



Informe Taller 3: Programación Concurrente

Santiago Avalo Monsalve - 2359442

Daniel Gómez Cano - 2359396

Edward Stivens Pinto - 2359431

Manuel Jesús Rosero Zuñiga - 2176007

Trabajo presentado a:
Carlos Andrés Delgado S.

Universidad del Valle
Fundamentos de Programación Funcional y Concurrente
Ingeniería de Sistemas
Tuluá - Valle del Cauca
2024

1. Informe de Procesos

1.1 MultMatriz

Se implementó la Función multMatriz que recibe dos “Type Matriz” compuesto de un Vector de Vectores, utilizando las funciones “Transpuesta” y “ProdPunto” para realizar la multiplicación de un vector. En este caso vamos a utilizar los vectores m1 y m2 cuyos valores son:

$$m1 = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{ y } m2 = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

Local

```
> m1 = Vector1@1968 "Vector(Vector(1, 2), Vector(3, 4))"  
> m2 = Vector1@1969 "Vector(Vector(5, 6), Vector(7, 8))"  
> this = App$@1970
```

Tal como se puede observar la función recibe dichos vectores asignados para realizar las posteriores operaciones necesarias de la operación.

Local

```
> m1 = Vector1@1968 "Vector(Vector(1, 2), Vector(3, 4))"  
> m2 = Vector1@1969 "Vector(Vector(5, 6), Vector(7, 8))"  
> transpuestaM2 = Vector1@2005 "Vector(Vector(5, 7), Vector(6, 8))"  
> this = App$@1970
```

El siguiente paso es llamar a la función Transpuesta para cambiar el valor de m2 y asignarlo a un nuevo Val el cual es “llamado transpuestaM2”, esto debido al funcionamiento de Producto punto y la multiplicación de matrices. Por lo que necesitamos pasar de:

$$m2 = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \text{ a } transpuestaM2 = \begin{pmatrix} 5 & 7 \\ 6 & 8 \end{pmatrix}$$

Una vez hecho esto se crea un vector nuevo con la función tabulate que será el resultado de la función, el cual va tabulando en una nueva matriz el valor del producto punto entre m1 y transpuestaM2. Dando como resultado una matriz.

$$multMatrix = \begin{matrix} 19 & 22 \\ 43 & 50 \end{matrix}$$

1.2 MultMatrizPar

Se implementó la Función multMatrizPar que recibe dos “Type Matriz” para realizar la multiplicación de dos matrices, esta vez usando paralelización. En este caso vamos a volver a utilizar los valores de las matrices m1 y m2, por lo que nos debe de retornar los valores anteriores:

```
> m1 = Vector1@1993 "Vector(Vector(1, 2), Vector(3, 4))"
> m2 = Vector1@1994 "Vector(Vector(5, 6), Vector(7, 8))"
> transpuestaM2 = Vector1@1995 "Vector(Vector(5, 7), Vector(6, 8))"
> this = App$@1996
```

Tal como en el caso anterior lo primero que hace la función es recoger los valores de las matrices, para posteriormente utilizar las funciones adecuadas para la operación. O sea transpuesta y producto punto.

```
> m1$3 = Vector1@1993 "Vector(Vector(1, 2), Vector(3, 4))"
> transpuestaM2$2 = Vector1@1995 "Vector(Vector(5, 7), Vector(6, 8))"
  i = 0
  j = 0
> MODULE$ = App$@1996
```

Luego se empieza a llamar al bucle for cuyos valores son asignados a “i” y “j” respectivamente para explorar la matriz e ir usando producto Punto, al final nos retorna el valor de la multiplicación el cual es:

$$multMatrix = \begin{matrix} 19 & 22 \\ 43 & 50 \end{matrix}$$

1.3 MultMatrizRec

Se implemento la Función multMatrizRec que recibe dos “Matriz” y sigue las reglas establecidas anteriormente, solo que en este caso el punto de esta operación es realizar una multiplicación de matrices bajo una forma distinta, en este caso la manera recursiva que separa la matriz en varias submatrices que luego se sumarán y multiplicaran para llegar al objetivo, reutilizaremos las dos matrices anteriores para este punto. Además de esto se utilizaron dos nuevas funciones siendo la primera subMatriz y la segunda sumMatriz.

```
> m1 = Vector1@1984 "Vector(Vector(1, 2), Vector(3, 4))"  
> m2 = Vector1@1985 "Vector(Vector(5, 6), Vector(7, 8))"  
n = 2  
mitad = 1  
> a11 = Vector1@1986 "Vector(Vector(1))"  
> a12 = Vector1@1987 "Vector(Vector(2))"  
> a21 = Vector1@1988 "Vector(Vector(3))"  
> a22 = Vector1@1989 "Vector(Vector(4))"  
> b11 = Vector1@1990 "Vector(Vector(5))"  
> b12 = Vector1@1991 "Vector(Vector(6))"  
> b21 = Vector1@1992 "Vector(Vector(7))"  
> this = App$@1983
```

Lo primero que hace la función es obtener el valor del tamaño de m1 para saber en cuántas mitades toca partir la matriz, lo cual se asigna a los Val “n” y “mitad” respectivamente. En caso de que sea un n de tamaño 1 retornará una multiplicación de los vectores en su primer elemento.

Posteriormente se aplica la función subMatriz a ambas matrices retornando variables con el valor de cada sección de la matriz, en este caso cada una equivale a un solo valor, pero la función puede retornar más de uno.

```

> m1 = Vector1@1984 "Vector(Vector(1, 2), Vector(3, 4))"
> m2 = Vector1@1985 "Vector(Vector(5, 6), Vector(7, 8))"
n = 2
mitad = 1
> a11 = Vector1@1986 "Vector(Vector(1))"
> a12 = Vector1@1987 "Vector(Vector(2))"
> a21 = Vector1@1988 "Vector(Vector(3))"
> a22 = Vector1@1989 "Vector(Vector(4))"
> b11 = Vector1@1990 "Vector(Vector(5))"
> b12 = Vector1@1991 "Vector(Vector(6))"
> b21 = Vector1@1992 "Vector(Vector(7))"
> b22 = Vector1@2048 "Vector(Vector(8))"
> c11 = Vector1@2049 "Vector(Vector(19))"
> c12 = Vector1@2050 "Vector(Vector(22))"
> c21 = Vector1@2051 "Vector(Vector(43))"
> c22 = Vector1@2052 "Vector(Vector(50))"

```

Luego se realizan las sumas y multiplicaciones de las submatrices, aunque al ser una operación recursiva se tiene que realizar un poco de recursión lineal para el cálculo, por lo que cada una de las submatrices tiene que realizar todo el proceso desde cero, una vez hecho eso y teniendo el valor de la submatriz se realiza la suma de dos de ellas según la fórmula de multiplicación recursiva. Cabe aclarar que la función sumMatriz utiliza la función de tabulate para poder asignarle a una nueva matriz el valor de dicha suma.

▽ Local

```

> x0$1 = Tuple2@2069 "(Vector(19),Vector(22))"
> MODULE$ = App$@1983

```

Finalmente se tiene que hacer la combinación de las operaciones y lo primero que se tiene que hacer es la creación de las filas de la nueva matriz, la cual es en este caso compuesta por c11 y c21.

✓ Local

```
> x0$1 = Tuple2@2075 "(Vector(43),Vector(50))"  
> MODULE$ = App$@1983
```

Después se asigna el valor de la siguiente fila de submatrices siendo en este caso las de c12 y c22 respectivamente. Finalmente se tiene que combinar ambas filas por lo que se tiene que usar la función map que nos permite concatenar cada una de las filas necesarias para la operación.

1.4 MultMatrizRecPar

Se implemento la Función multMatrizRecPar que recibe dos “Matriz” y sigue las reglas establecidas anteriormente, es más esta es una versión modificada de la anterior función solo que aplicando los principios de paralelización, por lo que vamos a utilizar de nuevo las mismas matrices. Ya no solo para ahorrar trabajo sino para poder comparar de mejor manera

✓ Local

```
> m1 = Vector1@1977 "Vector(Vector(1, 2), Vector(3, 4))"  
> m2 = Vector1@1978 "Vector(Vector(5, 6), Vector(7, 8))"  
n = 2  
mitad = 1  
> this = App$@1979
```

Tal como se puede observar lo primero que hace la función es lo mismo que la anterior función al inicio, hallar el valor de la mitad y no se usa paralelización aún porque es innecesario al ser solo una división.

✓ VARIABLES

✓ Local

```
> m1 = Vector1@1977 "Vector(Vector(1, 2), Vector(3, 4))"  
> m2 = Vector1@1978 "Vector(Vector(5, 6), Vector(7, 8))"  
n = 2  
mitad = 1  
> a11 = Vector1@2025 "Vector(Vector(1))"  
> a12 = Vector1@2026 "Vector(Vector(2))"  
> a21 = Vector1@2027 "Vector(Vector(3))"  
> a22 = Vector1@2028 "Vector(Vector(4))"  
> this = App$@1979
```

La función procede a usar la paralelización para realizar el cálculo de las submatrices “a” que son retornadas de esta manera, cada una para ser utilizadas posteriormente.

```

▽ Local
> m1 = Vector1@1977 "Vector(Vector(1, 2), Vector(3, 4))"
> m2 = Vector1@1978 "Vector(Vector(5, 6), Vector(7, 8))"
n = 2
mitad = 1
> a11 = Vector1@2025 "Vector(Vector(1))"
> a12 = Vector1@2026 "Vector(Vector(2))"
> a21 = Vector1@2027 "Vector(Vector(3))"
> a22 = Vector1@2028 "Vector(Vector(4))"
> b11 = Vector1@2047 "Vector(Vector(5))"
> b12 = Vector1@2048 "Vector(Vector(6))"
> b21 = Vector1@2049 "Vector(Vector(7))"
> b22 = Vector1@2050 "Vector(Vector(8))"
> c11 = Vector1@2121 "Vector(Vector(19))"
> c12 = Vector1@2122 "Vector(Vector(22))"
> c21 = Vector1@2123 "Vector(Vector(43))"
> c22 = Vector1@2124 "Vector(Vector(50))"
> this = App$@1979

```

Nuevamente se realiza lo mismo que en la función secuencial, siendo retornado el valor ahora de las submatrices “b” y posteriormente calculado los valores de “c” mediante la misma fórmula que se usó con anterioridad.

```

▽ Local
> x0$1 = Tuple2@2140 "(Vector(19),Vector(22))"
> MODULE$ = App$@1984

```

Al tener ya todas las operaciones hechas se pasa a la creación de las matrices primero uniendo los resultados de c11 y c21, resultando en la fila izquierda

✓ Local

```
> x0$1 = Tuple2@2142 "(Vector(43),Vector(50))"  
> MODULE$ = App$@1984
```

Al igual que antes también se pasa a tomar los valores del resultado de c12 y c22, concatenando antes de volver a utilizar la última parte de la función. Esta última parte de la función no necesita parallel ni task ni nada por el estilo para realizar lo propuesto, por lo que no fue modificada.

1.5 MultStrassen

Se implementó la Función multStrassen que recibe dos “Matriz” y sigue las reglas establecidas anteriormente, solo que en esta ocasión lo que se va a realizar es una serie de operaciones que incluye restaMatrices, pero tal como antes usaremos las mismas matrices.

✓ Local

```
> m1 = Vector1@1979 "Vector(Vector(1, 2), Vector(3, 4))"  
> m2 = Vector1@1980 "Vector(Vector(5, 6), Vector(7, 8))"  
  n = 2  
  mitad = 1  
> a11 = Vector1@1981 "Vector(Vector(1))"  
> a12 = Vector1@1982 "Vector(Vector(2))"  
> a21 = Vector1@1983 "Vector(Vector(3))"  
> a22 = Vector1@1984 "Vector(Vector(4))"  
> b11 = Vector1@1985 "Vector(Vector(5))"  
> b12 = Vector1@1986 "Vector(Vector(6))"  
> b21 = Vector1@1987 "Vector(Vector(7))"  
> b22 = Vector1@1988 "Vector(Vector(8))"  
> this = App$@1989
```

La función empieza tal como las dos anteriores, obteniendo la mitad de las matrices y recogiendo las submatrices de ambas para tenerlas listas para la operación.

✓ Local

```
> m1 = Vector1@1979 "Vector(Vector(1, 2), Vector(3, 4))"  
> m2 = Vector1@1980 "Vector(Vector(5, 6), Vector(7, 8))"  
n = 2  
mitad = 1  
> a11 = Vector1@1981 "Vector(Vector(1))"  
> a12 = Vector1@1982 "Vector(Vector(2))"  
> a21 = Vector1@1983 "Vector(Vector(3))"  
> a22 = Vector1@1984 "Vector(Vector(4))"  
> b11 = Vector1@1985 "Vector(Vector(5))"  
> b12 = Vector1@1986 "Vector(Vector(6))"  
> b21 = Vector1@1987 "Vector(Vector(7))"  
> b22 = Vector1@1988 "Vector(Vector(8))"  
> p1 = Vector1@2021 "Vector(Vector(-2))"  
> p2 = Vector1@2022 "Vector(Vector(24))"  
> p3 = Vector1@2023 "Vector(Vector(35))"  
> p4 = Vector1@2024 "Vector(Vector(8))"  
> p5 = Vector1@2025 "Vector(Vector(65))"  
> p6 = Vector1@2026 "Vector(Vector(-30))"  
> p7 = Vector1@2027 "Vector(Vector(-22))"  
> this = App$@1989
```

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

Ahora lo que sigue es hacer las operaciones del algoritmo de Strassen, por lo que se usan las siguientes funciones para que se lleve a cabo de la manera correcta: sumMatriz y restaMatriz, ambas con un funcionamiento similar pero cuyo resultado debe ser distinto por obvias razones, luego se hace un llamado recursivo a la propia multiplicación en cada una de las multiplicaciones del algoritmo, ya sea $A11 \cdot B12$ o $S2 \cdot B22$. El resultado de cada una de estas operaciones posteriormente es asignado a una variable Px para realizar la última parte del algoritmo.

```
> a11 = Vector1@1981 "Vector(Vector(1))"  
> a12 = Vector1@1982 "Vector(Vector(2))"  
> a21 = Vector1@1983 "Vector(Vector(3))"  
> a22 = Vector1@1984 "Vector(Vector(4))"  
> b11 = Vector1@1985 "Vector(Vector(5))"  
> b12 = Vector1@1986 "Vector(Vector(6))"  
> b21 = Vector1@1987 "Vector(Vector(7))"  
> b22 = Vector1@1988 "Vector(Vector(8))"  
> p1 = Vector1@2021 "Vector(Vector(-2))"  
> p2 = Vector1@2022 "Vector(Vector(24))"  
> p3 = Vector1@2023 "Vector(Vector(35))"  
> p4 = Vector1@2024 "Vector(Vector(8))"  
> p5 = Vector1@2025 "Vector(Vector(65))"  
> p6 = Vector1@2026 "Vector(Vector(-30))"  
> p7 = Vector1@2027 "Vector(Vector(-22))"  
> c11 = Vector1@2045 "Vector(Vector(19))"  
> c12 = Vector1@2046 "Vector(Vector(22))"  
> c21 = Vector1@2047 "Vector(Vector(43))"  
> c22 = Vector1@2048 "Vector(Vector(50))"  
> this = App$@1989
```

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

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

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

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

Lo último previo a la concatenación es esta parte en la que se realizan las operaciones y se toma el orden otorgado en cada una para su realización, por ejemplo, en el caso de C11 tenemos el código de esta manera:

```
val c11 = sumMatriz(restaMatriz(sumMatriz(p5, p4), p2), p6)
```

en el que primero se suma P5 y P4, para luego restarse a P2 y finalmente sumar todo con P6. Una vez cada una de las operaciones está realizada se asigna a cada uno de los valores a un Val Cx.

```
✓ Local
> x0$1 = Tuple2@2071 "(Vector(19),Vector(22))"
> MODULE$ = App$@1989
```

```
✓ Local
> x0$1 = Tuple2@2073 "(Vector(43),Vector(50))"
> MODULE$ = App$@1989
```

Finalmente, y para no repetir lo dicho en las anteriores partes del informe, el programa recoge lo retornado por todas las operaciones para luego empezar a concatenar, asignando a C11 y C21 como fila izquierda para luego al resto asignarlo como fila derecha. Retornando el valor de la matriz resultante.

1.7 MultStrassenPar

Se implementó la Función multStrassenPar que recibe dos “Matriz” y sigue las reglas establecidas anteriormente, en esta ocasión seguirá lo establecido por el algoritmo de Strassen a la par de una paralelización, se seguirán usando las mismas matrices para acelerar el proceso.

```
> m1 = Vector1@2021 "Vector(Vector(1, 2), Vector(3, 4))"  
> m2 = Vector1@2022 "Vector(Vector(5, 6), Vector(7, 8))"  
n = 2  
mitad = 1  
> a11 = Vector1@2023 "Vector(Vector(1))"  
> a12 = Vector1@2024 "Vector(Vector(2))"  
> a21 = Vector1@2025 "Vector(Vector(3))"  
> a22 = Vector1@2026 "Vector(Vector(4))"  
> b11 = Vector1@2027 "Vector(Vector(5))"  
> b12 = Vector1@2028 "Vector(Vector(6))"  
> b21 = Vector1@2029 "Vector(Vector(7))"  
> b22 = Vector1@2030 "Vector(Vector(8))"  
> this = App$@2031
```

Tal como antes la operación prosigue con el asignamiento de la Val Mitad para poder realizar el uso de subMatriz antes de poder proseguir, asignando una submatriz a Axx o Bxx utilizando la paralelización al llamar a la función subMatriz.

```

> m1 = Vector1@2021 "Vector(Vector(1, 2), Vector(3, 4))"
> m2 = Vector1@2022 "Vector(Vector(5, 6), Vector(7, 8))"
n = 2
mitad = 1

> a11 = Vector1@2023 "Vector(Vector(1))"
> a12 = Vector1@2024 "Vector(Vector(2))"
> a21 = Vector1@2025 "Vector(Vector(3))"
> a22 = Vector1@2026 "Vector(Vector(4))"
> b11 = Vector1@2027 "Vector(Vector(5))"
> b12 = Vector1@2028 "Vector(Vector(6))"
> b21 = Vector1@2029 "Vector(Vector(7))"
> b22 = Vector1@2030 "Vector(Vector(8))"
> p1 = Vector1@2062 "Vector(Vector(-2))"
> p2 = Vector1@2063 "Vector(Vector(24))"
> p3 = Vector1@2064 "Vector(Vector(35))"
> p4 = Vector1@2065 "Vector(Vector(8))"
> p5 = Vector1@2066 "Vector(Vector(65))"
> p6 = Vector1@2067 "Vector(Vector(-30))"
> p7 = Vector1@2068 "Vector(Vector(-22))"
> this = App$@2031

```

Posteriormente se utilizan las funciones sumMatriz y restaMatriz junto a un llamado recursivo a la función principal para poder sacar los valores de P, en este caso no se necesitó el uso de paralelización.

```

> a22 = Vector1@2026 "Vector(Vector(4))"
> b11 = Vector1@2027 "Vector(Vector(5))"
> b12 = Vector1@2028 "Vector(Vector(6))"
> b21 = Vector1@2029 "Vector(Vector(7))"
> b22 = Vector1@2030 "Vector(Vector(8))"
> p1 = Vector1@2062 "Vector(Vector(-2))"
> p2 = Vector1@2063 "Vector(Vector(24))"
> p3 = Vector1@2064 "Vector(Vector(35))"
> p4 = Vector1@2065 "Vector(Vector(8))"
> p5 = Vector1@2066 "Vector(Vector(65))"
> p6 = Vector1@2067 "Vector(Vector(-30))"
> p7 = Vector1@2068 "Vector(Vector(-22))"
> c11 = Vector1@2111 "Vector(Vector(19))"
> c12 = Vector1@2112 "Vector(Vector(22))"
> c21 = Vector1@2113 "Vector(Vector(43))"
> c22 = Vector1@2114 "Vector(Vector(50))"
> this = App$@2031

```

Para lo que sí se realizó el uso de la paralelización fue esta sección, la de cada una de las Cx se les asignó un valor mediante las distintas operaciones necesarias del algoritmo. Antes de ser concatenadas.

```

> x0$1 = Tuple2@2137 "(Vector(19),Vector(22))"
> MODULE$ = App$@2031

```

```

> x0$1 = Tuple2@2139 "(Vector(43),Vector(50))"
> MODULE$ = App$@2031

```

Ya para la concatenación se realizó el mismo procedimiento de antes, recogiendo C11 y C21 para asignarlo a la fila Izquierda dejando al resto como Fila Derecha. Concatenando mediante un map.

2. Informe de Paralelización

2.1 prodPuntoParD

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

Ley de Amdahl

$$S = 1/(1 - P) + P/N$$

$P = 0,95$ //Puesto que la paralelización está aplicada para que se aplique la multiplicación en paralelo menos la suma final

$N = 8$ //Cantidad estimada de procesadores

$$S = 1/(1 - 0,95) + 0,95/8$$

$$S = 6,71$$

Benchmarking para vectores de tamaño 58:	Benchmarking para vectores de tamaño 79:	Benchmarking para vectores de tamaño 52:
Unable to create a system terminal		
Iteración 1: Secuencial: 0.0997 ms Paralelo: 0.9417 ms	Iteración 1: Secuencial: 0.0197 ms Paralelo: 0.4441 ms	Iteración 1: Secuencial: 0.015 ms Paralelo: 0.3405 ms
Iteración 2: Secuencial: 0.0438 ms Paralelo: 0.718 ms	Iteración 2: Secuencial: 0.0187 ms Paralelo: 0.3782 ms	Iteración 2: Secuencial: 0.0146 ms Paralelo: 0.3224 ms
Iteración 3: Secuencial: 0.0361 ms Paralelo: 0.5842 ms	Iteración 3: Secuencial: 0.0196 ms Paralelo: 0.4 ms	Iteración 3: Secuencial: 0.012 ms Paralelo: 0.1627 ms
Iteración 4: Secuencial: 0.0288 ms Paralelo: 0.5432 ms	Iteración 4: Secuencial: 0.022 ms Paralelo: 0.4571 ms	Iteración 4: Secuencial: 0.0161 ms Paralelo: 0.1866 ms
Iteración 5: Secuencial: 0.0336 ms Paralelo: 0.4693 ms	Iteración 5: Secuencial: 0.0153 ms Paralelo: 0.4403 ms	Iteración 5: Secuencial: 0.018 ms Paralelo: 0.0838 ms
Iteración 6: Secuencial: 0.0233 ms Paralelo: 0.6079 ms	Iteración 6: Secuencial: 0.013 ms Paralelo: 0.4622 ms	Iteración 6: Secuencial: 0.0268 ms Paralelo: 0.1184 ms
Iteración 7: Secuencial: 0.0149 ms Paralelo: 0.6804 ms	Iteración 7: Secuencial: 0.011 ms Paralelo: 0.3011 ms	Iteración 7: Secuencial: 0.0116 ms Paralelo: 0.0982 ms
Iteración 8: Secuencial: 0.0225 ms Paralelo: 0.4888 ms	Iteración 8: Secuencial: 0.0159 ms Paralelo: 0.402 ms	Iteración 8: Secuencial: 0.0122 ms Paralelo: 0.1026 ms
Iteración 9: Secuencial: 0.0289 ms Paralelo: 0.4665 ms	Iteración 9: Secuencial: 0.0225 ms Paralelo: 0.3402 ms	Iteración 9: Secuencial: 0.0094 ms Paralelo: 0.0922 ms
Iteración 10: Secuencial: 0.0144 ms Paralelo: 2.3042 ms	Iteración 10: Secuencial: 0.0168 ms Paralelo: 0.3663 ms	Iteración 10: Secuencial: 0.0174 ms Paralelo: 0.0932 ms

Benchmarking para vectores de tamaño 40:

Iteración 1:
Secuencial: 0.0125 ms
Paralelo: 0.1071 ms

Iteración 2:
Secuencial: 0.0308 ms
Paralelo: 0.0831 ms

Iteración 3:
Secuencial: 0.0079 ms
Paralelo: 0.1158 ms

Iteración 4:
Secuencial: 0.0111 ms
Paralelo: 0.0771 ms

Iteración 5:
Secuencial: 0.0092 ms
Paralelo: 0.0844 ms

Iteración 6:
Secuencial: 0.0096 ms
Paralelo: 0.0873 ms

Iteración 7:
Secuencial: 0.0098 ms
Paralelo: 0.0914 ms

Iteración 8:
Secuencial: 0.0132 ms
Paralelo: 0.1259 ms

Iteración 9:
Secuencial: 0.0105 ms
Paralelo: 0.0941 ms

Iteración 10:
Secuencial: 0.0115 ms
Paralelo: 0.0742 ms

Benchmarking para vectores de tamaño 81:

Iteración 1:
Secuencial: 0.0125 ms
Paralelo: 0.1519 ms

Iteración 2:
Secuencial: 0.0147 ms
Paralelo: 0.3487 ms

Iteración 3:
Secuencial: 0.0175 ms
Paralelo: 0.3653 ms

Iteración 4:
Secuencial: 0.0477 ms
Paralelo: 0.219 ms

Iteración 5:
Secuencial: 0.0141 ms
Paralelo: 0.346 ms

Iteración 6:
Secuencial: 0.0187 ms
Paralelo: 0.6318 ms

Iteración 7:
Secuencial: 0.0149 ms
Paralelo: 0.2668 ms

Iteración 8:
Secuencial: 0.0126 ms
Paralelo: 0.3475 ms

Iteración 9:
Secuencial: 0.0195 ms
Paralelo: 0.316 ms

Iteración 10:
Secuencial: 0.0155 ms
Paralelo: 0.3701 ms

2.2 multMatrizRecPar

```
def multMatrizRecPar(m1: Matriz, m2: Matriz): Matriz = {
    val n = m1.length

    if (n == 1) {
        Vector(Vector(m1(0)(0) * m2(0)(0)))
    } else {
        val mitad = n / 2

        val ((a11,a12),(a21,a22)) = parallel(
            parallel(subMatriz(m1, 0, 0, mitad),subMatriz(m1, 0, mitad, mitad)),
            parallel(subMatriz(m1, mitad, 0, mitad),subMatriz(m1, mitad, mitad, mitad))
        )

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

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

        (c11 ++ c21).zip(c12 ++ c22).map { case (filaIzq, filaDer) => filaIzq ++ filaDer }
    }
}
```

Para la multiplicación de matrices recursivas se utilizó parallel en el conjunto de métodos para hallar los valores de a, b y c.

Ley de Amdahl

$$S = 1/(1 - P) + P/N$$

P = 0,7 //Puesto que la paralelización está aplicada para cada método menos para la organización final de las matrices, menos en los llamados recursivos

N = 8 //Cantidad estimada de procesadores

$$S = 1/(1 - 0,7) + 0,7/8$$

$$S = 3,16$$

Benchmarking para matrices :	vectores tamaño 2^3:	vectores tamaño 2^4:
vectores tamaño 2^2:	Iteración 1: Secuencial: 0.4268 ms Paralelo: 0.4347 ms	Iteración 1: Secuencial: 1.8621 ms Paralelo: 0.8014 ms
Unable to create a system tensor.	Iteración 2: Secuencial: 0.5544 ms Paralelo: 0.3355 ms	Iteración 2: Secuencial: 1.5368 ms Paralelo: 0.644 ms
Iteración 1: Secuencial: 0.585 ms Paralelo: 0.3531 ms	Iteración 3: Secuencial: 0.684 ms Paralelo: 0.3758 ms	Iteración 3: Secuencial: 1.7137 ms Paralelo: 0.7926 ms
Iteración 2: Secuencial: 0.1233 ms Paralelo: 0.4417 ms	Iteración 4: Secuencial: 0.5907 ms Paralelo: 0.4676 ms	Iteración 4: Secuencial: 1.7401 ms Paralelo: 0.6631 ms
Iteración 3: Secuencial: 0.1159 ms Paralelo: 0.2958 ms	Iteración 5: Secuencial: 0.5393 ms Paralelo: 0.3467 ms	Iteración 5: Secuencial: 1.9564 ms Paralelo: 0.6472 ms
Iteración 4: Secuencial: 0.0856 ms Paralelo: 0.2217 ms	Iteración 6: Secuencial: 0.6624 ms Paralelo: 0.3966 ms	Iteración 6: Secuencial: 2.8313 ms Paralelo: 0.6854 ms
Iteración 5: Secuencial: 0.0648 ms Paralelo: 0.4547 ms	Iteración 7: Secuencial: 0.3218 ms Paralelo: 0.2813 ms	Iteración 7: Secuencial: 1.7838 ms Paralelo: 2.4801 ms
Iteración 6: Secuencial: 0.0526 ms Paralelo: 0.1305 ms	Iteración 8: Secuencial: 0.3384 ms Paralelo: 0.3574 ms	Iteración 8: Secuencial: 1.5485 ms Paralelo: 0.594 ms
Iteración 7: Secuencial: 0.0742 ms Paralelo: 0.1865 ms	Iteración 9: Secuencial: 0.4143 ms Paralelo: 0.4013 ms	Iteración 9: Secuencial: 1.7892 ms Paralelo: 0.5699 ms
Iteración 8: Secuencial: 0.035 ms Paralelo: 0.1354 ms	Iteración 10: Secuencial: 0.2626 ms Paralelo: 0.3576 ms	Iteración 10: Secuencial: 1.4962 ms Paralelo: 0.6784 ms
Iteración 9: Secuencial: 0.0457 ms Paralelo: 0.1996 ms		
Iteración 10: Secuencial: 0.0341 ms Paralelo: 0.1313 ms		

vectores tamaño 2^5:	vectores tamaño 2^6:
Iteración 1: Secuencial: 12.9029 ms Paralelo: 4.047 ms	Iteración 1: Secuencial: 106.2269 ms Paralelo: 34.2627 ms
Iteración 2: Secuencial: 13.51 ms Paralelo: 4.9679 ms	Iteración 2: Secuencial: 108.3149 ms Paralelo: 35.4146 ms
Iteración 3: Secuencial: 13.1092 ms Paralelo: 4.1444 ms	Iteración 3: Secuencial: 107.4828 ms Paralelo: 32.7485 ms
Iteración 4: Secuencial: 14.5234 ms Paralelo: 3.9654 ms	Iteración 4: Secuencial: 105.0222 ms Paralelo: 33.0443 ms
Iteración 5: Secuencial: 13.6407 ms Paralelo: 3.997 ms	Iteración 5: Secuencial: 109.9982 ms Paralelo: 32.3677 ms
Iteración 6: Secuencial: 14.9686 ms Paralelo: 4.0261 ms	Iteración 6: Secuencial: 106.7442 ms Paralelo: 34.2858 ms
Iteración 7: Secuencial: 12.273 ms Paralelo: 4.0575 ms	Iteración 7: Secuencial: 106.6171 ms Paralelo: 33.294 ms
Iteración 8: Secuencial: 14.2569 ms Paralelo: 4.4706 ms	Iteración 8: Secuencial: 106.2953 ms Paralelo: 32.672 ms
Iteración 9: Secuencial: 15.6111 ms Paralelo: 4.136 ms	Iteración 9: Secuencial: 105.6442 ms Paralelo: 33.1572 ms
Iteración 10: Secuencial: 13.7088 ms Paralelo: 4.2795 ms	Iteración 10: Secuencial: 106.0924 ms Paralelo: 32.7962 ms

2.3 multStrassenPar

```
def multStrassenPar(m1: Matriz, m2: Matriz) : Matriz = {
    val n = m1.length
    if (n == 1) {
        Vector(Vector(m1(0)(0) * m2(0)(0)))
    } else {
        val mitad = n / 2

        val ((a11,a12),(a21,a22)) = parallel(
            parallel(subMatriz(m1, 0, 0, mitad),subMatriz(m1, 0, mitad, mitad)),
            parallel(subMatriz(m1, mitad, 0, mitad),subMatriz(m1, mitad, mitad, mitad))
        )

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

        val p1 = multStrassenPar(a11, restaMatriz(b12, b22))
        val p2 = multStrassenPar(sumMatriz(a11, a12), b22)
        val p3 = multStrassenPar(sumMatriz(a21, a22), b11)
        val p4 = multStrassenPar(a22, restaMatriz(b21, b11))
        val p5 = multStrassenPar(sumMatriz(a11, a22), sumMatriz(b11, b22))
        val p6 = multStrassenPar(restaMatriz(a12, a22), sumMatriz(b21, b22))
        val p7 = multStrassenPar(restaMatriz(a11, a21), sumMatriz(b11, b12))

        val ((c11,c12),(c21,c22)) = parallel(
            parallel(
                sumMatriz(restaMatriz(sumMatriz(p5, p4), p2), p6),
                sumMatriz(p1, p2)
            ),
            parallel(
                sumMatriz(p3, p4),
                restaMatriz(restaMatriz(sumMatriz(p5, p1), p3), p7)
            )
        )

        (c11 ++ c21).zip(c12 ++ c22).map { case (filaIzq, filaDer) => filaIzq ++ filaDer }
    }
}
```

Para la multiplicación de matrices de strassen se utilizó parallel en el conjunto de métodos para hallar los valores de a, b y c.

Ley de Amdahl

$$S = 1/(1 - P) + P/N$$

P = 0,9 //Puesto que la paralelización está aplicada para cada método menos para la organización final de las matrices, incluso en los llamados recursivos

N = 8 //Cantidad estimada de procesadores

$$S = 1/(1 - 0,9) + 0,9/8$$

$$S = 4,71$$

Benchmarking para matrices vectores tamaño 2^1: Unable to create a system	vectores tamaño 2^2: Iteración 1: Secuencial: 0.2441 ms Paralelo: 0.3017 ms	vectores tamaño 2^3: Iteración 1: Secuencial: 0.701 ms Paralelo: 3.6517 ms
Iteración 2: Secuencial: 0.146 ms Paralelo: 0.4329 ms	Iteración 2: Secuencial: 0.0777 ms Paralelo: 0.7228 ms	Iteración 2: Secuencial: 0.4569 ms Paralelo: 4.023 ms
Iteración 3: Secuencial: 0.0652 ms Paralelo: 0.2865 ms	Iteración 3: Secuencial: 0.0884 ms Paralelo: 1.0036 ms	Iteración 3: Secuencial: 0.3095 ms Paralelo: 4.1327 ms
Iteración 4: Secuencial: 0.0593 ms Paralelo: 0.2816 ms	Iteración 4: Secuencial: 0.1561 ms Paralelo: 1.5079 ms	Iteración 4: Secuencial: 0.3332 ms Paralelo: 4.1479 ms
Iteración 5: Secuencial: 0.0594 ms Paralelo: 0.2318 ms	Iteración 5: Secuencial: 0.0856 ms Paralelo: 0.888 ms	Iteración 5: Secuencial: 0.2995 ms Paralelo: 4.5364 ms
Iteración 6: Secuencial: 0.0568 ms Paralelo: 0.2364 ms	Iteración 6: Secuencial: 0.136 ms Paralelo: 0.7238 ms	Iteración 6: Secuencial: 0.2514 ms Paralelo: 3.548 ms
Iteración 7: Secuencial: 0.0409 ms Paralelo: 0.201 ms	Iteración 7: Secuencial: 0.1223 ms Paralelo: 0.686 ms	Iteración 7: Secuencial: 0.2356 ms Paralelo: 3.8489 ms
Iteración 8: Secuencial: 0.1674 ms Paralelo: 0.2158 ms	Iteración 8: Secuencial: 0.1029 ms Paralelo: 0.6471 ms	Iteración 8: Secuencial: 0.3027 ms Paralelo: 3.5722 ms
Iteración 9: Secuencial: 0.017 ms Paralelo: 0.1245 ms	Iteración 9: Secuencial: 0.1092 ms Paralelo: 1.0178 ms	Iteración 9: Secuencial: 0.2263 ms Paralelo: 3.8057 ms
Iteración 10: Secuencial: 0.0169 ms Paralelo: 0.1147 ms	Iteración 10: Secuencial: 0.1223 ms Paralelo: 0.7162 ms	Iteración 10: Secuencial: 0.233 ms Paralelo: 3.6382 ms
 vectores tamaño 2^4: Iteración 1: Secuencial: 1.6551 ms Paralelo: 24.7877 ms	 vectores tamaño 2^5: Iteración 1: Secuencial: 13.4505 ms Paralelo: 171.9023 ms	
Iteración 2: Secuencial: 1.607 ms Paralelo: 24.6232 ms	Iteración 2: Secuencial: 11.2646 ms Paralelo: 178.3751 ms	
Iteración 3: Secuencial: 1.6356 ms Paralelo: 25.365 ms	Iteración 3: Secuencial: 10.8936 ms Paralelo: 175.0005 ms	
Iteración 4: Secuencial: 1.6102 ms Paralelo: 25.9288 ms	Iteración 4: Secuencial: 14.8533 ms Paralelo: 184.1707 ms	
Iteración 5: Secuencial: 1.6318 ms Paralelo: 33.5447 ms	Iteración 5: Secuencial: 12.3508 ms Paralelo: 174.2309 ms	
Iteración 6: Secuencial: 1.8062 ms Paralelo: 27.5514 ms	Iteración 6: Secuencial: 11.3582 ms Paralelo: 175.5331 ms	
Iteración 7: Secuencial: 2.4259 ms Paralelo: 25.763 ms	Iteración 7: Secuencial: 12.3487 ms Paralelo: 174.3174 ms	
Iteración 8: Secuencial: 1.6707 ms Paralelo: 33.9851 ms	Iteración 8: Secuencial: 10.2626 ms Paralelo: 193.7365 ms	
Iteración 9: Secuencial: 2.2948 ms Paralelo: 25.8173 ms	Iteración 9: Secuencial: 11.7246 ms Paralelo: 175.1681 ms	
Iteración 10: Secuencial: 1.9131 ms Paralelo: 24.6705 ms	Iteración 10: Secuencial: 11.7569 ms Paralelo: 176.2801 ms	

3. Informe de Corrección

3.1 Transpuesta

$f: Matriz(nxn)(i, j) \rightarrow Matriz(nxn)(j, i)$

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

$P_f:$

$n =$ Tamaño de la matriz

$Vector.tabulate(n, n)((i, j) \Rightarrow m(j)(i)) // Crea un nueva matriz nxn$

$((i, j) \Rightarrow m(j)(i)) // Aplica a cada posición (i, j) de la matriz inicial el valor transpuesto$

Caso base

$m1: \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$

$Vector.tabulate(2, 2)((i, j) \Rightarrow m(j)(i))$

$m2: \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$

3.2 prodPunto

$f: V_1(n) \cdot V_2(n) \rightarrow V_1(0) * V_2(0) + \dots + V_1(n) * V_2(n)$

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

$P_f:$

$(v1 zip v2).map { case (i, j) \Rightarrow i * j }.sum$

$(v1 zip v2) // Combina cada valor de posiciones iguales con el otro valor del vector$
 $map { case (i, j) \Rightarrow i * j } // Aplica para cada par anteriormente combinado,$
 $multiplica cada par, genera un nuevo vector con la multiplicación de los dos datos$
 $sum // Suma cada valor del vector$

Caso base

$$v1: \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad v2: \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

$$(v1 \text{ zip } v2): \begin{pmatrix} (1, 1) \\ (0, 1) \\ (1, 0) \end{pmatrix}$$

$$\text{map } \{ \text{case } (i, j) \Rightarrow i * j \}: \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

$$\text{sum: } 1 + 0 + 0 = 1 //$$

3.3 multMatriz

$f: \text{Matriz}_1(n \times n) \times \text{Matriz}_2(n \times n) = \text{Matriz}_3(n \times n)$

```
def multMatriz(m1: Matriz, m2: Matriz): Matriz = {
    val transpuestaM2 = transpuesta(m2)
    Vector.tabulate(m1.length, m2.length) { (i, j) =>
        prodPunto(m1(i), transpuestaM2(j))
    }
}
```

$P_f:$

Vector.tabulate(m1.length, m2.length) { (i, j) => prodPunto(m1(i), transpuestaM2(j)) }
(i, j) => prodPunto(m1(i), transpuestaM2(j)) //Para cada valor de la nueva matriz se realiza el valor de producto punto entre la fila de m_1 y la columna de m_2

Caso base

$$m1: \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \quad m2: \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Vector.tabulate(2, 2) { (i, j) => prodPunto(m1(i), transpuestaM2(j))}

$$\text{mR: } \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}$$

3.4 SubMatriz

f: Matriz (n x n) -> submatriz(m x m, m <= n)

```
def subMatriz(m: Matriz, i: Int, j: Int, l: Int): Matriz = {
    m.slice(i, i + l).map(_.slice(j, j + l))
}
```

P_f:

Parámetros:

m = Matriz (n x n), donde "n" es igual a: 2^m, para cualquier número natural de "m"

i = posición de inicio de la submatriz en i, filas

j = posición de inicio de la submatriz en j, columnas

l = largo de la matriz a partir de los puntos (i, j)

m.slice(i, i + l).map(_.slice(j, j + l))

m.slice(i, i + l) = filtra todas las filas de la matriz entre los valores "i" y "i + l"

map(_.slice(j, j + l)) = después para cada fila anteriormente filtrada, se filtra los valores o las columnas de la matriz entre los valores "j" y "j + l"

Caso base

Matriz identidad 3x3

m =

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

subMatriz(m, 0, 1, 2)

m.slice(i, i + l)

```
m.slice(0, 2) // excluye el dato en posición 2
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \textcolor{red}{0} & \textcolor{red}{0} & \textcolor{red}{1} \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

```
map(_.slice(j, j + l))  
map(_.slice(1, 3)) //excluye para cada fila el dato que no esté en las posiciones 1 y 2
```

$$\begin{pmatrix} \textcolor{red}{1} & 0 & 0 \\ \textcolor{red}{0} & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

Esto devuelve un vector de vectores de enteros (matriz), de menor o igual tamaño que la original.

3.5 sumMatriz

$f: Matriz_1(n \times n) + Matriz_2(n \times n) = Matriz_3(n \times n)$

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

P_f :

```
sumMatriz(m1: Matriz, m2: Matriz) //m1 y m2 tiene las mismas dimensiones (n x n)  
Vector.tabulate(m1.length, m1.head.length) { (i, j) => m1(i)(j) + m2(i)(j)}
```

.tabulate crea un nuevo vector de dimensiones 2 en este caso (matriz), a través de una función como el .map para listas.

`Vector.tabulate(dim1, dim2)(func)`

$(i, j) \Rightarrow m1(i)(j) + m2(i)(j)$ // Para cada valor en posición (i, j) le suma los valores en las mismas posiciones de las matrices $m1$ y $m2$

Caso base

Matrices (2x2)

$$m1 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \quad m2 = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Vector.tabulate(m1.length, m1.head.length) { (i, j) => m1(i)(j) + m2(i)(j)}

$$\begin{pmatrix} 1+0 & 0+0 \\ 1+1 & 1+0 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}$$

3.6 multMatrizRec

$f: Matriz_1(n \times n) \times Matriz_2(n \times n) = Matriz_3(n \times n)$

```
def multMatrizRec(m1: Matriz, m2: Matriz): Matriz = {
    val n = m1.length

    if (n == 1) {
        Vector(Vector(m1(0)(0) * m2(0)(0)))
    } else {
        val mitad = n / 2

        val a11 = subMatriz(m1, 0, 0, mitad)
        val a12 = subMatriz(m1, 0, mitad, mitad)
        val a21 = subMatriz(m1, mitad, 0, mitad)
        val a22 = subMatriz(m1, mitad, mitad, mitad)

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

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

        (c11 ++ c21).zip(c12 ++ c22).map { case (filaIzq, filaDer) => filaIzq ++ filaDer }
    }
}
```

$P_f:$

$if(n == 1) // n = m_1.length, largo de la matriz 1$

$Vector(Vector(m1(0)(0) * m2(0)(0))) // Crea una nueva matriz con la multiplicación de los únicos valores de las 2 matrices$

else:

val a11 = subMatriz(m1, 0, 0, mitad) //Crea una submatriz con la mitad de la matriz 1, en el caso de ser una matriz (2x2) se devuelven los 4 valores de la matriz como los valores de "a". Lo mismo para la matriz 2.

val c11 = sumMatriz(multMatrizRec(a11, b11), multMatrizRec(a12, b21)) //Se hace un llamado recursivo para devolver valores mínimos de las matrices, luego se devuelve la multiplicación de los valores y se suman en un nuevo valor

(c11 ++ c21).zip(c12 ++ c22).map { case (filaIzq, filaDer) => filaIzq ++ filaDer }

(c11 ++ c21) //concatena las filas izquierdas en un vector de vectores de lado izquierdo .zip(c12 ++ c22) //combina cada par del vector de filas izquierda con el vector de filas derechas

map { case (filaIzq, filaDer) => filaIzq ++ filaDer } //concatena las anteriormente combinadas filas derecha e izquierda formando la matriz final

Caso base de prueba

matrices 2x2

$$m1: \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad m2: \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

$n = 2$

$$a_{11} = 1, a_{12} = 0, a_{21} = 1, a_{22} = 1$$

$$b_{11} = 1, b_{12} = 1, b_{21} = 0, b_{22} = 1$$

$$c_{11} = \text{sumMatriz}(\text{multMatrizRec}(a11, b11), \text{multMatrizRec}(a12, b21))$$

$$c_{11} = ((a11 * b11) + (a12 * b21)) = ((1 * 1) + (0 * 0)) = 1$$

$$c_{12} = ((a11 * b12) + (a12 * b22)) = ((1 * 1) + (0 * 1)) = 1$$

$$c_{21} = ((a21 * b11) + (a22 * b21)) = ((1 * 1) + (1 * 0)) = 1$$

$$c_{22} = ((a21 * b12) + (a22 * b22)) = ((1 * 1) + (1 * 1)) = 2$$

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

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

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

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

$$m_R = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$$

NOTA: Tenga en cuenta que durante el proceso de submatrices y suma de matrices se invierten las filas y columnas de la 2 matriz con cada llamado de “b”

3.7 restaMatriz

$$f: Matriz_1(n \times n) + Matriz_2(n \times n) = Matriz_3(n \times n)$$

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

P_f :

sumMatriz(m1: Matriz, m2: Matriz) //m1 y m2 tiene las mismas dimensiones (n x n)
Vector.tabulate(m1.length, m1.head.length) { (i, j) => m1(i)(j) - m2(i)(j)}

//Para cada posición (i, j) de las 2 matrices, se restan y se ubica en la nueva matriz

Caso base

Matrices (2x2)

$$m1: \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \quad m2: \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

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

$$\begin{pmatrix} 1 - 0 & 0 - 0 \\ 1 - 1 & 0 - 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

3.8 multStrassen

$$f: Matriz_1(n \times n) \times Matriz_2(n \times n) = Matriz_3(n \times n)$$

```

def multStrassen(m1: Matriz, m2: Matriz): Matriz = {
  val n = m1.length
  if (n == 1) {
    Vector(Vector(m1(0)(0) * m2(0)(0)))
  } else {
    val mitad = n / 2

    val a11 = subMatriz(m1, 0, 0, mitad)
    val a12 = subMatriz(m1, 0, mitad, mitad)
    val a21 = subMatriz(m1, mitad, 0, mitad)
    val a22 = subMatriz(m1, mitad, mitad, mitad)

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

    val p1 = multStrassen(a11, restaMatriz(b12, b22))
    val p2 = multStrassen(sumMatriz(a11, a12), b22)
    val p3 = multStrassen(sumMatriz(a21, a22), b11)
    val p4 = multStrassen(a22, restaMatriz(b21, b11))
    val p5 = multStrassen(sumMatriz(a11, a22), sumMatriz(b11, b22))
    val p6 = multStrassen(restaMatriz(a12, a22), sumMatriz(b21, b22))
    val p7 = multStrassen(restaMatriz(a11, a21), sumMatriz(b11, b12))

    val c11 = sumMatriz(restaMatriz(sumMatriz(p5, p4), p2), p6)
    val c12 = sumMatriz(p1, p2)
    val c21 = sumMatriz(p3, p4)
    val c22 = restaMatriz(restaMatriz(sumMatriz(p5, p1), p3), p7)

    (c11 ++ c21).zip(c12 ++ c22).map { case (filaIzq, filaDer) => filaIzq ++ filaDer }
  }
}

```

P_f :

if(n == 1) // n = m₁.length, largo de la matriz 1

*Vector(Vector(m1(0)(0) * m2(0)(0))) //Crea un nueva nueva matriz con la multiplicación de los únicos valores de las 2 matrices*

else:

val a11 = subMatriz(m1, 0, 0, mitad) //Crea una submatriz con la mitad de la matriz 1, en el caso de ser una matriz (2x2) se devuelve los 4 valores de la matriz como los valores de "a" Lo mismo para la matriz 2.

$$\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}$$

val p1 = multStrassen(a11, restaMatriz(b12, b22)) //Realiza un llamado recursivo, para multiplicar los valores de las matrices o aplicar el mismo método a las submatrices

restaMatriz(b12, b22) //Nótese que por fórmula esto sería $S_1 = b12 - b22$

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 \\
P_2 &= S_2 \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 \\
P_5 &= S_5 \cdot S_6 \\
P_6 &= S_7 \cdot S_8 \\
P_7 &= S_9 \cdot S_{10}
\end{aligned}$$

//Se aplica esta operación a los 7 P

```

val c11 = sumMatriz(restaMatriz(sumMatriz(p5, p4), p2), p6)
val c12 = sumMatriz(p1, p2)
val c21 = sumMatriz(p3, p4)
val c22 = restaMatriz(restaMatriz(sumMatriz(p5, p1), p3), p7)

```

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

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

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

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

```
(c11 ++ c21).zip(c12 ++ c22).map { case (filaIzq, filaDer) => filaIzq ++ filaDer }
```

// De la misma forma que en el método multMatrizRec concatena las filas y columnas de los vectores para formar una matriz.

Caso base de prueba

$$\text{m1: } \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad \text{m2: } \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

$$\begin{aligned} n &= 2 \\ a_{11} &= 1, a_{12} = 0, a_{21} = 1, a_{22} = 1 \\ b_{11} &= 1, b_{12} = 1, b_{21} = 0, b_{22} = 1 \end{aligned}$$

//Para este caso, como la matriz es la más pequeña posible, 2^n , entonces no se realiza llamado recursivo.

//Fórmula

$$\begin{aligned} \text{val } p1 &= \text{multStrassen}(a11, \text{restaMatriz}(b12, b22)) = a_{11} * (b_{12} - b_{22}) \\ \text{val } p2 &= \text{multStrassen}(\text{sumMatriz}(a11, a12), b22) = (a_{11} + a_{12}) * b_{22} \\ \text{val } p3 &= \text{multStrassen}(\text{sumMatriz}(a21, a22), b11) = (a_{21} + a_{22}) * b_{11} \\ \text{val } p4 &= \text{multStrassen}(a22, \text{restaMatriz}(b21, b11)) = a_{22} * (b_{21} - b_{11}) \\ \text{val } p5 &= \text{multStrassen}(\text{sumMatriz}(a11, a22), \text{sumMatriz}(b11, b22)) = \\ &(a_{11} + a_{22}) * (b_{11} + b_{22}) \\ \text{val } p6 &= \text{multStrassen}(\text{restaMatriz}(a12, a22), \text{sumMatriz}(b21, b22)) = \\ &(a_{12} - a_{22}) * (b_{21} + b_{22}) \\ \text{val } p7 &= \text{multStrassen}(\text{restaMatriz}(a11, a21), \text{sumMatriz}(b11, b12)) = \\ &(a_{11} - a_{21}) * (b_{11} + b_{12}) \end{aligned}$$

//Valores reemplazados

$$\begin{aligned} \text{val } p1 &= \text{multStrassen}(1, \text{restaMatriz}(1, 1)) = (1 * (1 - 1)) = 1 * 0 = 0 \\ \text{val } p2 &= \text{multStrassen}(\text{sumMatriz}(1, 0), 1) = (1 + 0) * 1 = 1 * 1 = 1 \\ \text{val } p3 &= \text{multStrassen}(\text{sumMatriz}(1, 1), 1) = (1 + 1) * 1 = 2 * 1 = 2 \\ \text{val } p4 &= \text{multStrassen}(1, \text{restaMatriz}(0, 1)) = 1 * (0 - 1) = 1 * (-1) = -1 \\ \text{val } p5 &= \text{multStrassen}(\text{sumMatriz}(1, 1), \text{sumMatriz}(1, 1)) = (1 + 1) * (1 + 1) = 4 \\ \text{val } p6 &= \text{multStrassen}(\text{restaMatriz}(0, 1), \text{sumMatriz}(0, 1)) = (0 - 1) * (0 + 1) = -1 \end{aligned}$$

```
val p7 = multStrassen(restaMatriz(1, 1), sumMatriz(1, 1) = (1 - 1) * (1 + 1) = 0
```

//Fórmula

```
val c11 = sumMatriz(restaMatriz(sumMatriz(p5, p4), p2), p6)
val c12 = sumMatriz(p1, p2)
val c21 = sumMatriz(p3, p4)
val c22 = restaMatriz(restaMatriz(sumMatriz(p5, p1), p3), p7)
```

//Valores reemplazados

```
val c11 = sumMatriz(restaMatriz(sumMatriz(4, -1), 1), -1)
((4 + (-1)) - 1) + (-1) = 3 - 1 - 1 = 1
val c12 = sumMatriz(0, 1) = (0 + 1) = 1
val c21 = sumMatriz(2, -1) = 2 + (-1) = 1
val c22 = restaMatriz(restaMatriz(sumMatriz(4, 0), 2), 0)
((4 + 0) - 2) - 0 = 2
```

$$\text{mR: } \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$$

4. Conclusiones

4.1 ¿Cuál de las implementaciones es más rápida?

4.1.1 Benchmarking vectores

```
def benchmarkingVectores():Unit = {

    val size = 100 // Tamaño del vector

    println(s"Benchmarking para vectores de tamaño $size:")

    for (i <- 1 to 10) {
        // Generar vectores aleatorios
        val v1 = App.vectorAlAzar(size, 100)
        val v2 = App.vectorAlAzar(size, 100)
        val parV1: ParVector[Int] = v1.par
        val parV2: ParVector[Int] = v2.par

        val seqTime = withWarmer(new Warmer.Default) measure {
            prodPunto(v1, v2)
        }
    }
}
```

```

val parTime = withWarmer(new Warmer.Default) measure {
    prodPuntoParD(parV1, parV2)
}

// Imprimir resultados
println(s"Iteración $i:")
println(s" Secuencial: $seqTime")
println(s" Paralelo:   $parTime")
}
}

```

Se define el tamaño del vector como 100, haremos 7 iteraciones para tener un gran número de casos pero teniendo en cuenta la memoria de computo y poder sacar mejores conclusiones, se definen los vectores v1 y v2 con la función vectorAlAzar() para generar vectores aleatorios, luego se definen estos vectores paralelizados con “.par”

Los valores seqTime y ParTime serán los encargados de medir el tiempo de ejecución con un Warmer, posterior a esto, se imprimen los resultados de cada iteración con el tiempo secuencial y paralelo.

Resultados

Tamaño Vector	Tiempo Secuencial	Tiempo Paralelizado
2^1	0,0817 ms	1,8173 ms
2^2	0,0329 ms	2,1542 ms
2^3	0,0317 ms	2,191 ms
2^4	0,0316 ms	2,1546 ms
2^5	0,0402 ms	1,7959 ms
2^6	0,0895 ms	1.8088 ms
2^7	0,0609 ms	2.0536 ms

Conclusiones

En este caso concreto, no es viable implementar versiones de los algoritmos de multiplicación de matrices usando ParVector y prodPuntoParD, pues la paralelización no ofrece una mayor eficiencia en comparación con la composición

secuencial, especialmente en vectores de menor tamaño. El rendimiento no aumenta y el costo de paralelizar empeora el rendimiento.

4.1.2 Benchmark para Matrices función Rec

```
/Benchmarking para la multiplicación de matrices
def benchmarkingMultMatrizRec():Unit = {

    println(s"Benchmarking para matrices :")

    for (i <- 1 to 7) {
        // Generar matrices aleatorias con límite de 2^i
        // Se generan matrices de 2^i x 2

        val m1 = App.matrizAlAzar(math.pow(2, i).toInt, 2)
        val m2 = App.matrizAlAzar(math.pow(2, i).toInt, 2)
        val parM1: App.Matriz = m1.par.map(_.toVector).toVector
        val parM2: App.Matriz = m2.par.map(_.toVector).toVector

        val seqTime = withWarmer(new Warmer.Default) measure {
            App.multMatrizRec(m1, m2)
        }

        val parTime = withWarmer(new Warmer.Default) measure {
            App.multMatrizRecPar(parM1, parM2)
        }

        // Imprimir resultados
        println(f"Iteración $i:")
        println(f" Secuencial: $seqTime")
        println(f" Paralelo: $parTime")
    }
}
```

Se realiza un ciclo con un iterador de 1 hasta 7, este iterador posteriormente nos indicará el exponente el cuál se elevará para obtener el tamaño de la matriz, se utilizó el límite de 7 debido al tiempo de ejecución de los benchmarks, un número más alto significa matrices más grandes, por lo que se tomará en consecuencia más tiempo, por motivos de facilitar la vista y los tiempos tomados, se limita hasta 7. Se definen entonces las dos matrices a multiplicar y se “calienta” con el warmer las funciones multMatrizRec y multMatrizRecPar, las matrices iniciales se evalúan en estas funciones y se imprime el tiempo en pantalla.

Resultados

Tamaño Matriz	Tiempo Secuencial	Tiempo Paralelizado
2^1	0,2056	0,3867
2^2	0,2482	0,5373
2^3	1,9144	1,9628
2^4	18,896	5,7424
2^5	28,1116	7,8781
2^6	161,9023	54,8575
2^7	1359,0686	450,2858

Conclusiones

En este caso de Benchmark se puede evidenciar mucho mejor la ventaja que consigue la paralelización en un volumen de procesos y datos más amplio, incluso tomando en cuenta la gestión de los hilos, en los valores más pequeños el tiempo secuencial era menor, hasta un punto donde suele tener un tiempo parecido al tiempo paralelizado, a partir de aquí la paraleliza tendrá un tiempo mucho menor en comparación a la evaluación secuencial.

4.1.3 Benchmark para matrices función Strassen

```
//Benchmarking para la multiplicación de matrices con
//metodo Strassen
def benchmarkingMultMatrizStrassen():Unit = {

    println(s"Benchmarking para matrices con metodo
Strassen:")

    for (i <- 1 to 6) { // hasta 6 por temas de memoria :c
        // Generar matrices aleatorias con limite de  $2^i$ 
        // Se generan matrices de  $2^i \times 2^i$ 

        val m1 = App.matrizAlAzar(math.pow(2, i).toInt, 2)
        val m2 = App.matrizAlAzar(math.pow(2, i).toInt, 2)
        val parM1: App.Matriz = m1.par.map(_.toVector).toVector
        val parM2: App.Matriz = m2.par.map(_.toVector).toVector
    }
}
```

```

val seqTime = withWarmer(new Warmer.Default) measure {
    App.multStrassen(m1, m2)
}

val parTime = withWarmer(new Warmer.Default) measure {
    App.multStrassenPar(parM1, parM2)
}

// Imprimir resultados
println(f"Iteración $i:")
println(f" Secuencial: $seqTime")
println(f" Paralelo: $parTime")
println("Resultado Matriz:")
App.printMatriz(App.multMatrizRec(m1, m2))
}
}

```

Similar a la función anterior, se realiza un ciclo con un iterador de 1 hasta 7, este iterador posteriormente nos indicará el exponente el cuál se elevará para obtener el tamaño de la matriz, se utilizó el límite de 7 debido al tiempo de ejecución de los benchmarks, un número más alto significa matrices más grandes, por lo que se tomará en consecuencia más tiempo, por motivos de facilitar la vista y los tiempos tomados, se limita hasta 7. Se definen entonces las dos matrices a multiplicar y se “calienta” con el warmer las funciones multStrassen y multStrassenPar, las matrices iniciales se evalúan en estas funciones y se imprime el tiempo en pantalla.

Resultados

Tamaño Matriz Strass	Tiempo Secuencial	Tiempo Paralelizado
2^1	0,0385	0,3059
2^2	0,7605	1,6072
2^3	0,7058	9,6464 ms
2^4	3,9142	45,5669 ms
2^5	19,9746 ms	332,5778 ms
2^6	150,2219 ms	2266,1932 ms

Conclusiones

Este es otro caso donde se evidencia que la paralelización no siempre será mejor que una solución secuencial, la gestión de los hilos en procesos y datos relativamente pequeños, hace que la paralelización pierda poder de minimizar tiempo, el método de Strassen de por sí es un método que simplifica la multiplicación de matrices, por lo que la paralelización no es tan necesaria (incluso es contraproducente) en este caso.

4.2 ¿De qué depende que la aceleración sea mejor?

La aceleración depende del tamaño de las matrices, la cantidad de núcleos disponibles, la sobrecarga asociada a la gestión de hilos y sincronización, y la estructura de los datos. La aceleración será mejor en cuanto el coste de paralelización no supere la ganancia, ósea cuando el número de hilos no genere un desgaste al tener que manejarlo, o cuando el paralelizar las matrices no cueste más que calcularlas de manera secuencial

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

a. Multiplicación de matrices convencionales.

Secuencial:

- Cuando las matrices son pequeñas
- Sistemas con pocos núcleos o procesadores.
- Cuando la sobrecarga de paralelización supera los beneficios.

Paralelo:

- Para matrices grandes, donde hay suficiente trabajo para dividir entre los núcleos.
- En arquitecturas con alto número de núcleos y buen acceso a memoria.
- Cuando el tiempo de ejecución es crítico y se puede tolerar un mayor uso de recursos.

b. Método de Strassen

El método de Strassen es útil porque reduce el número de multiplicaciones necesarias a gastos de realizar más sumas y restas. Sin embargo, su eficiencia varía:

Secuencial:

-Para matrices pequeñas, donde la sobrecarga de las sumas y restas adicionales no compensa la reducción de multiplicaciones.

Paralelo:

-para matrices muy grandes, donde la reducción de multiplicaciones tiene un impacto significativo.