 82, 82, 76, 97, 83, 65, 44, 83, 96, 66, 81, 63, 51, 84, 81, 83, 74, 62, 76, 86, 69, 76, 72, 98, 90, 77)
Vector(64, 79, 85, 86, 58, 73, 78, 83, 83, 84, 96, 67, 80, 93, 87, 78, 82, 67, 53, 100, 79, 69, 71, 70, 88, 97, 76, 60, 67, 84, 84, 83)
Vector(56, 50, 71, 70, 52, 53, 63, 53, 59, 60, 73, 45, 53, 69, 66, 57, 60, 45, 54, 70, 64, 66, 53, 50, 71, 70, 48, 43, 50, 71, 60, 53)
Vector(67, 57, 79, 93, 62, 59, 82, 82, 77, 88, 80, 53, 70, 80, 64, 74, 56, 64, 54, 94, 80, 70, 76, 55, 75, 76, 63, 56, 58, 68, 80, 74)
Vector(61, 76, 75, 74, 67, 90, 85, 86, 82, 84, 56, 51, 84, 88, 72, 81, 74, 50, 94, 82, 68, 64, 65, 81, 87, 66, 64, 75, 85, 84, 76)
Vector(71, 68, 77, 76, 53, 62, 76, 74, 80, 67, 68, 48, 56, 93, 67, 53, 78, 54, 53, 80, 76, 63, 58, 71, 80, 78, 68, 59, 76, 66, 66, 62)
Vector(83, 74, 99, 86, 67, 76, 85, 79, 91, 82, 76, 58, 67, 102, 82, 67, 83, 55, 64, 98, 85, 74, 80, 70, 99, 91, 73, 79, 70, 95, 74, 73)
Vector(55, 52, 70, 58, 54, 53, 61, 60, 63, 54, 62, 42, 54, 77, 61, 61, 67, 40, 39, 67, 70, 53, 57, 51, 66, 61, 62, 64, 55, 71, 58, 55)
Vector(63, 62, 70, 84, 42, 61, 70, 63, 68, 70, 78, 52, 56, 66, 64, 69, 61, 53, 46, 81, 67, 72, 63, 56, 64, 63, 64, 43, 49, 67, 82, 67)
Vector(67, 66, 76, 85, 59, 47, 89, 72, 77, 87, 78, 53, 65, 90, 78, 53, 79, 71, 53, 87, 81, 69, 76, 62, 75, 79, 49, 53, 70, 75, 78, 67)
Vector(72, 61, 87, 86, 72, 72, 78, 88, 83, 77, 72, 44, 65, 90, 82, 62, 81, 73, 48, 97, 77, 87, 60, 74, 86, 74, 51, 64, 73, 81, 85, 72)
Vector(75, 69, 88, 102, 83, 62, 87, 91, 80, 71, 77, 51, 68, 95, 88, 73, 78, 65, 59, 118, 75, 79, 81, 69, 94, 85, 72, 61, 70, 82, 78, 58)
Vector(82, 85, 95, 99, 78, 79, 104, 100, 105, 89, 97, 60, 78, 126, 96, 86, 98, 79, 64, 114, 94, 88, 85, 81, 103, 106, 87, 73, 85, 100, 100, 84)
Vector(77, 79, 106, 83, 75, 71, 105, 88, 91, 98, 85, 64, 61, 104, 93, 77, 95, 69, 69, 90, 104, 90, 77, 91, 98, 68, 83, 94, 95, 90, 74)
Vector(71, 72, 101, 82, 68, 77, 85, 84, 94, 88, 79, 55, 66, 96, 80, 66, 97, 62, 62, 87, 82, 69, 79, 71, 79, 83, 74, 70, 94, 81, 79, 78)
Vector(63, 57, 82, 62, 57, 54, 66, 64, 58, 70, 76, 57, 44, 81, 66, 56, 71, 47, 47, 72, 71, 68, 63, 46, 74, 69, 63, 59, 57, 83, 74, 63)
Vector(53, 47, 66, 68, 57, 57, 57, 74, 52, 53, 57, 41, 60, 73, 62, 51, 61, 55, 42, 79, 61, 53, 54, 52, 56, 58, 41, 49, 53, 57, 63, 49)
Vector(68, 56, 77, 69, 65, 81, 78, 76, 79, 69, 77, 55, 58, 92, 63, 64, 85, 49, 54, 87, 67, 59, 57, 64, 80, 81, 73, 68, 75, 76, 83, 71)
Vector(73, 81, 96, 86, 76, 88, 95, 87, 106, 102, 99, 68, 54, 103, 97, 77, 92, 76, 59, 95, 93, 79, 67, 76, 108, 114, 86, 74, 89, 102, 92, 89)
Vector(70, 65, 106, 86, 67, 87, 81, 75, 88, 93, 95, 59, 59, 87, 87, 79, 84, 69, 54, 87, 98, 87, 74, 67, 96, 88, 88, 77, 78, 101, 84, 78)
Vector(76, 74, 100, 83, 70, 83, 89, 80, 95, 91, 82, 62, 56, 97, 90, 73, 87, 66, 55, 94, 92, 77, 67, 71, 99, 91, 70, 79, 79, 96, 85)
Vector(56, 72, 80, 78, 67, 55, 80, 88, 83, 77, 79, 49, 49, 96, 80, 61, 88, 53, 51, 93, 75, 68, 66, 66, 79, 85, 69, 50, 76, 77, 60)
Vector(81, 85, 98, 99, 83, 93, 92, 106, 91, 97, 88, 63, 82, 92, 98, 83, 79, 84, 54, 104, 82, 74, 85, 78, 94, 100, 69, 81, 83, 81, 84)
Vector(59, 78, 95, 82, 75, 64, 78, 81, 81, 84, 81, 53, 51, 85, 98, 73, 81, 62, 58, 86, 73, 69, 80, 62, 86, 83, 75, 62, 77, 91, 65, 76)
Vector(72, 83, 96, 92, 63, 55, 86, 72, 84, 96, 81, 58, 61, 95, 83, 69, 98, 68, 49, 91, 77, 83, 75, 66, 83, 97, 85, 75, 80, 93, 89, 78)
Vector(67, 69, 88, 83, 58, 68, 74, 85, 92, 85, 81, 50, 49, 104, 83, 68, 81, 62, 57, 96, 85, 80, 66, 80, 80, 75, 71, 67, 70, 90, 84, 81)
Vector(42, 57, 68, 62, 55, 50, 64, 61, 64, 70, 52, 35, 43, 67, 69, 48, 52, 55, 36, 69, 55, 59, 54, 48, 66, 67, 43, 47, 60, 64, 65, 56)
Vector(80, 83, 106, 94, 76, 68, 98, 92, 94, 83, 84, 57, 66, 114, 95, 81, 109, 65, 67, 98, 95, 83, 91, 71, 87, 96, 86, 78, 88, 98, 95, 76)
Vector(56, 67, 72, 60, 55, 60, 76, 71, 72, 72, 68, 56, 54, 80, 66, 52, 79, 40, 45, 75, 75, 64, 53, 65, 75, 75, 49, 62, 57, 72, 73, 65)
Vector(66, 67, 72, 77, 54, 58, 73, 83, 85, 64, 71, 44, 66, 95, 76, 66, 71, 57, 48, 90, 69, 65, 64, 69, 87, 83, 70, 58, 57, 75, 60, 68)
Vector(59, 74, 86, 94, 85, 79, 85, 88, 94, 88, 89, 52, 68, 99, 89, 82, 81, 72, 48, 99, 87, 82, 72, 68, 104, 98, 66, 63, 76, 88, 94, 84)
Vector(74, 63, 101, 90, 75, 73, 95, 78, 85, 90, 82, 61, 69, 90, 75, 66, 90, 67, 61, 88, 90, 72, 82, 52, 93, 93, 64, 67, 84, 85, 87, 73)

```

Lín. 38, col. 28 Espacios: 2 UTF-8 CRLF Scs

Tiempo de ejecución de Strassen: 10.6734
Tiempo de ejecución de Strassen Paralelo: 6.897

A pesar de que el tiempo del algoritmo secuencial se reduce un poco, el tiempo del algoritmo paralelo sigue siendo el mejor.

Matrices 64x64

```

//matrices 64x64
val m11 = mat1.matrizAlAzar(64, 3)
val m12 = mat1.matrizAlAzar(64, 3)

//matrices 128x128
val m13 = mat1.matrizAlAzar(128, 3)
val m14 = mat1.matrizAlAzar(128, 3)

// 10 Pruebas para la función Strassen
def pruebas(): Unit = {

    println("Multiplicacion Strassen 64x64")
    println("Matriz 1: ")
    m11.foreach(row => println(row))
    println("Matriz 2: ")
    m12.foreach(row => println(row))
    println("Resultado: ")
    (mat1.multStrassen(m11, m12)).foreach(row => println(row))
    val comp9 = mat1.compararAlgoritmos(mat1.multStrassen, mat1.multStrassenPar)(m11, m12)
    println("Tiempo de ejecución de Strassen: " + comp9(0))
    println("Tiempo de ejecución de Strassen Paralelo: " + comp9(1))
}

```

Al ser matrices tan grandes, es difícil adjuntar una captura visualizando cada una, así que solo se mostrará el resultado obtenido.

```
Tiempo de ejecución de Strassen: 80.8519
Tiempo de ejecución de Strassen Paralelo: 56.3721
```

```
BUILD SUCCESSFUL in 18s
2 actionable tasks: 2 executed
```

```
master* ✘ ⑧ 0 ▲ 0 『 0 『 Live Share 『 Java: Ready
```

Recordar que el tiempo es en ms, en este caso, la versión paralela se desempeña mucho mejor, su aceleración es significativa con respecto al segundo y si hay ganancias.

Matrices 128x128

```
Tiempo de ejecución de Strassen: 563.7584
Tiempo de ejecución de Strassen Paralelo: 337.3889
```

```
BUILD SUCCESSFUL in 19s
2 actionable tasks: 2 executed
```

```
『 Live Share 『 Java: Ready
```

Al tratar con datos mucho más grandes, los tiempos de ejecución de ambos algoritmos crecen, sin embargo, el algoritmo paralelo es más eficiente para estos casos que la versión secuencial. Las diferencias son significativas, su aceleración es de 1,6709.

- **ANÁLISIS - Multiplicación Strassen**

¿Cuál de las implementaciones es más rápida?

R/ Depende de la dimensión de las matrices a tratar, para casos donde los datos son mayores, el algoritmo paralelizado resulta siendo más rápido.

¿De qué depende que la aceleración sea mejor?

R/ Depende de factores como la dimensión de las matrices, en estos casos, la aceleración del algoritmo paralelo es mucho mejor que cuando las dimensiones son muy pequeñas.

¿Puede caracterizar los casos en que es mejor usar la versión secuencial/paralela de cada algoritmo de multiplicación de matrices?

R/ Se ha probado con 7 matrices de dimensiones 2, 4, 8, 16, 32, 64, 128, la lista nos retorna el tiempo en ms del algoritmo secuencial de Strassen, el tiempo del algoritmo paralelo, la aceleración del algoritmo paralelo con respecto al secuencial, y la dimensión de las matrices tratadas.

```
(List(0.0329, 0.0912, 0.3607456140350877),2)
(List(0.1619, 0.1432, 1.130586592178771),4)
(List(0.283, 0.1938, 1.460268317853457),8)
(List(2.9426, 2.9137, 1.0099186601228678),16)
(List(9.3821, 8.2235, 1.1408889159117164),32)
(List(70.81, 47.5988, 1.4876425456103937),64)
(List(538.4742, 339.877, 1.5843207984064822),128)
```

Se puede apreciar, cómo a medida que la matriz va creciendo sus dimensiones, el algoritmo de Strassen, en su versión paralela, resulta siendo más eficiente que el algoritmo secuencial.

- **Función multMatrizRecPar**

La estrategia usada para parallelizar en esta función fue con la abstracción `parallel`, que se utilizó para ejecutar simultáneamente cuatro cálculos independientes que corresponder a la creación de las submatrices c_{11} , c_{12} , c_{21} y c_{22} . Esto permite aprovechar los múltiples núcleos del procesador para mejorar el rendimiento.

En este caso, `parallel` coordina la ejecución de las tareas en diferentes hilos del sistema, asignándolas a los núcleos disponibles del procesador, al ser las tareas independientes entre sí, se pueden ejecutar sin generar un problema en la función.

Una vez todas las tareas han terminado `parallel` recoge los resultados de cada tarea, es decir, de las submatrices c_{11} , c_{12} , c_{21} y c_{22} y devuelve los respectivos resultados en una tupla. Reduciendo así el tiempo total de la ejecución y optimizando el uso del hardware disponible.

- **CASOS DE PRUEBA**

Se debe aclarar que para todos los casos de prueba de esta función se usó la siguiente función:

```
def pruebasRec(): Unit = {  
  
    val mt1 = mat1.matrizAlAzar(2, 5)  
    val mt2 = mat1.matrizAlAzar(2, 5)  
  
    println("Multiplicacion recursiva 2x2")  
    println("Matriz 1: ")  
    mt1.foreach(row => println(row))  
    println("Matriz 2: ")  
    mt2.foreach(row => println(row))  
    println("Resultado: ")  
    (mat1.multMatrizRec(mt1, mt2)).foreach(row => println(row))  
    val compa1 = mat1.compararAlgoritmos(mat1.multMatrizRec, mat1.multMatrizRecPar)(mt1, mt2)  
    println("Tiempo de ejecución de recursiva: " + compa1(0))  
    println("Tiempo de ejecución de recursiva Paralelo: " + compa1(1))  
    println("Aceleracion: " + compa1(2))  
  
    println("Multiplicacion recursiva 2x2")  
    println("Matriz 1: ")  
    mt2.foreach(row => println(row))  
    println("Matriz 2: ")  
    mt1.foreach(row => println(row))  
    println("Resultado: ")  
    (mat1.multMatrizRec(mt2, mt1)).foreach(row => println(row))  
    val compa2 = mat1.compararAlgoritmos(mat1.multMatrizRec, mat1.multMatrizRecPar)(mt2, mt1)  
    println("Tiempo de ejecución de recursiva: " + compa2(0))  
    println("Tiempo de ejecución de recursiva Paralelo: " + compa2(1))  
    println("Aceleracion: " + compa2(2))} You, 1 second ago • Uncommitted changes
```

En este captura en concreto se muestra la ejecución de una matriz 2x2, pero esta misma función se puede usar para otro tamaño de matrices, y se usará, solo se debe cambiar el tamaño de la matriz generada al azar y el nombre de las matrices, cambiando la multiplicación recursiva 2x2 por la multiplicación respectiva.

Para generar los 10 casos de prueba se hará la multiplicación en cada tamaño de matriz cambiando el orden, ya que al ser una operación no conmutativa la matriz resultante será diferente.

Además, para todos los casos los resultados serán los correctos, puesto que las funciones ya han sido probadas y ya se ha demostrado la función en la sección anterior.

- Matrices 2 x 2

```
Multiplicacion recursiva 2x2
Matriz 1:
Vector(4, 4)
Vector(2, 0)
Matriz 2:
Vector(2, 3)
Vector(4, 1)
Resultado:
Vector(24, 16)
Vector(4, 6)
Tiempo de ejecución de recursiva: 0.0474
Tiempo de ejecución de recursiva Paralelo: 0.1033
Aceleracion: 0.4588576960309777
```

```
Multiplicacion recursiva 2x2
Matriz 1:
Vector(2, 3)
Vector(4, 1)
Matriz 2:
Vector(4, 4)
Vector(2, 0)
Resultado:
Vector(14, 8)
Vector(18, 16)
Unable to create a system terminal
Tiempo de ejecución de recursiva: 0.1926
Tiempo de ejecución de recursiva Paralelo: 0.2403
Aceleracion: 0.8014981273408239
```

En este caso, no hay ganancia en la versión paralela, en ninguno de los casos esto se debe que al ser un tamaño poco significativo, la versión paralela no es eficaz y generalmente demora más tiempo.

- Matrices 4 x 4

```
Multiplicacion recursiva 4x4
Matriz 1:
Vector(0, 0, 1, 1)
Vector(1, 0, 0, 1)
Vector(2, 0, 2, 2)
Vector(1, 0, 0, 2)
Matriz 2:
Vector(0, 2, 2, 0)
Vector(2, 0, 0, 2)
Vector(2, 2, 0, 1)
Vector(0, 1, 0, 0)
Resultado:
Vector(2, 3, 0, 1)
Vector(0, 3, 2, 0)
Vector(4, 10, 4, 2)
Vector(0, 4, 2, 0)
Unable to create a system terminal
Tiempo de ejecución de recursiva: 0.3246
Tiempo de ejecución de recursiva Paralelo: 1.3507
Aceleracion: 0.24031983416006514
```

```
Multiplicacion recursiva 4x4
Matriz 1:
Vector(0, 2, 2, 0)
Vector(2, 0, 0, 2)
Vector(2, 2, 0, 1)
Vector(0, 1, 0, 0)
Matriz 2:
Vector(0, 0, 1, 1)
Vector(1, 0, 0, 1)
Vector(2, 0, 2, 2)
Vector(1, 0, 0, 2)
Resultado:
Vector(6, 0, 4, 6)
Vector(2, 0, 2, 6)
Vector(3, 0, 2, 6)
Vector(1, 0, 0, 1)
Tiempo de ejecución de recursiva: 0.0623
Tiempo de ejecución de recursiva Paralelo: 0.1149
Aceleracion: 0.5422106179286336
```

En este caso, sigue sin haber ganancia en ninguno de los casos, ya que el tamaño de las matrices sigue sin ser lo suficientemente considerable.

- Matrices 8 x 8

Multiplicacion recursiva 8x8

Matriz 1:

```
Vector(0, 1, 1, 2, 3, 3, 3, 3)  
Vector(2, 0, 0, 0, 3, 1, 0, 2)  
Vector(0, 1, 2, 0, 2, 0, 0, 3)  
Vector(3, 2, 2, 1, 2, 0, 0, 1)  
Vector(2, 3, 0, 0, 2, 0, 0, 0)  
Vector(1, 2, 0, 1, 3, 0, 0, 1)  
Vector(2, 1, 1, 2, 3, 3, 3)  
Vector(3, 2, 1, 1, 2, 1, 1, 2)
```

Matriz 2:

```
Vector(0, 3, 3, 0, 3, 2, 0, 3)  
Vector(3, 3, 1, 1, 3, 2, 1, 3)  
Vector(1, 3, 0, 0, 1, 1, 0, 3)  
Vector(3, 0, 1, 0, 0, 2, 3, 2)  
Vector(0, 1, 2, 2, 3, 2, 0, 0)  
Vector(0, 1, 2, 0, 0, 3, 3, 2)  
Vector(3, 0, 3, 3, 2, 1, 1, 0)  
Vector(3, 1, 2, 2, 1, 3, 2, 2)
```

Resultado:

```
Vector(28, 15, 30, 22, 22, 34, 25, 22)  
Vector(6, 12, 18, 10, 17, 19, 7, 12)  
Vector(14, 14, 11, 11, 14, 17, 7, 15)  
Vector(14, 24, 18, 8, 24, 21, 7, 25)  
Vector(9, 17, 13, 7, 21, 14, 3, 15)  
Vector(12, 13, 14, 10, 19, 17, 7, 13)  
Vector(28, 21, 36, 22, 28, 38, 25, 28)  
Vector(19, 23, 25, 13, 26, 27, 13, 26)  
Unable to create a system terminal  
Tiempo de ejecución de recursiva: 0.9142  
Tiempo de ejecución de recursiva Paralelo: 0.7746  
Aceleracion: 1.1802220500903693
```

Multiplicacion recursiva 8x8

Matriz 1:

```
Vector(0, 3, 3, 0, 3, 2, 0, 3)  
Vector(3, 3, 1, 1, 3, 2, 1, 3)  
Vector(1, 3, 0, 0, 1, 1, 0, 3)  
Vector(3, 0, 1, 0, 0, 2, 3, 2)  
Vector(0, 1, 2, 2, 3, 2, 0, 0)  
Vector(0, 1, 2, 0, 0, 3, 3, 2)  
Vector(3, 0, 3, 3, 2, 1, 1, 0)  
Vector(3, 1, 2, 2, 1, 3, 2, 2)
```

Matriz 2:

```
Vector(0, 1, 1, 2, 3, 3, 3, 3)  
Vector(2, 0, 0, 0, 3, 1, 0, 2)  
Vector(0, 1, 2, 0, 2, 0, 0, 3)  
Vector(3, 2, 2, 1, 2, 0, 0, 1)  
Vector(2, 3, 0, 0, 2, 0, 0, 0)  
Vector(1, 2, 0, 1, 3, 0, 0, 1)  
Vector(2, 1, 1, 2, 3, 3, 3)  
Vector(3, 2, 1, 1, 2, 1, 1, 2)
```

Resultado:

```
Vector(23, 22, 9, 5, 33, 6, 3, 23)  
Vector(28, 26, 11, 14, 43, 18, 15, 30)  
Vector(18, 12, 4, 6, 23, 9, 6, 16)  
Vector(14, 15, 10, 16, 30, 20, 20, 27)  
Vector(16, 19, 8, 4, 23, 1, 0, 12)  
Vector(17, 15, 9, 11, 29, 12, 11, 24)  
Vector(16, 21, 16, 12, 31, 12, 12, 25)  
Vector(23, 24, 15, 17, 41, 18, 17, 32)  
Tiempo de ejecución de recursiva: 0.3269  
Tiempo de ejecución de recursiva Paralelo: 0.2004  
Aceleracion: 1.6312375249500999
```

En estos dos casos, se puede presenciar una ganancia del tiempo de ejecución de la versión paralela en comparación con la recursiva, esto se debe a que el tamaño de las matrices ahora sí es significativo y, por tanto, la función paralela comienza a ser útil para disminuir el tiempo de ejecución.

- Matrices 16 x 16

Matriz 1:

```
Vector(2, 0, 1, 4, 0, 0, 4, 4, 2, 3, 1, 3, 0, 3, 0, 0)  
Vector(1, 0, 0, 1, 0, 1, 2, 0, 0, 0, 4, 1, 1, 0, 4, 4)  
Vector(2, 2, 0, 1, 0, 0, 3, 2, 4, 2, 4, 0, 4, 2, 0, 3)  
Vector(0, 4, 3, 0, 2, 3, 2, 3, 3, 2, 3, 1, 3, 3, 4, 3)  
Vector(3, 4, 4, 1, 0, 3, 3, 2, 3, 0, 0, 4, 3, 2, 0, 2)  
Vector(0, 0, 3, 0, 0, 3, 1, 2, 1, 0, 0, 4, 1, 1, 3, 3)  
Vector(4, 2, 2, 4, 4, 3, 3, 2, 1, 0, 0, 4, 4, 1, 0, 3)  
Vector(3, 0, 4, 2, 1, 2, 0, 0, 4, 0, 4, 3, 0, 4, 4, 0)  
Vector(0, 2, 3, 2, 1, 1, 2, 3, 1, 3, 0, 2, 1, 3, 3, 4)  
Vector(2, 4, 1, 0, 4, 4, 4, 2, 3, 3, 3, 4, 0, 3, 3)  
Vector(4, 0, 3, 0, 1, 1, 3, 4, 1, 4, 2, 2, 4, 2, 2, 2)  
Vector(3, 0, 1, 1, 4, 2, 3, 3, 1, 1, 3, 1, 2, 4, 1, 3)  
Vector(4, 3, 3, 4, 1, 4, 0, 0, 3, 4, 2, 3, 1, 3, 2, 2)  
Vector(4, 1, 1, 1, 4, 1, 1, 4, 3, 1, 3, 3, 1, 0, 2, 2)  
Vector(4, 4, 0, 0, 4, 0, 4, 0, 1, 0, 0, 4, 0, 1, 1, 1)  
Vector(3, 2, 3, 2, 2, 0, 1, 2, 2, 4, 2, 3, 4, 0, 1, 0)
```

Matriz 2:

```
Vector(0, 2, 2, 2, 4, 2, 1, 4, 3, 1, 4, 0, 3, 4, 2, 0)  
Vector(2, 4, 0, 3, 0, 0, 0, 4, 2, 2, 1, 2, 0, 1, 3)  
Vector(4, 3, 4, 2, 1, 2, 2, 2, 0, 3, 4, 3, 0, 1, 4)  
Vector(2, 4, 2, 4, 1, 1, 1, 2, 2, 1, 1, 2, 1, 3, 2)  
Vector(4, 4, 2, 4, 3, 1, 1, 3, 0, 4, 4, 0, 1, 3, 1, 1)  
Vector(1, 0, 3, 2, 1, 3, 1, 3, 4, 1, 0, 4, 2, 1, 2, 2)  
Vector(1, 1, 0, 4, 3, 3, 4, 0, 2, 2, 2, 3, 4, 2, 0, 0)  
Vector(4, 1, 3, 3, 1, 0, 3, 2, 4, 1, 1, 0, 4, 2, 2, 3)  
Vector(3, 3, 2, 1, 0, 1, 0, 0, 3, 2, 0, 1, 4, 1, 2)  
Vector(3, 1, 0, 1, 3, 4, 3, 2, 2, 3, 1, 4, 4, 0, 2, 0)  
Vector(0, 3, 4, 2, 0, 3, 3, 2, 2, 2, 0, 0, 0, 3, 2, 0)  
Vector(2, 1, 2, 2, 3, 3, 4, 4, 0, 2, 2, 0, 4, 0, 4, 2)  
Vector(0, 1, 4, 1, 4, 4, 4, 1, 1, 3, 4, 3, 0, 3, 0)  
Vector(1, 4, 3, 3, 0, 2, 3, 3, 2, 1, 4, 0, 0, 1, 4, 0)  
Vector(1, 1, 4, 0, 4, 1, 2, 2, 4, 4, 0, 2, 0, 0, 0, 3)  
Vector(4, 2, 4, 2, 3, 1, 4, 4, 4, 3, 3, 1, 4, 4, 3, 3)
```

Resultado:

```
Vector(56, 58, 53, 74, 49, 52, 71, 51, 54, 48, 52, 32, 75, 42, 59, 34)  
Vector(27, 34, 61, 35, 47, 39, 55, 45, 54, 47, 26, 27, 38, 38, 34, 30)  
Vector(49, 65, 74, 66, 55, 57, 77, 47, 62, 57, 60, 39, 68, 61, 58, 31)  
Vector(81, 84, 106, 83, 70, 68, 91, 73, 97, 80, 72, 65, 81, 55, 68, 70)  
Vector(67, 71, 82, 81, 64, 63, 78, 67, 76, 52, 78, 56, 93, 48, 69, 62)  
Vector(51, 33, 69, 42, 49, 42, 61, 57, 55, 41, 39, 40, 59, 26, 46, 52)  
Vector(74, 80, 89, 97, 86, 71, 85, 84, 76, 68, 89, 54, 101, 62, 80, 57)  
Vector(52, 77, 96, 64, 52, 56, 64, 71, 61, 59, 60, 34, 46, 51, 61, 51)  
Vector(77, 66, 78, 70, 62, 53, 80, 66, 79, 64, 62, 52, 80, 39, 62, 61)  
Vector(83, 82, 101, 95, 97, 86, 101, 84, 96, 95, 81, 73, 104, 69, 73, 64)  
Vector(67, 61, 90, 72, 85, 79, 97, 78, 80, 62, 75, 63, 97, 54, 69, 45)  
Vector(64, 74, 88, 84, 68, 64, 84, 79, 73, 65, 77, 39, 73, 68, 68, 39)  
Vector(74, 90, 94, 85, 72, 76, 77, 87, 89, 75, 76, 61, 86, 56, 83, 62)  
Vector(70, 69, 84, 75, 71, 51, 71, 75, 67, 70, 66, 27, 75, 69, 59, 51)  
Vector(45, 58, 38, 67, 60, 40, 50, 53, 46, 55, 65, 19, 61, 45, 40, 32)  
Vector(62, 66, 72, 66, 71, 67, 73, 60, 57, 59, 65, 53, 82, 40, 61, 43)
```

Tiempo de ejecución de recursiva: 1.3237

Tiempo de ejecución de recursiva Paralelo: 0.7889

Aceleración: 1.6779059449866904

Matriz 1:

```
Vector(4, 3, 1, 2, 3, 1, 0, 1, 0, 0, 3, 2, 4, 0, 0, 2)  
Vector(4, 3, 4, 3, 2, 0, 2, 1, 4, 3, 1, 1, 2, 4, 2, 3)  
Vector(1, 4, 3, 0, 4, 0, 0, 1, 3, 3, 0, 0, 3, 0, 1, 3)  
Vector(4, 4, 2, 0, 2, 3, 0, 0, 2, 3, 2, 3, 3, 2, 1)  
Vector(0, 1, 0, 0, 1, 3, 1, 3, 1, 0, 4, 2, 4, 3, 4, 0)  
Vector(2, 1, 4, 4, 1, 2, 1, 3, 3, 2, 1, 4, 0, 3, 0, 4)  
Vector(1, 0, 2, 0, 3, 2, 2, 4, 3, 1, 2, 3, 4, 0, 1, 4)  
Vector(0, 4, 3, 1, 0, 1, 3, 1, 0, 0, 1, 3, 2, 0)  
Vector(3, 3, 4, 0, 2, 3, 2, 0, 3, 4, 1, 2, 1, 2, 2, 4)  
Vector(2, 3, 0, 3, 1, 4, 4, 2, 4, 4, 4, 4, 2, 4, 0)  
Vector(2, 3, 0, 4, 2, 4, 2, 3, 4, 4, 0, 2, 3, 4, 3, 1)  
Vector(4, 0, 4, 1, 4, 4, 0, 2, 0, 3, 2, 0, 3, 0, 0, 0)  
Vector(0, 0, 0, 0, 0, 1, 2, 4, 1, 3, 1, 1, 2, 4, 0)  
Vector(1, 0, 3, 4, 0, 2, 4, 3, 4, 2, 2, 1, 2, 4, 1)  
Vector(0, 2, 1, 0, 4, 0, 3, 2, 4, 0, 4, 4, 0, 0, 2, 3)  
Vector(4, 1, 2, 3, 1, 2, 4, 2, 3, 0, 0, 1, 3, 2, 0, 1)
```

Matriz 2:

```
Vector(4, 2, 3, 3, 4, 2, 4, 0, 2, 4, 1, 4, 0, 2, 3, 3)  
Vector(4, 4, 0, 2, 3, 4, 2, 3, 4, 3, 2, 4, 1, 4)  
Vector(1, 3, 2, 1, 1, 1, 2, 3, 2, 2, 4, 4, 1, 4, 1, 2)  
Vector(4, 4, 4, 1, 2, 2, 2, 3, 3, 3, 3, 0, 3, 3, 2, 1)  
Vector(2, 4, 2, 1, 1, 0, 4, 2, 2, 0, 4, 1, 4, 0, 1, 2)  
Vector(2, 3, 0, 3, 3, 1, 4, 2, 2, 1, 4, 2, 2, 3, 4, 1)  
Vector(1, 0, 3, 3, 1, 4, 3, 2, 2, 1, 0, 2, 0, 1, 3)  
Vector(4, 0, 4, 3, 1, 0, 3, 0, 2, 1, 1, 0, 4, 4, 0, 2)  
Vector(2, 3, 0, 2, 4, 1, 2, 3, 1, 3, 1, 2, 1, 3, 2, 0)  
Vector(4, 3, 1, 3, 2, 3, 0, 1, 1, 0, 0, 3, 3, 3, 4, 2)  
Vector(4, 2, 1, 0, 3, 2, 2, 2, 1, 4, 0, 1, 3, 3, 0, 4)  
Vector(3, 2, 1, 0, 0, 2, 4, 2, 3, 1, 4, 0, 1, 3, 0, 1)  
Vector(1, 1, 4, 4, 3, 1, 4, 0, 1, 0, 0, 2, 4, 4, 2, 4)  
Vector(1, 0, 3, 0, 3, 1, 4, 1, 3, 1, 4, 4, 0, 1, 3, 1)  
Vector(2, 2, 1, 4, 4, 2, 1, 4, 1, 4, 0, 2, 2, 4, 4, 2)  
Vector(2, 1, 1, 1, 2, 1, 2, 2, 4, 0, 0, 1, 1, 3, 4, 0)
```

Resultado:

```
Vector(75, 62, 55, 48, 62, 42, 81, 40, 55, 57, 51, 45, 60, 70, 41, 75)  
Vector(97, 86, 76, 72, 97, 68, 99, 75, 80, 88, 76, 85, 67, 99, 82, 86)  
Vector(64, 69, 40, 55, 61, 41, 62, 51, 50, 48, 49, 55, 60, 71, 51, 67)  
Vector(89, 80, 52, 67, 89, 61, 96, 60, 70, 67, 74, 81, 61, 95, 74, 79)  
Vector(64, 44, 52, 58, 70, 39, 88, 47, 48, 51, 45, 44, 65, 81, 50, 63)  
Vector(92, 75, 67, 51, 70, 52, 94, 67, 77, 74, 81, 57, 60, 91, 64, 66)  
Vector(75, 58, 62, 65, 40, 93, 52, 56, 58, 50, 40, 75, 83, 52, 76)  
Vector(47, 49, 30, 39, 55, 36, 47, 47, 45, 43, 51, 53, 34, 65, 42, 39)  
Vector(87, 83, 50, 70, 86, 65, 98, 72, 71, 77, 72, 78, 60, 91, 82, 82)  
Vector(120, 96, 78, 98, 112, 88, 118, 86, 85, 90, 77, 74, 99, 127, 90, 97)  
Vector(107, 90, 78, 90, 103, 70, 110, 75, 84, 74, 83, 75, 87, 111, 94, 78)  
Vector(71, 68, 57, 60, 61, 34, 80, 38, 47, 41, 57, 61, 66, 74, 56, 63)  
Vector(47, 32, 30, 40, 55, 30, 43, 41, 28, 47, 19, 32, 39, 57, 37, 36)  
Vector(86, 69, 70, 72, 81, 61, 85, 75, 65, 79, 60, 54, 70, 97, 69, 65)  
Vector(74, 62, 40, 43, 58, 48, 77, 65, 56, 68, 53, 30, 62, 67, 35, 67)  
Vector(68, 55, 70, 63, 69, 49, 90, 47, 60, 60, 58, 52, 54, 70, 55, 62)
```

Tiempo de ejecución de recursiva: 1.014

Tiempo de ejecución de recursiva Paralelo: 0.6066

Aceleración: 1.671612265084075

Como en las matrices 8 x 8 se vuelve a presenciar en ambos casos que el tiempo de ejecución paralelo es menor al secuencial, con esta vez una mayor diferencia en los tiempos de ejecución

- Matrices 32 x 32

```
Multiplicacion recursiva 32x32
Matriz 1:
Vector(0, 3, 1, 3, 1, 1, 0, 3, 2, 2, 3, 3, 0, 1, 0, 3, 1, 3, 2, 0, 1, 1, 3, 1, 2, 3, 1, 0, 2, 2, 0)
Vector(3, 2, 1, 3, 2, 1, 1, 0, 0, 2, 2, 2, 0, 0, 3, 1, 3, 2, 1, 2, 2, 2, 3, 1, 0, 0, 2, 2, 1)
Vector(2, 2, 1, 3, 0, 2, 0, 1, 2, 2, 1, 0, 0, 3, 3, 3, 0, 0, 2, 3, 3, 1, 1, 1, 3, 3, 1, 1, 0, 0, 1)
Vector(1, 3, 0, 0, 1, 2, 3, 3, 2, 3, 3, 1, 3, 3, 0, 0, 3, 2, 2, 2, 1, 2, 2, 0, 2, 0, 0, 1, 0, 1, 2, 2)
Vector(1, 0, 1, 3, 2, 2, 1, 2, 2, 1, 2, 0, 0, 0, 2, 0, 2, 1, 2, 3, 2, 0, 1, 1, 1, 0, 3, 3, 2, 1, 1)
Vector(1, 3, 2, 3, 3, 2, 1, 2, 1, 2, 3, 3, 2, 3, 0, 2, 1, 3, 1, 2, 3, 0, 1, 0, 3, 2, 2, 0, 1, 1, 3)
Vector(0, 3, 0, 1, 1, 2, 0, 2, 2, 2, 3, 1, 1, 1, 0, 0, 2, 0, 2, 1, 0, 1, 2, 1, 1, 3, 2, 0, 3, 2, 2)
Vector(1, 0, 2, 0, 1, 3, 3, 0, 1, 1, 1, 2, 1, 2, 2, 2, 0, 3, 0, 1, 1, 3, 0, 1, 3, 2, 3, 0, 0, 1, 3, 0)
Vector(3, 1, 3, 2, 1, 2, 2, 2, 0, 1, 2, 3, 1, 1, 1, 0, 1, 0, 3, 3, 0, 2, 2, 3, 2, 3, 2, 1, 1, 2, 0, 3)
Vector(0, 3, 1, 2, 2, 1, 2, 2, 1, 3, 0, 3, 1, 2, 0, 3, 0, 3, 1, 3, 0, 3, 2, 2, 2, 1, 1, 1, 2, 2, 3, 2)
Vector(0, 3, 2, 3, 0, 2, 1, 0, 2, 2, 0, 3, 3, 0, 0, 2, 3, 0, 0, 2, 0, 1, 3, 2, 0, 1, 0, 3, 3, 3)
Vector(3, 2, 0, 3, 0, 1, 3, 1, 1, 0, 1, 2, 1, 0, 2, 0, 2, 3, 3, 2, 3, 3, 2, 2, 3, 2, 0, 2, 3, 2, 2)
Vector(1, 3, 2, 1, 2, 3, 2, 1, 0, 1, 1, 2, 0, 3, 2, 3, 0, 2, 0, 1, 3, 1, 1, 1, 1, 2, 1, 1, 3, 0, 2, 1, 2)
Vector(0, 2, 2, 3, 0, 3, 3, 3, 1, 0, 3, 2, 3, 0, 3, 2, 2, 0, 0, 0, 1, 3, 3, 1, 2, 1, 1, 1, 0, 3, 1, 1)
Vector(2, 1, 3, 1, 0, 0, 0, 0, 3, 1, 1, 2, 1, 3, 3, 2, 0, 3, 2, 3, 0, 0, 3, 1, 2, 1, 0, 3, 2, 3, 2)
Vector(2, 3, 2, 3, 3, 0, 0, 0, 3, 0, 3, 1, 1, 3, 3, 0, 0, 1, 1, 2, 3, 3, 0, 2, 1, 1, 3, 1, 1, 3)
Vector(1, 3, 3, 3, 0, 2, 3, 1, 0, 2, 1, 0, 3, 1, 0, 0, 1, 2, 0, 2, 3, 2, 3, 0, 0, 0, 2, 1, 2, 2, 3, 3)
Vector(3, 0, 3, 3, 1, 3, 1, 1, 2, 1, 3, 2, 1, 2, 0, 2, 2, 2, 3, 2, 0, 3, 0, 0, 1, 2, 2, 3, 1, 0, 0, 1)
Vector(1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 1, 2, 2, 1, 1, 2, 2, 1, 2, 1, 3, 0, 3, 2, 3)
Vector(1, 1, 0, 0, 1, 1, 1, 0, 0, 2, 0, 0, 2, 3, 1, 1, 0, 1, 0, 3, 2, 1, 2, 1, 1, 0, 1, 0, 2, 3, 1)
Vector(0, 3, 0, 0, 1, 2, 3, 3, 0, 2, 0, 0, 3, 2, 0, 2, 1, 0, 1, 1, 3, 3, 0, 2, 1, 2, 2, 3, 1, 3, 0, 2)
Vector(2, 2, 2, 3, 1, 0, 3, 0, 0, 0, 1, 2, 2, 2, 2, 3, 0, 0, 0, 1, 2, 0, 1, 0, 2, 1, 1, 0, 3, 0, 1, 0)
Vector(0, 2, 1, 1, 3, 0, 1, 1, 1, 2, 3, 1, 3, 3, 3, 0, 0, 1, 2, 1, 2, 0, 2, 2, 3, 1, 3, 2, 0, 1)
Vector(0, 3, 2, 2, 3, 2, 3, 2, 3, 0, 1, 0, 0, 3, 1, 0, 0, 1, 3, 1, 3, 2, 2, 2, 0, 0, 0, 1, 1)
Vector(0, 3, 1, 3, 3, 3, 2, 0, 3, 1, 0, 0, 2, 2, 2, 0, 1, 1, 1, 2, 0, 1, 3, 1, 1, 3, 2, 2, 2, 3, 3)
Vector(0, 3, 2, 3, 0, 1, 0, 3, 1, 3, 2, 0, 0, 3, 2, 3, 0, 2, 2, 3, 0, 3, 2, 2, 1, 2, 0, 3, 1, 0)
Vector(3, 0, 3, 2, 3, 0, 1, 2, 3, 3, 1, 2, 0, 2, 1, 2, 0, 0, 1, 0, 1, 0, 2, 1, 3, 1, 1, 2, 1, 1, 2)
Vector(1, 3, 2, 3, 2, 0, 0, 2, 3, 2, 1, 1, 0, 3, 0, 1, 2, 2, 2, 0, 3, 1, 1, 1, 2, 3, 2, 1, 2, 1, 3, 0)
Vector(2, 3, 0, 2, 3, 2, 3, 2, 1, 0, 2, 3, 2, 0, 1, 0, 2, 3, 0, 1, 1, 3, 0, 0, 2, 3, 3, 2, 2, 1, 3, 1)
Vector(1, 2, 0, 3, 3, 1, 2, 0, 1, 1, 3, 0, 3, 1, 2, 2, 0, 1, 2, 2, 1, 0, 0, 2, 1, 0, 3, 0, 2, 0, 3, 1)
Vector(0, 0, 2, 2, 3, 0, 3, 2, 3, 1, 2, 2, 2, 0, 3, 3, 1, 0, 1, 0, 0, 0, 1, 2, 2, 3, 2, 3, 0, 1, 1, 3)
Vector(1, 3, 2, 3, 2, 0, 2, 0, 1, 2, 3, 2, 1, 2, 2, 3, 1, 2, 1, 1, 2, 2, 1, 0, 3, 1, 1, 1, 3, 0, 2)
```

Matriz 2:

```
Vector(0, 2, 3, 1, 0, 2, 2, 0, 0, 3, 1, 1, 1, 0, 2, 3, 3, 2, 1, 0, 2, 0, 1, 1, 1, 1, 3, 3, 0, 0, 3)  
Vector(0, 2, 1, 0, 1, 2, 3, 0, 1, 2, 2, 1, 1, 1, 0, 3, 1, 1, 0, 0, 0, 3, 1, 0, 1, 2, 2, 3, 1, 3, 2, 1, 2)  
Vector(2, 1, 1, 3, 2, 3, 2, 0, 0, 0, 0, 1, 0, 3, 1, 1, 3, 3, 2, 2, 1, 1, 0, 1, 3, 2, 0, 2, 0, 0, 3, 0)  
Vector(0, 2, 2, 0, 1, 1, 2, 1, 1, 0, 3, 1, 3, 3, 0, 3, 1, 3, 2, 0, 3, 1, 1, 1, 2, 1, 0, 3, 2, 3, 1)  
Vector(0, 2, 2, 1, 3, 1, 2, 3, 1, 0, 0, 1, 2, 3, 1, 0, 1, 2, 2, 0, 3, 3, 2, 1, 0, 3, 2, 2, 1, 3, 0)  
Vector(2, 2, 0, 1, 2, 2, 0, 1, 2, 1, 2, 0, 1, 3, 0, 0, 0, 1, 2, 3, 1, 0, 3, 0, 1, 0, 3, 0, 1, 0, 1, 2)  
Vector(0, 1, 3, 2, 2, 0, 0, 3, 0, 3, 1, 1, 1, 0, 0, 1, 0, 2, 0, 3, 1, 1, 3, 0, 1, 1, 0, 3, 1, 2, 1, 2)  
Vector(3, 2, 2, 1, 3, 0, 3, 2, 1, 3, 2, 3, 1, 0, 2, 2, 3, 2, 0, 0, 0, 2, 2, 0, 2, 3, 3, 2, 1, 1, 2, 0)  
Vector(3, 2, 2, 0, 1, 2, 1, 3, 1, 0, 0, 1, 1, 0, 0, 2, 3, 3, 1, 0, 2, 0, 0, 1, 2, 0, 2, 1, 2, 1, 0, 1)  
Vector(0, 1, 3, 0, 2, 2, 0, 0, 1, 0, 3, 2, 1, 1, 1, 3, 3, 3, 0, 0, 2, 0, 3, 0, 0, 0, 3, 1, 0, 2)  
Vector(1, 0, 0, 3, 0, 0, 1, 0, 3, 2, 3, 0, 2, 2, 1, 3, 2, 3, 1, 2, 3, 3, 3, 1, 1, 3, 3)  
Vector(1, 1, 2, 2, 2, 1, 0, 1, 3, 3, 0, 0, 2, 3, 0, 2, 1, 2, 1, 1, 2, 1, 0, 1, 0, 3, 1, 3, 0, 2)  
Vector(2, 2, 0, 1, 0, 3, 0, 0, 1, 3, 0, 0, 1, 3, 0, 3, 1, 1, 2, 0, 1, 0, 1, 2, 3, 0, 2, 2, 1, 1, 3, 0)  
Vector(0, 0, 0, 1, 3, 3, 1, 1, 0, 1, 0, 2, 1, 0, 2, 2, 2, 0, 2, 1, 3, 0, 0, 1, 3, 1, 3, 0, 2)  
Vector(1, 0, 3, 1, 1, 0, 1, 0, 1, 1, 2, 0, 1, 0, 2, 2, 2, 3, 1, 3, 3, 2, 2, 2, 0, 0, 1, 1, 3, 3, 2, 2)  
Vector(1, 0, 1, 2, 1, 3, 0, 3, 3, 0, 2, 0, 1, 1, 1, 2, 3, 0, 0, 2, 1, 0, 3, 3, 3, 2, 0, 3, 3)  
Vector(0, 1, 3, 3, 0, 1, 0, 3, 2, 2, 1, 1, 2, 3, 3, 2, 3, 1, 3, 1, 2, 3, 0, 0, 1, 1, 0, 2, 1, 3, 0, 0)  
Vector(2, 2, 1, 0, 0, 2, 0, 0, 1, 2, 0, 3, 3, 1, 2, 3, 0, 2, 1, 2, 2, 0, 2, 2, 2, 3, 3, 0, 1, 0, 2, 0)  
Vector(0, 0, 2, 1, 0, 3, 3, 1, 2, 3, 2, 2, 3, 2, 0, 1, 0, 3, 1, 1, 2, 3, 3, 3, 2, 1, 1, 3, 0, 1, 2, 1)  
Vector(1, 0, 1, 0, 3, 2, 0, 0, 3, 2, 0, 0, 2, 0, 1, 1, 1, 2, 3, 0, 0, 2, 1, 0, 3, 3, 3, 2, 0, 3, 3)  
Vector(0, 1, 3, 3, 0, 1, 0, 3, 2, 2, 2, 2, 1, 0, 0, 2, 2, 3, 3, 2, 1, 3, 1, 0, 0, 1, 1, 0, 2, 1, 3, 0, 0)  
Vector(2, 3, 1, 3, 3, 1, 1, 3, 2, 2, 3, 2, 0, 3, 0, 1, 3, 1, 0, 1, 2, 0, 3, 3, 1, 1, 3, 2, 2, 3, 2, 0)  
Vector(2, 0, 0, 2, 2, 3, 1, 1, 2, 1, 3, 1, 3, 0, 2, 1, 1, 0, 2, 1, 3, 2, 3, 0, 3, 1, 2, 0, 3, 1, 2, 0)  
Vector(2, 0, 2, 2, 0, 0, 3, 3, 3, 2, 0, 0, 2, 1, 3, 1, 3, 3, 0, 0, 3, 1, 2, 1, 3, 3, 2, 0, 1, 2, 1, 3)  
Vector(1, 0, 3, 2, 2, 1, 0, 2, 3, 1, 2, 0, 1, 0, 2, 2, 3, 2, 0, 1, 0, 3, 1, 1, 2, 3, 3, 2, 1, 1, 3, 0, 2)  
Vector(2, 3, 1, 3, 3, 1, 1, 3, 2, 2, 3, 2, 0, 3, 0, 1, 3, 1, 0, 1, 2, 0, 3, 3, 1, 1, 3, 2, 2, 3, 2, 0)  
Vector(2, 0, 0, 2, 2, 3, 1, 1, 2, 1, 3, 1, 3, 0, 2, 1, 1, 0, 2, 1, 3, 2, 3, 0, 3, 1, 2, 0, 3, 1, 2, 0)  
Vector(3, 2, 0, 0, 3, 1, 3, 2, 0, 1, 1, 3, 2, 3, 2, 0, 1, 0, 3, 1, 1, 2, 3, 3, 2, 0, 1, 0, 3, 2, 0, 1, 0, 0)  
Vector(0, 2, 3, 2, 1, 0, 3, 3, 1, 0, 0, 0, 0, 1, 1, 2, 2, 0, 0, 2, 1, 2, 0, 0, 2, 1, 3, 2, 1, 3, 2, 0)  
Vector(0, 3, 0, 0, 0, 3, 3, 1, 0, 1, 0, 2, 1, 3, 0, 1, 1, 2, 0, 3, 3, 0, 0, 1, 1, 3, 1, 2, 0, 1, 0)  
Vector(3, 1, 3, 3, 3, 0, 1, 0, 0, 3, 1, 1, 0, 1, 0, 2, 1, 3, 1, 1, 0, 0, 0, 2, 0, 2, 3, 1, 2, 0)  
Vector(1, 1, 2, 3, 0, 1, 3, 0, 1, 2, 2, 0, 3, 0, 3, 2, 1, 3, 1, 0, 1, 0, 2, 0, 0, 2, 0, 3)
```

Resultado:

```
Vector(63, 60, 72, 65, 67, 79, 69, 63, 70, 89, 67, 54, 84, 98, 52, 82, 73, 87, 84, 55, 85, 68, 77, 51, 78, 68, 99, 90, 85, 75, 70, 62)  
Vector(48, 62, 76, 63, 69, 87, 56, 61, 61, 93, 67, 48, 79, 88, 47, 71, 64, 85, 89, 74, 76, 58, 81, 54, 64, 68, 93, 92, 82, 63, 74, 74)  
Vector(44, 53, 72, 59, 76, 81, 61, 81, 57, 63, 82, 44, 67, 67, 50, 66, 87, 87, 81, 68, 81, 58, 73, 38, 51, 62, 83, 86, 76, 79, 54, 84)  
Vector(52, 57, 74, 58, 72, 82, 65, 71, 50, 85, 53, 57, 82, 77, 50, 71, 67, 86, 88, 72, 75, 68, 70, 40, 77, 63, 81, 88, 64, 68, 57, 70)  
Vector(45, 65, 72, 46, 72, 68, 63, 73, 56, 64, 54, 47, 63, 63, 45, 60, 59, 89, 76, 64, 62, 52, 73, 38, 58, 45, 90, 71, 60, 54, 60, 64)  
Vector(55, 70, 85, 77, 90, 101, 70, 80, 75, 88, 72, 59, 92, 93, 58, 82, 81, 104, 87, 85, 87, 67, 92, 52, 80, 78, 114, 101, 86, 84, 82, 82)  
Vector(57, 53, 51, 47, 63, 67, 60, 52, 48, 63, 54, 48, 71, 66, 50, 59, 72, 67, 59, 68, 49, 64, 34, 58, 58, 93, 69, 69, 56, 48, 59)  
Vector(52, 45, 62, 63, 72, 70, 37, 60, 51, 67, 58, 34, 62, 60, 46, 49, 49, 73, 67, 77, 65, 41, 76, 31, 50, 53, 77, 78, 68, 52, 59, 64)  
Vector(54, 58, 77, 76, 74, 84, 79, 74, 69, 90, 72, 53, 71, 95, 57, 63, 83, 94, 85, 71, 79, 74, 87, 48, 74, 63, 95, 100, 71, 75, 65, 85)  
Vector(57, 62, 80, 60, 86, 87, 71, 81, 64, 90, 68, 46, 85, 86, 76, 60, 69, 62, 91, 80, 77, 77, 65, 81, 36, 71, 73, 100, 87, 83, 70, 70, 82)  
Vector(49, 62, 75, 67, 56, 77, 64, 72, 52, 76, 68, 38, 77, 84, 49, 63, 59, 84, 80, 65, 76, 49, 59, 34, 68, 41, 81, 69, 73, 63, 64, 57)  
Vector(54, 72, 95, 70, 72, 89, 79, 77, 71, 105, 82, 64, 87, 81, 61, 76, 79, 100, 89, 73, 105, 76, 94, 56, 65, 69, 101, 101, 82, 90, 71, 88)  
Vector(58, 66, 58, 83, 63, 72, 51, 74, 68, 47, 72, 69, 54, 65, 61, 88, 78, 89, 67, 68, 77, 35, 54, 59, 98, 85, 68, 67, 61, 74)  
Vector(60, 66, 65, 67, 73, 73, 65, 46, 83, 72, 51, 73, 82, 58, 68, 69, 78, 89, 60, 88, 61, 77, 45, 64, 62, 94, 82, 70, 72, 78, 72)  
Vector(42, 50, 91, 77, 62, 76, 61, 70, 60, 84, 66, 36, 69, 70, 45, 67, 77, 97, 80, 82, 70, 58, 73, 44, 64, 55, 77, 97, 74, 74, 69, 77)  
Vector(56, 64, 74, 73, 60, 85, 83, 82, 58, 81, 65, 50, 70, 83, 58, 80, 89, 82, 92, 66, 83, 63, 69, 58, 70, 67, 101, 95, 74, 82, 78, 85)  
Vector(52, 65, 69, 58, 73, 87, 71, 57, 41, 77, 60, 54, 79, 73, 49, 64, 68, 87, 79, 69, 81, 58, 66, 35, 64, 67, 76, 78, 64, 69, 64, 72)  
Vector(53, 59, 64, 65, 70, 86, 74, 72, 56, 70, 63, 58, 76, 91, 46, 68, 77, 96, 85, 66, 74, 64, 45, 72, 64, 90, 89, 66, 57, 66, 74)  
Vector(57, 48, 56, 61, 56, 68, 66, 61, 42, 84, 57, 51, 68, 68, 51, 55, 63, 73, 74, 65, 63, 56, 63, 42, 52, 54, 86, 84, 55, 57, 55, 62)  
Vector(34, 35, 54, 47, 51, 47, 48, 49, 33, 57, 50, 35, 48, 40, 37, 45, 45, 57, 59, 64, 54, 48, 53, 31, 32, 38, 59, 66, 45, 53, 45, 53)  
Vector(45, 64, 59, 52, 74, 80, 66, 85, 51, 77, 72, 46, 60, 59, 55, 60, 63, 74, 73, 66, 63, 54, 79, 27, 60, 48, 96, 85, 56, 65, 53, 68)  
Vector(28, 41, 66, 54, 59, 57, 45, 52, 46, 64, 49, 25, 50, 52, 26, 57, 52, 72, 51, 55, 58, 45, 58, 33, 44, 51, 64, 79, 63, 64, 62, 64)  
Vector(43, 53, 74, 70, 79, 82, 65, 62, 68, 73, 64, 39, 73, 72, 66, 64, 78, 81, 76, 67, 81, 74, 78, 42, 55, 63, 99, 96, 78, 91, 65, 76)  
Vector(54, 55, 73, 61, 76, 74, 56, 75, 51, 76, 63, 49, 62, 72, 42, 64, 64, 76, 83, 77, 67, 46, 83, 50, 64, 59, 86, 87, 63, 66, 73, 71)  
Vector(57, 64, 87, 70, 90, 81, 74, 86, 63, 83, 50, 88, 74, 63, 71, 66, 99, 81, 97, 74, 63, 97, 38, 64, 70, 105, 94, 86, 82, 73, 84)  
Vector(63, 59, 80, 66, 73, 84, 78, 70, 68, 89, 73, 54, 82, 97, 53, 82, 81, 97, 99, 73, 95, 73, 57, 68, 65, 101, 99, 81, 87, 71, 74)  
Vector(53, 52, 80, 59, 75, 76, 64, 74, 48, 44, 55, 43, 71, 69, 54, 59, 71, 95, 78, 73, 52, 58, 68, 38, 62, 51, 80, 85, 70, 63, 59, 65)  
Vector(60, 64, 82, 60, 76, 78, 63, 63, 73, 66, 54, 74, 79, 55, 74, 75, 95, 77, 58, 92, 63, 75, 53, 73, 64, 91, 83, 87, 71, 77, 65)  
Vector(65, 79, 81, 70, 93, 92, 75, 93, 63, 88, 82, 59, 87, 77, 62, 79, 79, 96, 82, 78, 82, 61, 87, 39, 72, 78, 113, 98, 91, 74, 75, 78)  
Vector(41, 43, 78, 59, 58, 66, 53, 54, 57, 73, 56, 35, 70, 63, 37, 66, 50, 78, 65, 66, 67, 49, 45, 57, 59, 80, 87, 75, 61, 72, 78, 70)  
Vector(67, 58, 86, 81, 76, 63, 70, 92, 62, 80, 66, 51, 70, 75, 62, 70, 78, 95, 75, 80, 66, 53, 90, 49, 67, 53, 102, 95, 74, 83, 78, 71)  
Vector(44, 63, 68, 63, 67, 87, 78, 74, 49, 76, 68, 52, 80, 80, 54, 74, 75, 87, 81, 73, 87, 66, 75, 56, 67, 70, 98, 85, 80, 77, 74, 80)
```

Tiempo de ejecución de recursiva Paralelo: 10.9159

Tiempo de ejecución de recursiva Paralelo: 5.6844

Aceleracion: 1.920325803954683

Multiplicacion recursiva 32x32

Matriz 1:

```
Vector(0, 3, 1, 3, 1, 1, 0, 3, 2, 2, 3, 3, 3, 0, 1, 0, 3, 1, 3, 2, 0, 1, 1, 3, 1, 2, 3, 1, 0, 2, 2, 0)
Vector(3, 2, 1, 3, 2, 1, 1, 0, 0, 2, 2, 2, 0, 0, 3, 1, 3, 1, 3, 2, 1, 2, 2, 2, 3, 1, 0, 0, 2, 2, 1)
Vector(2, 2, 1, 3, 0, 2, 0, 1, 2, 2, 1, 0, 0, 3, 3, 3, 3, 0, 0, 2, 3, 3, 1, 1, 1, 3, 3, 1, 1, 0, 0, 1)
Vector(1, 3, 0, 0, 1, 2, 3, 3, 2, 3, 3, 1, 3, 3, 0, 0, 0, 3, 2, 2, 2, 1, 2, 2, 0, 2, 0, 0, 1, 0, 1, 2, 2)
Vector(1, 0, 1, 3, 2, 2, 1, 2, 2, 2, 1, 2, 0, 0, 0, 2, 0, 2, 1, 2, 3, 2, 0, 1, 1, 1, 0, 3, 3, 2, 1, 1)
Vector(1, 3, 2, 3, 3, 2, 1, 2, 1, 2, 3, 3, 2, 3, 0, 2, 1, 3, 1, 2, 3, 0, 1, 0, 3, 2, 2, 0, 1, 1, 3)
Vector(0, 3, 0, 1, 1, 2, 0, 2, 2, 2, 3, 1, 1, 1, 0, 0, 2, 0, 2, 1, 0, 1, 2, 1, 1, 3, 2, 0, 3, 2, 2)
Vector(1, 0, 2, 0, 1, 3, 3, 0, 1, 1, 1, 2, 1, 2, 2, 2, 0, 3, 0, 1, 1, 3, 0, 1, 3, 2, 3, 0, 0, 1, 3, 0)
Vector(3, 1, 3, 2, 1, 2, 2, 2, 0, 1, 2, 3, 1, 1, 1, 0, 1, 0, 3, 3, 0, 2, 2, 3, 2, 3, 2, 1, 1, 2, 0, 3)
Vector(0, 3, 1, 2, 2, 1, 2, 2, 1, 3, 0, 3, 1, 2, 0, 3, 0, 3, 1, 3, 0, 3, 2, 2, 2, 1, 1, 1, 2, 2, 3, 2)
Vector(0, 3, 2, 3, 0, 2, 1, 0, 2, 2, 0, 3, 3, 0, 0, 2, 3, 0, 3, 0, 0, 2, 0, 1, 3, 2, 0, 1, 0, 3, 3, 3)
Vector(3, 2, 0, 3, 0, 1, 3, 1, 1, 0, 1, 2, 1, 0, 2, 0, 2, 3, 3, 2, 3, 3, 2, 2, 3, 2, 0, 2, 3, 2, 2)
Vector(1, 3, 2, 1, 2, 3, 2, 1, 0, 1, 1, 2, 0, 3, 2, 3, 0, 2, 0, 1, 3, 1, 1, 1, 2, 1, 1, 3, 0, 2, 1, 2)
Vector(0, 2, 2, 3, 0, 3, 3, 1, 0, 3, 2, 3, 0, 3, 2, 2, 0, 0, 0, 1, 3, 3, 1, 2, 1, 1, 1, 0, 3, 1, 1)
Vector(2, 1, 3, 1, 0, 0, 0, 0, 3, 1, 1, 2, 1, 3, 3, 2, 0, 3, 2, 3, 0, 0, 3, 1, 2, 1, 0, 3, 2, 3, 2)
Vector(2, 3, 2, 3, 3, 0, 0, 0, 3, 0, 3, 0, 3, 1, 1, 3, 3, 0, 0, 1, 1, 2, 3, 3, 0, 2, 1, 1, 3, 1, 1, 3)
Vector(1, 3, 3, 3, 0, 2, 3, 1, 0, 2, 1, 0, 3, 1, 0, 0, 1, 2, 0, 2, 3, 2, 3, 0, 0, 0, 2, 1, 2, 2, 3, 3)
Vector(3, 0, 3, 3, 1, 3, 1, 1, 2, 1, 3, 2, 1, 2, 0, 2, 2, 2, 3, 2, 0, 3, 0, 0, 1, 2, 2, 3, 1, 0, 0, 1)
Vector(1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 1, 2, 2, 1, 1, 2, 2, 1, 2, 1, 3, 0, 3, 2, 3)
Vector(1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 2, 0, 0, 0, 2, 3, 1, 1, 0, 1, 0, 3, 2, 1, 2, 1, 1, 0, 1, 0, 2, 3, 1)
Vector(0, 3, 0, 0, 1, 2, 3, 3, 0, 2, 0, 0, 3, 2, 0, 2, 1, 0, 1, 1, 3, 3, 0, 2, 1, 2, 2, 3, 1, 3, 0, 2)
Vector(2, 2, 2, 3, 1, 0, 3, 0, 0, 0, 1, 2, 2, 2, 2, 0, 3, 0, 0, 1, 2, 0, 1, 0, 2, 1, 1, 0, 3, 0, 1, 0)
Vector(0, 2, 1, 1, 3, 0, 1, 1, 1, 1, 2, 3, 1, 3, 3, 3, 3, 0, 0, 1, 2, 1, 2, 0, 2, 2, 3, 1, 3, 2, 0, 1)
Vector(0, 3, 2, 2, 3, 2, 3, 2, 0, 1, 0, 0, 3, 1, 0, 0, 1, 3, 1, 3, 2, 2, 2, 0, 0, 0, 0, 1, 1)
Vector(0, 3, 1, 3, 3, 3, 2, 0, 3, 1, 0, 0, 2, 2, 2, 0, 1, 1, 1, 2, 0, 1, 3, 1, 2, 2, 2, 3, 3, 2, 2, 3)
Vector(0, 3, 2, 3, 0, 1, 0, 1, 3, 1, 3, 1, 0, 0, 3, 2, 2, 3, 0, 2, 2, 3, 0, 3, 2, 2, 1, 2, 0, 3, 1, 0)
Vector(3, 0, 3, 2, 3, 0, 1, 2, 3, 3, 1, 2, 0, 2, 1, 2, 0, 0, 1, 0, 1, 0, 2, 1, 3, 1, 1, 2, 1, 1, 1, 2)
Vector(1, 3, 2, 3, 2, 0, 0, 2, 3, 2, 2, 1, 1, 0, 3, 0, 1, 2, 2, 2, 0, 3, 1, 1, 2, 3, 2, 1, 2, 1, 3, 0)
Vector(2, 3, 0, 2, 2, 3, 2, 3, 2, 2, 1, 0, 2, 3, 0, 3, 2, 3, 0, 1, 1, 3, 0, 0, 0, 2, 3, 3, 2, 2, 1, 3, 1)
Vector(1, 2, 0, 3, 3, 1, 2, 0, 1, 1, 3, 0, 3, 1, 2, 2, 0, 1, 2, 2, 1, 0, 0, 2, 1, 0, 3, 0, 2, 0, 3, 1)
Vector(0, 0, 2, 2, 3, 0, 3, 2, 3, 1, 2, 2, 2, 0, 3, 3, 1, 0, 1, 0, 0, 0, 1, 2, 2, 3, 2, 3, 3, 0, 1, 3)
Vector(1, 3, 2, 3, 2, 0, 2, 0, 1, 2, 3, 2, 1, 2, 2, 3, 1, 2, 1, 1, 2, 2, 1, 0, 3, 1, 1, 1, 3, 0, 2)
```

Matriz 2:

```
Vector(0, 2, 3, 1, 0, 2, 2, 0, 0, 3, 1, 1, 1, 0, 2, 3, 3, 2, 1, 0, 2, 0, 1, 1, 1, 1, 3, 3, 0, 0, 3)
Vector(0, 2, 1, 0, 1, 2, 3, 0, 1, 2, 2, 1, 1, 1, 0, 3, 1, 1, 0, 0, 3, 1, 0, 1, 2, 2, 3, 1, 3, 2, 1, 2)
Vector(2, 1, 1, 3, 2, 3, 2, 0, 0, 0, 0, 1, 0, 3, 1, 1, 3, 3, 2, 2, 1, 1, 0, 1, 3, 2, 0, 2, 0, 0, 3, 0)
Vector(0, 2, 2, 0, 1, 1, 2, 1, 1, 0, 3, 1, 1, 3, 3, 0, 0, 3, 1, 3, 2, 0, 3, 1, 1, 1, 1, 1, 2, 1, 0, 3, 2, 3, 1)
Vector(0, 2, 2, 1, 3, 3, 1, 2, 3, 1, 0, 0, 1, 2, 3, 1, 0, 1, 2, 2, 0, 3, 3, 2, 1, 0, 3, 2, 2, 1, 3, 0)
Vector(2, 2, 0, 1, 2, 2, 0, 1, 2, 1, 2, 0, 1, 3, 0, 0, 0, 1, 2, 3, 1, 0, 3, 0, 1, 0, 3, 0, 1, 0, 1, 2)
Vector(0, 1, 3, 2, 2, 0, 0, 3, 0, 3, 1, 1, 1, 0, 0, 1, 0, 2, 0, 3, 1, 1, 3, 0, 1, 1, 0, 3, 1, 2, 1, 2)
Vector(3, 2, 2, 1, 3, 0, 3, 2, 1, 3, 2, 3, 1, 0, 2, 2, 3, 2, 0, 0, 0, 2, 2, 0, 2, 3, 3, 2, 1, 1, 2, 0)
Vector(3, 2, 2, 0, 1, 2, 1, 3, 1, 0, 0, 1, 1, 0, 0, 2, 3, 3, 1, 0, 2, 0, 0, 1, 2, 0, 2, 1, 2, 1, 0, 1)
Vector(0, 1, 3, 0, 2, 2, 0, 2, 0, 0, 1, 0, 3, 2, 1, 1, 1, 3, 3, 3, 0, 0, 2, 0, 3, 0, 0, 0, 3, 1, 0, 2)
Vector(1, 0, 0, 3, 0, 0, 3, 0, 0, 1, 0, 3, 2, 3, 0, 2, 2, 1, 3, 2, 3, 1, 2, 3, 3, 3, 3, 1, 1, 3, 3)
Vector(1, 1, 2, 2, 2, 1, 0, 1, 3, 3, 0, 0, 2, 3, 0, 2, 1, 2, 1, 1, 1, 2, 1, 0, 1, 0, 3, 1, 3, 3, 0, 2)
Vector(2, 2, 0, 1, 0, 3, 0, 0, 1, 3, 0, 0, 1, 3, 0, 3, 1, 1, 2, 0, 1, 0, 1, 2, 3, 0, 2, 2, 1, 1, 3, 0)
Vector(0, 0, 0, 1, 3, 3, 1, 3, 1, 0, 1, 0, 2, 1, 0, 2, 2, 2, 0, 2, 1, 3, 0, 0, 1, 3, 1, 3, 1, 3, 0, 2)
Vector(1, 0, 3, 1, 1, 0, 1, 0, 1, 1, 2, 0, 1, 0, 2, 2, 2, 3, 1, 3, 3, 2, 2, 2, 0, 0, 1, 1, 3, 3, 2, 2)
Vector(1, 0, 1, 2, 1, 2, 1, 3, 0, 3, 0, 2, 0, 1, 1, 1, 1, 2, 3, 0, 0, 2, 1, 0, 3, 3, 2, 0, 3, 3)
Vector(0, 1, 3, 3, 0, 1, 0, 3, 2, 2, 1, 1, 2, 3, 3, 2, 3, 1, 3, 1, 2, 3, 0, 0, 1, 1, 0, 2, 1, 3, 0, 0)
Vector(2, 2, 1, 0, 0, 2, 0, 0, 1, 2, 0, 3, 3, 1, 2, 3, 0, 2, 1, 2, 2, 0, 2, 2, 3, 3, 0, 1, 0, 2, 0)
Vector(0, 0, 2, 1, 0, 3, 3, 1, 2, 3, 2, 2, 3, 0, 1, 0, 3, 1, 1, 2, 3, 3, 2, 1, 1, 3, 0, 1, 2, 1)
Vector(0, 0, 1, 0, 3, 2, 0, 0, 3, 2, 0, 0, 0, 2, 0, 0, 2, 2, 1, 0, 2, 3, 1, 0, 1, 2, 0, 3, 1, 0, 0, 2)
Vector(0, 2, 3, 1, 3, 3, 0, 2, 2, 2, 2, 2, 1, 0, 0, 2, 2, 3, 3, 2, 3, 1, 3, 1, 0, 0, 2, 3, 0, 3, 1, 2)
Vector(0, 1, 0, 0, 1, 2, 3, 2, 0, 0, 2, 1, 0, 0, 2, 0, 1, 0, 3, 0, 3, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 3)
Vector(3, 0, 0, 0, 2, 3, 2, 0, 0, 3, 1, 2, 3, 3, 2, 0, 1, 0, 3, 1, 2, 3, 2, 0, 3, 1, 2, 0, 3, 1, 2)
Vector(2, 0, 2, 2, 0, 0, 0, 3, 3, 3, 2, 0, 0, 2, 1, 3, 1, 1, 3, 3, 0, 0, 3, 1, 2, 1, 3, 3, 1, 2, 1, 3)
Vector(1, 0, 3, 2, 1, 0, 2, 3, 1, 2, 0, 1, 2, 3, 0, 0, 2, 3, 2, 2, 0, 3, 0, 2, 0, 2, 3, 1, 1, 2, 1)
Vector(2, 3, 1, 3, 3, 1, 1, 3, 2, 2, 3, 2, 0, 3, 0, 1, 3, 1, 0, 1, 2, 0, 3, 3, 1, 1, 3, 2, 2, 3, 0, 2)
Vector(2, 0, 0, 2, 2, 3, 1, 1, 2, 1, 3, 1, 3, 1, 3, 0, 2, 1, 1, 0, 2, 1, 3, 0, 0, 3, 3, 3, 2, 0, 2)
Vector(3, 2, 0, 0, 3, 1, 3, 2, 0, 1, 1, 3, 2, 3, 2, 2, 1, 3, 3, 2, 0, 2, 1, 0, 1, 0, 3, 2, 0, 1, 0, 0)
Vector(0, 2, 3, 2, 1, 0, 3, 3, 3, 1, 0, 0, 0, 0, 1, 1, 2, 2, 0, 0, 2, 1, 2, 0, 2, 1, 3, 2, 1, 3, 2, 3)
Vector(0, 3, 0, 0, 0, 3, 3, 1, 0, 1, 1, 0, 2, 1, 3, 0, 1, 1, 2, 0, 3, 3, 0, 0, 1, 1, 3, 1, 2, 0, 1, 0)
Vector(3, 1, 3, 3, 3, 0, 1, 0, 0, 3, 1, 1, 3, 1, 0, 1, 0, 2, 1, 3, 1, 1, 0, 0, 0, 2, 0, 2, 3, 1, 2, 0)
Vector(1, 1, 2, 3, 0, 1, 3, 3, 0, 1, 2, 2, 2, 0, 3, 0, 3, 2, 1, 3, 1, 0, 1, 0, 1, 0, 2, 0, 0, 2, 0, 3)
```

```

Resultado:
Vector(63, 60, 72, 65, 67, 79, 69, 63, 70, 89, 67, 54, 84, 98, 52, 82, 73, 87, 84, 55, 85, 68, 77, 51, 78, 68, 99, 90, 85, 75, 70, 62)
Vector(48, 62, 76, 63, 69, 87, 56, 61, 61, 93, 67, 48, 79, 88, 47, 71, 64, 85, 89, 74, 76, 58, 81, 54, 64, 68, 93, 92, 82, 63, 74, 74)
Vector(44, 53, 72, 59, 76, 81, 61, 81, 57, 63, 82, 44, 67, 67, 50, 66, 87, 87, 81, 68, 81, 58, 73, 38, 51, 62, 83, 86, 76, 79, 54, 84)
Vector(52, 57, 74, 58, 72, 82, 65, 71, 50, 85, 53, 57, 82, 77, 50, 71, 67, 86, 80, 72, 75, 68, 70, 40, 77, 63, 81, 88, 64, 68, 57, 70)
Vector(45, 65, 72, 46, 72, 68, 63, 73, 56, 64, 54, 47, 63, 63, 45, 60, 59, 89, 76, 64, 62, 52, 73, 30, 58, 45, 90, 71, 60, 54, 66, 64)
Vector(55, 70, 85, 77, 90, 101, 70, 80, 75, 88, 72, 59, 92, 93, 58, 82, 81, 104, 87, 85, 87, 67, 92, 52, 80, 78, 114, 101, 86, 84, 82, 82)
Vector(57, 53, 51, 47, 63, 67, 60, 52, 48, 63, 54, 48, 71, 66, 50, 59, 59, 72, 67, 59, 68, 49, 64, 34, 58, 58, 93, 69, 69, 56, 48, 59)
Vector(52, 45, 62, 63, 72, 70, 37, 68, 51, 67, 58, 34, 62, 68, 46, 49, 49, 73, 67, 77, 65, 41, 76, 31, 50, 53, 77, 78, 68, 52, 59, 64)
Vector(54, 58, 77, 76, 74, 84, 79, 74, 69, 90, 72, 53, 71, 95, 57, 63, 83, 94, 85, 71, 79, 74, 87, 48, 74, 63, 95, 100, 71, 75, 65, 85)
Vector(57, 62, 80, 66, 86, 87, 71, 81, 64, 98, 68, 46, 85, 76, 60, 69, 62, 91, 80, 77, 77, 65, 81, 36, 71, 73, 100, 87, 83, 70, 70, 82)
Vector(49, 62, 75, 67, 56, 77, 64, 72, 52, 76, 68, 38, 77, 84, 49, 63, 59, 84, 80, 65, 76, 49, 59, 34, 68, 41, 81, 69, 73, 63, 64, 57)
Vector(54, 72, 95, 70, 72, 89, 79, 77, 71, 105, 82, 64, 87, 81, 61, 76, 79, 100, 89, 73, 105, 76, 94, 56, 65, 69, 101, 101, 82, 90, 90, 71, 88)
Vector(50, 58, 66, 58, 83, 83, 62, 71, 52, 74, 68, 47, 72, 69, 54, 65, 61, 88, 78, 89, 67, 60, 77, 35, 54, 59, 98, 85, 68, 67, 61, 74)
Vector(62, 60, 65, 65, 67, 73, 73, 65, 46, 83, 72, 51, 73, 82, 58, 68, 69, 78, 89, 69, 88, 61, 77, 45, 64, 62, 94, 82, 70, 72, 78, 72)
Vector(42, 50, 91, 77, 62, 76, 61, 70, 60, 84, 66, 36, 69, 70, 45, 67, 77, 97, 80, 82, 70, 58, 73, 44, 64, 55, 77, 97, 74, 74, 69, 77)
Vector(56, 64, 74, 73, 60, 85, 83, 82, 58, 81, 65, 58, 70, 83, 58, 80, 89, 82, 92, 66, 83, 63, 69, 58, 70, 67, 101, 95, 74, 82, 78, 85)
Vector(52, 65, 69, 58, 73, 87, 71, 57, 41, 77, 60, 54, 79, 73, 49, 64, 68, 87, 79, 69, 81, 58, 66, 35, 64, 67, 76, 78, 64, 69, 64, 72)
Vector(53, 59, 64, 65, 70, 86, 74, 72, 56, 70, 63, 58, 76, 91, 46, 68, 77, 96, 85, 66, 74, 64, 45, 72, 64, 90, 89, 66, 57, 66, 74)
Vector(57, 48, 56, 61, 56, 68, 66, 61, 42, 84, 57, 51, 68, 68, 51, 55, 63, 73, 74, 65, 63, 56, 63, 42, 52, 54, 86, 84, 55, 57, 55, 62)
Vector(34, 35, 54, 47, 51, 47, 48, 49, 33, 57, 50, 35, 48, 40, 37, 45, 45, 57, 59, 64, 54, 48, 53, 31, 32, 38, 59, 66, 45, 53, 45, 53)
Vector(45, 64, 59, 52, 74, 80, 66, 85, 51, 77, 72, 46, 60, 59, 55, 60, 63, 74, 73, 66, 63, 54, 79, 27, 60, 48, 96, 85, 56, 65, 53, 68)
Vector(28, 41, 66, 54, 59, 57, 45, 52, 46, 64, 49, 25, 50, 52, 26, 57, 52, 72, 51, 55, 58, 45, 58, 33, 44, 51, 64, 79, 63, 64, 62, 64)
Vector(43, 53, 74, 70, 79, 82, 65, 82, 68, 73, 64, 39, 73, 72, 66, 64, 78, 81, 76, 67, 81, 74, 78, 42, 55, 63, 99, 96, 78, 91, 65, 76)
Vector(54, 55, 73, 61, 76, 74, 56, 75, 51, 76, 63, 49, 62, 72, 42, 64, 64, 76, 83, 77, 67, 46, 83, 50, 64, 59, 86, 87, 63, 66, 73, 71)
Vector(57, 64, 87, 70, 90, 81, 74, 86, 63, 83, 83, 50, 88, 74, 63, 71, 66, 99, 81, 97, 74, 63, 97, 38, 64, 70, 105, 94, 86, 82, 73, 84)
Vector(63, 59, 80, 66, 73, 84, 70, 70, 68, 89, 73, 54, 82, 97, 53, 82, 81, 97, 99, 73, 95, 73, 79, 57, 68, 65, 101, 99, 81, 87, 71, 74)
Vector(53, 52, 80, 59, 75, 76, 64, 74, 48, 64, 55, 43, 71, 69, 54, 59, 71, 95, 78, 73, 52, 58, 68, 38, 62, 51, 80, 85, 70, 63, 59, 65)
Vector(60, 64, 82, 60, 76, 76, 78, 63, 63, 73, 66, 54, 74, 79, 55, 74, 75, 95, 77, 58, 92, 63, 75, 53, 73, 64, 91, 83, 87, 71, 77, 65)
Vector(65, 79, 81, 70, 93, 92, 75, 93, 63, 68, 88, 82, 59, 87, 77, 62, 79, 79, 96, 82, 78, 82, 61, 87, 39, 72, 78, 113, 98, 91, 74, 75, 78)
Vector(41, 43, 70, 59, 58, 66, 53, 54, 57, 73, 56, 35, 70, 63, 37, 66, 50, 78, 65, 66, 67, 49, 76, 45, 57, 59, 80, 87, 75, 61, 72, 70)
Vector(67, 58, 86, 81, 76, 63, 70, 92, 62, 88, 66, 51, 70, 75, 62, 70, 78, 95, 75, 80, 66, 53, 90, 49, 67, 53, 102, 95, 74, 83, 78, 71)
Vector(44, 63, 68, 63, 67, 87, 78, 74, 49, 76, 68, 52, 80, 80, 54, 74, 75, 87, 81, 73, 87, 66, 75, 56, 67, 70, 98, 85, 80, 77, 74, 80)

Tiempo de ejecución de recursiva: 11.4926
Tiempo de ejecución de recursiva Paralelo: 5.1042
Aceleracion: 2.2515967242662907

```

En estos dos últimos casos, donde las matrices son las más grandes (32 x 32) es donde se ve la ganancia más significativa de la versión paralela, lo que confirma que entre más grande sean las matrices más eficiente es la versión paralela

• ANÁLISIS - Multiplicación Recursiva

¿Cuál de las implementaciones es más rápida?

Eso depende del tamaño de las matrices en los que se use la implementación, en caso de matrices con un alto número de datos la mejor opción es el algoritmo paralelizado, pero para matrices con bajo número de datos es mejor la recursiva.

¿De qué depende que la aceleración sea mejor?

Eso depende de factores como el tamaño de las matrices, en los casos probados, la aceleración del algoritmo fue mucho mejor para dimensiones grandes que cuando son pequeñas.

¿Puede caracterizar los casos en que es mejor usar la versión secuencial/paralela de cada algoritmo de multiplicación de matrices?

Se ha probado con 5 matrices de tamaño 2x2, 4x4, 8x8, 16x16, 32x32, la lista retorna el tiempo en ms del algoritmo de multiplicación recursiva de matrices,

el tiempo del algoritmo paralelo, la aceleración del algoritmo paralelo con respecto al secuencial, y la dimensión de las matrices tratadas.

Entendiendo esto, en los casos evaluados para las matrices 2x2 y 4x4, la mejor versión a usar es la secuencial, ya que los valores de la versión paralela fueron mayores.

Pero para los casos de las matrices 8x8, 16x16 y 32x32, la mejor versión a usar es la paralela, puesto que al ser matrices de tamaños más grandes, la parallelizada tiene tiempos de ejecución menores a la secuencial.

- **Funcion multiplicaciónMatrizEstandarPar**

La estrategia que se llevó a cabo para parallelizar esta función, fue con la abstracción de par (parallelización de datos). Implica dividir un conjunto de datos en partes más pequeñas que pueden ser procesadas simultáneamente por múltiples hilos lo que mejora el rendimiento al aprovechar múltiples núcleos de procesamiento. En este caso, se aplica a las filas de m1 y a las columnas de m2T. Aquí, m1.par y m2T.par crean colecciones paralelas de las filas de m1 y las columnas de m2T, respectivamente.

El producto punto entre una fila de m1 y una columna de m2T se calcula utilizando la función prodPuntoParD. Esta función parallelizada también utiliza la abstracción par para parallelizar el cálculo del producto punto entre dos vectores. Finalmente, el resultado de las operaciones se convierte en Vector para obtener una matriz de tipo Vector[Vector[Int]].

- **CASOS DE PRUEBA**

Los tiempos de ejecución fueron tomados en la medida ms.

Para la toma de medida del tiempo, se usó el whitWarmer(calentador), para evitar tomas incorrectas

- **Matrices de 2x2**

```

def pruebasMultiplicacionEstandar(): Unit = {
    println("1. Multiplicacion Estandar 2x2")
    println("\nMatriz 1: ")
    val E1 = mat1.matrizAlAzar(2, 2)
    E1.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E2 = mat1.matrizAlAzar(2, 2)
    E2.foreach(row => println(row))
    val comp = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E1, E2)
    println("Resultado: ")
    (mat1.multMatrizEstandarSec(E1, E2)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + comp(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp(1))
    println("Aceleracion: " + comp(2))

    println("\n2. Multiplicacion Estandar 2x2")
    println("Matriz 1: ")
    val E3 = mat1.matrizAlAzar(2, 2)
    E3.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E4 = mat1.matrizAlAzar(2, 2)
    E4.foreach(row => println(row))
    val comp1 = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E3, E4)
    println("Resultado: ")
    (mat1.multMatrizEstandarSec(E3, E4)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + comp1(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp1(1))
    println("Aceleracion: " + comp1(2))
}

```

- Para todos los casos, los resultados siempre serán correctos, ya que se ha evaluado todos estos. Por otro lado, en cuanto a la eficiencia no hay ganancia obtenida, en su lugar, la versión paralelizada no ofrece beneficios en este contexto debido al overhead asociado con el paralelismo, especialmente en tareas de pequeña escala como las matrices 2x2.

```

1. Multiplicacion Estandar 2x2

Matriz 1:
Vector(0, 0)
Vector(1, 0)

Matriz 2:
Vector(0, 1)
Vector(1, 1)
Unable to create a system terminal
Resultado:
Vector(0, 0)
Vector(0, 1)

Tiempo de ejecución de Multiplicación Estandar: 0.167799
Tiempo de ejecución de Multiplicación Estandar Paralelo: 2.378
Aceleracion: 0.07056307821698907

```

2.

```
2. Multiplicacion Estandar 2x2
```

```
Matriz 1:
```

```
Vector(1, 1)  
Vector(0, 1)
```

```
Matriz 2:
```

```
Vector(1, 1)  
Vector(0, 0)
```

```
Resultado:
```

```
Vector(1, 1)  
Vector(0, 0)
```

```
Tiempo de ejecución de Multiplicación Estandar: 0.0635
```

```
Tiempo de ejecución de Multiplicación Estandar Paralelo: 0.5145
```

```
Aceleracion: 0.12342079689018466
```

- Matrices 4x4

```
def pruebasMultiplicacionEstandar(): Unit = {
    println("1. Multiplicacion Estandar 4x4")
    println("\nMatriz 1: ")
    val E1 = mat1.matrizAlAzar(4, 4)
    E1.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E2 = mat1.matrizAlAzar(4, 4)
    E2.foreach(row => println(row))
    val comp = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E1, E2)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E1, E2)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + comp(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp(1))
    println("Aceleracion: " + comp(2))

    println("\n2. Multiplicacion Estandar 4x4")
    println("Matriz 1: ")
    val E3 = mat1.matrizAlAzar(4, 4)
    E3.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E4 = mat1.matrizAlAzar(4, 4)
    E4.foreach(row => println(row))
    val comp1 = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E3, E4)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E3, E4)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + comp1(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp1(1))
    println("Aceleracion: " + comp1(2))
```

1. Similar al caso anterior, el costo de gestionar hilos y dividir el trabajo supera cualquier posible ganancia en rendimiento para matrices de este tamaño.

1. Multiplicacion Estandar 4x4

```
Matriz 1:  
Vector(3, 0, 3, 0)  
Vector(2, 3, 1, 2)  
Vector(2, 0, 3, 2)  
Vector(0, 1, 3, 2)  
  
Matriz 2:  
Vector(3, 2, 2, 1)  
Vector(1, 0, 1, 0)  
Vector(1, 2, 2, 1)  
Vector(1, 2, 2, 2)  
Unable to create a system terminal  
  
Resultado:  
Vector(12, 12, 12, 6)  
Vector(12, 10, 13, 7)  
Vector(11, 14, 14, 9)  
Vector(6, 10, 11, 7)
```

```
Tiempo de ejecución de Multiplicación Estandar: 0.1027  
Tiempo de ejecución de Multiplicación Estandar Paralelo: 1.6604  
Aceleracion: 0.061852565646832086
```

```

2. Multiplicacion Estandar 2x2
Matriz 1:
Vector(2, 1, 0, 3)
Vector(2, 1, 3, 0)
Vector(1, 2, 2, 3)
Vector(1, 1, 0, 2)

Matriz 2:
Vector(1, 3, 1, 3)
Vector(3, 1, 2, 1)
Vector(0, 0, 0, 0)
Vector(1, 0, 2, 3)

Resultado:
Vector(8, 7, 10, 16)
Vector(5, 7, 4, 7)
Vector(10, 5, 11, 14)
Vector(6, 4, 7, 10)

Tiempo de ejecución de Multiplicación Estandar: 0.040201
Tiempo de ejecución de Multiplicación Estandar Paralelo: 0.4539
Aceleracion: 0.08856796651244767

```

2.

- Matrices 8x8

```

def pruebasMultiplicacionEstandar(): Unit = {
  println("1. Multiplicacion Estandar 8x8")
  println("\nMatriz 1: ")
  val E1 = mat1.matrizAlAzar(8, 8)
  E1.foreach(row => println(row))
  println("\nMatriz 2: ")
  val E2 = mat1.matrizAlAzar(8, 8)
  E2.foreach(row => println(row))
  val comp = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E1, E2)
  println("\nResultado: ")
  (mat1.multMatrizEstandarSec(E1, E2)).foreach(row => println(row))
  println("\nTiempo de ejecución de Multiplicación Estandar: " + comp(0))
  println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp(1))
  println("Aceleracion: " + comp(2))

  println("\n2. Multiplicacion Estandar 8x8")
  println("Matriz 1: ")
  val E3 = mat1.matrizAlAzar(8, 8)
  E3.foreach(row => println(row))
  println("\nMatriz 2: ")
  val E4 = mat1.matrizAlAzar(8, 8)
  E4.foreach(row => println(row))
  val comp1 = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E3, E4)
  println("\nResultado: ")
  (mat1.multMatrizEstandarSec(E3, E4)).foreach(row => println(row))
  println("\nTiempo de ejecución de Multiplicación Estandar: " + comp1(0))
  println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp1(1))
  println("Aceleracion: " + comp1(2))
}

```

1. Se obtienen matrices correctas en ambas implementaciones. El costo de gestionar hilos y dividir el trabajo supera cualquier posible ganancia en rendimiento para matrices de este tamaño.

1. Multiplicacion Estandar 8x8

Matriz 1:

```
Vector(6, 0, 6, 1, 0, 1, 3, 3)
Vector(7, 4, 5, 0, 6, 7, 7, 1)
Vector(5, 4, 4, 0, 4, 7, 6, 3)
Vector(1, 5, 0, 4, 5, 1, 6, 5)
Vector(0, 3, 5, 3, 4, 0, 2, 3)
Vector(2, 4, 1, 5, 7, 7, 2, 3)
Vector(0, 7, 3, 1, 2, 6, 6, 2)
Vector(4, 4, 7, 2, 3, 1, 3, 3)
```

Matriz 2:

```
Vector(2, 7, 2, 0, 3, 5, 6, 6)
Vector(5, 7, 7, 0, 6, 1, 6, 0)
Vector(2, 5, 3, 7, 2, 1, 6, 6)
Vector(4, 1, 5, 3, 7, 2, 4, 4)
Vector(6, 3, 6, 5, 5, 2, 5, 7)
Vector(7, 1, 3, 1, 3, 0, 6, 6)
Vector(1, 4, 0, 5, 5, 6, 4, 7)
Vector(4, 7, 5, 4, 6, 6, 7, 5)
Unable to create a system terminal
```

Resultado:

```
Vector(50, 107, 53, 73, 73, 74, 115, 118)
Vector(140, 162, 119, 111, 147, 104, 203, 210)
Vector(129, 147, 110, 97, 136, 95, 185, 181)
Vector(106, 121, 115, 88, 149, 94, 142, 130)
Vector(75, 90, 90, 86, 97, 52, 109, 99)
Vector(151, 109, 138, 86, 151, 69, 168, 158)
Vector(113, 115, 103, 78, 125, 64, 148, 124)
Vector(90, 136, 103, 98, 115, 77, 152, 137)
```

Tiempo de ejecución de Multiplicación Estandar: 0.1698

Tiempo de ejecución de Multiplicación Estandar Paralelo: 2.3388

Aceleracion: 0.07260133401744484

2.

2. Multiplicacion Estandar 8x8

Matriz 1:

```
Vector(6, 5, 3, 2, 6, 6, 7, 2)
Vector(5, 7, 1, 7, 3, 1, 3, 0)
Vector(5, 6, 1, 2, 1, 2, 6, 5)
Vector(6, 0, 7, 1, 1, 5, 1, 1)
Vector(3, 4, 2, 0, 4, 1, 5, 0)
Vector(0, 5, 5, 2, 2, 7, 1, 2)
Vector(7, 5, 6, 2, 7, 3, 2, 6)
Vector(3, 4, 3, 6, 3, 4, 3, 6)
```

Matriz 2:

```
Vector(6, 4, 3, 0, 3, 1, 7, 4)
Vector(3, 4, 4, 1, 7, 2, 2, 6)
Vector(3, 2, 3, 2, 3, 4, 7, 1)
Vector(7, 6, 3, 7, 5, 0, 2, 7)
Vector(1, 5, 5, 7, 0, 6, 5, 4)
Vector(4, 4, 2, 5, 6, 1, 5, 1)
Vector(5, 6, 3, 2, 6, 2, 1, 7)
Vector(0, 5, 0, 6, 4, 0, 6, 0)
```

Resultado:

```
Vector(139, 168, 116, 123, 158, 84, 156, 150)
Vector(125, 129, 93, 90, 126, 48, 93, 146)
Vector(104, 132, 75, 81, 138, 41, 109, 119)
Vector(90, 80, 60, 61, 84, 47, 130, 54)
Vector(69, 86, 68, 51, 79, 54, 73, 90)
Vector(79, 96, 68, 92, 116, 51, 107, 71)
Vector(118, 161, 112, 135, 138, 90, 193, 123)
Vector(115, 149, 84, 135, 142, 51, 136, 118)
```

Tiempo de ejecución de Multiplicación Estandar: 0.0926

Tiempo de ejecución de Multiplicación Estandar Paralelo: 1.1146

Aceleracion: 0.0830791315270052

- **Matrices 16x16:** en este caso tampoco hay ganancias

```

def pruebasMultiplicacionEstandar(): Unit = {
    println("1. Multiplicacion Estandar 16x16")
    println("\nMatriz 1: ")
    val E1 = mat1.matrizAlAzar(16, 16)
    E1.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E2 = mat1.matrizAlAzar(16, 16)
    E2.foreach(row => println(row))
    val comp = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E1, E2)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E1, E2)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + comp(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp(1))
    println("Aceleracion: " + comp(2))

    println("\n2. Multiplicacion Estandar 16x16")
    println("Matriz 1: ")
    val E3 = mat1.matrizAlAzar(16, 16)
    E3.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E4 = mat1.matrizAlAzar(16, 16)
    E4.foreach(row => println(row))
    val compl = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E3, E4)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E3, E4)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + compl(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + compl(1))
    println("Aceleracion: " + compl(2))
}

```

1. En este caso no hay ganancias en cuanto a la eficiencia.

Matriz 1:

```

Vector(0, 15, 0, 15, 12, 0, 1, 7, 12, 12, 7, 1, 7, 4, 8, 2)
Vector(4, 12, 5, 4, 12, 9, 6, 14, 13, 5, 11, 9, 4, 13, 1, 3)
Vector(10, 14, 5, 5, 1, 9, 8, 14, 5, 2, 0, 15, 1, 4, 1, 15)
Vector(10, 6, 12, 12, 13, 3, 15, 13, 0, 3, 3, 2, 12, 4, 13, 3)
Vector(3, 2, 3, 5, 13, 0, 13, 0, 2, 7, 5, 13, 12, 8, 11, 6)
Vector(3, 7, 7, 4, 2, 3, 14, 8, 14, 10, 11, 13, 5, 2, 14, 6)
Vector(6, 6, 8, 2, 2, 0, 10, 0, 8, 2, 6, 8, 4, 14, 8, 1)
Vector(15, 7, 14, 5, 4, 2, 6, 3, 14, 1, 2, 5, 3, 6, 14, 13)
Vector(10, 9, 6, 5, 2, 7, 7, 13, 9, 9, 15, 14, 5, 4, 2, 14)
Vector(15, 15, 8, 0, 1, 6, 7, 5, 14, 5, 9, 4, 3, 3, 14, 1)
Vector(10, 7, 9, 5, 8, 4, 11, 13, 15, 2, 10, 7, 10, 3, 4, 0)
Vector(7, 4, 13, 3, 8, 4, 10, 14, 11, 7, 14, 2, 13, 8, 0, 12)
Vector(5, 4, 5, 2, 1, 0, 1, 11, 12, 9, 10, 12, 8, 8, 5, 14)
Vector(9, 15, 14, 15, 4, 13, 3, 11, 14, 0, 11, 10, 3, 8, 11, 5)
Vector(5, 12, 5, 0, 7, 14, 9, 2, 15, 2, 4, 9, 2, 2, 6, 5)
Vector(1, 1, 2, 12, 6, 10, 13, 0, 1, 0, 3, 6, 2, 3, 3, 11)

```

Matriz 2:

```

Vector(3, 12, 10, 9, 0, 11, 7, 8, 6, 1, 5, 0, 0, 13, 2, 3)
Vector(6, 12, 2, 10, 13, 3, 9, 2, 11, 9, 3, 0, 5, 0, 3, 15)
Vector(8, 3, 4, 4, 9, 11, 6, 13, 4, 13, 15, 7, 1, 15, 0, 9)
Vector(14, 14, 12, 10, 5, 10, 14, 13, 15, 3, 9, 10, 0, 9, 2, 4)
Vector(4, 6, 13, 9, 7, 10, 6, 5, 12, 15, 0, 4, 15, 7, 2, 9)
Vector(9, 4, 15, 10, 13, 5, 12, 7, 14, 7, 3, 4, 0, 14, 13, 8)
Vector(3, 15, 6, 15, 5, 4, 15, 8, 8, 15, 2, 6, 5, 9, 11, 15)
Vector(14, 6, 9, 10, 12, 4, 11, 14, 13, 10, 9, 5, 5, 14, 0, 4)
Vector(12, 1, 6, 8, 5, 15, 14, 14, 8, 7, 0, 9, 6, 12, 5, 7)
Vector(11, 15, 9, 13, 5, 12, 15, 0, 15, 2, 7, 5, 0, 2, 12, 1)
Vector(12, 1, 11, 3, 3, 9, 5, 14, 0, 4, 7, 15, 14, 12, 9, 2)
Vector(10, 2, 9, 7, 7, 3, 0, 5, 3, 10, 2, 12, 0, 8, 0, 3)
Vector(6, 10, 5, 4, 7, 8, 6, 9, 7, 11, 0, 11, 10, 4, 3, 8)
Vector(13, 5, 14, 1, 4, 2, 4, 4, 13, 0, 13, 4, 7, 1, 8, 7)
Vector(8, 2, 14, 5, 8, 2, 10, 14, 11, 6, 14, 3, 5, 1, 2, 1)
Vector(9, 0, 6, 2, 3, 3, 2, 13, 2, 5, 0, 7, 6, 2, 15, 4)

```

Resultado:

```
Vector(995, 826, 916, 849, 726, 823, 1034, 879, 1105, 726, 544, 655, 615, 630, 476, 649)
Vector(1184, 769, 1144, 949, 903, 870, 1050, 1062, 1156, 962, 645, 800, 731, 1029, 646, 860)
Vector(962, 716, 878, 873, 800, 607, 846, 945, 908, 835, 491, 613, 363, 874, 590, 741)
Vector(980, 1007, 1119, 1005, 858, 845, 1130, 1189, 1177, 1090, 792, 715, 650, 997, 494, 858)
Vector(815, 732, 962, 757, 607, 654, 784, 836, 868, 871, 505, 724, 622, 618, 542, 655)
Vector(1114, 768, 1044, 973, 810, 852, 1119, 1178, 993, 971, 681, 877, 590, 946, 670, 738)
Vector(733, 544, 752, 570, 499, 553, 668, 733, 686, 611, 572, 543, 434, 596, 421, 592)
Vector(941, 645, 958, 767, 674, 834, 922, 1170, 882, 809, 701, 634, 486, 894, 541, 700)
Vector(1227, 800, 1111, 971, 834, 902, 1031, 1215, 991, 916, 640, 929, 617, 1094, 778, 740)
Vector(897, 728, 925, 860, 745, 810, 1014, 1017, 919, 772, 658, 579, 495, 888, 519, 720)
Vector(1020, 800, 986, 941, 794, 930, 1063, 1157, 981, 1001, 595, 805, 655, 1110, 491, 810)
Vector(1172, 815, 1066, 906, 815, 995, 1068, 1281, 1015, 1029, 682, 926, 793, 1125, 758, 846)
Vector(1069, 549, 897, 679, 640, 746, 782, 1030, 797, 702, 562, 800, 557, 779, 605, 526)
Vector(1390, 864, 1326, 1058, 1069, 1025, 1259, 1446, 1295, 1043, 934, 933, 628, 1274, 642, 928)
Vector(807, 573, 852, 814, 746, 689, 890, 848, 853, 834, 384, 579, 465, 824, 585, 756)
Vector(628, 526, 723, 603, 467, 439, 652, 697, 655, 601, 316, 522, 332, 582, 547, 528)
```

Tiempo de ejecución de Multiplicación Estandar: 0.6521

Tiempo de ejecución de Multiplicación Estandar Paralelo: 2.9584

Aceleracion: 0.2204232017306652

2.

Matriz 2:

```
Vector(3, 1, 2, 3, 11, 2, 11, 2, 15, 1, 4, 4, 1, 2, 4, 14)
Vector(10, 8, 7, 14, 1, 1, 6, 1, 5, 3, 2, 0, 3, 14, 11, 11)
Vector(5, 8, 11, 10, 10, 7, 4, 6, 12, 6, 15, 1, 6, 7, 13, 6)
Vector(4, 0, 6, 9, 5, 10, 8, 3, 1, 4, 9, 11, 0, 13, 11, 8)
Vector(11, 12, 7, 1, 7, 6, 7, 0, 11, 10, 18, 8, 2, 8, 3, 4)
Vector(6, 2, 10, 6, 15, 11, 11, 7, 5, 14, 3, 12, 1, 13, 1, 3)
Vector(15, 15, 10, 11, 6, 2, 3, 12, 15, 1, 5, 4, 13, 13, 14, 2)
Vector(4, 5, 8, 10, 4, 13, 1, 6, 4, 9, 7, 3, 2, 12, 9, 9)
Vector(3, 1, 10, 5, 1, 15, 3, 5, 0, 0, 6, 8, 13, 12, 10, 14)
Vector(6, 2, 2, 15, 11, 9, 4, 14, 13, 13, 9, 15, 12, 10, 14, 2)
Vector(13, 7, 7, 1, 6, 11, 15, 15, 13, 5, 1, 5, 8, 9, 6, 15)
Vector(12, 1, 10, 6, 11, 5, 13, 3, 15, 11, 7, 8, 4, 5, 10, 0)
Vector(4, 12, 1, 6, 3, 1, 11, 4, 10, 14, 7, 5, 14, 14, 4, 14)
Vector(15, 8, 4, 3, 10, 0, 1, 3, 1, 1, 2, 9, 5, 0, 15, 6)
Vector(14, 0, 15, 13, 3, 11, 3, 14, 6, 4, 7, 10, 2, 6, 1, 1)
Vector(9, 6, 3, 7, 10, 5, 3, 14, 14, 13, 6, 10, 9, 4, 4, 5)
```

2. Multiplicacion Estandar 16x16

```
Matriz 1:
Vector(1, 12, 10, 13, 5, 4, 2, 3, 9, 1, 9, 13, 3, 5, 1, 6)
Vector(10, 6, 11, 3, 0, 8, 13, 10, 11, 3, 12, 15, 4, 2, 9, 14)
Vector(5, 15, 10, 2, 5, 1, 1, 8, 13, 8, 3, 1, 2, 2, 2, 6)
Vector(4, 1, 6, 7, 2, 15, 2, 4, 15, 7, 9, 0, 9, 3, 6, 3)
Vector(14, 12, 8, 5, 4, 2, 15, 5, 15, 14, 14, 9, 13, 11, 2, 7)
Vector(1, 12, 6, 14, 11, 1, 13, 10, 15, 15, 3, 6, 12, 0, 15, 3)
Vector(10, 13, 5, 2, 7, 11, 9, 9, 15, 6, 3, 12, 7, 13, 7, 1)
Vector(14, 7, 12, 12, 8, 3, 6, 0, 12, 1, 4, 12, 2, 10, 7, 1)
Vector(3, 4, 6, 12, 9, 4, 15, 10, 0, 10, 11, 4, 14, 12, 15, 2)
Vector(11, 6, 4, 5, 8, 1, 4, 13, 6, 0, 6, 9, 12, 13, 9, 10)
Vector(15, 3, 14, 5, 2, 5, 12, 11, 5, 14, 7, 10, 14, 3, 9, 15)
Vector(2, 8, 9, 11, 8, 6, 14, 5, 6, 14, 6, 1, 10, 13, 11, 4)
Vector(13, 2, 7, 5, 5, 6, 3, 0, 8, 3, 9, 2, 13, 11, 8, 0)
Vector(1, 3, 15, 6, 3, 0, 7, 14, 10, 10, 5, 13, 3, 0, 13, 2)
Vector(2, 15, 9, 2, 9, 2, 8, 15, 7, 5, 12, 5, 0, 1, 6, 0)
Vector(8, 8, 15, 12, 11, 10, 6, 4, 3, 9, 14, 7, 1, 14, 6, 15)
```

Resultado:

```
Vector(807, 489, 734, 784, 646, 683, 709, 541, 775, 614, 625, 628, 519, 899, 883, 739)
Vector(1125, 671, 1038, 989, 959, 949, 886, 1039, 1302, 861, 792, 843, 836, 1124, 1847, 923)
Vector(589, 449, 591, 725, 496, 621, 435, 509, 655, 581, 568, 487, 539, 810, 773, 723)
Vector(634, 406, 686, 654, 658, 831, 656, 698, 666, 667, 565, 747, 627, 945, 668, 776)
Vector(1250, 912, 942, 1105, 1016, 907, 1007, 1046, 1428, 889, 863, 976, 1128, 1343, 1383, 1246)
Vector(1102, 758, 1048, 1244, 738, 1044, 770, 963, 1098, 899, 969, 994, 933, 1441, 1187, 940)
Vector(1101, 694, 969, 961, 900, 847, 814, 727, 1016, 778, 731, 871, 764, 1158, 1115, 944)
Vector(919, 522, 847, 751, 794, 709, 759, 556, 925, 527, 735, 731, 553, 871, 979, 824)
Vector(1230, 851, 935, 1045, 857, 850, 813, 997, 1136, 873, 833, 945, 816, 1212, 1102, 864)
Vector(1002, 679, 769, 778, 773, 784, 732, 700, 993, 764, 693, 751, 651, 917, 893, 923)
Vector(1113, 794, 940, 1163, 1098, 931, 930, 1134, 1514, 1086, 983, 988, 988, 1227, 1166, 1014)
Vector(1146, 886, 900, 1077, 865, 830, 708, 947, 1047, 819, 833, 958, 855, 1199, 1169, 832)
Vector(762, 525, 618, 577, 673, 586, 708, 583, 799, 549, 556, 652, 589, 772, 704, 839)
Vector(849, 482, 916, 953, 674, 900, 585, 807, 936, 705, 822, 703, 653, 968, 964, 637)
Vector(866, 615, 811, 816, 555, 762, 593, 666, 837, 575, 610, 512, 539, 988, 871, 756)
Vector(1276, 794, 986, 1008, 1159, 946, 935, 1028, 1303, 973, 908, 1049, 758, 1130, 1200, 957)
```

Tiempo de ejecución de Multiplicación Estandar: 0.2453

Tiempo de ejecución de Multiplicación Estandar Paralelo: 1.7712

Aceleracion: 0.13849367660343267

- **Matrices 32x32:** En este caso tampoco mejora

```
def pruebasMultiplicacionEstandar(): Unit = {
    println("1. Multiplicacion Estandar 32x32")
    println("\nMatriz 1: ")
    val E1 = mat1.matrizAlAzar(32, 32)
    E1.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E2 = mat1.matrizAlAzar(32, 32)
    E2.foreach(row => println(row))
    val comp = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E1, E2)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E1, E2)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + comp(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp(1))
    println("Aceleracion: " + comp(2))

    println("\n2. Multiplicacion Estandar 32x32")
    println("Matriz 1: ")
    val E3 = mat1.matrizAlAzar(32, 32)
    E3.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E4 = mat1.matrizAlAzar(32, 32)
    E4.foreach(row => println(row))
    val comp1 = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E3, E4)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E3, E4)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + comp1(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp1(1))
    println("Aceleracion: " + comp1(2))
```

1.

1. Multiplicacion Estandar 32x32

Resultado:

Vector(9, 8, 10, 8, 7, 9, 9, 10, 4, 7, 9, 11, 9, 8, 9, 8, 7, 7, 11, 9, 10, 8, 9, 7, 10, 12, 10, 7, 8, 8)
Vector(11, 10, 10, 6, 12, 9, 10, 11, 7, 10, 12, 10, 10, 6, 9, 7, 11, 7, 6, 10, 10, 7, 12, 10, 10, 11, 13, 11, 10, 8, 8, 11)
Vector(12, 9, 12, 9, 11, 8, 10, 10, 7, 11, 11, 11, 10, 8, 8, 6, 12, 13, 9, 8, 11, 6, 10, 14, 11, 8, 13, 13, 10, 10, 13)
Vector(12, 10, 14, 10, 10, 10, 11, 6, 10, 13, 13, 11, 8, 10, 8, 13, 11, 10, 9, 14, 10, 9, 12, 9, 8, 14, 15, 15, 10, 12, 12)
Vector(6, 8, 11, 9, 10, 9, 9, 7, 5, 5, 9, 9, 7, 4, 6, 6, 10, 11, 7, 7, 7, 8, 6, 8, 6, 9, 10, 18, 6, 8, 11)
Vector(5, 9, 9, 7, 9, 9, 10, 7, 4, 7, 10, 9, 11, 6, 11, 5, 8, 9, 6, 8, 8, 11, 8, 9, 5, 9, 12, 12, 12, 8, 7, 10)
Vector(9, 6, 8, 8, 9, 7, 9, 8, 5, 8, 9, 10, 12, 9, 9, 3, 7, 8, 9, 6, 9, 7, 7, 8, 7, 8, 12, 12, 11, 8, 7, 6)
Vector(12, 10, 13, 9, 13, 10, 13, 10, 9, 10, 15, 11, 12, 7, 9, 8, 16, 14, 7, 12, 14, 8, 11, 14, 13, 9, 15, 15, 15, 10, 10, 17)
Vector(10, 11, 11, 9, 11, 10, 12, 10, 5, 7, 11, 11, 10, 8, 9, 7, 10, 11, 8, 9, 10, 8, 10, 9, 10, 9, 11, 13, 12, 12, 10, 9)
Vector(10, 6, 8, 6, 7, 7, 5, 5, 2, 5, 7, 4, 7, 4, 4, 4, 8, 8, 5, 8, 7, 6, 6, 7, 7, 6, 8, 9, 9, 7, 4, 9)
Vector(8, 9, 9, 8, 10, 7, 12, 8, 7, 3, 9, 8, 8, 7, 2, 9, 10, 7, 5, 9, 7, 6, 9, 5, 11, 10, 8, 7, 8, 8)
Vector(10, 10, 11, 9, 12, 10, 14, 8, 8, 9, 12, 10, 11, 6, 9, 4, 13, 11, 7, 9, 11, 10, 9, 10, 10, 9, 12, 12, 13, 8, 9, 11)
Vector(7, 7, 8, 8, 9, 6, 8, 8, 4, 5, 6, 9, 6, 7, 8, 4, 12, 10, 7, 6, 10, 6, 8, 9, 7, 6, 11, 13, 9, 8, 7, 9)
Vector(13, 9, 12, 12, 12, 9, 9, 8, 4, 10, 13, 10, 12, 8, 7, 8, 11, 11, 10, 9, 10, 6, 11, 10, 11, 8, 13, 13, 10, 9, 11)
Vector(4, 6, 7, 7, 9, 6, 5, 3, 3, 7, 5, 6, 4, 6, 5, 8, 7, 3, 6, 4, 8, 7, 6, 4, 8, 8, 8, 4, 4, 7)
Vector(12, 10, 13, 9, 10, 9, 12, 10, 7, 7, 12, 11, 10, 8, 9, 9, 13, 9, 7, 8, 14, 10, 11, 11, 11, 7, 12, 13, 11, 8, 8, 13)
Vector(10, 10, 13, 12, 12, 10, 13, 11, 6, 6, 9, 12, 11, 8, 11, 7, 11, 12, 9, 7, 12, 9, 10, 11, 10, 8, 13, 16, 11, 8, 11, 11)
Vector(8, 9, 6, 5, 11, 6, 9, 9, 4, 7, 10, 8, 8, 10, 8, 4, 10, 7, 9, 8, 9, 6, 9, 9, 7, 8, 11, 10, 9, 7, 8, 9)
Vector(11, 7, 12, 11, 8, 7, 7, 10, 3, 8, 10, 10, 9, 8, 8, 8, 12, 12, 8, 8, 10, 9, 10, 10, 8, 7, 12, 13, 12, 9, 7, 11)
Vector(7, 9, 8, 5, 8, 6, 9, 6, 5, 9, 8, 8, 9, 6, 10, 3, 7, 7, 4, 11, 7, 6, 8, 9, 8, 8, 10, 10, 9, 6, 6, 7)
Vector(11, 10, 14, 11, 12, 8, 11, 11, 7, 8, 12, 12, 9, 9, 9, 8, 10, 10, 9, 8, 13, 6, 8, 11, 13, 6, 12, 15, 12, 9, 12, 12)
Vector(11, 7, 11, 6, 9, 10, 9, 8, 6, 8, 11, 9, 11, 5, 7, 7, 8, 8, 7, 10, 8, 8, 11, 10, 7, 8, 10, 9, 12, 8, 9, 10)
Vector(7, 7, 7, 5, 9, 4, 10, 4, 8, 6, 10, 6, 6, 4, 5, 4, 10, 7, 4, 7, 8, 2, 6, 8, 9, 5, 7, 7, 8, 5, 8, 11)
Vector(9, 9, 8, 5, 9, 7, 7, 8, 6, 8, 11, 7, 11, 5, 7, 5, 12, 10, 5, 8, 10, 7, 11, 9, 7, 12, 10, 9, 3, 10)
Vector(7, 10, 7, 6, 10, 6, 10, 6, 7, 8, 9, 7, 10, 3, 6, 2, 10, 11, 7, 7, 7, 8, 7, 9, 7, 9, 8, 11, 6, 6, 9)
Vector(9, 11, 11, 7, 10, 10, 14, 7, 8, 7, 14, 8, 11, 8, 10, 6, 10, 11, 6, 8, 13, 8, 8, 11, 11, 6, 12, 11, 12, 8, 9, 13)
Vector(11, 8, 11, 7, 5, 6, 7, 7, 3, 8, 9, 8, 8, 6, 7, 8, 8, 8, 5, 9, 9, 6, 10, 12, 8, 4, 10, 10, 10, 8, 8, 11)
Vector(11, 12, 13, 9, 14, 11, 13, 14, 8, 9, 13, 13, 11, 7, 9, 7, 13, 13, 7, 11, 12, 9, 11, 10, 13, 10, 14, 15, 12, 11, 9, 11)
Vector(11, 7, 10, 8, 6, 8, 7, 10, 2, 6, 7, 7, 7, 6, 7, 8, 9, 4, 7, 9, 7, 7, 11, 8, 5, 9, 11, 8, 8, 7, 9)
Vector(9, 7, 10, 5, 6, 9, 6, 7, 3, 7, 8, 8, 7, 5, 7, 5, 8, 6, 5, 8, 7, 9, 9, 7, 6, 8, 9, 8, 10, 6, 6, 9)
Vector(10, 12, 11, 10, 10, 8, 11, 9, 5, 9, 11, 11, 10, 8, 11, 6, 12, 10, 8, 9, 11, 9, 11, 10, 9, 8, 16, 13, 14, 8, 10, 10)
Vector(8, 9, 8, 8, 10, 8, 10, 7, 6, 7, 10, 8, 10, 5, 6, 5, 9, 11, 9, 6, 10, 7, 10, 7, 11, 6, 11, 10, 11, 6, 6, 11)

Tiempo de ejecución de Multiplicación Estandar: 1.2346

Tiempo de ejecución de Multiplicación Estandar Paralelo: 6.754

Aceleracion: 0.1827953805152502

2.

2. Multiplicacion Estandar 32x32

Matriz 1:

Resultado:
Vector(8, 9, 6, 8, 9, 4, 11, 9, 6, 9, 6, 12, 8, 10, 10, 10, 9, 7, 9, 11, 4, 5, 9, 6, 8, 4, 9, 7, 7, 5, 7, 5)
Vector(6, 7, 7, 10, 11, 5, 11, 12, 7, 10, 7, 13, 11, 11, 13, 8, 7, 9, 10, 14, 4, 9, 11, 7, 9, 7, 12, 8, 8, 7, 10, 8)
Vector(9, 7, 5, 12, 10, 5, 5, 8, 8, 7, 7, 11, 10, 8, 6, 8, 7, 5, 8, 11, 7, 6, 9, 6, 5, 5, 10, 7, 7, 7, 5)
Vector(10, 10, 9, 12, 10, 8, 10, 8, 9, 11, 12, 13, 13, 13, 11, 12, 10, 11, 11, 14, 7, 5, 12, 9, 12, 5, 13, 11, 7, 7, 10, 6)
Vector(6, 7, 8, 11, 11, 6, 9, 10, 8, 10, 10, 11, 11, 8, 14, 10, 6, 10, 11, 12, 2, 10, 12, 11, 8, 6, 14, 9, 8, 10, 12, 4)
Vector(12, 10, 11, 14, 13, 10, 12, 12, 10, 13, 11, 17, 14, 16, 12, 10, 12, 12, 13, 17, 7, 9, 14, 9, 11, 9, 12, 12, 10, 10, 11, 10)
Vector(10, 11, 8, 13, 13, 8, 11, 10, 14, 14, 12, 16, 15, 14, 11, 16, 12, 10, 10, 13, 10, 8, 10, 10, 9, 5, 17, 11, 9, 9, 13, 6)
Vector(8, 7, 8, 8, 7, 4, 7, 8, 5, 8, 7, 8, 8, 10, 6, 8, 4, 6, 5, 10, 2, 6, 6, 5, 7, 5, 9, 3, 5, 6, 6, 3)
Vector(9, 9, 7, 11, 11, 9, 5, 7, 9, 6, 7, 9, 11, 10, 9, 9, 6, 4, 9, 9, 6, 5, 8, 4, 9, 5, 10, 8, 8, 4, 6, 6)
Vector(9, 7, 8, 10, 12, 6, 8, 8, 6, 9, 11, 10, 10, 11, 7, 9, 9, 10, 5, 7, 8, 8, 10, 5, 12, 11, 8, 8, 9, 5)
Vector(7, 8, 6, 8, 7, 8, 8, 5, 11, 9, 8, 13, 12, 11, 6, 10, 9, 10, 9, 13, 8, 5, 7, 8, 4, 7, 7, 7, 9, 8, 4)
Vector(6, 5, 7, 9, 7, 8, 6, 7, 9, 9, 9, 10, 11, 10, 8, 9, 8, 8, 10, 5, 6, 7, 6, 6, 5, 7, 8, 4, 9, 8, 3)
Vector(10, 9, 9, 12, 9, 5, 9, 10, 9, 11, 12, 12, 11, 13, 9, 8, 9, 10, 14, 5, 7, 11, 8, 9, 6, 12, 8, 9, 7, 10, 6)
Vector(10, 8, 5, 11, 6, 6, 8, 6, 9, 9, 6, 12, 10, 11, 7, 10, 9, 11, 9, 13, 4, 7, 9, 9, 8, 8, 9, 5, 9, 9, 9, 4)
Vector(7, 7, 4, 8, 8, 6, 9, 8, 9, 11, 7, 11, 13, 9, 9, 8, 9, 11, 8, 11, 6, 6, 6, 8, 6, 5, 8, 9, 8, 7, 12, 7)
Vector(8, 10, 8, 10, 9, 8, 11, 7, 9, 9, 7, 12, 7, 12, 8, 9, 11, 8, 10, 13, 7, 5, 10, 8, 7, 6, 9, 6, 6, 6, 9, 7)
Vector(8, 7, 10, 10, 8, 8, 9, 10, 8, 9, 10, 13, 13, 10, 11, 10, 10, 11, 8, 15, 4, 7, 14, 8, 8, 8, 10, 9, 8, 10, 11, 6)
Vector(10, 9, 8, 13, 11, 8, 9, 7, 8, 8, 12, 11, 12, 9, 10, 9, 8, 11, 13, 5, 6, 12, 8, 11, 9, 10, 9, 8, 8, 7, 6)
Vector(10, 10, 6, 11, 10, 8, 10, 11, 9, 13, 8, 16, 16, 12, 10, 10, 11, 9, 14, 6, 7, 11, 7, 10, 7, 10, 12, 11, 9, 9, 7)
Vector(9, 8, 10, 10, 9, 8, 8, 9, 7, 8, 11, 11, 11, 11, 9, 11, 6, 9, 10, 5, 7, 9, 5, 9, 6, 8, 9, 8, 10, 7, 4)
Vector(9, 8, 8, 11, 10, 7, 13, 10, 12, 9, 9, 14, 15, 15, 11, 10, 11, 13, 8, 17, 6, 10, 13, 12, 10, 9, 12, 11, 9, 11, 9, 7)
Vector(7, 9, 8, 9, 9, 5, 11, 10, 9, 12, 9, 12, 13, 12, 10, 10, 9, 10, 8, 15, 4, 8, 11, 9, 9, 6, 9, 10, 8, 7, 9, 6)
Vector(6, 3, 4, 8, 7, 5, 6, 7, 7, 8, 7, 9, 8, 7, 7, 7, 7, 5, 8, 5, 3, 7, 7, 4, 3, 10, 7, 7, 5, 7, 5)
Vector(6, 6, 6, 6, 5, 5, 7, 6, 4, 7, 6, 8, 5, 6, 5, 7, 8, 5, 4, 6, 2, 4, 7, 4, 4, 7, 4, 7, 6, 8, 4)
Vector(10, 10, 7, 12, 9, 6, 8, 7, 9, 11, 11, 16, 10, 10, 9, 12, 9, 9, 11, 12, 8, 5, 9, 7, 8, 6, 12, 8, 11, 9, 8, 4)
Vector(10, 12, 8, 12, 10, 7, 12, 9, 10, 10, 11, 12, 12, 13, 10, 11, 11, 9, 9, 14, 7, 9, 8, 9, 10, 5, 10, 9, 9, 7, 9, 7)
Vector(7, 7, 4, 10, 8, 7, 9, 10, 10, 9, 7, 12, 11, 10, 10, 8, 7, 6, 7, 10, 6, 7, 7, 6, 5, 9, 8, 9, 7, 7, 5)
Vector(7, 7, 8, 6, 8, 8, 6, 3, 7, 4, 7, 8, 9, 10, 6, 6, 8, 6, 6, 10, 5, 6, 8, 6, 7, 7, 7, 7, 5, 7, 5, 4)
Vector(8, 5, 7, 9, 9, 5, 9, 9, 5, 6, 9, 9, 8, 9, 11, 6, 9, 9, 8, 12, 3, 7, 11, 7, 6, 8, 9, 7, 8, 7, 7)
Vector(6, 5, 6, 5, 5, 4, 11, 6, 8, 6, 6, 9, 6, 10, 5, 6, 9, 7, 6, 9, 5, 3, 7, 7, 6, 3, 7, 6, 4, 4, 7, 5)
Vector(9, 12, 6, 11, 11, 10, 11, 8, 11, 12, 10, 13, 12, 13, 10, 12, 11, 11, 10, 14, 6, 7, 11, 11, 9, 7, 11, 10, 7, 9, 12, 5)
Vector(11, 6, 9, 13, 10, 7, 9, 9, 8, 7, 10, 11, 12, 12, 10, 9, 7, 9, 11, 14, 5, 6, 11, 10, 8, 7, 11, 8, 7, 9, 7, 5)

```
Tiempo de ejecución de Multiplicación Estandar: 0.8594
Tiempo de ejecución de Multiplicación Estandar Paralelo: 4.0301
Aceleracion: 0.21324532890002731
```

- **Matrices 64x64:** Dado que las matrices son demasiado grandes para adjuntarlas y queden con buena calidad, por lo que solo se ponen los resultados. En cuanto a la eficiencia del algoritmo, no hay ganancias.

```
def pruebasMultiplicacionEstandar(): Unit = {
    println("1. Multiplicacion Estandar 64x64")
    println("\nMatriz 1: ")
    val E1 = mat1.matrizAlAzar(64, 2)
    E1.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E2 = mat1.matrizAlAzar(64, 2)
    E2.foreach(row => println(row))
    val comp = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E1, E2)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E1, E2)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + comp(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp(1))
    println("Aceleracion: " + comp(2))

    println("\n2. Multiplicacion Estandar 64x64")
    println("Matriz 1: ")
    val E3 = mat1.matrizAlAzar(64, 2)
    E3.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E4 = mat1.matrizAlAzar(64, 2)
    E4.foreach(row => println(row))
    val compl = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E3, E4)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E3, E4)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + compl(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + compl(1))
    println("Aceleracion: " + compl(2))
```

1.

```
Tiempo de ejecución de Multiplicación Estandar: 5.6728
Tiempo de ejecución de Multiplicación Estandar Paralelo: 21.6707
Aceleracion: 0.26177280844642764
```

2.

```
Tiempo de ejecución de Multiplicación Estandar: 5.0943
Tiempo de ejecución de Multiplicación Estandar Paralelo: 20.0548
Aceleracion: 0.2540189879729541
```

- **Matrices 128x128:** en este caso la aceleración es muy parecida a la anterior.

```
def pruebasMultiplicacionEstandar(): Unit = {
    println("1. Multiplicacion Estandar 128x128")
    println("\nMatriz 1: ")
    val E1 = mat1.matrizAlAzar(128, 2)
    E1.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E2 = mat1.matrizAlAzar(128, 2)
    E2.foreach(row => println(row))
    val comp = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E1, E2)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E1, E2)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + comp(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + comp(1))
    println("Aceleracion: " + comp(2))

    println("\n2. Multiplicacion Estandar 128x128")
    println("Matriz 1: ")
    val E3 = mat1.matrizAlAzar(128, 2)
    E3.foreach(row => println(row))
    println("\nMatriz 2: ")
    val E4 = mat1.matrizAlAzar(128, 2)
    E4.foreach(row => println(row))
    val compl = mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(E3, E4)
    println("\nResultado: ")
    (mat1.multMatrizEstandarSec(E3, E4)).foreach(row => println(row))
    println("\nTiempo de ejecución de Multiplicación Estandar: " + compl(0))
    println("Tiempo de ejecución de Multiplicación Estandar Paralelo: " + compl(1))
    println("Aceleracion: " + compl(2))
```

1.

```
1. Multiplicacion Estandar 128x128
Unable to create a system terminal

Resultados:

Tiempo de ejecución de Multiplicación Estandar: 32.3196
Tiempo de ejecución de Multiplicación Estandar Paralelo: 70.9253
Aceleracion: 0.4556850658368735
```

2.

```
2. Multiplicacion Estandar 128x128

Resultados:

Tiempo de ejecución de Multiplicación Estandar: 35.2057
Tiempo de ejecución de Multiplicación Estandar Paralelo: 74.1596
Aceleracion: 0.47472882809508143
```

● ANÁLISIS - Multiplicación Estándar

¿Cuál de las implementaciones es más rápida?

R/ En caso del algoritmo estándar de multiplicación de matrices, el algoritmo no paralelizado es el más rápido.

¿De qué depende que la aceleración sea mejor?

R/ Depende de factores como la dimensión de las matrices. En este caso, la aceleración del algoritmo sin paralelizar es mucho mejor que el algoritmo no paralelizado incluso cuando la dimensión de las matrices es demasiado grande.

¿Puede caracterizar los casos en que es mejor usar la versión secuencial/paralela de cada algoritmo de multiplicación de matrices?

R/

mat1.compararAlgoritmos(mat1.multMatrizEstandarSec, mat1.multMatrizEstandarPar)(m1, m2)

Los algoritmos se han probado con 14 matrices de dimensiones 2, 4, 8, 16, 32, 64, 128, la lista nos retorna el tiempo en ms del algoritmo de multiplicación estándar y su versión paraleizada, y la aceleración del algoritmo paralelo con respecto al secuencial.

Resultados de matrices 2x2:

(0.0608, 0.6848, 0.08878504672897197)

Resultados de matrices 4x4:

(0.0806, 0.5873, 0.13723820875191553)

Resultados de matrices 8x8:

(0.1187, 1.0747, 0.11044942774727831)

Resultados de matrices 16x16:

(0.2688, 4.3466, 0.06184143928587862)

Resultados de matrices 32x32:

(0.6061, 7.5205, 0.08059304567515457)

Resultados matrices 64x64:

(6.4674, 17.099, 0.37823264518392885)

Resultados de matrices 128x128:

(30.5214, 81.97, 0.3723484201537148)

Se puede apreciar, cómo a medida que las dimensiones de la matriz aumentan, el tiempo en ambos algoritmos se incrementa. Se puede apreciar que el algoritmo paralelizado es más deficiente que el que no lo es.

- **Función prodPuntoParD**

La estrategia usada para paralelizar en esta función fue mediante el uso de la abstracción ParVector, una implementación de vectores paralelos en Scala que permite ejecutar operaciones de forma concurrente. Al convertir los vectores v1 y v2 en ParVector, las operaciones como zip, map y sum son ejecutadas en paralelo, aprovechando los múltiples núcleos del procesador para mejorar el rendimiento.

En este caso, ParVector divide los datos en fragmentos que pueden ser procesados simultáneamente por diferentes hilos. Esto permite que las operaciones de cálculo del producto de elementos correspondientes y su suma posterior se distribuyan entre los núcleos disponibles, optimizando así el tiempo de ejecución de la función.

Una vez finalizadas todas las operaciones en paralelo, **ParVector** sincroniza los resultados parciales, asegurando que la suma final sea consistente y correcta. Esta estrategia es especialmente eficiente para operaciones sobre grandes conjuntos de datos, ya que reduce significativamente el tiempo de procesamiento al aprovechar al máximo el hardware disponible.

- **CASOS DE PRUEBA**

Se debe aclarar que para todos los casos de prueba de esta función se usó la siguiente función:

```
def pruebasProductoPunto(): Unit = {  
    val v1 = mat1.vectorAlAzar(10,10)  
    val v2 = mat1.vectorAlAzar(10,10)  
    val comp = mat1.compararProdPunto(mat1.prodPunto, mat1.prodPuntoParD)(v1, v2)  
    println("Producto Punto (10):")  
    println("Vector 1: " + v1.mkString(" "))  
    println("Vector 2: " + v2.mkString(" "))  
    println("Resultado Producto Punto:" + mat1.prodPunto(v1, v2))  
    println("Tiempo de ejecución de Producto Punto " + comp(0))  
    println("Tiempo de ejecución de Producto Punto Paralelo " + comp(1))  
    println("Aceleración: " + comp(2))  
}
```

En este captura en concreto se muestra la ejecución de un vector de tamaño 10, pero esta misma función se puede usar para otro tamaño de vectores, y se usará, solo se debe cambiar el tamaño del vector generado al azar y el nombre del vector, cambiando él (10) producto punto respectivo.

Para generar los 10 casos de prueba se ejecutará el código dos veces para cada tamaño, ya que al ser commutativas, el resultado no es diferente al cambiar posiciones.

Además, para todos los casos los resultados serán los correctos, puesto que las funciones ya han sido probadas y ya se ha demostrado la función en la sección anterior.

- Vector tamaño 10

Producto Punto (10):

Vector 1: 7 8 0 1 6 1 4 7 1 2

Vector 2: 3 6 8 7 2 1 3 3 1 6

Resultado Producto Punto:135

Tiempo de ejecución de Producto Punto 0.0641

Tiempo de ejecución de Producto Punto Paralelo 0.6636

Aceleración: 0.0965943339361061

Producto Punto (10):

Vector 1: 5 0 1 9 6 8 8 8 4 0

Vector 2: 2 7 6 4 8 5 5 0 4 0

Resultado Producto Punto:196

Tiempo de ejecución de Producto Punto 0.0499

Tiempo de ejecución de Producto Punto Paralelo 1.1674

Aceleración: 0.0427445605619325

En este caso, no hay ganancia en la versión paralela, en ninguno de los casos, esto se debe al pequeño número de datos, la versión paralela no es eficaz y generalmente demora más tiempo.

- Vector tamaño 100

Producto Punto (100):

Vector 1: 8 7 7 7 4 1 9 8 7 6 7 4 7 4 3 9 7 4 2 9 8 9 5 3 7 5 1 1 7 6 7 0 2 7 7 7 3 8 4 9 2 9 1 4 1 5 2 9

1 7 1 1 1 5 8 9 1 6 1 4 0 7 3 2 6 7 2 2 6 5 0 2 3 3 5 7 6 8 8 2 7 9 5 2 9 8 0 2 3 6 8 3 4 5 2 6 7 3 8 0

Vector 2: 0 7 9 0 3 1 8 9 5 7 6 1 7 7 3 6 3 2 9 7 2 9 2 4 4 9 6 1 9 2 1 5 3 8 9 7 2 4 4 0 0 1 9 7 7 5 6 1

9 9 0 1 1 1 1 3 2 4 9 1 3 1 0 8 2 7 3 8 9 3 1 7 7 9 3 2 6 9 3 2 9 6 0 8 1 6 1 7 4 9 1 4 9 0 8 0 2 3 2 8

Resultado Producto Punto:2186

Tiempo de ejecución de Producto Punto 0.1141

Tiempo de ejecución de Producto Punto Paralelo 0.7658

Aceleración: 0.1489945155393053

Producto Punto (100):

Vector 1: 3 6 2 1 3 3 3 7 1 4 2 6 8 3 1 9 3 5 0 1 4 6 2 3 3 7 0 3 3 5 5 7 3 3 4 9 7 4 4 4 3 5 6 5 7 5 2 4

2 9 4 7 9 7 0 3 7 0 8 3 9 4 7 4 7 7 5 3 2 0 4 7 0 6 6 4 9 2 0 4 4 8 8 3 3 8 9 0 9 8 4 4 1 9 4 3 1 0 3 5

Vector 2: 8 2 9 8 0 4 4 6 4 2 0 3 9 1 8 8 9 7 9 7 4 7 9 1 4 8 4 7 4 5 3 6 9 2 3 7 9 0 5 3 2 3 8 2 6 1 7 6

9 3 1 4 4 0 7 1 3 1 1 3 7 6 3 9 0 5 3 5 2 1 0 0 2 5 8 6 3 9 0 4 6 2 8 3 4 6 3 5 1 2 2 8 4 6 6 4 6 2 6 9

Resultado Producto Punto:1969

Tiempo de ejecución de Producto Punto 0.0958

Tiempo de ejecución de Producto Punto Paralelo 0.5792

Aceleración: 0.16540055248618782

En este caso, sigue sin haber ganancia en ninguno de los casos, ya que el tamaño de las matrices sigue sin ser lo suficientemente considerable.

- Vector tamaño 1000

```
Resultado Producto Punto:20957
Tiempo de ejecución de Producto Punto 0.3168
Tiempo de ejecución de Producto Punto Paralelo 1.2367
Aceleración: 0.25616560200533683

Producto Punto (1000):
Vector 1: 4 6 0 2 8 0 9 3 6 4 2 1 7 7 4 8 3 1 5 0 3 2 5 8 6 6 3 2 3 5 9 2 3 3 9 8 8 3 0 9 1 1 0 1 0 0 0 7 8 7 2 3 9 9 6 8 6 7 1 1 3 8 9 4 6 2 9 2 6 8 6 3 4 7 1 6 7 9 5 2 7 2 1
5 9 8 0 6 0 3 4 0 7 1 2 3 5 3 8 3 7 3 1 0 2 6 9 7 8 1 1 7 2 5 2 3 5 8 1 5 0 3 9 3 4 3 9 1 0 7 0 3 9 9 0 1 3 6 2 8 2 2 1 8 6 0 8 9 2 1 6 4 4 7 0 8 6 9 9 3 0 6 8 9 1 3 2 1
0 2 5 6 2 0 6 4 5 8 0 9 8 6 7 3 4 8 8 9 7 4 8 3 1 7 1 1 7 4 2 7 0 5 0 4 9 2 5 4 3 2 7 6 6 6 7 0 0 8 1 9 1 7 5 5 3 3 8 9 8 8 5 7 8 0 4 8 1 4 1 0 8 4 1 8 4 8 3 5 2 7 7 2 9 4 6
0 0 8 6 1 5 6 5 9 5 9 0 3 2 0 1 3 1 6 2 4 5 4 9 5 3 9 6 3 8 0 3 8 2 7 2 9 6 1 8 6 4 7 1 9 2 0 3 5 5 8 0 0 6 0 8 9 3 9 9 1 0 7 1 4 2 9 5 9 7 5 2 8 3 8 2 5 0 2 6 2 9 4 5 5
2 6 7 5 8 0 1 7 9 1 9 3 1 8 4 5 5 8 7 9 4 1 2 8 3 3 5 1 3 1 6 2 3 1 5 4 5 2 4 3 4 4 5 0 4 0 5 9 5 7 6 0 9 9 0 3 5 2 3 1 9 5 1 4 6 0 1 4 7 7 1 9 4 4 6 3 7 7 4 2 0 2
2 7 1 6 5 1 7 7 7 7 7 8 7 4 9 6 1 3 2 7 0 9 5 8 4 9 0 7 2 8 8 8 7 6 9 3 6 5 6 8 8 2 8 9 5 7 8 1 8 2 6 9 9 5 8 2 9 9 0 9 3 3 6 5 9 7 9 3 1 7 9 6 8 5 0 9 8 3 6 6 6 3 3 2 7 1 7 1
2 0 8 9 6 1 4 6 2 3 7 4 9 2 3 5 4 1 4 8 6 1 6 2 0 5 6 2 9 3 5 0 0 4 8 0 4 1 9 3 6 2 4 4 8 8 7 4 5 3 6 6 0 7 6 9 4 0 5 6 1 3 5 4 2 7 2 7 0 8 0 3 2 2 4 6 3 3 9 1 0 2 7 4 1 1 2
1 2 0 8 7 0 2 0 4 7 7 9 3 5 4 6 0 6 4 6 1 3 5 2 0 2 9 9 6 7 7 6 9 1 7 4 1 2 5 2 1 0 9 2 9 9 2 4 5 5 2 0 0 1 7 0 6 2 6 4 3 2 2 5 0 0 6 1 3 8 2 9 8 0 9 5 4 2 4 5 6 7 3 7 0 8 7
9 6 3 4 5 4 3 2 4 6 3 3 6 2 2 4 2 6 8 3 3 2 1 9 7 5 0 9 7 2 5 2 7 0 3 5 6 9 2 0 0 7 6 1 9 4 3 7 2 2 3 7 9 8 7 8 5 9 6 8 7 3 5 9 0 4 2 4 5 7 3 3 6 1 4 2 6 0 8 5 0 1
4 5 8 0 9 5 6 0 9 5 2 2 6 3 5 2 9 0 6 7 7 3 2 7 6 6 9 4 8 2 6 4 5 4 2 0 6 8 5 8 0 4 4 2 5 3 7 2 0 4 0 0 6 5 0 3 1 2 0 7 9 5 8 6 4 2 3 3 9 9 8 5 3 9 8 4 0 3 9 5 0 1 3 7 3 3 8
0 7 1 9 8 2 9 6 0 9 4 0 7 5 5 3 7 5 0 4 9 2 7 6 0 7 6 0 9 2 6 0 7 6 4 5 8 2 0 6 8 2 5 9 0 4 0 4 5 5 6 3 0 2 6 5 7 8 9 7 5 4 0 8 3 7 9 2 0 0 7 2 0 2 6 4 7 6 9 4 8 6 1 9 6 8 9
4 1 0 3 7 3 8 1 8 3 6 2 2 8 6 2 1 2 9 2 2 6 6 6 7 9 4 8 9 6 1 5 5 9 9 8 0
Vector 2: 0 5 9 7 4 2 7 3 3 1 7 0 7 7 6 9 2 7 1 4 5 4 7 5 0 8 4 8 8 9 1 3 7 6 3 2 5 4 7 9 3 0 5 4 5 7 1 2 8 7 2 2 6 5 8 8 8 7 4 7 3 2 7 6 6 8 9 9 7 2 3 1 3 5 9 8 3 4 3 0 5 6 8 5
0 0 9 9 2 8 4 9 2 1 9 3 7 5 4 7 6 9 1 9 7 0 0 4 9 5 1 5 2 3 3 9 4 9 5 1 5 9 8 3 5 5 3 1 4 3 1 0 8 2 8 9 3 9 4 6 8 7 0 9 0 2 6 9 9 3 5 5 9 0 2 9 5 0 3 3 6 3 7 1 2 8 2 1 6 9 1
2 1 7 3 1 9 2 7 2 8 5 6 6 2 3 5 2 0 1 1 8 2 7 3 2 4 0 4 9 4 3 7 7 0 5 3 6 4 7 7 5 6 1 3 6 2 9 9 7 3 3 7 8 5 8 6 5 0 7 8 0 5 6 8 4 0 5 6 7 9 1 3 8 9 0 5 8 0 7 4 1 3 6 7 7 7 0
3 2 1 8 9 5 6 4 9 2 9 4 7 6 1 5 1 0 1 9 2 8 2 9 0 5 0 6 0 6 5 4 9 2 5 2 8 2 9 9 0 6 9 1 8 9 7 3 9 3 0 2 8 4 5 1 6 6 8 8 9 8 8 5 1 2 7 5 7 0 0 9 9 6 5 8 7 4 6 8 2 4 3 9 1 1 0 5
7 5 9 8 1 7 2 9 8 9 3 4 8 9 3 2 2 3 3 3 7 9 5 1 7 8 0 8 5 9 4 4 8 9 2 4 4 9 8 0 9 6 1 6 7 6 5 2 2 9 6 5 3 2 8 3 0 2 6 3 4 7 3 4 7 0 3 7 9 0 8 0 6 9 3 9 7 1 5 6 6 7 9 2 6 7 3 5
4 3 6 0 7 7 9 5 7 9 8 3 0 7 4 8 3 1 2 6 9 5 8 0 5 8 9 2 0 9 9 9 6 8 0 8 6 9 8 7 0 9 7 8 2 4 1 4 4 6 9 7 4 6 9 8 9 7 0 2 0 4 3 6 8 9 8 2 8 3 8 5 2 8 4 6 0 4 6 6 4 5 4 7 8 7 5
1 8 7 6 7 1 0 3 4 7 3 7 7 2 1 2 6 8 0 4 9 7 2 7 6 0 9 2 6 0 7 6 4 5 8 2 0 6 8 2 5 9 0 4 0 4 5 5 6 3 0 2 6 5 7 8 9 7 5 4 0 8 3 7 9 2 0 0 7 2 0 2 6 4 7 6 9 4 8 6 1 9 6 8 9
3 3 2 2 6 1 3 6 0 5 1 4 1 3 6 7 6 0 1 5 4 0 1 8 4 2 3 2 8 0 1 0 7 0 3 9 6 5 8 1 9 8 3 5 4 2 9 9 7 9 7 5 6 1 0 7 5 9 2 2 4 3 1 0 1 6 3 9 1 3 7 8 0 3 5 4 6 9 2 6 9 5 8
0 8 4 9 4 2 8 0 2 6 9 5 6 7 0 2 8 9 7 7 1 4 5 1 8 3 4 8 0 3 2 0 9 4 7 1 3 0 8 2 7 2 0 6 8 9 2 2 5 7 1 4 5 3 9 2 3 0 6 3 4 8 6 8 8 9 8 8 4 1 9 7 4 5 3 0 2 4 2 4 1 9 5 1 2 0
7 4 8 7 3 3 2 9 6 2 0 1 0 9 5 9 4 3 2 9 2 8 0 6 5 4 9 1 2 6 4 9 9 1 0 3 5 6 2 3 4 2 6 6 8 6 2 1 4 8 2 7 4 4 7 8 5 0 0 4 7 0 2 6 1 7 6 8 5 3 2 1 8 7 5 2 6 7 7 3 0 0 2 0 6 1 3 1
3 4 4 6 7 1 7 9 0 3 9 1 2 5 6 0 8 8 3 5 1 0 9 3 0 0 4 5 0 9 1 4 8 7 2 4 1 3 1 3 8 3 8 2 2 2 2 3 9 9 2 0 4 5 9 3 5 2 8 3 0 3 5 9 7 4 1 9 8 3 9 5 0 7 6 5 1 3 3 7 4 5 6 0 4 2 0 9
9 7 6 1 0 7 0 3 3 4 4 6 9 8 9 7 7 0 9 7 2 0 7 4 8 8 5 7 9 8 9 7 3 7 0 5 8
Resultado Producto Punto:20957
```

```
Tiempo de ejecución de Producto Punto 0.3168
Tiempo de ejecución de Producto Punto Paralelo 1.2367
Aceleración: 0.25616560200533683

Producto Punto (1000):
Vector 1: 1 4 5 1 3 5 5 5 0 9 6 0 2 9 0 5 1 7 8 2 1 0 3 6 8 8 3 2 7 2 5 2 8 0 4 0 6 1 9 1 8 0 0 7 4 1 7 8 8 3 5 0 5 6 0 3 5 6 4 6 7 3 5 5 1 6 2 8 2 0 9 1 4 9 7 1 3 6 6 4 0 4
0 1 4 5 1 0 6 4 8 7 6 7 9 8 6 7 1 6 0 7 4 0 3 9 2 4 7 9 3 1 2 4 0 6 4 8 3 8 5 4 0 5 7 6 0 5 7 5 3 5 8 0 3 5 0 1 6 0 6 6 6 9 2 3 3 1 6 1 7 5 8 6 3 5 8 3 7 4 7 6 9 7 4 8 7 3 3
5 3 6 2 6 2 0 2 0 5 9 7 5 0 1 7 4 1 3 5 7 0 7 4 1 9 9 6 3 1 6 5 9 2 9 8 6 8 4 4 6 8 7 6 0 9 7 4 0 1 1 9 4 3 4 6 1 5 4 6 2 6 0 5 2 5 9 4 9 7 8 1 6 1 9 9 2 1 4 8 7 5 5 1 3 9 2 4
4 3 8 3 0 3 1 4 4 5 7 1 2 0 2 6 6 2 4 3 6 7 1 8 9 4 5 1 2 7 9 6 6 7 8 0 9 8 7 3 0 4 3 5 2 5 9 8 7 9 8 0 9 0 2 4 4 4 7 4 3 5 5 7 5 4 8 3 9 0 0 8 6 8 7 4 3 7 3
6 8 2 7 4 5 8 9 3 0 7 3 9 0 7 7 7 0 7 9 7 2 5 5 3 2 4 0 7 0 6 4 2 1 1 5 4 8 7 7 5 5 7 7 9 9 4 8 0 6 5 5 5 0 6 5 2 0 2 8 3 4 5 6 1 7 2 5 7 8 3 4 1 3 8 4 7 8 0 6 7 6 9 9 7 3 1 3 3
6 5 2 4 6 3 8 8 3 7 6 5 9 9 8 0 5 3 6 2 1 2 0 6 8 5 4 7 7 7 7 3 0 9 1 1 8 6 2 1 8 2 7 0 7 6 0 6 6 0 1 7 9 2 0 9 7 5 2 4 0 5 5 8 7 4 0 5 9 3 4 0 6 4 4 3 0 8 7 3 5 0 8 4 2 6 4
2 2 8 1 9 7 4 2 9 7 6 2 8 7 9 3 9 3 9 7 4 7 2 2 6 8 7 5 5 9 1 3 6 3 9 4 7 8 5 5 3 9 7 6 4 7 6 9 2 3 8 8 8 5 8 6 3 5 7 3 6 5 4 4 8 9 0 8 3 0 4 9 8 8 5 0 8 8 8 4 9 3 6 7 5 8 5 8 5
6 7 2 3 3 8 1 7 3 2 4 5 4 9 2 3 8 6 7 3 7 5 9 1 3 5 9 8 0 8 0 3 1 5 2 9 1 3 7 9 8 1 5 6 3 9 5 5 0 6 5 2 0 2 0 8 3 4 5 6 1 7 2 5 7 8 3 8 3 9 2 5 8 8 6 4 5 5 7 4 0 0 8 9 5 3 8 9 8
2 4 5 8 6 4 4 6 0 2 8 2 2 2 9 3 6 7 8 3 4 8 8 2 4 1 8 9 5 8 8 7 4 8 2 8 8 7 2 6 2 8 5 8 4 5 7 1 4 7 2 5 1 9 4 0 5 2 6 1 6 6 2 2 1 2 3 3 7 6 9 0 9 5 3 5 8 1 2 5 3 3 3 6 4
3 1 3 2 1 9 4 3 9 6 7 2 1 3 7 0 6 7 0 0 9 2 7 4 9 7 4 2 3 2 9 1 2 9 4 8 4 5 6 3 4 9 6 8 1 6 1 2 8 9 0 0 5 6 4 5 4 6 2 5 2 1 1 0 2 9 5 2 7 6 8 2 0 7 6 6 9 6 2 0 9 6 1 4 4 1
1 8 8 8 0 8 9 6 1 9 5 1 7 1 1 0 1 5 6 6 6 3 0 0 5 1 5 1 4 0 7 2 8 9 6 8 1 1 9 4 3 6 8 4 2 4 4 3 7 7 8 3 0 7 5 1 5 2 7 9 5 2 5 6 2 6 4 2 4 7 7 9 0 4 0 2 8 5 8 1 2 2 9 1 9 3 5
2 2 1 6 8 1 7 4 2 8 4 9 9 7 3 6 9 0 7 8 8 2 2 7 7 1 8 1 9 5 2 0 3 0 0 4 6
Vector 2: 1 7 2 2 7 8 6 3 4 9 4 4 3 1 3 6 9 8 4 9 2 5 3 1 8 4 0 3 4 1 4 5 1 0 1 2 7 6 0 9 2 4 0 6 4 5 0 9 8 2 5 1 4 9 2 3 4 0 3 8 1 0 9 1 8 3 3 1 4 4 7 8 4 0 9 0
2 2 2 4 1 3 9 8 1 0 5 7 5 6 5 4 5 8 3 5 8 2 9 5 6 4 0 5 5 0 9 7 1 7 3 5 4 1 1 7 9 6 5 3 2 7 6 5 1 9 8 6 4 2 2 8 4 6 5 4 9 1 2 8 1 3 3 2 6 8 3 4 7 2 2 5 5 7 6 1 3 8 4 3
2 2 2 7 6 1 5 4 5 6 7 0 3 9 1 3 0 0 1 9 0 9 7 7 6 9 5 8 5 7 9 2 9 3 9 5 7 6 9 2 1 1 7 9 9 1 3 5 4 1 4 1 9 5 6 5 9 9 3 1 7 3 4 0 0 1 8 6 8 4 5 6 5 0 1 8 9 7 9 0 4 0 5 0 4 0 9 4 9 8 8 9 3 0 3
5 6 2 4 6 4 2 2 7 2 1 3 2 5 9 4 6 9 2 2 0 8 4 3 3 9 9 8 2 4 2 1 2 8 9 6 4 4 9 5 9 7 5 1 3 2 8 2 7 5 2 9 3 0 8 4 6 8 0 6 6 9 5 9 6 8 9 3 4 7 6 7 8 9 1 8 6 6 9 9 6 8 8 3 1 1 9 2
2 9 2 0 0 2 8 3 4 8 7 6 5 0 9 7 0 3 7 5 2 0 7 8 8 1 1 1 1 4 7 1 5 5 1 7 7 3 5 7 2 1 4 4 2 2 6 9 3 5 0 3 2 0 9 2 8 7 7 0 0 7 2 2 0 9 1 2 2 3 2 1 8 5 4 5 3 0 6 2 3 7 6 0 6 9 8 7
7 4 0 8 9 1 3 4 9 6 8 9 2 4 0 9 7 6 9 9 7 3 9 3 9 6 2 0 0 7 3 9 9 5 8 6 6 4 1 5 1 4 1 9 5 6 5 9 9 3 1 7 3 4 0 0 1 8 6 8 4 5 6 5 0 1 8 9 7 9 0 4 0 5 0 4 0 9 4 9 8 8 9 3 0 3
6 1 6 3 9 2 6 4 0 6 1 6 4 9 0 9 6 5 0 9 4 0 3 5 4 3 0 2 3 9 1 9 1 5 3 7 7 9 6 7 2 7 8 2 8 9 9 4 6 7 7 5 9 5 7 2 3 5 0 1 0 9 4 5 3 9 6 4 6 3 5 0 5 2 7 7 0 0 1 4 4 2 8 6 8 0 3
6 9 4 2 0 8 3 6 2 0 7 4 1 0 8 1 1 7 6 4 4 4 5 0 6 7 2 5 9 9 0 5 0 1 4 6 3 5 2 6 7 1 7 0 7 1 4 9 1 2 9 0 9 4 4 3 4 1 3 0 5 5 3 6 4 2 5 1 7 9 2 6 3 9 4 8 2 4 8 8 1 1 4 6 2
2 7 0 7 6 3 5 4 9 4 7 1 0 6 1 2 6 0 3 7 3 9 8 6 6 0 6 7 1 7 4 1 3 9 6 5 8 2 4 6 8 4 8 7 5 4 7 3 5 4 9 2 1 8 1 0 8 4 4 7 1 4 8 2 8 6 3 3 7 9 7 0 9 0 2 6 0 4 8 2 7 9 0 7 5 1 7
1 9 7 8 2 5 6 0 3 7 1 1 8 7 2 6 8 4 7 0 1 7 8 5 8 0 4 7 8 1 5 1 0 1 1 2 5 9 2 0 6 2 7 0 8 9 1 0 5 3 4 0 1 2 8 0 8 0 1 8 4 2 0 6 6 1 5 3 0 2 6 0 5 0 8 1 7 6 1 5 9 7 0 1 5 2 1 6
7 7 8 0 4 6 2 2 1 5 5 7 0 8 0 4 3 8 4 4 7 0 7 8 1 7 0 1 4 2 2 7 1 7 6 1 4 1 8 7 3 3 4 4 0 4 3 6 7 0 7 2 5 2 9 0 4 2 3 8 7 2 2 8 6 7 2 9 0 9 0 0 9 6 9 5 9 8 0 5 8 7 3 3 9 1 6
1 4 4 4 3 1 5 5 0 2 6 0 2 1 2 8 4 3 9 7 5 0 7 5 3 0 3 7 4 1 4 7 2 8 0 4 7
Resultado Producto Punto:20957
```

En ambos casos se puede ver que sigue sin haber una ganancia en la versión paralela, lo que hace dudar si es efectivo en el caso de la función de producto punto, efectuar la paralelización, aunque al ser una operación y función más sencilla, posiblemente se necesite un mayor número de datos para que sea efectiva la paralelización.

- Vector tamaño 10000

Debido al tamaño de los vectores, en este grupo de vectores y los siguientes no es ni será posible mostrar los vectores generados de manera clara, por tanto, se mostrara solo los resultados dados.

```
Resultado Producto Punto:203595
Tiempo de ejecución de Producto Punto 1.0364
Tiempo de ejecución de Producto Punto Paralelo 1.2366
Aceleración: 0.8381044800258775
```

```
Resultado Producto Punto:205150
Tiempo de ejecución de Producto Punto 0.8534
Tiempo de ejecución de Producto Punto Paralelo 3.1577
Aceleración: 0.2702599993666276
```

En ambos casos sigue sin haber ganancia en la versión paralela, lo que comienza a plantear la idea de que la paralelización no es efectiva en este caso, esto que puede deberse al overhead de paralelización o a limitaciones de hardware.

- Vector tamaño 100000

Resultado Producto Punto:2024829

Tiempo de ejecución de Producto Punto 4.2849

Tiempo de ejecución de Producto Punto Paralelo 2.0791

Aceleración: 2.0609398297340196

Resultado Producto Punto:2024659

Tiempo de ejecución de Producto Punto 6.5718

Tiempo de ejecución de Producto Punto Paralelo 2.4789

Aceleración: 2.6510952438581628

En el último caso de prueba con vectores de tamaño 100000, se puede evidenciar una ganancia en ambos casos para el tiempo de ejecución en la versión paralela, lo que confirma que al ser una función y operación más sencilla se necesita de una gran cantidad de datos para que la paralelización sea efectiva

• ANÁLISIS - Producto punto

¿Cuál de las implementaciones es más rápida?

Depende del tamaño de datos del vector, aunque según los casos de pruebas se requiere de un gran tamaño de los vectores para que la versión paralelizada sea el más rápido, por tanto, para vectores de gran tamaño el mejor es el algoritmo paralelizado, pero para vectores de un tamaño más regular o pequeño es mejor el secuencial.

¿De qué depende que la aceleración sea mejor?

Depende de factores como el tamaño de los vectores, en estos casos, la aceleración del algoritmo paralelo es mucho mejor cuando los vectores tienen un gran tamaño.

¿Puede caracterizar los casos en que es mejor usar la versión secuencial/paralela de cada algoritmo de multiplicación de matrices?

Se ha probado con 10 vectores de dimensiones 10, 100, 1000, 10000, 100000, la lista retorno, el tiempo en ms del producto punto secuencial, el tiempo del algoritmo paralelo, y la aceleración del algoritmo paralelo con respecto al secuencial

Entendiendo esto, en los casos evaluados para los vectores de tamaño 10, 100, 1000, 10000, la mejor versión a usar es la secuencial, ya que los valores de la versión paralela fueron mayores.

Pero para los casos de los vectores de tamaño 10000, la mejor versión a usar es la paralela, puesto que al ser vectores de gran tamaño, la paralelizada tiene tiempos de ejecución menores a la secuencial.

ARGUMENTACIÓN SOBRE LA CORRECCIÓN

- **Demostración por inducción estructural de la función subMatriz:**

La función subMatriz toma una matriz m y genera una submatriz de tamaño $l \times l$, empezando desde la posición (i,j) en la matriz original. Demostraremos por inducción estructural que esta función produce una submatriz válida de tamaño $l \times l$ para toda matriz m que sea válida (es decir, todas sus filas tienen igual longitud y tiene dimensiones suficientes para contener la submatriz extraída).

Caso base: $l = 1$

Si $l = 1$, la submatriz debe contener solo un elemento, el correspondiente a la posición (i,j) en m .

La función evalúa:

```
Vector.tabulate(1, 1)((a, b) => m(i + a)(j + b))
```

En este caso:

$a=0$ y $b=0$, por lo que se accede al elemento $m(i+0)(j+0) = m(i)(j)$.

La función genera un vector de 1 fila y 1 columna, equivalente a una submatriz $[[m(i)(j)]]$. Esto coincide exactamente con la definición esperada de la submatriz. Por lo tanto, el caso base es válido.

Hipótesis de inducción

Supongamos que la función funciona correctamente para una submatriz de tamaño $l \times l$. Es decir, produce una matriz válida $l \times l$, con los elementos correspondientes de m , comenzando en la posición (i,j) .

Paso inductivo: Demostración para $l + 1$

Queremos demostrar que la función genera correctamente una submatriz de tamaño $(l+1) \times (l+1)$.

Para $l + 1$, la función evalúa:

```
Vector.tabulate(l + 1, l + 1)((a, b) => m(i + a)(j + b))
```

- Dimensiones generadas: Vector.tabulate($l + 1, l + 1$) asegura que el resultado es una matriz de $(l+1)$ filas y $(l+1)$ columnas.
- Elementos generados: para $0 \leq a, b < l + 1$, la posición (a,b) en la submatriz se calcula como $m(i+a)(j+b)$. Esto asegura que:
 - Los primeros l elementos de cada fila y columna son los mismos que en la submatriz de tamaño $l \times l$ (por la hipótesis de inducción).

- La nueva fila (para $a = l$) y la nueva columna (para $b = l$) se extraen correctamente de m , según el patrón establecido.

- Propiedades heredadas:

- Si m es una matriz válida y tiene dimensiones suficientes para extraer una submatriz $(l+1) \times (l+1)$, todos los índices $i+a$ y $j+b$ están dentro de los límites de m , garantizando que no se accede a posiciones inválidas.

Por lo tanto, la función produce correctamente una submatriz de tamaño $l+1$, completando el paso inductivo.

- **Demostración por inducción estructural de la función sumMatriz:**

La función sumMatriz toma dos matrices $m1$ y $m2$ de las mismas dimensiones y genera una nueva matriz resultante de sumar sus elementos posición por posición. Demostremos por inducción estructural que esta función produce una matriz válida con las dimensiones de entrada, cuyas entradas son las sumas correspondientes de $m1$ y $m2$.

Caso base: $m1$ y $m2$ son matrices 1×1

Si $m1$ y $m2$ son matrices 1×1 , entonces cada una contiene un único elemento, $m1(0)(0)$ y $m2(0)(0)$. La función evalúa:

`Vector.tabulate(1, 1)((i, j) => m1(i)(j) + m2(i)(j))`

Aquí:

- $i=0$ y $j=0$, lo que significa que la única operación realizada es:

$$\text{resultado}(0)(0) = m1(0)(0) + m2(0)(0).$$

El resultado es una matriz 1×1 , válida y consistente con la suma de $m1$ y $m2$. El caso base es válido.

Hipótesis de inducción

Supongamos que la función sumMatriz funciona correctamente para matrices $m1$ y $m2$ de dimensiones $n \times n$. Es decir, genera una matriz $n \times n$ en la que cada entrada (i,j) satisface:

$$\text{resultado}(i)(j) = m1(i)(j) + m2(i)(j) \quad \text{para } 0 \leq i, j < n.$$

Paso inductivo: Demostración para $(n+1) \times (n+1)$

Queremos demostrar que la función produce correctamente la suma de dos matrices $m1$ y $m2$ de dimensiones $(n+1) \times (n+1)$. La función evalúa:

`Vector.tabulate(n + 1, n + 1)((i, j) => m1(i)(j) + m2(i)(j))`

- Dimensiones generadas:

`Vector.tabulate(n + 1, n + 1)` asegura que la matriz resultante tiene $n+1$ filas y $n+1$ columnas.

- Cálculo de elementos:

Para $0 \leq i, j < n + 1$, el elemento en la posición (i, j) se calcula como:
 $\text{resultado}(i)(j) = m1(i)(j) + m2(i)(j)$.

Por la hipótesis de inducción, para las primeras n filas y columnas ($0 \leq i, j < n$), la suma es correcta.

Para la fila n y la columna n , $i = n$ o $j = n$, la función aplica el mismo patrón: accede a $m1(n)(j)$ y $m2(n)(j)$ o $m1(i)(n)$ y $m2(i)(n)$, y suma sus valores.

- Validez estructural:

Dado que las dimensiones de $m1$ y $m2$ son $(n+1) \times (n+1)$ y ambas matrices son válidas, cada acceso $m1(i)(j)$ y $m2(i)(j)$ está dentro de los límites, asegurando que no se accede a índices fuera de rango.

Por lo tanto, la matriz resultante es válida y satisface la propiedad de ser la suma, posición por posición de $m1$ y $m2$.

- **Demostración por inducción estructural de la función `restaMatriz`:**

La función `restaMatriz` toma dos matrices $m1$ y $m2$ de las mismas dimensiones y genera una nueva matriz resultante de restar sus elementos posición por posición. Demostremos por inducción estructural que esta función produce una matriz válida con las dimensiones de entrada, cuyas entradas son las restas correspondientes de $m1$ y $m2$.

Caso base: $m1$ y $m2$ son matrices 1×1

Si $m1$ y $m2$ son matrices 1×1 , entonces cada una contiene un único elemento, $m1(0)(0)$ y $m2(0)(0)$. La función evalúa:

`Vector.tabulate(1, 1)((i, j) => m1(i)(j) - m2(i)(j))`

Aquí:

- $i=0$ y $j=0$, lo que significa que la única operación realizada es:

$$\text{resultado}(0)(0) = m1(0)(0) - m2(0)(0).$$

El resultado es una matriz 1×1 , válida y consistente con la resta de $m1$ y $m2$. El caso base es válido.

Hipótesis de inducción

Supongamos que la función `restaMatriz` funciona correctamente para matrices $m1$ y $m2$ de dimensiones $n \times n$. Es decir, genera una matriz $n \times n$ en la que cada entrada (i,j) satisface:

$$\text{resultado}(i)(j) = m1(i)(j) - m2(i)(j) \quad \text{para } 0 \leq i, j < n.$$

Paso inductivo: Demostración para $(n+1) \times (n+1)$

Queremos demostrar que la función produce correctamente la resta de dos matrices $m1$ y $m2$ de dimensiones $(n+1) \times (n+1)$. La función evalúa:

`Vector.tabulate(n + 1, n + 1)((i, j) => m1(i)(j) - m2(i)(j))`

- Dimensiones generadas:

`Vector.tabulate(n + 1, n + 1)` asegura que la matriz resultante tiene $n+1$ filas y $n+1$ columnas.

- Cálculo de elementos:

Para $0 \leq i, j < n + 1$, el elemento en la posición (i, j) se calcula como:

$$\text{resultado}(i)(j) = m1(i)(j) - m2(i)(j).$$

Por la hipótesis de inducción, para las primeras n filas y columnas ($0 \leq i, j < n$), la resta es correcta.

Para la fila n y la columna n , $i = n$ o $j = n$, la función aplica el mismo patrón: accede a $m1(n)(j)$ y $m2(n)(j)$ o $m1(i)(n)$ y $m2(i)(n)$, y resta sus valores.

- Validez estructural:

Dado que las dimensiones de $m1$ y $m2$ son $(n+1) \times (n+1)$ y ambas matrices son válidas, cada acceso $m1(i)(j)$ y $m2(i)(j)$ está dentro de los límites, asegurando que no se accede a índices fuera de rango.

Por lo tanto, la matriz resultante es válida y satisface la propiedad de ser la resta, posición por posición de $m1$ y $m2$.

- **Demostración por inducción estructural de la función multStrassen:**

Demostraremos la función multStrassen mediante inducción estructural.

Caso base:

Cuando $n=1$, las matrices $m1$ y $m2$ tienen tamaño 1×1 .

En este caso, el algoritmo simplemente multiplica los únicos elementos disponibles de ambas matrices:

`Vector(Vector(m1(0)(0) * m2(0)(0)))`

La operación es correcta porque la multiplicación entre dos números escalares está definida y produce el resultado esperado para matrices de 1×1 .

Por lo tanto, el algoritmo es correcto en el caso base.

Paso inductivo:

Supongamos que la función multStrassen es correcta para matrices de tamaño $n/2$, es decir, que para matrices $m1$ y $m2$ de tamaño $n/2 \times n/2$, devuelve el producto correcto $m1 \times m2$. Debemos demostrar que multStrassen también produce el resultado correcto para matrices de tamaño n .

1. Descomposición en submatrices:

Dado que n es una potencia de 2, cada matriz $m1$ y $m2$ se divide en cuatro submatrices cuadradas de tamaño $n/2 \times n/2$:

- a, b, c, d : submatrices de $m1$.
- e, f, g, h : submatrices de $m2$.

Esto es válido porque la función subMatriz divide correctamente las matrices originales.

2. Cálculo de los productos intermedios:

El algoritmo utiliza las siguientes fórmulas de Strassen para calcular productos intermedios $p1, \dots, p7$, que son combinaciones específicas de

sumas, restas y multiplicaciones recursivas entre las submatrices:

$$\begin{array}{lll} S_1 & = & B_{12} - B_{22} \\ S_2 & = & A_{11} + A_{12} \\ S_3 & = & A_{21} + A_{22} \\ S_4 & = & B_{21} - B_{11} \\ S_5 & = & A_{11} + A_{22} \\ S_6 & = & B_{11} + B_{22} \\ S_7 & = & A_{12} - A_{22} \\ S_8 & = & B_{21} + B_{22} \\ S_9 & = & A_{11} - A_{21} \\ S_{10} & = & B_{11} + B_{12} \end{array} \quad \begin{array}{lll} P_1 & = & A_{11} \cdot S_1 \\ P_2 & = & S_2 \cdot B_{22} \\ P_3 & = & S_3 \cdot B_{11} \\ P_4 & = & A_{22} \cdot S_4 \\ P_5 & = & S_5 \cdot S_6 \\ P_6 & = & S_7 \cdot S_8 \\ P_7 & = & S_9 \cdot S_{10} \end{array}$$

Por hipótesis inductiva, asumimos que cada multiplicación recursiva de matrices de tamaño $n/2$ es correcta.

3. Cálculo de las submatrices resultantes:

Las submatrices del resultado C se obtienen mediante:

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

Estas fórmulas son las definiciones estándar del método de Strassen, que garantiza que las submatrices $c_{11}, c_{12}, c_{21}, c_{22}$ se ensamblen correctamente para formar la matriz resultado C.

4. Combinación de las submatrices:

Finalmente, el algoritmo combina las submatrices $c_{11}, c_{12}, c_{21}, c_{22}$, en la matriz resultante C usando la operación:

```
val c1 = c11.zip(c12).map({ case (i, j) => i ++ j })
```

```
val c2 = c21.zip(c22).map({ case (i, j) => i ++ j })
```

```
c1 ++ c2
```

La operación de concatenación preserva la estructura de la matriz y garantiza que los elementos estén correctamente ubicados.

Por la hipótesis inductiva, sabemos que las multiplicaciones recursivas son correctas para matrices más pequeñas ($n/2$). Al combinar correctamente las submatrices mediante las fórmulas de Strassen y al ensamblar la matriz resultado, demostramos que el algoritmo es correcto para matrices de tamaño n .

- **Demostración por inducción estructural de la función multStrassenPar:**

Para demostrar por inducción estructural la corrección de la función multStrassenPar, vamos a probar que produce la matriz correcta resultante de la multiplicación de dos matrices cuadradas de tamaño $n \times n$ (donde n es una potencia de 2) utilizando la descomposición recursiva y la parallelización en cuatro tareas.

Paso base: matrices de dimensiones 2x2

Cuando $n=2$, las matrices tienen tamaño 2x2. En este caso, las submatrices resultantes de cada matriz será de tamaño uno, y al hacer los cálculos correspondientes para hallar las 4 submatrices de la matriz resultante, es decir, del producto de las dos matrices, se reducirá a la multiplicación de un elemento por otro al invocar la función multStrassen, la cual se paralleliza en 4 tareas:

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

$c11 = \text{sumMatriz}(\text{Vector}(\text{Vector}(a11(0)*b11(0))\text{, Vector}(\text{Vector}(a12(0)*b21(0))))$

Y así en adelante, según el orden, será lo mismo para $c12, c21$ y $c22$.

Todos tendrán esta estructura, resultando submatrices de tamaño 1x1, que después, al unirlas nos dará una matriz de mismas dimensión que las matrices pasadas como parámetros.

Este cálculo es correcto, ya que es la definición directa del producto de matrices en 1×1 . Por lo tanto, el caso base se cumple.

Hipótesis de inducción

Supongamos que la función multStrassenPar es correcta para todas las matrices $n \times n$, donde n es una potencia de 2 menor o igual a k . Es decir, para matrices de tamaño $k \times k$, multStrassenPar produce la matriz correcta utilizando la paralelización y las fórmulas de Strassen.

Paso inductivo

Queremos demostrar que la función es correcta para matrices de tamaño $(2k) \times (2k)$.

1. **División de las matrices:** La función divide las matrices m_1 y m_2 en 4 submatrices de tamaño $k \times k$:

$$m_1 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad m_2 = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

2. **Aplicación de las fórmulas de Strassen:** Utilizando las fórmulas de Strassen, se calculan los 7 productos intermedios p_1, p_2, \dots, p_7 recursivamente:

$$p_1 = \text{multStrassen}(a_{11}, b_{12} - b_{22}), \quad p_2 = \text{multStrassen}(a_{11} + a_{12}, a_{22}),$$

y así sucesivamente.

Por la hipótesis de inducción, asumimos que cada llamada recursiva a multStrassen produce resultados correctos para submatrices $k \times k$.

3. **Cálculo de las submatrices resultantes:** Las submatrices $c_{11}, c_{12}, c_{21}, c_{22}$ de la matriz resultado C se calculan correctamente combinando los productos intermedios p_1, \dots, p_7 según las definiciones de Strassen.

Por ejemplo:

$$c_{11} = p_5 + p_4 - p_2 + p_6$$

Estas combinaciones son correctas según las fórmulas de Strassen para multiplicar matrices.

4. **Paralelización de los cálculos:** La función utiliza parallel para calcular las submatrices $c_{11}, c_{12}, c_{21}, c_{22}$ simultáneamente. La paralelización no altera la validez matemática de los cálculos, ya que cada submatriz depende de valores independientes.
5. **Unión de las submatrices:** Finalmente, las submatrices se unen correctamente utilizando las operaciones:

```
val c1 = c11.zip(c12).map({case (i,j) => i ++ j})
```

```
val c2 = c21.zip(c22).map({case (i,j) => i ++ j})
```

```
c1 ++ c2
```

Esto garantiza que las submatrices $c_{11}, c_{12}, c_{21}, c_{22}$ se ensamblen en la matriz completa C de tamaño $2k \times 2k$.

- **Demostración Por Inducción matemática de la función multMatrizEstandarSec**

- Si A es una matriz de tamaño mxn y B es una matriz de tamaño nxp , entonces el producto $C = A * B$ es una matriz C de tamaño $m \times p$ donde cada elemento c_{ij} se calcula como la sumatoria desde $k=1$ hasta n de $a_{ik} * b_{kj}$. Donde a_{ik} es el elemento de la fila i y la columna k de la matriz A y b_{kj} es el elemento de la fila k y la columna j de la matriz B .
- Base de la Inducción: Consideremos el caso donde las matrices son de tamaño 1×1 . Es decir, tenemos: $m1 = [[a]], m2 = [[b]]$ el producto debe ser $multMatrizEstandarSec(m1, m2) = [[a * b]]$. La función, al aplicar la transportación y el producto punto, devuelve efectivamente el resultado esperado.
- Paso Inductivo: Supongamos que la afirmación es cierta para matrices de tamaño $k \times n$ y $n \times p$. Es decir, asumimos que: $multMatrizEstandarSec(m1, m2) = C$ donde C es una matriz de tamaño $k \times p$. Ahora consideremos matrices de tamaño $(k + 1) \times n$ y $n \times p$. Al aplicar $multMatrizEstandarSec$, la función iterará sobre cada fila de $m1$, que ahora tiene $k + 1$ filas. La última fila se puede considerar como un nuevo caso base.

Para cada fila i en $m1$, calculamos el producto punto con cada columna en $m1^T$. Por la hipótesis inductiva, sabemos que el producto punto para las primeras k filas se calculará correctamente. Para la fila adicional (la fila $k + 1$), también se aplicará el mismo proceso. Es decir, $C(k + 1)_j$ será igual a la sumatoria desde $l = 1$ hasta n de $m1[k][l] * m2[l][j]$. Esto asegura que cada elemento en la fila adicional se calcula correctamente, manteniendo así la propiedad del producto de matrices.

- **Demostración por Inducción matemática de la función multMatrizEstandarPar**

- Si A es una matriz de tamaño $m \times n$ y B es una matriz de tamaño $n \times p$, entonces el producto $C = A * B$ es una matriz C de tamaño $m \times p$ donde cada elemento c_{ij} se calcula como la sumatoria desde $k=1$ hasta n de $a_{ik} * b_{kj}$. Donde a_{ik} es el elemento de la fila i y la columna k de la matriz A y b_{kj} es el elemento de la fila k y la columna j de la matriz B.
- Base de la Inducción: Comenzamos con el caso base, que es cuando las matrices son de tamaño 1×1 . En este caso, si tenemos: $m1 = [[a]]$, $m2 = [[b]]$ el producto debe ser: $C = [[a * b]]$ La función *multMatrizEstandarPar* calcula esto correctamente, ya que:
 - Traspone $m2$, que sigue siendo $[[b]]$.
 - Calcula el producto punto entre las filas y columnas, que resulta en $[[a * b]]$.

Por lo tanto, el caso base se cumple.

- Paso Inductivo: Supongamos que la afirmación es cierta para matrices de tamaño $k \times k$. Es decir si $m1$ y $m2$ son matrices de tamaño $k \times k$, entonces *multMatrizEstandarPar*($m1, m2$) produce una matriz de tamaño $k \times k$. Ahora consideremos matrices de tamaño $(k+1) \times (k+1)$. Estas matrices pueden ser vistas como compuestas por submatrices:
 - Sea $m1$ dividido en cuatro bloques:
 - $A = m1[0:k][0:k]$
 - $B = m1[0:k][k]$
 - $C = m1[k][0:k]$
 - $D = m1[k][k]$
 - Similarmente para $m2$.

La multiplicación se puede expresar como:

$$C = A * E + B * F$$

$$C' = C * E + D * F$$

Donde E, F, C, D son submatrices correspondientes de $m2$.

Por hipótesis de inducción, sabemos que las multiplicaciones entre submatrices de tamaño $k \times k$ se calculan correctamente. Por lo tanto, al combinar estos resultados para formar la matriz resultante de tamaño $(k+1) \times (k+1)$, también se obtendrá correctamente.

- **Demostración por inducción estructural de la función multMatrizRec**

La función multMatrizRec realiza la multiplicación de dos matrices cuadradas m1 y m2, donde n una potencia de 2. La función tiene un enfoque recursivo, dividiendo matrices en submatrices hasta alcanzar el caso base de matrices 1X1

Caso base n = 1

Cuando n = 1, las matrices m1 y m2 tienen un solo elemento, evaluándose así la función:

$$\text{Vector}(\text{Vector}(m1(0)(0) * m2(0)(0)))$$

Esto representa una matriz 1X1, donde el único elemento es el producto de m1(0)(0) y m2(0)(0). Esta operación cumple la definición del producto de matrices escalares, por lo que el caso base es válido.

Hipótesis de inducción

Se supone que la función es correcta para matrices de tamaño n X n, es decir, para matrices cuadradas de tamaño n, esto produce una matriz resultante C tal que:

$$C[i][j] = \sum_{k=0}^{n-1} m1[i][k] \cdot m2[k][j]$$

Para $0 \leq i, j < n$.

Paso inductivo: Demostración para $2n$

Se quiere demostrar que la función también produce el resultado correcto para matrices $2n \times 2n$

División en submatrices

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Se tiene dos grupos de submatrices el grupo A y el grupo B, y la resultante de estas matrices siendo el grupo C.

El producto de estas matrices se representa explícitamente de la siguiente manera

$$\begin{aligned}
 C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
 C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
 C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
 C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
 \end{aligned}$$

Por hipótesis de inducción, se sabe que la función da resultados correctos para matrices $n \times n$. Por esto, cada cálculo intermedio es correcto.

Combinación de los bloques

Los bloques de la matriz resultante $C(C_{11}, C_{12}, C_{21}, C_{22})$ se combinan utilizando las operaciones de concatenación.

```

val c1 = c11.zip(c12).map({ case (i, j) => i ++ j })

val c2 = c21.zip(c22).map({ case (i, j) => i ++ j })

c1 ++ c2

```

Esto asegura que la matriz C se ensamble en una matriz cuadrada $2n \times 2n$.

Validez estructural

1. Acceso válido a las matrices: Ya que $m1$ y $m2$ son matrices válidas de tamaño $2n \times 2n$, todas las submatrices están dentro de los límites, asegurando que no haya accesos fuera de rango.
2. Dimensiones correctas: La división y recombinación garantizan que C tenga las dimensiones esperadas
3. Correctitud matemática: Por la hipótesis de inducción, los cálculos recursivos para $n \times n$ son correctos. La operación de bloques sigue las definiciones del producto matricial.

Por medio de la inducción estructural, se ha demostrado que la función `multMatrizRec` produce correctamente el producto de dos matrices cuadradas $m1$ y $m2$ de tamaño $n \times n$, donde n es una potencia de 2.

- **Demostración por inducción estructural de la función `multMatrizRecPar`**

La función realiza la multiplicación matricial. Además, busca paralelizar el cálculo de las submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ para optimizar el proceso.

Caso base n=1

En caso de matrices 1X1, el cálculo se reduce a:

$$\text{Vector}(\text{Vector}(m1(0)(0) \times m2(0)(0)))$$

Es correcto debido a que se trata de la multiplicación de escalares y, por tanto, cumple con la definición del producto matricial para matrices tamaño 1X1.

Hipótesis de inducción

Se supone que la función es correcta para tamaño $n \times n$, es decir, calcula correctamente el producto matricial $m1 \times m2$ para este tamaño, utilizando paralelización para submatrices si $k > 1$.

Paso inductivo: Demostración para $2n$

Para matrices $2n \times 2n$ la función divide las matrices $m1$ y $m2$ en submatrices de tamaño $n \times n$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Definiéndose de manera explícita:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

En esta función, los cálculos de las submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ se realizan de forma paralela utilizando la función `parallel`.

Corrección de cada cálculo parcial

1. Cada término `multMatrizRecPar(aij,bik)` realiza recursivamente el producto de dos matrices $n \times n$, y por la hipótesis de inducción, este cálculo es correcto.
2. La función `sumMatriz`, combina los resultados parciales $aij \times bki$, respetando la definición del producto matricial.

3. La función parallel garantiza que los cálculos C11, C12, C21, C22 se ejecuten simultáneamente. Ya que estos cálculos son independientes entre sí (ósea no comparten datos intermedios), la paralelización es válida y no afecta la corrección.

Combinación de resultados

Calculadas las submatrices de la matriz C, la función combina las submatrices utilizando operaciones como concat para ensamblar la matriz $2n \times 2n$. Esta operación es mecánica y no introduce errores, ya que ordena las submatrices en sus posiciones correctas.

Se demostró que la función es correcta para $n=1$ y también lo es para $n \times n$ como para $2n \times 2n$.

Por lo tanto, la función mulMatrizRecPar es correcta para cualquier matriz cuadrada de tamaño n , donde n es potencia de 2 y optimiza el cálculo mediante paralelización.

CONCLUSIONES

- Se estudió la implementación y comparación en la eficiencia del algoritmo de Strassen en sus variantes secuencial y paralela, estudiando los diferentes casos, en los que se pudo concluir que la paralelización de la función multStrassen es útil cuando se manejan matrices de tamaños grandes, que resulta siendo más eficiente que el algoritmo secuencial, se logró dividir problemas complejos en subproblemas más pequeños.
- El análisis realizado en torno al algoritmo de multiplicación estandar y su versión paralelizada, ha demostrado que la versión paralelizada de un algoritmo no siempre es la opción más eficiente en términos de tiempo de ejecución. A pesar de que la paralelización está diseñada para aprovechar múltiples núcleos de procesamiento y distribuir la carga de trabajo, esto

implica un costo adicional conocido como overhead, que incluye la creación, sincronización y comunicación entre hilos.

En los casos analizados con matrices de tamaños pequeños, medianos y grandes (2x2, 4x4, 8x8, 16x16, 32x32, 64x64, 128x128), el overhead asociado al paralelismo superó cualquier beneficio potencial, resultando en tiempos de ejecución más altos que los obtenidos con el algoritmo secuencial. Esto se reflejó en factores de aceleración menores a 1, lo que indica una desaceleración en lugar de mejora.