



Informe Taller 3: Programación Concurrente

Jeidy Nicol Murillo Murillo - 235910
Verónica Lorena Mujica Gavidia - 2359406
Karol Tatiana Burbano Nasner - 2359305
Sebastian Castro Rengifo - 2359435

14 de diciembre de 2024

1. Informe de Procesos

En esta parte del informe se presentan los procesos generados por los programas recursivos, se generan algunos ejemplos para cada ejercicio, y se muestra como se comporta el proceso generado por cada uno de los programas.

1.1. Funciones para Multiplicación *Estándar* de Matrices

1.1.1. Función para multiplicación de matrices estándar *secuencial*

La función `multMatriz` implementa la multiplicación de dos matrices cuadradas A y B de dimensión $n \times n$. Primero, calcula la transpuesta de B (B^T) para facilitar el acceso a sus columnas como filas. Luego, genera la matriz resultante C mediante la función `Vector.tabulate`, iterando sobre cada par de índices (i, j) . Para cada posición (i, j) , utiliza la función `prodPunto` para calcular el producto punto entre la fila i de A y la fila j de B^T , correspondiente a la columna j de B . Finalmente, devuelve C como el resultado de la multiplicación de A y B .

Ejemplo para `multMatriz(matriz1,matriz2)`

Se comienzan definiendo 2 matrices para poder ejecutar el primer test

```
1 val matriz1 = Vector(  
2   Vector(2, 4, 6, 8),  
3   Vector(10, 12, 14, 16),  
4   Vector(18, 20, 22, 24),  
5   Vector(26, 28, 30, 32)  
6 )  
7  
8 val matriz2 = Vector(  
9   Vector(1, 3, 5, 7),  
10  Vector(9, 11, 13, 15),  
11  Vector(17, 19, 21, 23),  
12  Vector(25, 27, 29, 31)  
13 )
```

Y se define el valor esperado de la multiplicación de las 2 matrices

```
1 val resultadoEsperado = Vector(  
2   Vector(340, 380, 420, 460),  
3   Vector(756, 860, 964, 1068),  
4   Vector(1172, 1340, 1508, 1676),
```

```

5   Vector(1588,1820,2052,2284)
6   )

```

```

Local
  m1 = Vector1@1992 "Vector(Vector(2, 4, 6, 8), Vector(10, 12, 14, 16), Vector(18, 20, 22, 24), Vector(26, 28, 30, 32))"
  prefix1 = Object[4]@2001
    0 = Vector1@2003 "Vector(2, 4, 6, 8)"
    1 = Vector1@2004 "Vector(10, 12, 14, 16)"
    2 = Vector1@2005 "Vector(18, 20, 22, 24)"
    3 = Vector1@2006 "Vector(26, 28, 30, 32)"
  m2 = Vector1@1993 "Vector(Vector(1, 3, 5, 7), Vector(9, 11, 13, 15), Vector(17, 19, 21, 23), Vector(25, 27, 29, 31))"
  prefix1 = Object[4]@2002
    0 = Vector1@2011 "Vector(1, 3, 5, 7)"
    1 = Vector1@2012 "Vector(9, 11, 13, 15)"
    2 = Vector1@2013 "Vector(17, 19, 21, 23)"
    3 = Vector1@2014 "Vector(25, 27, 29, 31)"
  this = Taller3@1994

```

Figura 1: Debug MultMatriz

```

Local
  m2T = Vector1@1993 "Vector(Vector(1, 3, 5, 7), Vector(9, 11, 13, 15), Vector(17, 19, 21, 23), Vector(25, 27, 29, 31))"
  prefix1 = Object[4]@2002
    0 = Vector1@2003 "Vector(1, 3, 5, 7)"
    1 = Vector1@2004 "Vector(9, 11, 13, 15)"
    2 = Vector1@2005 "Vector(17, 19, 21, 23)"
    3 = Vector1@2006 "Vector(25, 27, 29, 31)"
  this = Taller3@1994

```

Figura 2: Debug MultMatriz

Lo primero que hace la función `multMatriz` después de recibir como parámetros las 2 matrices anteriores es crear una variable `m2T` y llama a la función `transpuesta` para darle como parámetro la `matriz2` y así preparar esta matriz para la multiplicación.

```

Local
  m2T = Vector1@1993 "Vector(Vector(1, 3, 5, 7), Vector(9, 11, 13, 15), Vector(17, 19, 21, 23), Vector(25, 27, 29, 31))"
  prefix1 = Object[4]@2002
    0 = Vector1@2003 "Vector(1, 3, 5, 7)"
    1 = Vector1@2004 "Vector(9, 11, 13, 15)"
    2 = Vector1@2005 "Vector(17, 19, 21, 23)"
    3 = Vector1@2006 "Vector(25, 27, 29, 31)"
    l = 4
  this = Taller3@1994

```

Figura 3: Debug MultMatriz

```
Local
  m2T$1 = Vector1@1993 "Vector(Vector(1, 3, 5, 7), Vector(9, 11, 13, 15), Vector(17, 19, 21, 23), Vector(25, 27, 29, 31))"
  prefix1 = Object[4]@2002
    > 0 = Vector1@2003 "Vector(1, 3, 5, 7)"
    > 1 = Vector1@2004 "Vector(9, 11, 13, 15)"
    > 2 = Vector1@2005 "Vector(17, 19, 21, 23)"
    > 3 = Vector1@2006 "Vector(25, 27, 29, 31)"
    i = 0
    j = 0
```

Figura 4: Debug MultMatriz

La función `transpuesta` toma como parámetro una matriz (`m2T`) y devuelve la matriz transpuesta de la misma. La función procede de la siguiente manera:

Obtiene la longitud (`l`) de la matriz (`m2T`). Crea una nueva matriz utilizando `Vector.tabulate` con dimensiones $((l, l))$. En cada posición $((i, j))$ de la nueva matriz, asigna el valor de la posición $((j, i))$ de la matriz (`m2T`). De esta manera, la función `transpuesta` devuelve la matriz transpuesta de la matriz (`m2T`) original.

```
Local
  m2T$1 = Vector1@1993 "Vector(Vector(1, 3, 5, 7), Vector(9, 11, 13, 15), Vector(17, 19, 21, 23), Vector(25, 27, 29, 31))"
  prefix1 = Object[4]@2002
    > 0 = Vector1@2003 "Vector(1, 3, 5, 7)"
    > 1 = Vector1@2004 "Vector(9, 11, 13, 15)"
    > 2 = Vector1@2005 "Vector(17, 19, 21, 23)"
    > 3 = Vector1@2006 "Vector(25, 27, 29, 31)"
    i = 3
    j = 3
```

Figura 5: Debug MultMatriz

```

Local
m1 = Vector1@1992 "Vector(Vector(2, 4, 6, 8), Vector(10, 12, 14, 16), Vector(18, 20, 22, 24), Vector(26,
  prefix1 = Object[4]@2086
    > 0 = Vector1@2087 "Vector(2, 4, 6, 8)"
    > 1 = Vector1@2088 "Vector(10, 12, 14, 16)"
    > 2 = Vector1@2089 "Vector(18, 20, 22, 24)"
    > 3 = Vector1@2090 "Vector(26, 28, 30, 32)"
m2 = Vector1@1993 "Vector(Vector(1, 3, 5, 7), Vector(9, 11, 13, 15), Vector(17, 19, 21, 23), Vector(25,
  prefix1 = Object[4]@2002
    > 0 = Vector1@2003 "Vector(1, 3, 5, 7)"
    > 1 = Vector1@2004 "Vector(9, 11, 13, 15)"
    > 2 = Vector1@2005 "Vector(17, 19, 21, 23)"
    > 3 = Vector1@2006 "Vector(25, 27, 29, 31)"
m2T = Vector1@2073 "Vector(Vector(1, 9, 17, 25), Vector(3, 11, 19, 27), Vector(5, 13, 21, 29), Vector(7,
  prefix1 = Object[4]@2077
    > 0 = Vector1@2078 "Vector(1, 9, 17, 25)"
    > 1 = Vector1@2079 "Vector(3, 11, 19, 27)"
    > 2 = Vector1@2080 "Vector(5, 13, 21, 29)"
    > 3 = Vector1@2081 "Vector(7, 15, 23, 31)"
  > this = Taller3@1994

```

Figura 6: Debug MultMatriz

```

Local
m1 = Vector1@1992 "Vector(Vector(2, 4, 6, 8), Vector(10, 12, 14, 16), Vector(18, 20, 22, 24), Vector(26, 28, 30, 32))"
  prefix1 = Object[4]@2032
    > 0 = Vector1@2033 "Vector(2, 4, 6, 8)"
    > 1 = Vector1@2034 "Vector(10, 12, 14, 16)"
    > 2 = Vector1@2035 "Vector(18, 20, 22, 24)"
    > 3 = Vector1@2036 "Vector(26, 28, 30, 32)"
m2 = Vector1@1993 "Vector(Vector(1, 3, 5, 7), Vector(9, 11, 13, 15), Vector(17, 19, 21, 23), Vector(25, 27, 29, 31))"
  prefix1 = Object[4]@2019
    > 0 = Vector1@2020 "Vector(1, 3, 5, 7)"
    > 1 = Vector1@2021 "Vector(9, 11, 13, 15)"
    > 2 = Vector1@2022 "Vector(17, 19, 21, 23)"
    > 3 = Vector1@2023 "Vector(25, 27, 29, 31)"
m2T = Vector1@2028 "Vector(Vector(1, 9, 17, 25), Vector(3, 11, 19, 27), Vector(5, 13, 21, 29), Vector(7, 15, 23, 31))"
  prefix1 = Object[4]@2045
    > 0 = Vector1@2046 "Vector(1, 9, 17, 25)"
    > 1 = Vector1@2047 "Vector(3, 11, 19, 27)"
    > 2 = Vector1@2048 "Vector(5, 13, 21, 29)"
    > 3 = Vector1@2049 "Vector(7, 15, 23, 31)"
  n = 4
  > this = Taller3@1994

```

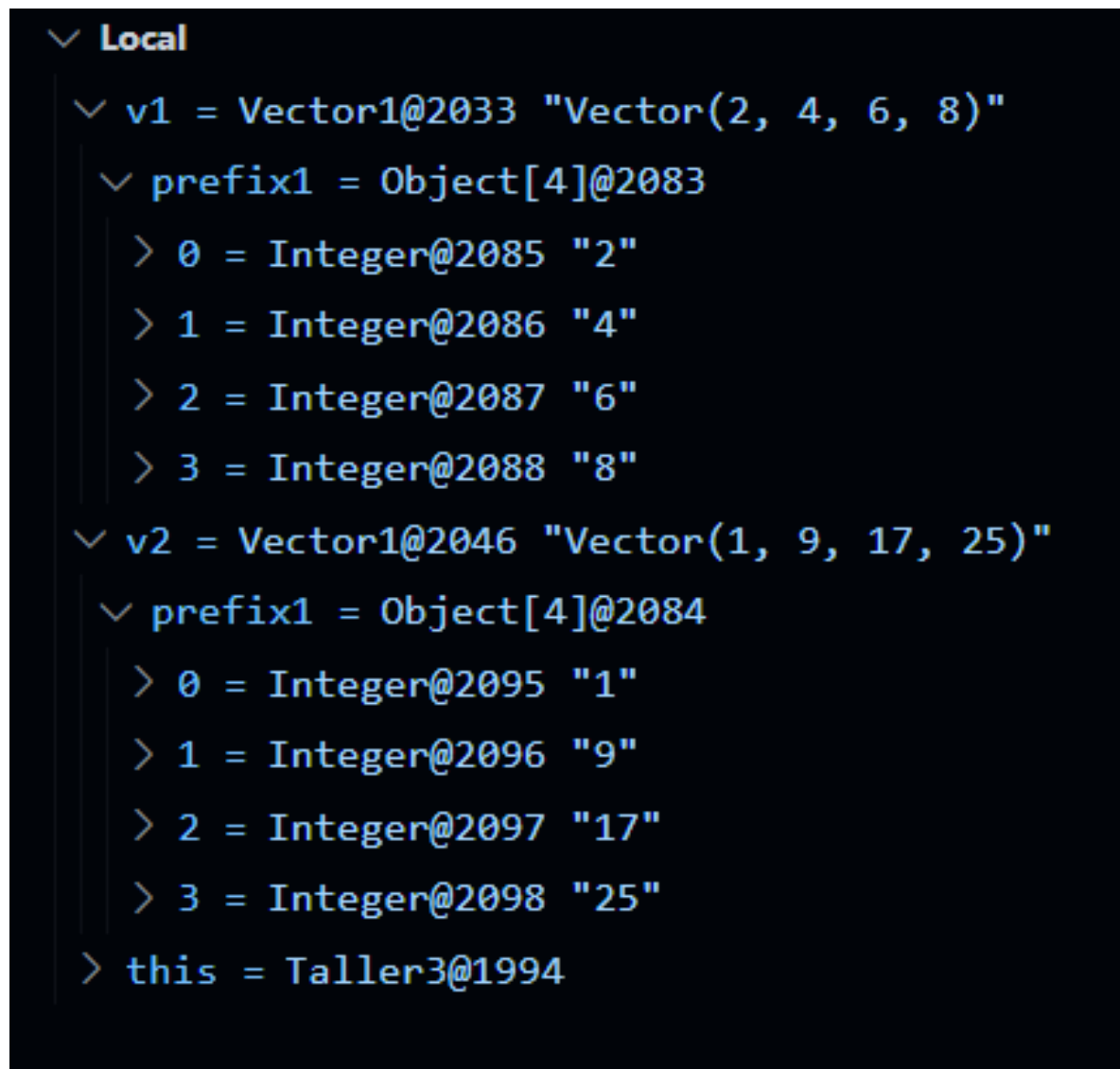
Figura 7: Debug MultMatriz

Luego, la función obtiene la longitud n de la matriz $\mathbf{m1}$. Después, la función crea una nueva matriz $\mathbf{m3}$ utilizando `Vector.tabulate`. Esta nueva matriz tiene dimensiones (n, n) , es decir, n filas y n columnas. Para cada posición (i, j) de la nueva matriz $\mathbf{m3}$, la función calcula el producto punto entre la fila i de $\mathbf{m1}$ y la columna j de $\mathbf{m2T}$ (la matriz transpuesta de $\mathbf{m2}$). Este resultado se asigna a la posición (i, j) de $\mathbf{m3}$.

La función `prodPunto` toma dos vectores de enteros `v1` y `v2` como entrada y devuelve un entero que representa el producto punto de los dos vectores. El producto punto se calcula de la siguiente manera:

1. Se utiliza la función `zip` para crear una lista de pares de elementos correspondientes de `v1` y `v2`.
2. Se aplica la función `map` a esta lista de pares, donde cada par (i, j) se multiplica y se devuelve el resultado.
3. Finalmente, se suman todos los resultados de la multiplicación utilizando la función `sum`.

En este ejemplo, se llama a la función `prodPunto` con dos vectores de enteros, `v1` y `v2`, que tienen la siguiente estructura:



```

✓ Local
  ✓ v1 = Vector1@2033 "Vector(2, 4, 6, 8)"
    ✓ prefix1 = Object[4]@2083
      > 0 = Integer@2085 "2"
      > 1 = Integer@2086 "4"
      > 2 = Integer@2087 "6"
      > 3 = Integer@2088 "8"
  ✓ v2 = Vector1@2046 "Vector(1, 9, 17, 25)"
    ✓ prefix1 = Object[4]@2084
      > 0 = Integer@2095 "1"
      > 1 = Integer@2096 "9"
      > 2 = Integer@2097 "17"
      > 3 = Integer@2098 "25"
  > this = Taller3@1994

```

Figura 8: Debug MultMatriz

- `v1: Vector(2, 4, 6, 8)`
- `v2: Vector(1, 9, 17, 25)`

El primer paso de la función `prodPunto` es utilizar `zip` para crear una lista de pares de elementos correspondientes de `v1` y `v2`. En este caso, la lista resultante es `List((2, 1), (4, 9), (6, 17), (8, 25))`.

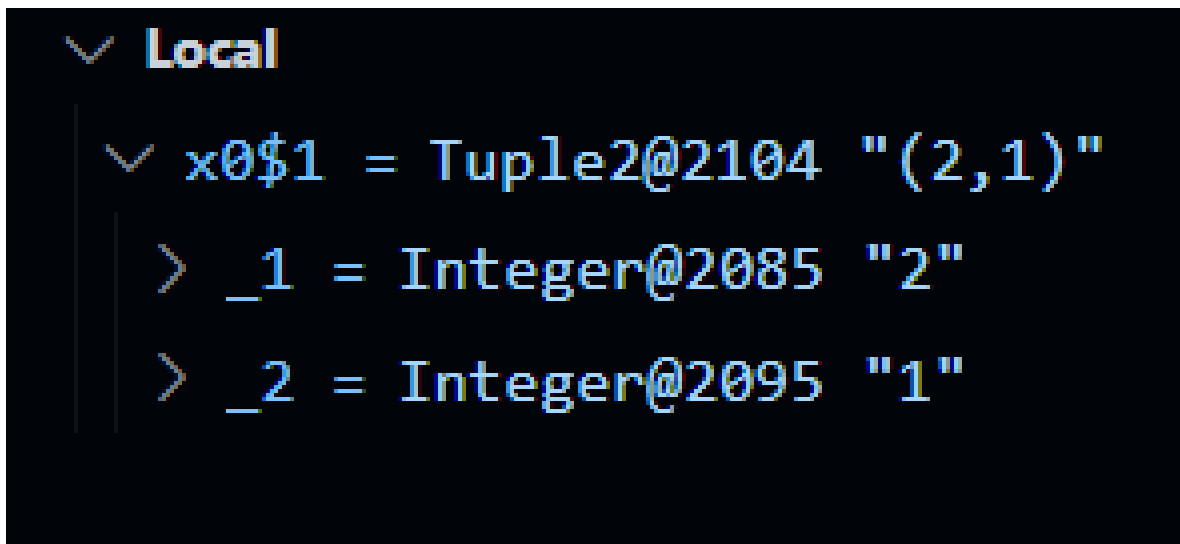


Figura 9: Debug MultMatriz

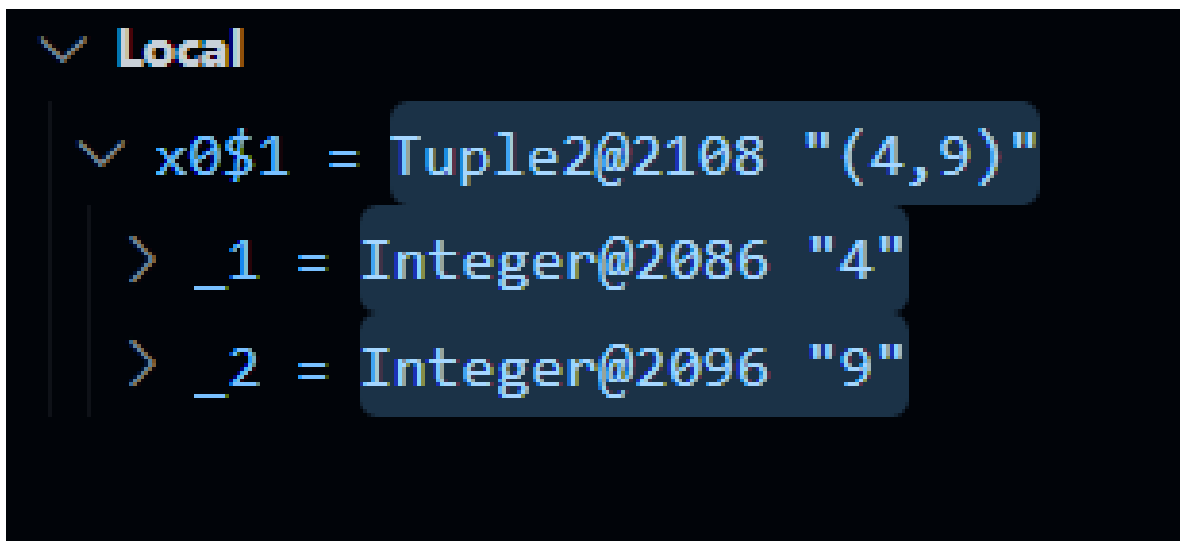
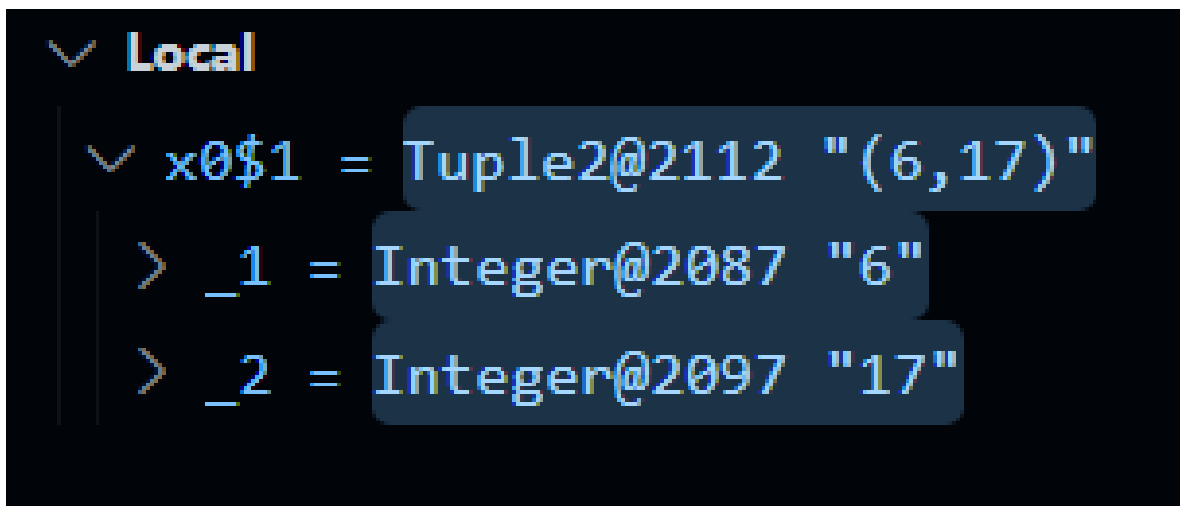
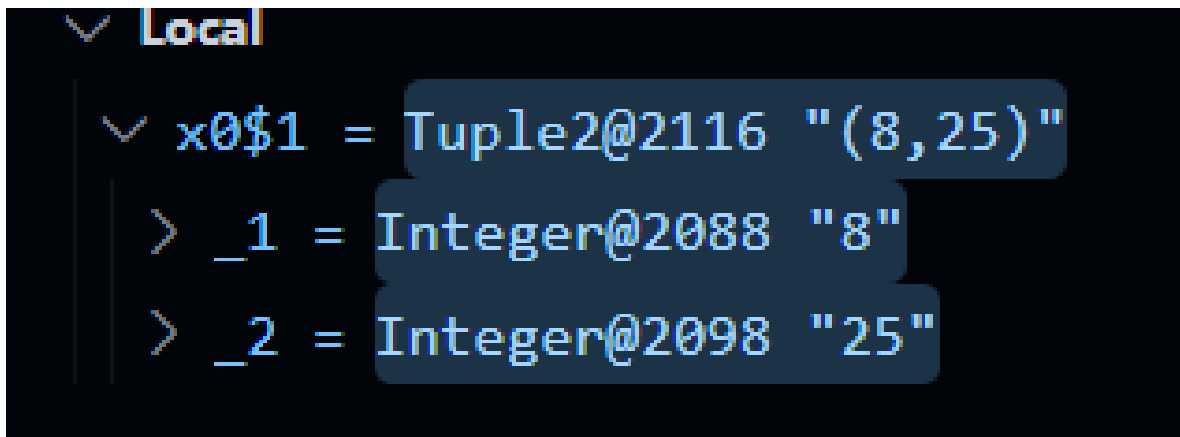


Figura 10: Debug MultMatriz



```
✓ Local
  ✓ x0$1 = Tuple2@2112 "(6,17)"
    > _1 = Integer@2087 "6"
    > _2 = Integer@2097 "17"
```

Figura 11: Debug MultMatriz



```
✓ Local
  ✓ x0$1 = Tuple2@2116 "(8,25)"
    > _1 = Integer@2088 "8"
    > _2 = Integer@2098 "25"
```

Figura 12: Debug MultMatriz

A continuación, la función `map` se aplica a esta lista de pares, donde cada par (i, j) se multiplica y se devuelve el resultado. Esto genera una nueva lista `List(2, 36, 102, 200)`.

Finalmente, la función `sum` se utiliza para sumar todos los elementos de esta lista, lo que da como resultado el producto punto final de 340.

```

> v1 = Vector1@2014 "Vector(2, 4, 6, 8)"
> v2 = Vector1@2015 "Vector(1, 9, 17, 25)"
    result = 340
> this = Taller3@1994

```

Figura 13: Debug MultMatriz

Luego de repetir el mismo proceso con los demás vectores, la función retorna una matriz $m3$ resultante de la multiplicación de las matrices anteriores.

```

Local
> m1 = Vector1@1992 "Vector(Vector(2, 4, 6, 8), Vector(10, 12, 14, 16), Vector(18, 20, 22, 24), Vector(26, 28, 30, 32))"
> m2 = Vector1@1993 "Vector(Vector(1, 3, 5, 7), Vector(9, 11, 13, 15), Vector(17, 19, 21, 23), Vector(25, 27, 29, 31))"
> m2T = Vector1@2028 "Vector(Vector(1, 9, 17, 25), Vector(3, 11, 19, 27), Vector(5, 13, 21, 29), Vector(7, 15, 23, 31))"
    n = 4
> m3 = Vector1@2223 "Vector(Vector(340, 380, 420, 460), Vector(756, 860, 964, 1068), Vector(1172, 1340, 1508, 1676), Vector(1588, 1820, 2052, 2284))"
  prefix1 = Object[4]@2228
    0 = Vector1@2229 "Vector(340, 380, 420, 460)"
    1 = Vector1@2230 "Vector(756, 860, 964, 1068)"
    2 = Vector1@2231 "Vector(1172, 1340, 1508, 1676)"
    3 = Vector1@2232 "Vector(1588, 1820, 2052, 2284)"
  > this = Taller3@1994

```

Figura 14: Debug MultMatriz

Para finalizar el test verifica que `multMatriz(matriz1, matriz2)` sea igual a `resultadoEsperado`, que hemos calculado como:

```

1  val resultadoEsperado = Vector(
2    Vector(340,380,420,460),
3    Vector(756,860,964,1068),
4    Vector(1172,1340,1508,1676),
5    Vector(1588,1820,2052,2284)
6  )

```

Como el cálculo cumple con esta igualdad, el `assert` pasa sin errores.

1.2. Funciones para Multiplicación *Paralela* de Matrices

1.2.1. Función para multiplicación de matrices estándar *paralela*

La función `multMatrizPar` implementa la multiplicación de dos matrices cuadradas A y B de dimensión $n \times n$ de forma paralela. A continuación, se detalla su funcionamiento:

1. *Transposición de la matriz B *: La función primero calcula la transpuesta de B , generando B^T . Esto facilita el acceso a las columnas de B como filas, lo cual es esencial para el cálculo del producto punto.
2. *Creación de la matriz resultante C *: Utiliza `Vector.tabulate` para crear una nueva matriz C con dimensiones $n \times n$. Cada posición (i, j) de C se calcula de manera paralela.
3. *Asignación de tareas paralelas*:

- Para cada elemento $C[i][j]$, se crea una tarea utilizando `task` que calcula el producto punto entre la fila i de A y la fila j de B^T .
 - Esta tarea se ejecuta de manera concurrente, lo que permite aprovechar los núcleos de procesamiento disponibles.
4. *Sincronización de tareas*: Una vez asignadas todas las tareas, se utiliza `join()` para garantizar que todas las operaciones paralelas finalicen antes de devolver el resultado.
 5. *Devolución del resultado*: Finalmente, se retorna la matriz C resultante de la multiplicación de A y B .

Ejemplo para `multMatrizPar(matriz1, matrizNegativa)`

Se comienzan definiendo 2 matrices para ejecutar el primer test:

```
1 val matriz1 = Vector.tabulate(16, 16)((i, j) => 2 * (i * 16 + j + 1))
1 val matrizNegativa = Vector.tabulate(16, 16)((i, j) => -2 * (i * 16 + j + 1))
```

Y se define el valor esperado de la multiplicación de las 2 matrices:

```
1 val resultado = Vector(
2   Vector(-87584, -88128, -88672, -89216, -89760, -90304, -90848, -91392, -91936,
3     -92480, -93024, -93568, -94112, -94656, -95200, -95744),
4   Vector(-211488, -213056, -214624, -216192, -217760, -219328, -220896, -222464,
5     -224032, -225600, -227168, -228736, -230304, -231872, -233440, -235008),
6   Vector(-335392, -337984, -340576, -343168, -345760, -348352, -350944, -353536,
7     -356128, -358720, -361312, -363904, -366496, -369088, -371680, -374272),
8   Vector(-459296, -462912, -466528, -470144, -473760, -477376, -480992, -484608,
9     -488224, -491840, -495456, -499072, -502688, -506304, -509920, -513536),
10  Vector(-583200, -587840, -592480, -597120, -601760, -606400, -611040, -615680,
11    -620320, -624960, -629600, -634240, -638880, -643520, -648160, -652800),
12  Vector(-707104, -712768, -718432, -724096, -729760, -735424, -741088, -746752,
13    -752416, -758080, -763744, -769408, -775072, -780736, -786400, -792064),
14  Vector(-831008, -837696, -844384, -851072, -857760, -864448, -871136, -877824,
15    -884512, -891200, -897888, -904576, -911264, -917952, -924640, -931328),
16  Vector(-954912, -962624, -970336, -978048, -985760, -993472, -1001184, -1008896,
17    -1016608, -1024320, -1032032, -1039744, -1047456, -1055168, -1062880, -1070592),
18  Vector(-1078816, -1087552, -1096288, -1105024, -1113760, -1122496, -1131232,
19    -1139968, -1148704, -1157440, -1166176, -1174912, -1183648, -1192384, -1201120,
20    -1209856),
21  Vector(-1202720, -1212480, -1222240, -1232000, -1241760, -1251520, -1261280,
22    -1271040, -1280800, -1290560, -1300320, -1310080, -1319840, -1329600, -1339360,
23    -1349120),
24  Vector(-1326624, -1337408, -1348192, -1358976, -1369760, -1380544, -1391328,
25    -1402112, -1412896, -1423680, -1434464, -1445248, -1456032, -1466816, -1477600,
26    -1488384),
27  Vector(-1450528, -1462336, -1474144, -1485952, -1497760, -1509568, -1521376,
28    -1533184, -1544992, -1556800, -1568608, -1580416, -1592224, -1604032, -1615840,
29    -1627648),
30  Vector(-1574432, -1587264, -1600096, -1612928, -1625760, -1638592, -1651424,
31    -1664256, -1677088, -1689920, -1702752, -1715584, -1728416, -1741248, -1754080,
32    -1766912),
33  Vector(-1698336, -1712192, -1726048, -1739904, -1753760, -1767616, -1781472,
34    -1795328, -1809184, -1823040, -1836896, -1850752, -1864608, -1878464, -1892320,
35    -1906176),
36  Vector(-1822240, -1837120, -1852000, -1866880, -1881760, -1896640, -1911520,
37    -1926400, -1941280, -1956160, -1971040, -1985920, -2000800, -2015680, -2030560,
38    -2045440),
39  Vector(-1946144, -1962048, -1977952, -1993856, -2009760, -2025664, -2041568,
40    -2057472, -2073376, -2089280, -2105184, -2121088, -2136992, -2152896, -2168800,
41    -2184704)
42 )
```

```

Local
> m1 = Vector1@2004 "Vector(Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32), Vector(34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56,...
> m2 = Vector1@2005 "Vector(Vector(-2, -4, -6, -8, -10, -12, -14, -16, -18, -20, -22, -24, -26, -28, -30, -32), Vector(-34, -36, -38, -40, -42, -44, -...
> this = Taller3@2006

```

Figura 15: Debug MultMatrizParalela

En esta imagen, se presenta la matriz inicial **m1**, que contiene los valores originales que serán utilizados como primera entrada para la operación de multiplicación de matrices. Cada fila de **m1** representará uno de los vectores necesarios para calcular los productos punto con las columnas de **m2**.

```

Local
> m1 = Vector1@2004 "Vector(Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32), Vector(34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56,...
> m2 = Vector1@2005 "Vector(Vector(-2, -4, -6, -8, -10, -12, -14, -16, -18, -20, -22, -24, -26, -28, -30, -32), Vector(-34, -36, -38, -40, -42, -44, -...
n = 16
> this = Taller3@2006

```

Figura 16: Debug MultMatrizParalela

Se muestra **m2**, la segunda matriz involucrada en la multiplicación. Aún no ha sido transpuesta, por lo que sus columnas no están accesibles como filas. Este paso es crucial para permitir una computación más eficiente del producto punto.

```

Local
> m2T = Vector1@2005 "Vector(Vector(-2, -4, -6, -8, -10, -12, -14, -16, -18, -20, -22, -24, -26, -28, -30, -32), Vector(-34, -36, -38, -40, -42, -44, ...
> this = Taller3@2006

```

Figura 17: Debug MultMatrizParalela

Aquí se muestra el resultado del proceso de transposición aplicado a **m2**. Al convertir **m2** en **m2T**, se reorganizan sus elementos de forma que las columnas de **m2** ahora se presenten como filas en **m2T**. Esto optimiza el acceso a los datos durante los cálculos paralelos.

```

Local
> m2T = Vector1@2005 "Vector(Vector(-2, -4, -6, -8, -10, -12, -14, -16, -18, -20, -22, -24, -26, -28, -30, -32), Vector(-34, -36, -38, -40, -42, -44, ...
l = 16
> this = Taller3@2006

```

Figura 18: Debug MultMatrizParalela

```

Local
> m2T$1 = Vector1@2005 "Vector(Vector(-2, -4, -6, -8, -10, -12, -14, -16, -18, -20, -22, -24, -26, -28, -30, -32), Vector(-34, -36, -38, -40, -42, -44...
i = 0
j = 0

```

Figura 19: Debug MultMatrizParalela

```

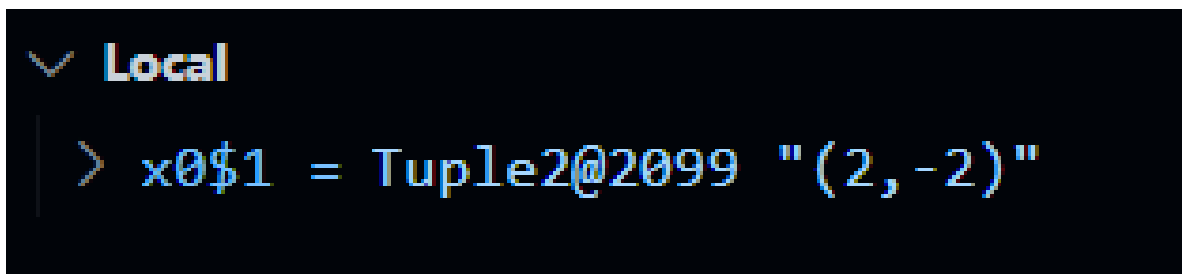
> m2T$1 = Vector1@2005 "Vector(Vector(-2, -4, -6, -8, -10, -12, -14, -16, -18, -20, -22, -24, -26, -28, -30, -32), Vector(-34, -36, -38, -40, -42, -44...
i = 15
j = 15

```

Figura 20: Debug MultMatrizParalela

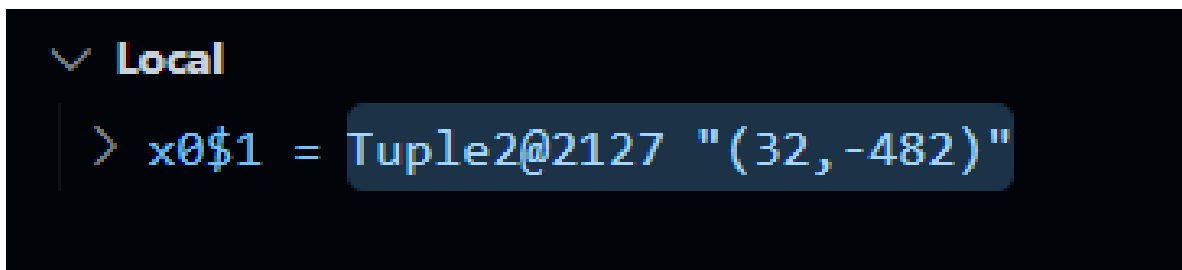
```

Local
> m2T = Vector1@2005 "Vector(Vector(-2, -4, -6, -8, -10, -12, -14, -16, -18, -20, -22, -24, -26, -28, -30, -32), Vector(-34, -36, -38, -40, -42, -44, -46, -48, -50, -52, -54, -56, -58, -60, -62, -64, -66, -68, -70, -72, -74, -76, -78, -80, -82, -84, -86, -88, -90, -92, -94, -96, -98, -100, -102, -104, -106, -108, -110, -112, -114, -116, -118, -120, -122, -124, -126, -128, -130, -132, -134, -136, -138, -140, -142, -144, -146, -148, -150, -152, -154, -156, -158, -160, -162, -164, -166, -168, -170, -172, -174, -176, -178, -180, -182, -184, -186, -188, -190, -192, -194, -196, -198, -200, -202, -204, -206, -208, -210, -212, -214, -216, -218, -220, -222, -224, -226, -228, -230, -232, -234, -236, -238, -240, -242, -244, -246, -248, -250, -252, -254, -256, -258, -260, -262, -264, -266, -268, -270, -272, -274, -276, -278, -280, -282, -284, -286, -288, -290, -292, -294, -296, -298, -300, -302, -304, -306, -308, -310, -312, -314, -316, -318, -320, -322, -324, -326, -328, -330, -332, -334, -336, -338, -340, -342, -344, -346, -348, -350, -352, -354, -356, -358, -360, -362, -364, -366, -368, -370, -372, -374, -376, -378, -380, -382, -384, -386, -388, -390, -392, -394, -396, -398, -400, -402, -404, -406, -408, -410, -412, -414, -416, -418, -420, -422, -424, -426, -428, -430, -432, -434, -436, -438, -440, -442, -444, -446, -448, -450, -452, -454, -456, -458, -460, -462, -464, -466, -468, -470, -472, -474, -476, -478, -480, -482, -484, -486, -488, -490, -492, -494, -496, -498, -500, -502, -504, -506, -508, -510, -512, -514, -516, -518, -520, -522, -524, -526, -528, -530, -532, -534, -536, -538, -540, -542, -544, -546, -548, -550, -552, -554, -556, -558, -560, -562, -564, -566, -568, -570, -572, -574, -576, -578, -580, -582, -584, -586, -588, -590, -592, -594, -596, -598, -600, -602, -604, -606, -608, -610, -612, -614, -616, -618, -620, -622, -624, -626, -628, -630, -632, -634, -636, -638, -640, -642, -644, -646, -648, -650, -652, -654, -656, -658, -660, -662, -664, -666, -668, -670, -672, -674, -676, -678, -680, -682, -684, -686, -688, -690, -692, -694, -696, -698, -700, -702, -704, -706, -708, -710, -712, -714, -716, -718, -720, -722, -724, -726, -728, -730, -732, -734, -736, -738, -740, -742, -744, -746, -748, -750, -752, -754, -756, -758, -760, -762, -764, -766, -768, -770, -772, -774, -776, -778, -780, -782, -784, -786, -788, -790, -792, -794, -796, -798, -800, -802, -804, -806, -808, -810, -812, -814, -816, -818, -820, -822, -824, -826, -828, -830, -832, -834, -836, -838, -840, -842, -844, -846, -848, -850, -852, -854, -856, -858, -860, -862, -864, -866, -868, -870, -872, -874, -876, -878, -880, -882, -884, -886, -888, -890, -892, -894, -896, -898, -900, -902, -904, -906, -908, -910, -912, -914, -916, -918, -920, -922, -924, -926, -928, -930, -932, -934, -936, -938, -940, -942, -944, -946, -948, -950, -952, -954, -956, -958, -960, -962, -964, -966, -968, -970, -972, -974, -976, -978, -980, -982, -984, -986, -988, -990, -992, -994, -996, -998, -1000, -1002, -1004, -1006, -1008, -1010, -1012, -1014, -1016, -1018, -1020, -1022, -1024, -1026, -1028, -1030, -1032, -1034, -1036, -1038, -1040, -1042, -1044, -1046, -1048, -1050, -1052, -1054, -1056, -1058, -1060, -1062, -1064, -1066, -1068, -1070, -1072, -1074, -1076, -1078, -1080, -1082, -1084, -1086, -1088, -1090, -1092, -1094, -1096, -1098, -1100, -1102, -1104, -1106, -1108, -1110, -1112, -1114, -1116, -1118, -1120, -1122, -1124, -1126, -1128, -1130, -1132, -1134, -1136, -1138, -1140, -1142, -1144, -1146, -1148, -1150, -1152, -1154, -1156, -1158, -1160, -1162, -1164, -1166, -1168, -1170, -1172, -1174, -1176, -1178, -1180, -1182, -1184, -1186, -1188, -1190, -1192, -1194, -1196, -1198, -1200, -1202, -1204, -1206, -1208, -1210, -1212, -1214, -1216, -1218, -1220, -1222, -1224, -1226, -1228, -1230, -1232, -1234, -1236, -1238, -1240, -1242, -1244, -1246, -1248, -1250, -1252, -1254, -1256, -1258, -1260, -1262, -1264, -1266, -1268, -1270, -1272, -1274, -1276, -1278, -1280, -1282, -1284, -1286, -1288, -1290, -1292, -1294, -1296, -1298, -1300, -1302, -1304, -1306, -1308, -1310, -1312, -1314, -1316, -1318, -1320, -1322, -1324, -1326, -1328, -1330, -1332, -1334, -1336, -1338, -1340, -1342, -1344, -1346, -1348, -1350, -1352, -1354, -1356, -1358, -1360, -1362, -1364, -1366, -1368, -1370, -1372, -1374, -1376, -1378, -1380, -1382, -1384, -1386, -1388, -1390, -1392, -1394, -1396, -1398, -1400, -1402, -1404, -1406, -1408, -1410, -1412, -1414, -1416, -1418, -1420, -1422, -1424, -1426, -1428, -1430, -1432, -1434, -1436, -1438, -1440, -1442, -1444, -1446, -1448, -1450, -1452, -1454, -1456, -1458, -1460, -1462, -1464, -1466, -1468, -1470, -1472, -1474, -1476, -1478, -1480, -1482, -1484, -1486, -1488, -1490, -1492, -1494, -1496, -1498, -1500, -1502, -1504, -1506, -1508, -1510, -1512, -1514, -1516, -1518, -1520, -1522, -1524, -1526, -1528, -1530, -1532, -1534, -1536, -1538, -1540, -1542, -1544, -1546, -1548, -1550, -1552, -1554, -1556, -1558, -1560, -1562, -1564, -1566, -1568, -1570, -1572, -1574, -1576, -1578, -1580, -1582, -1584, -1586, -1588, -1590, -1592, -1594, -1596, -1598, -1600, -1602, -1604, -1606, -1608, -1610, -1612, -1614, -1616, -1618, -1620, -1622, -1624, -1626, -1628, -1630, -1632, -1634, -1636, -1638, -1640, -1642, -1644, -1646, -1648, -1650, -1652, -1654, -1656, -1658, -1660, -1662, -1664, -1666, -1668, -1670, -1672, -1674, -1676, -1678, -1680, -1682, -1684, -1686, -1688, -1690, -1692, -1694, -1696, -1698, -1700, -1702, -1704, -1706, -1708, -1710, -1712, -1714, -1716, -1718, -1720, -1722, -1724, -1726, -1728, -1730, -1732, -1734, -1736, -1738, -1740, -1742, -1744, -1746, -1748, -1750, -1752, -1754, -1756, -1758, -1760, -1762, -1764, -1766, -1768, -1770, -1772, -1774, -1776, -1778, -1780, -1782, -1784, -1786, -1788, -1790, -1792, -1794, -1796, -1798, -1800, -1802, -1804, -1806, -1808, -1810, -1812, -1814, -1816, -1818, -1820, -1822, -1824, -1826, -1828, -1830, -1832, -1834, -1836, -1838, -1840, -1842, -1844, -1846, -1848, -1850, -1852, -1854, -1856, -1858, -1860, -1862, -1864, -1866, -1868, -1870, -1872, -1874, -1876, -1878, -1880, -1882, -1884, -1886, -1888, -1890, -1892, -1894, -1896, -1898, -1900, -1902, -1904, -1906, -1908, -1910, -1912, -1914, -1916, -1918, -1920, -1922, -1924, -1926, -1928, -1930, -1932, -1934, -1936, -1938, -1940, -1942, -1944, -1946, -1948, -1950, -1952, -1954, -1956, -1958, -1960, -1962, -1964, -1966, -1968, -1970, -1972, -1974, -1976, -1978, -1980, -1982, -1984, -1986, -1988, -1990, -1992, -1994, -1996, -1998, -2000, -2002, -2004, -2006, -2008, -2010, -2012, -2014, -2016, -2018, -2020, -2022, -2024, -2026, -2028, -2030, -2032, -2034, -2036, -2038, -2040, -2042, -2044, -2046, -2048, -2050, -2052, -2054, -2056, -2058, -2060, -2062, -2064, -2066, -2068, -2070, -2072, -2074, -2076, -2078, -2080, -2082, -2084, -2086, -2088, -2090, -2092, -2094, -2096, -2098, -2100, -2102, -2104, -2106, -2108, -2110, -2112, -2114, -2116, -2118, -2120, -2122, -2124, -2126, -2128, -2130, -2132, -2134, -2136, -2138, -2140, -2142, -2144, -2146, -2148, -2150, -2152, -2154, -2156, -2158, -2160, -2162, -2164, -2166, -2168, -2170, -2172, -2174, -2176, -2178, -2180, -2182, -2184, -2186, -2188, -2190, -2192, -2194, -2196, -2198, -2200, -2202, -2204, -2206, -2208, -2210, -2212, -2214, -2216, -2218, -2220, -2222, -2224, -2226, -2228, -2230, -2232, -2234, -2236, -2238, -2240, -2242, -2244, -2246, -2248, -2250, -2252, -2254, -2256, -2258, -2260, -2262, -2264, -2266, -2268, -2270, -2272, -2274, -2276, -2278, -2280, -2282, -2284, -2286, -2288, -2290, -2292, -2294, -2296, -2298, -2300, -2302, -2304, -2306, -2308, -2310, -2312, -2314, -2316, -2318, -2320, -2322, -2324, -2326, -2328, -2330, -2332, -2334, -2336, -2338, -2340, -2342, -2344, -2346, -2348, -2350, -2352, -2354, -2356, -2358, -2360, -2362, -2364, -2366, -2368, -2370, -2372, -2374, -2376, -2378, -2380, -2382, -2384, -2386, -2388, -2390, -2392, -2394, -2396, -2398, -2400, -2402, -2404, -2406, -2408, -2410, -2412, -2414, -2416, -2418, -2420, -2422, -2424, -2426, -2428, -2430, -2432, -2434, -2436, -2438, -2440, -2442, -2444, -2446, -2448, -2450, -2452, -2454, -2456, -2458, -2460, -2462, -2464, -2466, -2468, -2470, -2472, -2474, -2476, -2478, -2480, -2482, -2484, -2486, -2488, -2490, -2492, -2494, -2496, -2498, -2500, -2502, -2504, -2506, -2508, -2510, -2512, -2514, -2516, -2518, -2520, -2522, -2524, -2526, -2528, -2530, -2532, -2534, -2536, -2538, -2540, -2542, -2544, -2546, -2548, -2550, -2552, -2554, -2556, -2558, -2560, -2562, -2564, -2566, -2568, -2570, -2572, -2574, -2576, -2578, -2580, -2582, -2584, -2586, -2588, -2590, -2592, -2594, -2596, -2598, -2600, -2602, -2604, -2606, -2608, -2610, -2612, -2614, -2616, -2618, -2620, -2622, -2624, -2626, -2628, -2630, -2632, -2634, -2636, -2638, -2640, -2642, -2644, -2646, -2648, -2650, -2652, -2654, -2656, -2658, -2660, -2662, -2664, -2666, -2668, -2670, -2672, -2674, -2676, -2678, -2680, -2682, -2684, -2686, -2688, -2690, -2692, -2694, -2696, -2698, -2700, -2702, -2704, -2706, -2708, -2710, -2712, -2714, -2716, -2718, -2720, -2722, -2724, -2726, -2728, -2730, -2732, -2734, -2736, -2738, -2740, -2742, -2744, -2746, -2748, -2750, -2752, -2754, -2756, -2758, -2760, -2762, -2764, -2766, -2768, -2770, -2772, -2774, -2776, -2778, -2780, -2782, -2784, -2786, -2788, -2790, -2792, -2794, -2796, -2798, -2800, -2802, -2804, -2806, -2808, -2810, -2812, -2814, -2816, -2818, -2820, -2822, -2824, -2826, -2828, -2830, -2832, -2834, -2836, -2838, -2840, -2842, -2844, -2846, -2848, -2850, -2852, -2854, -2856, -2858, -2860, -2862, -2864, -2866, -2868, -2870, -2872, -2874, -2876, -2878, -2880, -2882, -2884, -2886, -2888, -2890, -2892, -2894, -2896, -2898, -2900, -2902, -2904, -2906, -2908, -2910, -2912, -2914, -2916, -2918, -2920, -2922, -2924, -2926, -2928, -2930, -2932, -2934, -2936, -2938, -2940, -2942, -2944, -2946, -2948, -2950, -2952, -2954, -2956, -2958, -2960, -2962, -2964, -2966, -2968, -2970, -2972, -2974, -2976, -2978, -2980, -2982, -2984, -2986, -2988, -2990, -2992, -2994, -2996, -2998, -3000, -3002, -3004, -3006, -3008, -3010, -3012, -3014, -3016, -3018, -3020, -3022, -3024, -3026, -3028, -3030, -3032, -3034, -3036, -3038, -3040, -3042, -3044, -3046, -3048, -3050, -3052, -3054, -3056, -3058, -3060, -3062, -3064, -3066, -3068, -3070, -3072, -3074, -3076, -3078, -3080, -3082, -3084, -3086, -3088, -3090, -3092, -3094, -3096, -3098, -3100, -3102, -3104, -3106, -3108, -3110, -3112, -3114, -3116, -3118, -3120, -3122, -3124, -3126, -3128, -3130, -3132, -3134, -3136, -3138, -3140, -3142, -3144, -3146, -3148, -3150, -3152, -3154, -3156, -3158, -3160, -3162, -3164, -3166, -3168, -3170, -3172, -3174, -3176, -3178, -3180, -3182, -3184, -3186, -3188, -3190, -3192, -3194, -3196, -3198, -3200, -3202, -3204, -3206, -3208, -3210, -3212, -3214, -3216, -3218, -3220, -3222, -3224, -3226, -3228, -3230, -3232, -3234, -3236, -3238, -3240, -3242, -3244, -3246, -3248, -3250, -3252, -3254, -3256, -3258, -3260, -3262, -3264, -3266, -3268, -3270, -3272, -3274, -3276, -3278, -3280, -3282, -3284, -3286, -3288, -3290, -3292, -3294, -3296, -3298, -3300, -3302, -3304, -3306, -3308, -3310, -3312, -3314, -3316, -3318, -3320, -3322, -3324, -3326, -3328, -3330, -3332, -3334, -3336, -3338, -3340, -3342, -3344, -3346, -3348, -3350, -3352, -3354, -3356, -3358, -3360, -3362, -3364, -3366, -3368, -3370, -3372, -3374, -3376, -3378, -3380, -3382, -3384, -3386, -3388, -3390, -3392, -3394, -3396, -3398, -3400, -3402, -3404, -3406, -3408, -3410, -3412, -3414, -3416, -3418, -3420, -3422, -3424, -3426, -3428, -3430, -3432, -3434, -3436, -3438, -3440, -3442, -3444, -3446, -3448, -3450, -3452, -3454, -3456, -3458, -3460, -3462, -3464, -3466, -3468, -3470, -3472, -3474, -3476, -3478, -3480, -3482, -3484, -3486, -3488, -3490, -3492, -3494, -3496, -3498, -3500, -3502, -3504, -3506, -3508, -3510, -3512, -3514, -3516, -3518, -3520, -3522, -3524, -3526, -3528, -3530, -3532, -3534, -3536, -3538, -3540, -3542, -3544, -3546, -3548, -3550, -3552, -3554, -3556, -3558, -3560, -3562, -3564, -3566, -3568, -3570, -3572, -3574, -3576, -3578, -3580, -3582, -3584, -3586, -3588, -3590, -3592, -3594, -3596, -3598, -3600, -3602, -3604, -3606, -3608, -3610, -3612, -3614, -3616, -3618, -3620, -3622, -3624, -3626, -3628, -3630, -3632, -3634, -3636, -3638, -3640, -3642, -3644, -3646, -3648, -3650, -3652, -3654, -3656, -3658, -3660, -3662, -3664, -3666, -3668, -3670, -3672, -3674, -3676, -3678, -3680, -3682, -3684, -3686, -3688, -3690, -3692, -3694, -3696, -3698, -3700, -3702, -3704, -3706, -3708, -3710, -3712, -3714, -3716, -3718, -3720, -3722, -3724, -3726, -3728, -3730, -3732, -3734, -3736, -3738, -3740, -3742, -3744, -3746, -3748, -3750, -3752, -3754, -3756, -3758, -3760, -3762, -3764, -3766, -3768, -3770, -3772, -3774, -3776, -3778, -3780, -3782, -3784, -3786, -3788, -3790, -3792, -3794, -3796, -3798, -3800, -3802, -3804, -3806, -3808, -3810, -3812, -3814, -3816, -3818, -3820, -3822, -3824, -3826, -3828, -3830, -3832, -3834, -3836, -3838, -3840, -3842, -3844, -3846, -3848, -3850, -3852, -3854, -3856, -3858, -3860, -3862, -3864, -3866, -3868, -3870, -3872, -3874, -3876, -3878, -3880, -3882, -3884, -3886, -3888, -3890, -3892, -3894, -3896, -3898, -3900, -3902, -3904, -3906, -3908, -3910, -3912, -3914, -3916, -3918, -3920, -3922, -3924, -3926, -3928, -3930, -3932, -3934, -3936, -3938, -3940, -3942, -3944, -3946, -3948, -3950, -3952, -3954, -3956, -3958, -3960, -3962, -3964, -3966, -3968, -3970, -3972, -3974, -3976, -3978, -3980, -3982, -3984, -3986, -3988, -3990, -3992, -3994, -3996, -3998, -4000, -4002, -4004, -4006, -4008, -4010, -4012, -4014, -4016, -4018, -4020, -4022, -4024, -4026, -4028, -4030, -4032, -4034, -4036, -4038, -4040, -4042, -4044, -4046, -4048, -4050, -4052, -4054, -4056, -4058, -4060, -4062, -4064, -4066, -4068, -4070, -4072, -4074, -4076, -4078, -4080, -4082, -4084, -4086, -4088, -4090, -4092, -4094, -4096, -4098, -4100, -4102, -4104, -4106, -4108, -4110, -4112, -4114, -4116, -4118, -4120, -4122, -4124, -4126, -4128, -4130, -4132, -4134, -4136, -4138, -4140, -4142, -4144, -4146, -4148, -4150, -4152, -4154, -4156, -4158, -4160, -4162, -4164, -4166, -4168, -4170, -4172, -4174, -4176, -4178, -4180, -4182, -4184, -4186, -4188, -4190, -4192, -4194, -4196, -4198, -4200, -4202, -4204, -4206, -4208, -4210, -4212, -4214, -4216, -4218, -4220, -4222, -4224, -4226, -4228, -4230, -4232, -4234, -4236, -4238, -4240, -4242, -4244, -4246, -4248, -4250, -4252, -4254, -4256, -4258, -4260, -4262, -4264, -4266, -4268, -4270, -4272, -4274, -4276, -4278, -4280, -4282, -4284, -4286, -4288, -4290, -4292, -4294, -4296,
```



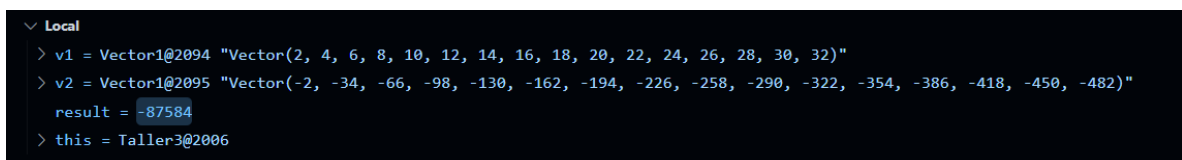
```
Local
> x0$1 = Tuple2@2099 "(2, -2)"
```

Figura 26: Debug MultMatrizParalela



```
Local
> x0$1 = Tuple2@2127 "(32, -482)"
```

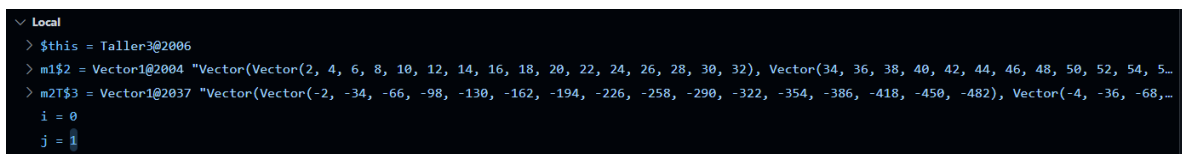
Figura 27: Debug MultMatrizParalela



```
Local
> v1 = Vector1@2094 "Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32)"
> v2 = Vector1@2095 "Vector(-2, -34, -66, -98, -130, -162, -194, -226, -258, -290, -322, -354, -386, -418, -450, -482)"
  result = -87584
> this = Taller3@2006
```

Figura 28: Debug MultMatrizParalela

Se observan resultados intermedios mientras se completan las tareas asignadas. Algunos elementos de C ya han sido calculados y almacenados, mientras que otras tareas continúan en ejecución. Este paso refleja cómo se actualiza gradualmente la matriz resultante.



```
Local
> $this = Taller3@2006
> m1$2 = Vector1@2004 "Vector(Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32), Vector(34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100))"
> m2T$3 = Vector1@2037 "Vector(Vector(-2, -34, -66, -98, -130, -162, -194, -226, -258, -290, -322, -354, -386, -418, -450, -482), Vector(-4, -36, -68, -100, -132, -164, -196, -228, -260, -292, -324, -356, -388, -420, -452, -484, -516, -548, -580, -612, -644, -676, -708, -740, -772, -804, -836, -868, -900, -932, -964, -996, -1028, -1060, -1092, -1124, -1156, -1188, -1220, -1252, -1284, -1316, -1348, -1380, -1412, -1444, -1476, -1508, -1540, -1572, -1604, -1636, -1668, -1700, -1732, -1764, -1796, -1828, -1860, -1892, -1924, -1956, -1988, -2020, -2052, -2084, -2116, -2148, -2180, -2212, -2244, -2276, -2308, -2340, -2372, -2404, -2436, -2468, -2500, -2532, -2564, -2596, -2628, -2660, -2692, -2724, -2756, -2788, -2820, -2852, -2884, -2916, -2948, -2980, -3012, -3044, -3076, -3108, -3140, -3172, -3204, -3236, -3268, -3300, -3332, -3364, -3396, -3428, -3460, -3492, -3524, -3556, -3588, -3620, -3652, -3684, -3716, -3748, -3780, -3812, -3844, -3876, -3908, -3940, -3972, -4004, -4036, -4068, -4100, -4132, -4164, -4196, -4228, -4260, -4292, -4324, -4356, -4388, -4420, -4452, -4484, -4516, -4548, -4580, -4612, -4644, -4676, -4708, -4740, -4772, -4804, -4836, -4868, -4900, -4932, -4964, -4996, -5028, -5060, -5092, -5124, -5156, -5188, -5220, -5252, -5284, -5316, -5348, -5380, -5412, -5444, -5476, -5508, -5540, -5572, -5604, -5636, -5668, -5700, -5732, -5764, -5796, -5828, -5860, -5892, -5924, -5956, -5988, -6020, -6052, -6084, -6116, -6148, -6180, -6212, -6244, -6276, -6308, -6340, -6372, -6404, -6436, -6468, -6500, -6532, -6564, -6596, -6628, -6660, -6692, -6724, -6756, -6788, -6820, -6852, -6884, -6916, -6948, -6980, -7012, -7044, -7076, -7108, -7140, -7172, -7204, -7236, -7268, -7300, -7332, -7364, -7396, -7428, -7460, -7492, -7524, -7556, -7588, -7620, -7652, -7684, -7716, -7748, -7780, -7812, -7844, -7876, -7908, -7940, -7972, -8004, -8036, -8068, -8100, -8132, -8164, -8196, -8228, -8260, -8292, -8324, -8356, -8388, -8420, -8452, -8484, -8516, -8548, -8580, -8612, -8644, -8676, -8708, -8740, -8772, -8804, -8836, -8868, -8900, -8932, -8964, -8996, -9028, -9060, -9092, -9124, -9156, -9188, -9220, -9252, -9284, -9316, -9348, -9380, -9412, -9444, -9476, -9508, -9540, -9572, -9604, -9636, -9668, -9700, -9732, -9764, -9796, -9828, -9860, -9892, -9924, -9956, -9988, -10020, -10052, -10084, -10116, -10148, -10180, -10212, -10244, -10276, -10308, -10340, -10372, -10404, -10436, -10468, -10500, -10532, -10564, -10596, -10628, -10660, -10692, -10724, -10756, -10788, -10820, -10852, -10884, -10916, -10948, -10980, -11012, -11044, -11076, -11108, -11140, -11172, -11204, -11236, -11268, -11300, -11332, -11364, -11396, -11428, -11460, -11492, -11524, -11556, -11588, -11620, -11652, -11684, -11716, -11748, -11780, -11812, -11844, -11876, -11908, -11940, -11972, -12004, -12036, -12068, -12100, -12132, -12164, -12196, -12228, -12260, -12292, -12324, -12356, -12388, -12420, -12452, -12484, -12516, -12548, -12580, -12612, -12644, -12676, -12708, -12740, -12772, -12804, -12836, -12868, -12900, -12932, -12964, -12996, -13028, -13060, -13092, -13124, -13156, -13188, -13220, -13252, -13284, -13316, -13348, -13380, -13412, -13444, -13476, -13508, -13540, -13572, -13604, -13636, -13668, -13700, -13732, -13764, -13796, -13828, -13860, -13892, -13924, -13956, -13988, -14020, -14052, -14084, -14116, -14148, -14180, -14212, -14244, -14276, -14308, -14340, -14372, -14404, -14436, -14468, -14500, -14532, -14564, -14596, -14628, -14660, -14692, -14724, -14756, -14788, -14820, -14852, -14884, -14916, -14948, -14980, -15012, -15044, -15076, -15108, -15140, -15172, -15204, -15236, -15268, -15300, -15332, -15364, -15396, -15428, -15460, -15492, -15524, -15556, -15588, -15620, -15652, -15684, -15716, -15748, -15780, -15812, -15844, -15876, -15908, -15940, -15972, -16004, -16036, -16068, -16100, -16132, -16164, -16196, -16228, -16260, -16292, -16324, -16356, -16388, -16420, -16452, -16484, -16516, -16548, -16580, -16612, -16644, -16676, -16708, -16740, -16772, -16804, -16836, -16868, -16900, -16932, -16964, -16996, -17028, -17060, -17092, -17124, -17156, -17188, -17220, -17252, -17284, -17316, -17348, -17380, -17412, -17444, -17476, -17508, -17540, -17572, -17604, -17636, -17668, -17700, -17732, -17764, -17796, -17828, -17860, -17892, -17924, -17956, -17988, -18020, -18052, -18084, -18116, -18148, -18180, -18212, -18244, -18276, -18308, -18340, -18372, -18404, -18436, -18468, -18500, -18532, -18564, -18596, -18628, -18660, -18692, -18724, -18756, -18788, -18820, -18852, -18884, -18916, -18948, -18980, -19012, -19044, -19076, -19108, -19140, -19172, -19204, -19236, -19268, -19300, -19332, -19364, -19396, -19428, -19460, -19492, -19524, -19556, -19588, -19620, -19652, -19684, -19716, -19748, -19780, -19812, -19844, -19876, -19908, -19940, -19972, -20004, -20036, -20068, -20100, -20132, -20164, -20196, -20228, -20260, -20292, -20324, -20356, -20388, -20420, -20452, -20484, -20516, -20548, -20580, -20612, -20644, -20676, -20708, -20740, -20772, -20804, -20836, -20868, -20900, -20932, -20964, -20996, -21028, -21060, -21092, -21124, -21156, -21188, -21220, -21252, -21284, -21316, -21348, -21380, -21412, -21444, -21476, -21508, -21540, -21572, -21604, -21636, -21668, -21700, -21732, -21764, -21796, -21828, -21860, -21892, -21924, -21956, -21988, -22020, -22052, -22084, -22116, -22148, -22180, -22212, -22244, -22276, -22308, -22340, -22372, -22404, -22436, -22468, -22500, -22532, -22564, -22596, -22628, -22660, -22692, -22724, -22756, -22788, -22820, -22852, -22884, -22916, -22948, -22980, -23012, -23044, -23076, -23108, -23140, -23172, -23204, -23236, -23268, -23300, -23332, -23364, -23396, -23428, -23460, -23492, -23524, -23556, -23588, -23620, -23652, -23684, -23716, -23748, -23780, -23812, -23844, -23876, -23908, -23940, -23972, -24004, -24036, -24068, -24100, -24132, -24164, -24196, -24228, -24260, -24292, -24324, -24356, -24388, -24420, -24452, -24484, -24516, -24548, -24580, -24612, -24644, -24676, -24708, -24740, -24772, -24804, -24836, -24868, -24900, -24932, -24964, -24996, -25028, -25060, -25092, -25124, -25156, -25188, -25220, -25252, -25284, -25316, -25348, -25380, -25412, -25444, -25476, -25508, -25540, -25572, -25604, -25636, -25668, -25700, -25732, -25764, -25796, -25828, -25860, -25892, -25924, -25956, -25988, -26020, -26052, -26084, -26116, -26148, -26180, -26212, -26244, -26276, -26308, -26340, -26372, -26404, -26436, -26468, -26500, -26532, -26564, -26596, -26628, -26660, -26692, -26724, -26756, -26788, -26820, -26852, -26884, -26916, -26948, -26980, -27012, -27044, -27076, -27108, -27140, -27172, -27204, -27236, -27268, -27300, -27332, -27364, -27396, -27428, -27460, -27492, -27524, -27556, -27588, -27620, -27652, -27684, -27716, -27748, -27780, -27812, -27844, -27876, -27908, -27940, -27972, -28004, -28036, -28068, -28100, -28132, -28164, -28196, -28228, -28260, -28292, -28324, -28356, -28388, -28420, -28452, -28484, -28516, -28548, -28580, -28612, -28644, -28676, -28708, -28740, -28772, -28804, -28836, -28868, -28900, -28932, -28964, -28996, -29028, -29060, -29092, -29124, -29156, -29188, -29220, -29252, -29284, -29316, -29348, -29380, -29412, -29444, -29476, -29508, -29540, -29572, -29604, -29636, -29668, -29700, -29732, -29764, -29796, -29828, -29860, -29892, -29924, -29956, -29988, -30020, -30052, -30084, -30116, -30148, -30180, -30212, -30244, -30276, -30308, -30340, -30372, -30404, -30436, -30468, -30500, -30532, -30564, -30596, -30628, -30660, -30692, -30724, -30756, -30788, -30820, -30852, -30884, -30916, -30948, -30980, -31012, -31044, -31076, -31108, -31140, -31172, -31204, -31236, -31268, -31300, -31332, -31364, -31396, -31428, -31460, -31492, -31524, -31556, -31588, -31620, -31652, -31684, -31716, -31748, -31780, -31812, -31844, -31876, -31908, -31940, -31972, -32004, -32036, -32068, -32100, -32132, -32164, -32196, -32228, -32260, -32292, -32324, -32356, -32388, -32420, -32452, -32484, -32516, -32548, -32580, -32612, -32644, -32676, -32708, -32740, -32772, -32804, -32836, -32868, -32900, -32932, -32964, -32996, -33028, -33060, -33092, -33124, -33156, -33188, -33220, -33252, -33284, -33316, -33348, -33380, -33412, -33444, -33476, -33508, -33540, -33572, -33604, -33636, -33668, -33700, -33732, -33764, -33796, -33828, -33860, -33892, -33924, -33956, -33988, -34020, -34052, -34084, -34116, -34148, -34180, -34212, -34244, -34276, -34308, -34340, -34372, -34404, -34436, -34468, -34500, -34532, -34564, -34596, -34628, -34660, -34692, -34724, -34756, -34788, -34820, -34852, -34884, -34916, -34948, -34980, -35012, -35044, -35076, -35108, -35140, -35172, -35204, -35236, -35268, -35300, -35332, -35364, -35396, -35428, -35460, -35492, -35524, -35556, -35588, -35620, -35652, -35684, -35716, -35748, -35780, -35812, -35844, -35876, -35908, -35940, -35972, -36004, -36036, -36068, -36100, -36132, -36164, -36196, -36228, -36260, -36292, -36324, -36356, -36388, -36420, -36452, -36484, -36516, -36548, -36580, -36612, -36644, -36676, -36708, -36740, -36772, -36804, -36836, -36868, -36900, -36932, -36964, -36996, -37028, -37060, -37092, -37124, -37156, -37188, -37220, -37252, -37284, -37316, -37348, -37380, -37412, -37444, -37476, -37508, -37540, -37572, -37604, -37636, -37668, -37700, -37732, -37764, -37796, -37828, -37860, -37892, -37924, -37956, -37988, -38020, -38052, -38084, -38116, -38148, -38180, -38212, -38244, -38276, -38308, -38340, -38372, -38404, -38436, -38468, -38500, -38532, -38564, -38596, -38628, -38660, -38692, -38724, -38756, -38788, -38820, -38852, -38884, -38916, -38948, -38980, -39012, -39044, -39076, -39108, -39140, -39172, -39204, -39236, -39268, -39300, -39332, -39364, -39396, -39428, -39460, -39492, -39524, -39556, -39588, -39620, -39652, -39684, -39716, -39748, -39780, -39812, -39844, -39876, -39908, -39940, -39972, -40004, -40036, -40068, -40100, -40132, -40164, -40196, -40228, -40260, -40292, -40324, -40356, -40388, -40420, -40452, -40484, -40516, -40548, -40580, -40612, -40644, -40676, -40708, -40740, -40772, -40804, -40836, -40868, -40900, -40932, -40964, -40996, -41028, -41060, -41092, -41124, -41156, -41188, -41220, -41252, -41284, -41316, -41348, -41380, -41412, -41444, -41476, -41508, -41540, -41572, -41604, -41636, -41668, -41700, -41732, -41764, -41796, -41828, -41860, -41892, -41924, -41956, -41988, -42020, -42052, -42084, -42116, -42148, -42180, -42212, -42244, -42276, -42308, -42340, -42372, -42404, -42436, -42468, -42500, -42532, -42564, -42596, -42628, -42660, -42692, -42724, -42756, -42788, -42820, -42852, -42884, -42916, -42948, -42980, -43012, -43044, -43076, -43108, -43140, -43172, -43204, -43236, -43268, -43300, -43332, -43364, -43396, -43428, -43460, -43492, -43524, -43556, -43588, -43620, -43652, -43684, -43716, -43748, -43780, -43812, -43844, -43876, -43908, -43940, -43972, -44004, -44036, -44068, -44100, -44132, -44164, -44196, -44228, -44260, -44292, -44324, -44356, -44388, -44420, -44452, -44484, -44516, -44548, -44580, -44612, -44644, -44676, -44708, -44740, -44772, -44804, -44836, -44868, -44900, -44932, -44964, -44996, -45028, -45060, -45092, -45124, -45156, -45188, -45220, -45252, -45284, -45316, -45348, -45380, -45412, -45444, -45476, -45508, -45540, -45572, -45604, -45636, -45668, -45700, -45732, -45764, -45796, -45828, -45860, -45892, -45924, -45956, -45988, -46020, -46052, -46084, -46116, -46148, -46180, -46212, -46244, -46276, -46308, -46340, -46372, -46404, -46436, -46468, -46500, -46532, -46564, -46596, -46628, -46660, -46692, -46724, -46756, -46788, -46820, -46852, -46884, -46916, -46948, -46980, -47012, -47044, -47076, -47108, -47140, -47172, -47204, -47236, -47268, -47300, -47332, -47364, -47396, -47428, -47460, -47492, -47524, -47556, -47588, -47620, -47652, -47684, -47716, -47748, -47780, -47812, -47844, -47876, -47908, -47940, -47972, -48004, -48036, -48068, -48100, -48132, -48164, -48196, -48228, -48260, -48292, -48324, -48356, -48388, -48420, -48452, -48484, -48516, -48548, -48580, -48612, -48644, -48676, -48708, -48740, -48772, -48804, -48836, -48868, -48900, -48932, -48964, -48996, -49028, -49060, -49092, -49124, -49156, -49188, -49220, -49252, -49284, -49316, -49348, -49380, -49412, -49444, -49476, -49508, -49540, -49572, -49604, -49636, -49668, -49700, -49732, -49764, -49796, -49828, -49860, -49892, -49924, -49956, -49988, -50020, -50052, -50084, -50116, -50148, -50180, -50212, -50244, -50276, -50308, -50340, -50372, -50404, -50436, -50468, -50500, -50532, -50564, -50596, -50628, -50660, -50692, -50724, -50756, -50788, -50820, -50852, -50884, -50916, -50948, -50980, -51012, -51044, -51076, -51108, -51140, -51172, -51204, -51236, -51268, -51300, -51332, -51364, -51396, -51428, -51460, -51492, -51524, -51556, -51588, -51620, -51652, -51684, -51716, -51748, -51780, -51812, -51844, -51876, -51908, -51940, -51972, -52004, -52036, -52068, -52100, -52132, -52164, -52196, -52228, -52260, -52292, -52324, -52356, -52388, -52420, -52452, -52484, -52516, -52548, -52580, -52612, -52644, -52676, -52708, -52740, -52772, -52804, -52836, -52868, -52900, -52932,
```

```

Local
> $this = Taller3@2006
> m1$2 = Vector1@2004 "Vector(Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32), Vector(34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, ...))
> m2T$3 = Vector1@2037 "Vector(Vector(-2, -34, -66, -98, -130, -162, -194, -226, -258, -290, -322, -354, -386, -418, -450, -482), Vector(-4, -36, -68, ...))
i = 15
j = 15

```

Figura 31: Debug MultMatrizParalela

```

Local
> $this = Taller3@2006
> m1$2 = Vector1@2004 "Vector(Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32), Vector(34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, ...))
i$1 = 15
> m2T$3 = Vector1@2037 "Vector(Vector(-2, -34, -66, -98, -130, -162, -194, -226, -258, -290, -322, -354, -386, -418, -450, -482), Vector(-4, -36, -68, ...))
j$1 = 15

```

Figura 32: Debug MultMatrizParalela

```

VARIABLES
Local
> $this = Taller3@2006
> m1$2 = Vector1@2004 "Vector(Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32), Vector(34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, ...))
> m2T$3 = Vector1@2037 "Vector(Vector(-2, -34, -66, -98, -130, -162, -194, -226, -258, -290, -322, -354, -386, -418, -450, -482), Vector(-4, -36, -68, ...))
i = 15
j = 15
> taskElement = package$DefaultTaskScheduler$anon$1@2414

```

Figura 33: Debug MultMatrizParalela

Finalmente, se muestra la matriz C completamente calculada. Todos los elementos han sido determinados por las tareas paralelas, y la matriz resultante se presenta lista para su validación o uso posterior en el programa. Este resultado confirma el correcto funcionamiento de la función `multMatrizPar` y la ejecución exitosa del test.

```

Local
> m1 = Vector1@2004 "Vector(Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32), Vector(34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, ...))
> m2 = Vector1@2005 "Vector(Vector(-2, -4, -6, -8, -10, -12, -14, -16, -18, -20, -22, -24, -26, -28, -30, -32), Vector(-34, -36, -38, -40, -42, -44, ...))
n = 16
> m2T = Vector1@2037 "Vector(Vector(-2, -34, -66, -98, -130, -162, -194, -226, -258, -290, -322, -354, -386, -418, -450, -482), Vector(-4, -36, -68, ...))
> m3 = Vector1@2417 "Vector(Vector(-87584, -88128, -88672, -89216, -89760, -90304, -90848, -91392, -91936, -92480, -93024, -93568, -94112, -94656, -95200, ...))
> this = Taller3@2006

```

Figura 34: Debug MultMatrizParalela

La función asegura un cálculo eficiente utilizando técnicas de paralelización, ideal para grandes matrices donde la computación secuencial sería lenta.

1.3. Funciones para Multiplicación *Recursiva* de Matrices

1.3.1. Función para extraer submatrices

Esta función tiene como objetivo extraer submatrices de diferentes tamaños a partir de una matriz principal previamente definida. Para ilustrar cómo opera esta función, consideremos la siguiente matriz como ejemplo:

```

1  val matriz = Vector(
2      Vector(1, 2, 3, 4),
3      Vector(5, 6, 7, 8),
4      Vector(9, 10, 11, 12),
5      Vector(13, 14, 15, 16)
6  )

```

Para verificar el correcto funcionamiento de la función, se utiliza un test donde se extrae una submatriz de tamaño 2×2 , comenzando desde la esquina superior izquierda de la matriz original. El código del test es el siguiente:

```

1  test("Extraer submatriz A11 de tama o 2x2 desde la esquina superior izquierda") {
2      val resultadoEsperado = Vector(
3          Vector(1, 2),
4          Vector(5, 6)
5      )
6      assert(taller3.subMatriz(matriz, 0, 0, 2) == resultadoEsperado)
7  }

```

El proceso comienza definiendo las variables necesarias. En este caso, la matriz original es representada por 'm'. Los valores 'i' (que indica la fila inicial) y 'j' (que especifica la columna inicial) determinan las coordenadas desde las cuales se iniciará la extracción de la submatriz. Finalmente, el parámetro 'l' indica el tamaño de la submatriz que se desea extraer, en este caso, $l = 2$, lo que resulta en una submatriz de dimensiones 2×2 .

```

Local
> m = Vector1@1980 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
i = 0
j = 0
l = 2
> this = Taller3@1981

```

Figura 35: Debug SubMatriz

El proceso se basa en la creación de una matriz de dimensiones 2×2 utilizando dos bucles anidados, los cuales son generados automáticamente por la función `Vector.tabulate`. Esta función recorre las posiciones necesarias para rellenar la submatriz, basándose en los valores de la matriz original.

```

Local
> m$2 = Vector1@1985 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
i$2 = 0
j$2 = 0
x = 0
y = 0

```

Figura 36: Debug SubMatriz

El cálculo comienza evaluando el primer índice $i = 0$, lo que inicia el procesamiento de la primera fila de la submatriz.

```

Local
> ->$anonfun$tabulate$2() = Integer@2033 "1"
n = 2
> f = IterableFactory$$$Lambda$158/0x000001d64d1878a8@2038 "scala.collection.IterableFactory$$$Lambda$158/0x000001d64d1...
> b = VectorBuilder@2039 "VectorBuilder(len1=0, lenRest=0, offset=0, depth=1)"
i = 0
> this = Vector$@2040

```

Figura 37: Debug SubMatriz

El primer valor evaluado es para las coordenadas $i = 0$ y $j = 0$, donde se toma el elemento $m(0)(0) = 1$. Este valor se coloca en la posición $(0,0)$ de la submatriz. Luego, se evalúa $i = 0$ y $j = 1$, donde $m(0)(1) = 2$. Este valor se inserta en la posición $(0,1)$ de la submatriz, completando así la primera fila.

```

Local
> m$2 = Vector1@1985 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
i$2 = 0
j$2 = 0
x = 0
y = 1

```

Figura 38: Debug SubMatriz

El procesamiento continúa ahora con la segunda fila, es decir, con el índice $i = 1$. El primer cálculo en esta fila es para $i = 1$ y $j = 0$, lo que corresponde al elemento $m(1)(0) = 5$. Este valor se inserta en la posición $(1,0)$ de la submatriz.

```

Local
> ->$apply() = Integer@2091 "5"
> f$1 = Taller3$$$Lambda$156/0x000001d64d181290@2014 "taller.Taller3$$$Lambda$156/0x000001d64d181290@21947940"
i1$1 = 1
x$1 = 0

```

Figura 39: Debug SubMatriz

El segundo cálculo en la segunda fila es para $i = 1$ y $j = 1$, obteniendo el elemento $m(1)(1) = 6$. Este valor se ubica en la posición $(1,1)$ de la submatriz, completando así la construcción de la segunda fila.

```

Local
->$anonfun$subMatriz$1() = 6
> v1 = Integer@2033 "1"
> v2 = Integer@2033 "1"
> this = Taller3$$$Lambda$156/0x000001d64d181290@2014 "taller.Taller3$$$Lambda$156/0x000001d64d181290@21947940"

```

Figura 40: Debug SubMatriz

El resultado es un nuevo **Vector** bidimensional que representa la submatriz, formado por dos filas: la primera fila $(1,2)$ y la segunda fila $(5,6)$. Esto genera la submatriz final de tamaño 2×2 , como se esperaba.

```

Local
> ->tabulate() = Vector1@2120 "Vector(5, 6)"
> $this = Vector$@2040
  n2$5 = 2
> f$1 = Taller3$$Lambda$156/0x000001d64d181290@2014 "taller.Taller3$$Lambda$156/0x000001d64d181290@21947940"
  i1 = 1

```

Figura 41: Debug SubMatriz

Finalmente, se realiza una verificación para asegurar que la submatriz generada cumple con los parámetros iniciales, tanto en tamaño como en contenido. Al comparar la submatriz obtenida con la solución esperada, se observa que coinciden perfectamente, lo que permite pasar el test de manera satisfactoria.

```

Local
> ->tabulate() = Vector1@2132 "Vector(Vector(1, 2), Vector(5, 6))"
  n1 = 2
  n2 = 2
> f = Taller3$$Lambda$156/0x000001d64d181290@2014 "taller.Taller3$$Lambda$156/0x000001d64d181290@21947940"
> this = Vector$@2040

```

Figura 42: Debug SubMatriz

1.3.2. Función para sumar matrices

La función `sumMatriz` suma elemento a elemento dos matrices cuadradas **m1** y **m2** de dimensión $n \times n$. Para ello, utiliza la estructura `Vector.tabulate` para iterar sobre los índices i y j de las matrices de entrada. En cada posición (i, j) , calcula el resultado como $\mathbf{m1}(i, j) + \mathbf{m2}(i, j)$, generando una nueva matriz resultante **m3**.

Ejemplo de prueba de la función Para verificar la funcionalidad de `sumMatriz`, se definen las matrices de entrada y el resultado esperado:

```

1  val matriz1 = Vector(
2      Vector(1, 2, 3, 4),
3      Vector(5, 6, 7, 8),
4      Vector(9, 10, 11, 12),
5      Vector(13, 14, 15, 16)
6  )

```

La matriz resultante esperada se define de la siguiente manera:

```

1  test("Suma de una matriz con una de valores negativos") {
2      val matrizNegativa = Vector(
3          Vector(-1, -2, -3, -4),
4          Vector(-5, -6, -7, -8),
5          Vector(-9, -10, -11, -12),
6          Vector(-13, -14, -15, -16)
7      )
8      val resultadoEsperado = Vector.fill(4, 4)(0)
9      assert(taller3.sumMatriz(matriz1, matrizNegativa) == resultadoEsperado)
10 }

```



```

▼ Local
  ▼ m1 = Vector1@1998 "Vector(Vector(1, 2, 3, 4), Vec..."
    ▼ prefix1 = Object[4]@2004
      > 0 = Vector1@2005 "Vector(1, 2, 3, 4)"
      > 1 = Vector1@2006 "Vector(5, 6, 7, 8)"
      > 2 = Vector1@2007 "Vector(9, 10, 11, 12)"
      > 3 = Vector1@2008 "Vector(13, 14, 15, 16)"
    ▼ m2 = Vector1@1999 "Vector(Vector(-1, -2, -3, -4),..."
      ▼ prefix1 = Object[4]@2003
        > 0 = Vector1@2013 "Vector(-1, -2, -3, -4)"
        > 1 = Vector1@2014 "Vector(-5, -6, -7, -8)"
        > 2 = Vector1@2015 "Vector(-9, -10, -11, -12)"
        > 3 = Vector1@2016 "Vector(-13, -14, -15, -16)"
      > this = Taller3@2000

```

Figura 43: Debug SumMatriz

Se observa el estado inicial de las matrices `m1` y `m2`. La matriz `m2` contiene los valores negativos de `m1`, por lo que el resultado esperado será una matriz de ceros.

La función `sumMatriz` comienza iterando sobre los índices i y j de las matrices de entrada. La primera iteración se realiza sobre la posición $(0, 0)$.

```

▼ Local
  > m1$3 = Vector1@1998 "Vector(Vector(1, 2, 3, 4), V..."
  > m2$1 = Vector1@1999 "Vector(Vector(-1, -2, -3, -4..."
  i = 0
  j = 0

```

Figura 44: Debug SumMatriz

En este caso:

$$m1(0, 0) + m2(0, 0) = 1 + (-1) = 0$$

En las Iteraciones posteriores, se observa el cálculo de la posición $(0, 2)$ durante la primera fila. Aquí:

$$m1(0, 2) + m2(0, 2) = 3 + (-3) = 0$$

```

▼ Local
  → $anonfun$sumMatriz$1() = 0
  > v1 = Integer@2038 "0"
  > v2 = Integer@2038 "0"
  > this = Taller3$Lambda$158/0x000002bb4d18a778@203...

```

Figura 45: Debug SumMatriz

De manera similar, la función continúa calculando cada elemento fila por fila. En la siguiente imagen, se analiza un cálculo intermedio:

```

v Local
> ->apply() = Integer@2038 "0"
v f$1 = Taller3$$Lambda$158/0x000002bb4d18a778@2039 "t...
v arg$1 = Vector1@1998 "Vector(Vector(1, 2, 3, 4), Ve...
v prefix1 = Object[4]@2004
> 0 = Vector1@2005 "Vector(1, 2, 3, 4)"
> 1 = Vector1@2006 "Vector(5, 6, 7, 8)"
> 2 = Vector1@2007 "Vector(9, 10, 11, 12)"
> 3 = Vector1@2008 "Vector(13, 14, 15, 16)"
v arg$2 = Vector1@1999 "Vector(Vector(-1, -2, -3, -4)-...
v prefix1 = Object[4]@2003
> 0 = Vector1@2013 "Vector(-1, -2, -3, -4)"
> 1 = Vector1@2014 "Vector(-5, -6, -7, -8)"
> 2 = Vector1@2015 "Vector(-9, -10, -11, -12)"
> 3 = Vector1@2016 "Vector(-13, -14, -15, -16)"
i1$1 = 0
x$1 = 0

```

Figura 46: Debug SumMatriz

Aquí, para la posición (1,1):

$$m1(1,1) + m2(1,1) = 5 + (-5) = 0$$

A medida que la función avanza, los elementos ya calculados se almacenan en la matriz resultante. Se muestra cómo la función utiliza `Vector.tabulate` para iterar sobre los índices y construir la matriz **m3**.

```

v Local
> ->$anonfun$tabulate$2() = Integer@2038 "0"
n = 4
v f = IterableFactory$$Lambda$160/0x000002bb4d18a778@2039
v arg$1 = Taller3$$Lambda$158/0x000002bb4d18a778@2039
> arg$1 = Vector1@1998 "Vector(Vector(1, 2, ..., 13, 14, 15, 16))"
> arg$2 = Vector1@1999 "Vector(Vector(-1, -2, ..., -13, -14, -15, -16))"
arg$2 = 0
> b = VectorBuilder@2071 "VectorBuilder(len1=0, lenRest=0, offset=0, depth=1)"
i = 0
> this = Vector$@2072

```

Figura 47: Debug SumMatriz

Se observa la construcción parcial de la matriz **m3**, donde las posiciones calculadas tienen un valor de 0. Esto es consistente con las expectativas dado que **m2** = -**m1**.

```

v Local
n = 4
v f = IterableFactory$$Lambda$160/0x000002bb4d18b108@2070 "scala.collection.Iterator"
v arg$1 = Taller3$$Lambda$158/0x000002bb4d18a778@2039 "taller.Taller3$$Lambda$158/0x000002bb4d18a778@2039"
> arg$1 = Vector1@1998 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
> arg$2 = Vector1@1999 "Vector(Vector(-1, -2, -3, -4), Vector(-5, -6, -7, -8), Vector(-9, -10, -11, -12), Vector(-13, -14, -15, -16))"
arg$2 = 0
> b = VectorBuilder@2071 "VectorBuilder(len1=1, lenRest=0, offset=0, depth=1)"
i = 0
> this = Vector$@2072

```

Figura 48: Debug SumMatriz

Finalmente, la matriz **m3** es completada, y se confirma que todos los valores son 0, como se esperaba.

```

Local
n = 4
f = IterableFactory$$Lambda$160/0x000002bb4d18b108@2070 "scala.collection.Iter...
arg$1 = Taller3$$Lambda$158/0x000002bb4d18a778@2039 "taller.Taller3$$Lambda$...
> arg$1 = Vector1@1998 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector...
> arg$2 = Vector1@1999 "Vector(Vector(-1, -2, -3, -4), Vector(-5, -6, -7, -8)...
arg$2 = 0
b = VectorBuilder@2071 "VectorBuilder(len1=1, lenRest=0, offset=0, depth=1)"
i = 1
> this = Vector$@2072

```

Figura 49: Debug SumMatriz

Resultado final La función finaliza tras completar todas las iteraciones, como se observa en las siguientes imagenes. La matriz resultante **m3** está compuesta únicamente por ceros, lo cual confirma que la operación se realizó correctamente.

```

Local
m1$3 = Vector1@1998 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
> prefix1 = Object[4]@2004
m2$1 = Vector1@1999 "Vector(Vector(-1, -2, -3, -4), Vector(-5, -6, -7, -8), Vector(-9, -10, -11, -12), Vector(-13, ...
> prefix1 = Object[4]@2003
i = 0
j = 1

```

Figura 50: Debug SumMatriz

```

Local
->$anonfun$sumMatriz$1() = 0
> v1 = Integer@2038 "0"
> v2 = Integer@2100 "1"
> this = Taller3$$Lambda$158/0x000002bb4d18a778@2039 "taller.Taller3$$Lambda$158/0x000002bb4d18a778@361a08d0"

```

Figura 51: Debug SumMatriz

```

Local
> ->apply() = Integer@2038 "0"
f$1 = Taller3$$Lambda$158/0x000002bb4d18a778@2039 "taller.Taller3$$Lambda$158/0x000002bb4d18a778@361a08d0"
> arg$1 = Vector1@1998 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, ...
> arg$2 = Vector1@1999 "Vector(Vector(-1, -2, -3, -4), Vector(-5, -6, -7, -8), Vector(-9, -10, -11, -12), Ve...
i1$1 = 0
x$1 = 1

```

Figura 52: Debug SumMatriz

```

✓ Local
> ->$anonfun$tabulate$2() = Integer@2038 "0"
  n = 4
> f = IterableFactory$$Lambda$160/0x000002bb4d18b108@2070 "scala.collection.Iter
> b = VectorBuilder@2071 "VectorBuilder(len1=1, lenRest=0, offset=0, depth=1)"
  i = 1
> this = Vector$@2072

```

Figura 53: Debug SumMatriz

Conclusión La función `sumMatriz` permite sumar correctamente dos matrices cuadradas `m1` y `m2` de dimensión $n \times n$, generando una nueva matriz resultante. El test verifica que el resultado sea igual al valor esperado, y el `assert` pasa sin errores.

1.3.3. Función para multiplicar matrices recursivamente (*secuencial*)

La función `multMatrizRec` emplea un enfoque de "divide y vencerás". El metodo lo que hace es dividir las matrices grandes en partes más pequeñas para facilitar su multiplicación. Esto se ejecuta de la siguiente manera:

1. **Tamaño de la matriz:** Si el tamaño n es 1 (caso base), multiplica directamente los valores correspondientes de las matrices y devuelve el resultado.
2. **División en submatrices:** Para matrices más grandes, divide las matrices `m1` y `m2` en 4 submatrices de tamaño $\frac{n}{2} \times \frac{n}{2}$ cada una:

$$m1 = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \text{ y } m2 = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

3. **Cálculo de submatrices del resultado:** Usa las submatrices para calcular $C_{11}, C_{12}, C_{21}, C_{22}$ mediante sumas y multiplicaciones recursivas.
4. **Combinación:** Integra las submatrices calculadas para formar la matriz resultante C .

A continuación se mostrará un ejemplo de la función `multMatrizRec`.

Ejemplo de prueba Multiplicación Recursiva Secuencial Identidad: Para ilustrar cómo opera esta función, decidimos mostrar la multiplicación recursiva de una matriz identidad. Así que definimos las siguientes matrices:

```

1  val matrizIdentidad = Vector(
2      Vector(1, 0, 0, 0),
3      Vector(0, 1, 0, 0),
4      Vector(0, 0, 1, 0),
5      Vector(0, 0, 0, 1)
6  )

```

```

1  val matriz1 = Vector(
2      Vector(1, 2, 3, 4),
3      Vector(5, 6, 7, 8),
4      Vector(9, 10, 11, 12),
5      Vector(13, 14, 15, 16)
6  )

```

Este es el test que comprueba el correcto funcionamiento de la función, definiendo el resultado correcto esperado. que en este caso es la misma matriz 1.

```

1  test("Multiplicación recursiva secuencial identidad") {
2      assert(taller3.multMatrizRec(matriz1, matrizIdentidad) == matriz1)
3  }

```

La función empieza recibiendo dos matrices $m1$ y $m2$, siendo $m1$ la matriz1 definida anteriormente y $m2$ una matriz identidad.

```

Local
> m1 = Vector1@2001 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
> m2 = Vector1@2002 "Vector(Vector(1, 0, 0, 0), Vector(0, 1, 0, 0), Vector(0, 0, 1, 0), Vector(0, 0, 0, 1))"
> this = Taller3@2003

```

Figura 54: Debug MultMatrizRecSec

Aquí se calcula el tamaño de la matriz ($m1$) es igual 4 debido a que las matrices que se están multiplicando son 4×4 , verificando que no sea igual a 1 (el caso base). El algoritmo pasa a dividir las matrices.

```

Local
> m1 = Vector1@2001 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
> m2 = Vector1@2002 "Vector(Vector(1, 0, 0, 0), Vector(0, 1, 0, 0), Vector(0, 0, 1, 0), Vector(0, 0, 0, 1))"
  n = 4
> this = Taller3@2003

```

Figura 55: Debug MultMatrizRecSec

Ya que el tamaño de la matriz no es igual a 1 ($n \neq 1$) el programa divide las matrices en 4 submatrices de tamaño $\frac{n}{2} \times \frac{n}{2}$, en este caso en tamaño 2×2 . Aquí se prepara la extracción de $A_{11}, A_{12}, A_{21}, A_{22}$ y sus correspondientes en $m2$:

```

Local
> m1 = Vector1@2001 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
> m2 = Vector1@2002 "Vector(Vector(1, 0, 0, 0), Vector(0, 1, 0, 0), Vector(0, 0, 1, 0), Vector(0, 0, 0, 1))"
  n = 4
  half = 2
> this = Taller3@2003

```

Figura 56: Debug MultMatrizRecSec

El programa usando la función *subMatriz* empieza a obtener las submatrices $A_{11}, A_{12}, A_{21}, A_{22}$ de $m1$ pasando por cada posición i y j hasta llegar al maximo.

```

Local
> m = Vector1@2001 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
  i = 0
  j = 0
  l = 2
> this = Taller3@2003

```

Figura 57: Debug MultMatrizRecSec

La función continúa recorriendo $m1$ fila por fila y columna por columna para identificar las regiones correspondientes a cada submatriz. Aquí se empieza a trabajar con submatrices más pequeñas que tendrán un tamaño de 2×2 .

```
> m$1 = Vector1@2001 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
i$2 = 2
j$2 = 2
x = 1
y = 1
```

Figura 58: Debug MultMatrizRecSec

Se muestran las submatrices $A_{11}, A_{12}, A_{21}, A_{22}$ y sus correspondientes valores en las posiciones $A_{11}, A_{12}, A_{21}, A_{22}$ que se extrajeron de $m1$.

```
< Local
> m1 = Vector1@2001 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
> m2 = Vector1@2002 "Vector(Vector(1, 0, 0, 0), Vector(0, 1, 0, 0), Vector(0, 0, 1, 0), Vector(0, 0, 0, 1))"
n = 4
half = 2
> a11 = Vector1@2049 "Vector(Vector(1, 2), Vector(5, 6))"
> a12 = Vector1@2050 "Vector(Vector(3, 4), Vector(7, 8))"
> a21 = Vector1@2051 "Vector(Vector(9, 10), Vector(13, 14))"
> a22 = Vector1@2052 "Vector(Vector(11, 12), Vector(15, 16))"
> this = Taller3@2003
```

Figura 59: Debug MultMatrizRecSec

El programa usando la función *subMatriz* empieza a obtener las submatrices $B_{11}, B_{12}, B_{21}, B_{22}$ de $m2$ pasando por cada posición i y j hasta llegar al maximo.

```
< Local
> m = Vector1@2002 "Vector(Vector(1, 0, 0, 0), Vector(0, 1, 0, 0), Vector(0, 0, 1, 0), Vector(0, 0, 0, 1))"
i = 0
j = 0
l = 2
> this = Taller3@2003
```

Figura 60: Debug MultMatrizRecSec

```
< Local
> m$1 = Vector1@2002 "Vector(Vector(1, 0, 0, 0), Vector(0, 1, 0, 0), Vector(0, 0, 1, 0), Vector(0, 0, 0, 1))"
i$2 = 0
j$2 = 0
x = 0
y = 0
```

Figura 61: Debug MultMatrizRecSec

La función continúa recorriendo $m2$ fila por fila y columna por columna para identificar las regiones correspondientes a cada submatriz. Aquí se empieza a trabajar con submatrices más pequeñas que tendrán un tamaño de 2×2 .

```

Local
> m$1 = Vector1@2002 "Vector(Vector(1, 0, 0, 0), Vector(0, 1, 0, 0), Vector(0, 0, 1, 0), Vector(0, 0, 0, 1))"
i$2 = 2
j$2 = 2
x = 1
y = 1

```

Figura 62: Debug MultMatrizRecSec

Al llegar a este proceso se muestran las submatrices $A_{11}, A_{12}, A_{21}, A_{22}$ con sus correspondientes valores en las posiciones $A_{11}, A_{12}, A_{21}, A_{22}$ que se extrajeron de $m1$ Y $B_{11}, B_{12}, B_{21}, B_{22}$ sus correspondientes valores en las posiciones $B_{11}, B_{12}, B_{21}, B_{22}$ que se extrajeron de $m2$.

```

Local
> m1 = Vector1@2001 "Vector(Vector(1, 2, 3, 4), Vector(5, 6, 7, 8), Vector(9, 10, 11, 12), Vector(13, 14, 15, 16))"
> m2 = Vector1@2002 "Vector(Vector(1, 0, 0, 0), Vector(0, 1, 0, 0), Vector(0, 0, 1, 0), Vector(0, 0, 0, 1))"
n = 4
half = 2
> a11 = Vector1@2049 "Vector(Vector(1, 2), Vector(5, 6))"
> a12 = Vector1@2050 "Vector(Vector(3, 4), Vector(7, 8))"
> a21 = Vector1@2051 "Vector(Vector(9, 10), Vector(13, 14))"
> a22 = Vector1@2052 "Vector(Vector(11, 12), Vector(15, 16))"
> b11 = Vector1@2097 "Vector(Vector(1, 0), Vector(0, 1))"
> b12 = Vector1@2098 "Vector(Vector(0, 0), Vector(0, 0))"
> b21 = Vector1@2099 "Vector(Vector(0, 0), Vector(0, 0))"
> b22 = Vector1@2100 "Vector(Vector(1, 0), Vector(0, 1))"
> this = Taller3@2003

```

Figura 63: Debug MultMatrizRecSec

Se muestran las submatrices resultantes de la selección anterior con tamaño 2×2

```

Local
> m1 = Vector1@2049 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2097 "Vector(Vector(1, 0), Vector(0, 1))"
> this = Taller3@2003

```

Figura 64: Debug MultMatrizRecSec

La función vuelve a contar el tamaño de la matriz esto asegura que el algoritmo siga dividiendo hasta llegar al caso base ($n = 1$). y como sigue siendo diferente de 1 ($n \neq 1$) se repite el proceso de división recursiva hasta llegar a una submatriz 1×1 .

```

> m1 = Vector1@2049 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2097 "Vector(Vector(1, 0), Vector(0, 1))"
  n = 2
> this = Taller3@2003

```

Figura 65: Debug MultMatrizRecSec

Este proceso de división recursiva continúa hasta que el tamaño de las submatrices sea 1. En ese momento Se llega al caso base ($n = 1$) donde el tamaño de las submatrices es 1×1 . Por ejemplo:

$$A_{11} = [1], \quad A_{12} = [2], \quad A_{21} = [5], \quad A_{22} = [6]$$

$$B_{11} = [1], \quad B_{12} = [0], \quad B_{21} = [0], \quad B_{22} = [1]$$

```

> m1 = Vector1@2049 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2097 "Vector(Vector(1, 0), Vector(0, 1))"
  n = 2
  half = 1
> this = Taller3@2003

```

Figura 66: Debug MultMatrizRecSec

El programa empieza a obtener las submatrices de $m1$ y $m2$ pasando por cada posición i y j hasta llegar al máximo para definir la nueva submatriz tamaño 1. en este punto se multiplica directamente.

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

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

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

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

Ahora, cada submatriz será tratada como una matriz independiente y enviada de forma recursiva a la función `multMatrizRec`. En la siguiente imagen se muestra el primer resultado para C_{11} de la primera submatriz resultante de cuatro tamaño 1 .

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

$$C_{11} = [(1 \cdot 1) + (2 \cdot 0)] = 1$$

$$C_{11} = 1$$


```

Local
> m1 = Vector1@2049 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2097 "Vector(Vector(1, 0), Vector(0, 1))"
  n = 2
  half = 1
> a11 = Vector1@2129 "Vector(Vector(1))"
> a12 = Vector1@2130 "Vector(Vector(2))"
> a21 = Vector1@2131 "Vector(Vector(5))"
> a22 = Vector1@2132 "Vector(Vector(6))"
> b11 = Vector1@2146 "Vector(Vector(1))"
> b12 = Vector1@2147 "Vector(Vector(0))"
> b21 = Vector1@2148 "Vector(Vector(0))"
> b22 = Vector1@2149 "Vector(Vector(1))"
> c11 = Vector1@2178 "Vector(Vector(1))"
> this = Taller3@2003

```

Figura 67: Debug MultMatrizRecSec

Este mismo proceso se realiza en la siguiente imagen donde se muestra el segundo resultado para el resultante de C_{12} .

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

$$C_{12} = [(1 \cdot 0) + (2 \cdot 1)] = 2$$

$$C_{12} = 2$$

```

Local
> m1 = Vector1@2049 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2097 "Vector(Vector(1, 0), Vector(0, 1))"
  n = 2
  half = 1
> a11 = Vector1@2129 "Vector(Vector(1))"
> a12 = Vector1@2130 "Vector(Vector(2))"
> a21 = Vector1@2131 "Vector(Vector(5))"
> a22 = Vector1@2132 "Vector(Vector(6))"
> b11 = Vector1@2146 "Vector(Vector(1))"
> b12 = Vector1@2147 "Vector(Vector(0))"
> b21 = Vector1@2148 "Vector(Vector(0))"
> b22 = Vector1@2149 "Vector(Vector(1))"
> c11 = Vector1@2178 "Vector(Vector(1))"
> c12 = Vector1@2208 "Vector(Vector(2))"
> this = Taller3@2003

```

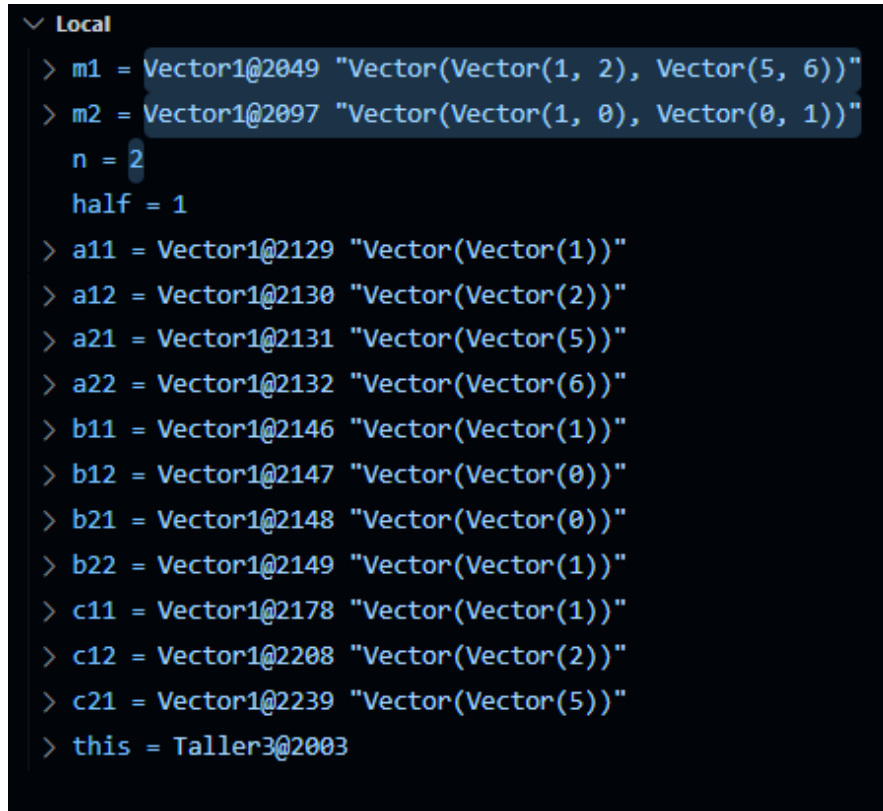
Figura 68: Debug MultMatrizRecSec

Seguimos repitiendo el mismo proceso para encontrar el siguiente valor. La siguiente imagen muestra el tercer resultado para el resultante de C_{21} .

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

$$C_{21} = [(5 \cdot 1) + (6 \cdot 0)] = 5$$

$$C_{21} = 5$$



```

Local
> m1 = Vector1@2049 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2097 "Vector(Vector(1, 0), Vector(0, 1))"
  n = 2
  half = 1
> a11 = Vector1@2129 "Vector(Vector(1))"
> a12 = Vector1@2130 "Vector(Vector(2))"
> a21 = Vector1@2131 "Vector(Vector(5))"
> a22 = Vector1@2132 "Vector(Vector(6))"
> b11 = Vector1@2146 "Vector(Vector(1))"
> b12 = Vector1@2147 "Vector(Vector(0))"
> b21 = Vector1@2148 "Vector(Vector(0))"
> b22 = Vector1@2149 "Vector(Vector(1))"
> c11 = Vector1@2178 "Vector(Vector(1))"
> c12 = Vector1@2208 "Vector(Vector(2))"
> c21 = Vector1@2239 "Vector(Vector(5))"
> this = Taller3@2003

```

Figura 69: Debug MultMatrizRecSec

Finalmente se realiza la primera submatriz resultante de cuatro de tamaño 1 repitiendo el mismo proceso. La siguiente imagen muestra el tercer resultado para el resultante de C_{22} .

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

$$C_{22} = [(5 \cdot 0) + (6 \cdot 1)] = 6$$

$$C_{22} = 6$$

```
Local
> m1 = Vector1@2049 "Vector(Vector(1, 2), Vector(5, 6))"
> m2 = Vector1@2097 "Vector(Vector(1, 0), Vector(0, 1))"
  n = 2
  half = 1
> a11 = Vector1@2129 "Vector(Vector(1))"
> a12 = Vector1@2130 "Vector(Vector(2))"
> a21 = Vector1@2131 "Vector(Vector(5))"
> a22 = Vector1@2132 "Vector(Vector(6))"
> b11 = Vector1@2146 "Vector(Vector(1))"
> b12 = Vector1@2147 "Vector(Vector(0))"
> b21 = Vector1@2148 "Vector(Vector(0))"
> b22 = Vector1@2149 "Vector(Vector(1))"
> c11 = Vector1@2178 "Vector(Vector(1))"
> c12 = Vector1@2208 "Vector(Vector(2))"
> c21 = Vector1@2239 "Vector(Vector(5))"
> c22 = Vector1@2271 "Vector(Vector(6))"
> this = Taller3@2003
```

Figura 70: Debug MultMatrizRecSec

- A continuación se muestra la primera submatriz de cuatro de tamaño 1 resultante

```
Local
  half$1 = 1
> c11$1 = Vector1@2178 "Vector(Vector(1))"
> c12$1 = Vector1@2208 "Vector(Vector(2))"
> c21$1 = Vector1@2239 "Vector(Vector(5))"
> c22$1 = Vector1@2271 "Vector(Vector(6))"
  i = 1
  j = 1
```

Figura 71: Debug MultMatrizRecSec

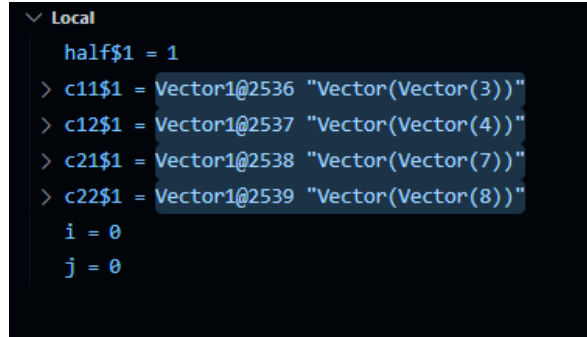
Una vez resueltas todas las multiplicaciones, los resultados se combinan según las formulas dadas anteriormente para obtener las submatrices del resultado final. Por último entramos a los condicionales, en donde se evalúa uno a uno si cumple o no la condición.

```

1 // Construir la matriz resultante
2 Vector.tabulate(n, n) { (i, j) => // Se genera una matriz de tamaño n x n
3   // Se combinan las submatrices en la matriz resultante usando Vector.tabulate
4   if (i < half && j < half) c11(i)(j)
5   else if (i < half) c12(i)(j - half)
6   else if (j < half) c21(i - half)(j)
7   else c22(i - half)(j - half)
8 }

```

El proceso realizado anteriormente es el mismo para las siguientes submatrices de tamaño 1. La siguiente imagen muestra la segunda submatriz de cuatro de tamaño 1 resultante.

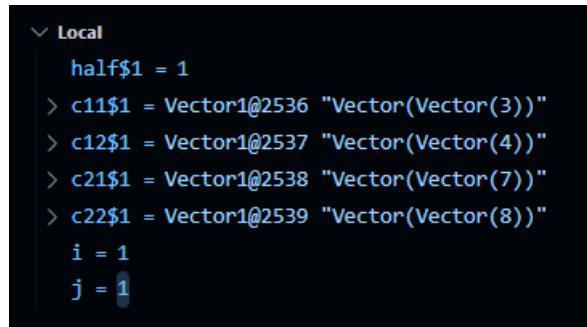


```

Local
half$1 = 1
> c11$1 = Vector1@2536 "Vector(Vector(3))"
> c12$1 = Vector1@2537 "Vector(Vector(4))"
> c21$1 = Vector1@2538 "Vector(Vector(7))"
> c22$1 = Vector1@2539 "Vector(Vector(8))"
i = 0
j = 0

```

Figura 72: Debug MultMatrizRecSec



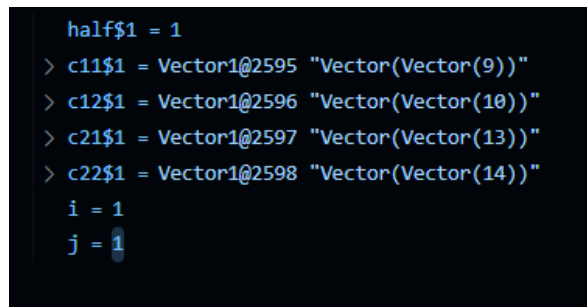
```

Local
half$1 = 1
> c11$1 = Vector1@2536 "Vector(Vector(3))"
> c12$1 = Vector1@2537 "Vector(Vector(4))"
> c21$1 = Vector1@2538 "Vector(Vector(7))"
> c22$1 = Vector1@2539 "Vector(Vector(8))"
i = 1
j = 1

```

Figura 73: Debug MultMatrizRecSec

Para la siguiente imagen se muestra la tercera submatriz de cuatro de tamaño 1 resultante.



```

Local
half$1 = 1
> c11$1 = Vector1@2595 "Vector(Vector(9))"
> c12$1 = Vector1@2596 "Vector(Vector(10))"
> c21$1 = Vector1@2597 "Vector(Vector(13))"
> c22$1 = Vector1@2598 "Vector(Vector(14))"
i = 1
j = 1

```

Figura 74: Debug MultMatrizRecSec

Finalmente para la siguiente imagen se muestra la cuarta submatriz de cuatro de tamaño 1 resultante.

```

Local
  half$1 = 1
  > c11$1 = Vector1@2784 "Vector(Vector(11))"
  > c12$1 = Vector1@2785 "Vector(Vector(12))"
  > c21$1 = Vector1@2786 "Vector(Vector(15))"
  > c22$1 = Vector1@2787 "Vector(Vector(16))"
  i = 1
  j = 1

```

Figura 75: Debug MultMatrizRecSec

Cuando este termina calculamos las submatrices del resultado :

```

half$1 = 2
> c11$1 = Vector1@2834 "Vector(Vector(1, 2), Vector(5, 6))"
> c12$1 = Vector1@2835 "Vector(Vector(3, 4), Vector(7, 8))"
> c21$1 = Vector1@2836 "Vector(Vector(9, 10), Vector(13, 14))"
> c22$1 = Vector1@2837 "Vector(Vector(11, 12), Vector(15, 16))"
i = 3
j = 3

```

Figura 76: Debug MultMatrizRecSec

Submatriz C_{11}

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 12 & 6 \end{bmatrix}$$

Submatriz C_{12}

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 10 & 5 \\ 24 & 10 \end{bmatrix}$$

Submatriz C_{21}

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} = \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 20 & 10 \\ 28 & 14 \end{bmatrix}$$

Submatriz C_{22}

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} = \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 26 & 13 \\ 36 & 16 \end{bmatrix}$$

Matriz resultante C : Una vez calculadas todas las 4 submatrices $C_{11}, C_{12}, C_{21}, C_{22}$, se ensamblan, es decir, se combinan para formar la matriz final C . El algoritmo termina y devuelve el resultado.

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 4 & 2 & 10 & 5 \\ 12 & 6 & 24 & 10 \\ 20 & 10 & 26 & 13 \\ 28 & 14 & 36 & 16 \end{bmatrix}$$

1.3.4. Función para multiplicar matrices recursivamente (*paralela*)

Similar al proceso secuencial Se reciben las matrices m_1 y m_2 y se calcula su tamaño ($n = 4$)., el tamaño de las matrices es calculado, y si $n \neq 1$, se dividen en submatrices A_{ij} y B_{ij} .

```
Local
  →multMatrizRecPar$default$3() = 64
  ✓ $this = MultMatrizRecParTest@2013 "MultMatriz
  > matriz1 = Vector1@2015 "Vector(Vector(1, 2,
  > matriz2 = Vector1@2016 "Vector(Vector(4, 3,
  > matrizCero = Vector1@2017 "Vector(Vector(0,
  > matrizGrande1 = Vector1@2018 "Vector(Vector(
  > matrizGrande2 = Vector1@2019 "Vector(Vector(
  ✓ matrizIdentidad = Vector1@2020 "Vector(Vector
  ✓ prefix1 = Object[4]@2035
    > 0 = Vector1@2036 "Vector(1, 0, 0, 0)"
    > 1 = Vector1@2037 "Vector(0, 1, 0, 0)"
    > 2 = Vector1@2038 "Vector(0, 0, 1, 0)"
    > 3 = Vector1@2039 "Vector(0, 0, 0, 1)"
  NoArgTest$module = null
```

Figura 77: Debug MultRecParalela

Las multiplicaciones de submatrices ($A_{ij} \cdot B_{ij}$) y las sumas ($A_{ij} + B_{ij}$) se realizan en paralelo utilizando la función `parallel`. Cada cálculo se delega a una tarea independiente. El programa comienza a dividir m_1 y m_2 en sus respectivas submatrices ($A_{11}, A_{12}, A_{21}, A_{22}$ y $B_{11}, B_{12}, B_{21}, B_{22}$). Aquí es donde se introduce la paralelización: Cada cálculo de C_{ij} (como $C_{11}, C_{12}, C_{21}, C_{22}$) se delega a una tarea independiente que se ejecuta al mismo tiempo.

```
Local
  ✓ m1 = Vector1@2015 "Vector(Vector(1, 2, 3, 4
  ✓ prefix1 = Object[4]@2046
    > 0 = Vector1@2047 "Vector(1, 2, 3, 4)"
    > 1 = Vector1@2048 "Vector(5, 6, 7, 8)"
    > 2 = Vector1@2049 "Vector(9, 10, 11, 12)"
    > 3 = Vector1@2050 "Vector(13, 14, 15, 16)"
  ✓ m2 = Vector1@2020 "Vector(Vector(1, 0, 0, 0
  ✓ prefix1 = Object[4]@2035
    > 0 = Vector1@2036 "Vector(1, 0, 0, 0)"
    > 1 = Vector1@2037 "Vector(0, 1, 0, 0)"
    > 2 = Vector1@2038 "Vector(0, 0, 1, 0)"
    > 3 = Vector1@2039 "Vector(0, 0, 0, 1)"
  umbral = 64
  ✓ this = Taller3@2008
  Class has no fields
```

Figura 78: Debug MultRecParalela

Muestra cómo las tareas paralelas están ejecutando los cálculos de las submatrices simultáneamente. Por ejemplo: Una tarea calcula C_{11} , mientras otra calcula C_{12} , y así sucesivamente.

Se muestran las submatrices C_{ij} calculadas en paralelo. Cada tarea ha procesado una porción de los datos, y las tareas sincronizan al finalizar.

```

Local
  m1 = Vector1@2015 "Vector(Vector(1, 2, 3, 4),
  prefix1 = Object[4]@2046
    > 0 = Vector1@2047 "Vector(1, 2, 3, 4)"
    > 1 = Vector1@2048 "Vector(5, 6, 7, 8)"
    > 2 = Vector1@2049 "Vector(9, 10, 11, 12)"
    > 3 = Vector1@2050 "Vector(13, 14, 15, 16)"
  m2 = Vector1@2020 "Vector(Vector(1, 0, 0, 0),
  prefix1 = Object[4]@2035
    > 0 = Vector1@2036 "Vector(1, 0, 0, 0)"
    > 1 = Vector1@2037 "Vector(0, 1, 0, 0)"
    > 2 = Vector1@2038 "Vector(0, 0, 1, 0)"
    > 3 = Vector1@2039 "Vector(0, 0, 0, 1)"
  umbral = 64
  n = 4
  this = Taller3@2008
    Class has no fields

```

Figura 79: Debug MultRecParalela

Las submatrices calculadas $C_{11}, C_{12}, C_{21}, C_{22}$ son combinadas en la matriz final igual que en el caso secuencial; las submatrices se colocan en sus posiciones correspondientes para formar C . Este paso no se paraleliza y ocurre de manera secuencial.

```

Local
  multMatrizRecPar() = Vector1@2278 "Vector(Vector(1, 2, 3, 4),
  prefix1 = Object[4]@2292
    > 0 = Vector1@2293 "Vector(1, 2, 3, 4)"
    > 1 = Vector1@2294 "Vector(5, 6, 7, 8)"
    > 2 = Vector1@2295 "Vector(9, 10, 11, 12)"
    > 3 = Vector1@2296 "Vector(13, 14, 15, 16)"
  $this = MultMatrizRecParTest@2013 "MultMatrizRecParTest"
    > matriz1 = Vector1@2015 "Vector(Vector(1, 2, 3, 4), Vector
    > matriz2 = Vector1@2016 "Vector(Vector(4, 3, 2, 1), Vector
    > matrizCero = Vector1@2017 "Vector(Vector(0, 0, 0, 0), Vec
    > matrizGrande1 = Vector1@2018 "Vector(Vector(1, 1, 1, 1, 1
    > matrizGrande2 = Vector1@2019 "Vector(Vector(2, 2, 2, 2, 2
    > matrizIdentidad = Vector1@2020 "Vector(Vector(1, 0, 0, 0)
  NoArgTest$module = null

```

Figura 80: Debug MultRecParalela

El algoritmo verifica que todos los cálculos paralelos se hayan completado y devuelve la matriz final. Aquí se observa el balance de carga entre las tareas.

```

v Local
> ->$anonfun$new$1() = Succeeded$@2024 "Succeeded"
> f = MultMatrizRecParTest$$Lambda$94/0x00000225a010f
> this = OutcomeOf$@2307

```

Figura 81: Debug MultRecParalela

Conclusión de las multiplicaciones recursivas Secuencial. Permite dividir matrices y calcula de forma recursiva, trabajando en un paso a la vez y ensambla resultados en pasos secuenciales. .

Paralela. Usa tareas concurrentes para calcular las submatrices simultáneamente, aprovechando múltiples núcleos de procesamiento para acelerar el cálculo. Cada figura representa un paso del proceso para dividir, calcular y combinar matrices, y la versión paralela optimiza el tiempo al realizar varios cálculos al mismo tiempo.

El algoritmo recursivo, especialmente en su versión paralela, demuestra ser una herramienta poderosa para la multiplicación de matrices grandes. Aunque su implementación es más compleja, la aceleración lograda en comparación con la versión secuencial justifica su uso en sistemas con recursos paralelos disponibles. Sin embargo, es crucial balancear el umbral de paralelización y tener en cuenta los costos asociados para garantizar un rendimiento óptimo.

1.4. Funciones para Multiplicación de matrices usando el algoritmo de Strassen

1.4.1. Función para restar matrices

Esta función permite restar dos matrices cuadradas de la misma dimensión y devuelve la matriz $A - B$. Para probar su correcto funcionamiento se definen las matrices m1 y m2 de tamaño 8×8 con valores grandes aleatorios y se define la matriz con el resultado correcto de la resta de ambas matrices.

```

1  test("Resta de matrices 8x8 con valores grandes y pequeños aleatorios") {
2      val m1 = Vector (
3          Vector(724,110,791,656,74,544,604,862),
4          Vector(293,579,154,424,189,245,439,770),
5          Vector(805,81,338,909,782,112,774,189),
6          Vector(330,769,143,583,283,305,239,351),
7          Vector(446,360,978,847,22,92,690,421),
8          Vector(277,280,781,923,285,970,691,496),
9          Vector(584,230,801,750,443,536,693,510),
10         Vector(302,355,594,989,147,531,325,635)
11     )
12     val m2 = Vector (
13         Vector(81,689,916,886,425,881,626,376),
14         Vector(323,440,557,745,81,712,658,806),
15         Vector(330,357,35,978,614,344,497,800),
16         Vector(356,967,261,174,1,0,361,321),
17         Vector(334,865,288,44,463,952,484,472),
18         Vector(525,550,462,693,25,647,699,42),
19         Vector(797,703,761,604,396,983,280,629),
20         Vector(738,179,495,649,638,954,571,509)
21     )
22
23     val resultado = taller3.restMatriz(m1, m2)
24
25     val esperado = Vector (
26         Vector(643,-579,-125,-230,-351,-337,-22,486),
27         Vector(-30,139,-403,-321,108,-467,-219,-36),
28         Vector(475,-276,303,-69,168,-232,277,-611),
29         Vector(-26,-198,-118,409,282,305,-122,30),
30         Vector(112,-505,690,803,-441,-860,206,-51),

```



```

31         Vector(-248,-270,319,230,260,323,-8,454),
32         Vector(-213,-473,40,146,47,-447,413,-119),
33         Vector(-436,176,99,340,-491,-423,-246,126)
34     )
35
36
37     assert(resultado == esperado)
38
39 }

```

El programa empieza llamando a las matrices ya definidas (m1 y m2) para ejecutar la resta.

```

✓ Local
  ✓ m1 = Vector1@1981 "Vector(Vector(724, 110, 791, 656, 74, 544, 604, 8
  ✓ prefix1 = Object[8]@1991
    > 0 = Vector1@1992 "Vector(724, 110, 791, 656, 74, 544, 604, 862)"
    > 1 = Vector1@1993 "Vector(293, 579, 154, 424, 189, 245, 439, 770)"
    > 2 = Vector1@1994 "Vector(805, 81, 338, 909, 782, 112, 774, 189)"
    > 3 = Vector1@1995 "Vector(330, 769, 143, 583, 283, 305, 239, 351)"
    > 4 = Vector1@1996 "Vector(446, 360, 978, 847, 22, 92, 690, 421)"
    > 5 = Vector1@1997 "Vector(277, 280, 781, 923, 285, 970, 691, 496)"
    > 6 = Vector1@1998 "Vector(584, 230, 801, 750, 443, 536, 693, 510)"
    > 7 = Vector1@1999 "Vector(302, 355, 594, 989, 147, 531, 325, 635)"
  ✓ m2 = Vector1@1982 "Vector(Vector(81, 689, 916, 886, 425, 881, 626, 3
  ✓ prefix1 = Object[8]@1990
    > 0 = Vector1@2008 "Vector(81, 689, 916, 886, 425, 881, 626, 376)"
    > 1 = Vector1@2009 "Vector(323, 440, 557, 745, 81, 712, 658, 806)"
    > 2 = Vector1@2010 "Vector(330, 357, 35, 978, 614, 344, 497, 800)"
    > 3 = Vector1@2011 "Vector(356, 967, 261, 174, 1, 0, 361, 321)"
    > 4 = Vector1@2012 "Vector(334, 865, 288, 44, 463, 952, 484, 472)"
    > 5 = Vector1@2013 "Vector(525, 550, 462, 693, 25, 647, 699, 42)"
    > 6 = Vector1@2014 "Vector(797, 703, 761, 604, 396, 983, 280, 629)"
    > 7 = Vector1@2015 "Vector(738, 179, 495, 649, 638, 954, 571, 509)"

```

Figura 82: Debug RestaMatriz

Empieza a hacer la resta por orden de filas y luego columnas, es decir, primero evaluará todos los resultados de la resta entre la matriz m1 en la posición $i = 0$ y $j = 0$ y la matriz m2 en la misma posición.

```

✓ Local
  > m1$4 = Vector1@1981 "Vector(Vector(724, 110, 791, 656, 74, 544, 604,...
  > m2$2 = Vector1@1982 "Vector(Vector(81, 689, 916, 886, 425, 881, 626,...
    i = 0
    j = 0

```

Figura 83: Debug RestaMatriz

En estas imágenes se puede ver el resultado de la resta de las matrices '643' y cómo se almacena en la posición (0,0) en la nueva matriz resultante.

```

✓ Local
  →$anonfun$restaMatriz$1() = 643
  > v1 = Integer@2050 "0"
  > v2 = Integer@2050 "0"
  > this = Taller3$$Lambda$156/0x0000001

```

Figura 84: Debug RestaMatriz

```

✓ Local
  > →apply() = Integer@2071 "643"
  ✓ f$1 = Taller3$$Lambda$156/0x0000001a289180d00@2051 "taller.Taller3$$L...
  > arg$1 = Vector1@1981 "Vector(Vector(724, 110, 791, 656, 74, 544, 60...
  > arg$2 = Vector1@1982 "Vector(Vector(81, 689, 916, 886, 425, 881, 62...
  i1$1 = 0
  x$1 = 0

```

Figura 85: Debug RestaMatriz

```

✓ Local
  > →$anonfun$tabulate$2() = Integer@2071 "643"
  n = 8
  > f = IterableFactory$$Lambda$158/0x0000001a2891878a8@2077 "scala.colle...
  > b = VectorBuilder@2078 "VectorBuilder(len1=0, lenRest=0, offset=0, d...
  i = 0
  > this = Vector$@2079

```

Figura 86: Debug RestaMatriz

El programa continúa ahora para evaluar la resta de la misma fila pero en la columna 1 y se almacena en la posición (0,1).

```

✓ Local
  n = 8
  > f = IterableFactory$$Lambda$158/0x0000001a2891878a8@2077 "scala.colle...
  > b = VectorBuilder@2078 "VectorBuilder(len1=1, lenRest=0, offset=0, d...
  i = 1
  > this = Vector$@2079

```

Figura 87: Debug RestaMatriz

```

v Local
  ->$anonfun$restaMatriz$1() = -579
  > v1 = Integer@2050 "0"
  > v2 = Integer@2097 "1"
  > this = Taller3$$Lambda$156/0x000001a289180d00@2051 "taller.Taller3$$L..."

```

Figura 88: Debug RestaMatriz

```

v Local
  > ->apply() = Integer@2101 "-579"
  > f$1 = Taller3$$Lambda$156/0x000001a289180d00@2051 "taller.Taller3$$L..."
  i1$1 = 0
  x$1 = 1

```

Figura 89: Debug RestaMatriz

```

v Local
  > ->$anonfun$tabulate$2() = Integer@2101 "-579"
  n = 8
  > f = IterableFactory$$Lambda$158/0x000001a2891878a8@2077 "scala.colle..."
  > b = VectorBuilder@2078 "VectorBuilder(len1=1, lenRest=0, offset=0, d..."
  i = 1
  > this = Vector$@2079

```

Figura 90: Debug RestaMatriz

```

v Local
  n = 8
  > f = IterableFactory$$Lambda$158/0x000001a2891878a8@2077 "scala.colle..."
  > b = VectorBuilder@2078 "VectorBuilder(len1=2, lenRest=0, offset=0, d..."
  i = 1
  > this = Vector$@2079

```

Figura 91: Debug RestaMatriz

La función continua evaluando la resta en todas las posiciones de la matriz y organizandolas en una nueva matriz. Hasta llegar a las posiciones finales (7,7).

```

v Local
  n = 8
  > f = IterableFactory$$Lambda$158/0x000001a2891878a8@2077 "scala.colle..."
  > b = VectorBuilder@2078 "VectorBuilder(len1=2, lenRest=0, offset=0, d..."
  i = 2
  > this = Vector$@2079

```

Figura 92: Debug RestaMatriz

```

▼ Local
> m1$4 = Vector1@1981 "Vector(Vector(724, 110, 791, 656, 74, 544, 604,...
> m2$2 = Vector1@1982 "Vector(Vector(81, 689, 916, 886, 425, 881, 626,...
  i = 0
  j = 2

```

Figura 93: Debug RestaMatriz

```

▼ Local
→$anonfun$restaMatriz$1() = -125
> v1 = Integer@2050 "0"
> v2 = Integer@2115 "2"
> this = Taller3$$Lambda$156/0x000001a289180d00@2051 "taller.Taller3$$L...

```

Figura 94: Debug RestaMatriz

```

▼ Local
> →apply() = Integer@2119 "-125"
> f$1 = Taller3$$Lambda$156/0x000001a289180d00@2051 "taller.Taller3$$L...
  i1$1 = 0
  x$1 = 2

```

Figura 95: Debug RestaMatriz

```

▼ Local
  n = 8
> f = IterableFactory$$Lambda$158/0x000001a2891878a8@2077 "scala.colle...
> b = VectorBuilder@2078 "VectorBuilder(len1=3, lenRest=0, offset=0, d...
  i = 3
> this = Vector$@2079

```

Figura 96: Debug RestaMatriz

```

▼ Local
→$anonfun$restaMatriz$1() = -230
> v1 = Integer@2050 "0"
> v2 = Integer@2150 "3"
> this = Taller3$$Lambda$156/0x000001a289180d00@2051 "taller.Taller3$$L...

```

Figura 97: Debug RestaMatriz

Se calcula la resta de las posiciones finales de la fila $i = 7$ y todas las columnas.

```

Local
  ->$anonfun$restaMatriz$1() = 126
  > v1 = Integer@2215 "7"
  > v2 = Integer@2215 "7"
  > this = Taller3$$Lambda$156/0x000001a289180d00@2051 "taller.Taller3$..."

```

Figura 98: Debug RestaMatriz

```

Local
  ->tabulate() = Vector1@2233 "Vector(-436, 176, 99, 340, -491, -423, ..."
  prefix1 = Object[8]@2236
    > 0 = Integer@2237 "-436"
    > 1 = Integer@2238 "176"
    > 2 = Integer@2239 "99"
    > 3 = Integer@2240 "340"
    > 4 = Integer@2241 "-491"
    > 5 = Integer@2242 "-423"
    > 6 = Integer@2243 "-246"
    > 7 = Integer@2219 "126"
  > $this = Vector$@2079
    n2$5 = 8
  > f$1 = Taller3$$Lambda$156/0x000001a289180d00@2051 "taller.Taller3$L..."
    i1 = 7

```

Figura 99: Debug RestaMatriz

El test evalúa todas las filas y columnas de la matriz resultante con la matriz esperada ya definida en la función.

```

Local
  ->tabulate() = Vector1@2265 "Vector(Vector(643, -579, -125, -230, -351, -337, -2..."
  prefix1 = Object[8]@2268
    > 0 = Vector1@2269 "Vector(643, -579, -125, -230, -351, -337, -22, 486)"
    > 1 = Vector1@2270 "Vector(-30, 139, -403, -321, 108, -467, -219, -36)"
    > 2 = Vector1@2271 "Vector(475, -276, 303, -69, 168, -232, 277, -611)"
    > 3 = Vector1@2272 "Vector(-26, -198, -118, 409, 282, 305, -122, 30)"
    > 4 = Vector1@2273 "Vector(112, -505, 690, 803, -441, -860, 206, -51)"
    > 5 = Vector1@2274 "Vector(-248, -270, 319, 230, 260, 323, -8, 454)"
    > 6 = Vector1@2275 "Vector(-213, -473, 40, 146, 47, -447, 413, -119)"
    > 7 = Vector1@2233 "Vector(-436, 176, 99, 340, -491, -423, -246, 126)"
  n1 = 8
  n2 = 8
  > f = Taller3$$Lambda$156/0x000001a289180d00@2051 "taller.Taller3$$Lambda$156/0x00..."
  > this = Vector$@2079

```

Figura 100: Debug RestaMatriz

El test se completa exitosamente.

```

Local
> ->$anonfun$new$1() = Succeeded$@2436 "Succeeded"
> f = RestaMatrizTest$$Lambda$89/0x000001a289102ba8@2437 "taller.RestaMatrizTest$$..."
> this = OutcomeOf$@2438

```

Figura 101: Debug RestaMatriz

1.4.2. Función aplicando algoritmo de Strassen(*secuencial*)

La función `multStrassen` implementa el algoritmo de Strassen para la multiplicación de matrices de dimensiones $n \times n$. Esta versión de la multiplicación es más eficiente que la multiplicación de matrices tradicional. Se verifica el correcto funcionamiento de esta función con matrices que incluyen elementos negativos.

Ejemplo de prueba de la función Para probar la funcionalidad de `multStrassen`, se define una matriz con elementos negativos y se llama a la función para multiplicarla con una segunda matriz (`matriz2`). El resultado se imprime y se compara con la matriz esperada, que en este caso es `matrizNegativa`.

```

test("Multiplicación de matrices con elementos negativos") {
    val matrizNegativa = Vector(
        Vector(-2, -4, -6, -8),
        Vector(-1, -3, -5, -7),
        Vector(-9, -11, -13, -15),
        Vector(-10, -12, -14, -16)
    )

    val resultado = taller3.multStrassen(matrizNegativa, matriz2)
    println("Resultado de la multiplicación de matrices con elementos negativos:")
    println(resultado)
    assert(resultado == matrizNegativa)
}

```

Figura 102: Test seleccionado para el debug

```

val matriz2 = Vector(
    Vector(1, 0, 0, 0),
    Vector(0, 1, 0, 0),
    Vector(0, 0, 1, 0),
    Vector(0, 0, 0, 1)
)

```

Figura 103: Matriz 2

Debug de la función: Estado inicial de las matrices La función comienza con dos matrices de entrada: `matrizNegativa` y `matriz2` la cual es llamada de primero. Se observa que `matrizNegativa` contiene valores negativos y se espera que el resultado final sea una matriz con los mismos elementos.

```

v ∞ matriz2 = {Vector1@2793} size = 4
  v 0 = {Vector1@2799} size = 4
    > 0 = {Integer@2803} 1
    > 1 = {Integer@2804} 0
    > 2 = {Integer@2804} 0
    > 3 = {Integer@2804} 0
  v 1 = {Vector1@2800} size = 4
    > 0 = {Integer@2804} 0
    > 1 = {Integer@2803} 1
    > 2 = {Integer@2804} 0
    > 3 = {Integer@2804} 0
  > 2 = {Vector1@2801} size = 4
  > 3 = {Vector1@2802} size = 4

```

Figura 104: Primera llamada a pila, Matriz 2

```

> Ⓟ $this = {MultStrassenTest@2791} MultStrassenTest
v 0 = {Vector1@2822} size = 4
  > 0 = {Vector1@2825} size = 4
  > 1 = {Vector1@2826} size = 4
  > 2 = {Vector1@2827} size = 4
  > 3 = {Vector1@2828} size = 4
> ∞ taller3 = {Taller3@2792} taller.Taller3@6179e425
> ∞ matriz2 = {Vector1@2793} size = 4

```

Figura 105: Matriz negativa

Paso 1: Inicialización La función `multStrassen` toma dos matrices cuadradas `m1` y `m2` de tamaño $n \times n$ y verifica que ambas sean de tamaño $2^k \times 2^k$. Se obtiene el tamaño de la matriz n .




```

// Verificar que las matrices son cuadradas y tienen dimensiones iguales
val n = m1.length    m1: size = 4    n: 4
require(n == m2.length && (n & (n - 1)) == 0)    m2: size = 4    n: 4

```

Figura 106: Debug MultStrassen

```

>  this = {Taller3@2738} taller.Taller3@f316aeb
>  m1 = {Vector1@2739} size = 4
>  m2 = {Vector1@2740} size = 4
10
01 n = 4

```

Figura 107: Debug MultStrassen

Paso 2: Se calcula $\text{half} = n / 2$.




```

// Dividir las matrices en submatrices de tamaño n/2
val half = n / 2  half: 2    n: 4

```

Figura 108: Debug MultStrassen

```

>  this = {Taller3@2738} taller.Taller3@f316aeb
>  m1 = {Vector1@2739} size = 4
>  m2 = {Vector1@2740} size = 4
10
01 n = 4
10
01 half = 2

```

Figura 109: Debug MultStrassen

Paso 3: División de las matrices Se divide m1 en submatrices a11, a12, a21, a22. Cada submatriz tiene dimensiones $\text{half} \times \text{half}$.

```

val (a11, a12, a21, a22) = (  a21: size = 2    a12: size = 2    a11: size = 2    a22: size = 2
  subMatriz(m1, 0, 0, half),
  subMatriz(m1, 0, half, half),
  subMatriz(m1, half, 0, half),
  subMatriz(m1, half, half, half)  m1: size = 4    half: 2

```

Figura 110: Debug MultStrassen


```

a11 = {Vector1@2789} size = 2
  0 = {Vector1@2794} size = 2
    0 = {Integer@2772} 2
    1 = {Integer@2773} 4
  1 = {Vector1@2795} size = 2
    0 = {Integer@2776} 1
    1 = {Integer@2777} 3

```

Figura 111: Debug MultStrassen

Paso 4: Se divide m_2 en submatrices b_{11} , b_{12} , b_{21} , b_{22} . De igual manera cada submatriz tiene dimensiones $half \times half$.

```

val (b11, b12, b21, b22) = ( b22: size = 2    b21: size = 2    b12: size = 2    b11: size = 2
  subMatriz(m2, 0, 0, half),
  subMatriz(m2, 0, half, half),
  subMatriz(m2, half, 0, half),
  subMatriz(m2, half, half, half)  m2: size = 4    half: 2

```

Figura 112: Debug MultStrassen

```

b11 = {Vector1@2800} size = 2
  0 = {Vector1@2805} size = 2
    0 = {Integer@2776} 1
    1 = {Integer@2807} 0
  1 = {Vector1@2806} size = 2
    0 = {Integer@2807} 0
    1 = {Integer@2776} 1

```

Figura 113: Debug MultStrassen

Paso 5: Cálculo de las 7 multiplicaciones de Strassen Se calculan las 7 multiplicaciones recursivas:

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

```
val p1 = multStrassen(a11, restaMatriz(b12, b22))  b22: size = 2  b12: size = 2  a11: size = 2
```

Figura 114: Debug MultStrassen

Así sucesivamente con el resto de multiplicaciones de Strassen. Observamos que ahora los vectores tienen un tamaño = 2.

```
Ⓟ m1 = {Vector1@2765} size = 2
> 0 = {Vector1@2787} size = 2
> 1 = {Vector1@2788} size = 2
Ⓟ m2 = {Vector1@2781} size = 2
> 0 = {Vector1@2791} size = 2
> 1 = {Vector1@2792} size = 2
10
01 n = 2
10
01 half = 1
```

Figura 115: Debug MultStrassen

Paso 6: Combinación de resultados para submatrices Se calculan las submatrices de la matriz resultante c11, c12, c21, c22 usando las multiplicaciones anteriores:

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

Explicación: En cada submatriz, se realiza una multiplicación similar a la de la multiplicación tradicional pero con una estrategia de divide y vencerás. A continuación, se analiza cómo se calcula el producto de una submatriz específica, por ejemplo, la posición (0,0):

$$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10}$$

```
val (a11, a12, a21, a22) = ( a22: size = 1    a21: size = 1    a12: size = 1    a11: size = 1
    subMatriz(m1, 0, 0, half),
    subMatriz(m1, 0, half, half),
    subMatriz(m1, half, 0, half),
    subMatriz(m1, half, half, half) half: 1~ m1: size = 2
```

Figura 116: Debug MultStrassen

El resultado es el siguiente:

```
> ≡ a11 = {Vector1@2797} size = 1
> ≡ a12 = {Vector1@2798} size = 1
> ≡ a21 = {Vector1@2799} size = 1
> ≡ a22 = {Vector1@2800} size = 1
> ≡ b11 = {Vector1@2807} size = 1
> ≡ b12 = {Vector1@2808} size = 1
> ≡ b21 = {Vector1@2809} size = 1
> ≡ b22 = {Vector1@2810} size = 1
```

Figura 117: Debug MultStrassen

Paso 7: Reconstrucción de la matriz resultante La reconstrucción de la matriz resultante implica combinar las cuatro submatrices calculadas previamente: c_{11} , c_{12} , c_{21} , y c_{22} , para formar una única matriz de dimensión $n \times n$.

Para ello, se sigue el esquema de ensamblaje de las submatrices en las posiciones correspondientes:

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

Donde cada submatriz ocupa una de las cuatro cuadrantes de la matriz C . Esto se realiza iterando sobre los índices de las submatrices y asignando sus valores en las posiciones correspondientes dentro de C . Por ejemplo:

- Los elementos de c_{11} se copian en la esquina superior izquierda.
- Los elementos de c_{12} se copian en la esquina superior derecha.
- Los elementos de c_{21} se copian en la esquina inferior izquierda.

- Los elementos de c22 se copian en la esquina inferior derecha.

```
// Combinar los resultados para obtener las submatrices de la matriz resultante
val c11 = sumMatriz(restaMatriz(sumMatriz(p5, p4), p2), p6)  c11: size = 1    p6: size = 1
val c12 = sumMatriz(p1, p2)  c12: size = 1    p2: size = 1
val c21 = sumMatriz(p3, p4)  c21: size = 1    p4: size = 1
val c22 = restaMatriz(restaMatriz(sumMatriz(p5, p1), p3), p7)  p7: size = 1    p5: size = 1    p3: size = 1    p1: size = 1
```

Figura 118: Debug MultStrassen

```
a11 = {Vector1@2797} size = 1
a12 = {Vector1@2798} size = 1
a21 = {Vector1@2799} size = 1
a22 = {Vector1@2800} size = 1
b11 = {Vector1@2807} size = 1
b12 = {Vector1@2808} size = 1
b21 = {Vector1@2809} size = 1
b22 = {Vector1@2810} size = 1
p1 = {Vector1@2817} size = 1
p2 = {Vector1@2825} size = 1
p3 = {Vector1@2832} size = 1
p4 = {Vector1@2839} size = 1
p5 = {Vector1@2847} size = 1
p6 = {Vector1@2855} size = 1
p7 = {Vector1@2863} size = 1
c11 = {Vector1@2865} size = 1
c12 = {Vector1@2867} size = 1
c21 = {Vector1@2869} size = 1
```

Figura 119: Debug MultStrassen

Los productos parciales se combinan para obtener la matriz final

$$C_{00} = M_1 + M_4 - M_5 + M_7$$

Resultado Final La función `multStrassen` realiza la multiplicación de matrices correctamente, incluso con elementos negativos. El test confirma que la matriz resultante es igual a la matriz esperada `matrizNegativa`, después del todo proceso realizado.

```
✓ Tests passed: 1 of 1 test – 92 ms

Resultado de la multiplicación de matrices con elementos negativos:
Vector(Vector(-2, -4, -6, -8), Vector(-1, -3, -5, -7), Vector(-9, -11, -13, -15), Vector(-10, -12, -14, -16))
> Task :app:test
BUILD SUCCESSFUL in 20s
```

Figura 120: Debug MultStrassen

Conclusión: La función `multStrassen` permite realizar la multiplicación de matrices, incluida la multiplicación con elementos negativos, de manera eficiente y correcta. La prueba muestra que la matriz resultante es la esperada, y el `assert` pasa sin errores.

1.4.3. Función aplicando algoritmo de Strassen (*paralela*)

Ejemplo de prueba de la función: Para este análisis y probar la funcionalidad de la `multStrassenPar`, utilizaremos dos matrices de tamaño 2×2 : El resultado esperado de la multiplicación $A \times B$ es A , ya que B es la matriz identidad.

```
test("Multiplicación de una matriz con la matriz identidad") {
    val resultadoEsperado = matriz1
    assert(taller3.multStrassenPar(matriz1, matriz2) == resultadoEsperado)
}
```

Figura 121: Test seleccionado para el debug.

```
val matriz1 = Vector(
    Vector(1, 2),
    Vector(3, 4),
)

val matriz2 = Vector(
    Vector(1, 0),
    Vector(0, 1)
)
```

Figura 122: Matrices utilizadas

Inicialización: Se calcula `half = n / 2`, en este caso igual a 1.

```
// Dividir las matrices en submatrices de tamaño n/2
val half = n / 2  half: 1    n: 2
```

Figura 123: Debug MultStrassenPar

```
Ⓟ m1 = {Vector1@2403} size = 2
Ⓟ m2 = {Vector1@2424} size = 2
10 01 n = 2
10 01 half = 1
```

Figura 124: Debug MultStrassenPar

La función `multStrassenPar` opera de la siguiente manera:

1. **Caso base:** Si el tamaño de las matrices es 1×1 , se realiza la multiplicación directa.
2. **Caso recursivo:** Si el tamaño de las matrices es mayor que 1, se dividen en submatrices y se calculan las 7 multiplicaciones de Strassen. En este caso procedemos a realizar las submatrices de igual manera que en la función anterior `MultStrassen`.

Las matrices A y B se dividen en submatrices de 1×1 de la siguiente manera:

$$\begin{array}{ll} A_{11} = [1], & A_{12} = [2] \\ A_{21} = [3], & A_{22} = [4] \\ B_{11} = [1], & B_{12} = [0] \\ B_{21} = [0], & B_{22} = [1] \end{array}$$

```
val (a11, a12, a21, a22) = ( a22: size = 1    a21: size = 1    a12: size = 1    a11: size = 1
  subMatriz(m1, 0, 0, half),
  subMatriz(m1, 0, half, half),
  subMatriz(m1, half, 0, half),
  subMatriz(m1, half, half, half)  m1: size = 2    half: 1
```

Figura 125: Debug MultStrassenPar

```
val (b11, b12, b21, b22) = ( b22: size = 1    b21: size = 1    b12: size = 1    b11: size = 1
  subMatriz(m2, 0, 0, half),
  subMatriz(m2, 0, half, half),
  subMatriz(m2, half, 0, half),
  subMatriz(m2, half, half, half)  m2: size = 2    half: 1
```

Figura 126: Debug MultStrassenPar

La división permite que las submatrices sean matrices de 1×1 , lo que facilita la multiplicación directa en el caso base, tal y como vemos en el resultado de dicha operación:

```

a11 = {Vector1@2771} size = 1
a12 = {Vector1@2772} size = 1
a21 = {Vector1@2773} size = 1
a22 = {Vector1@2774} size = 1
b11 = {Vector1@2782} size = 1
b12 = {Vector1@2783} size = 1
b21 = {Vector1@2784} size = 1
b22 = {Vector1@2785} size = 1

```

Figura 127: Debug MultStrassenPar

Aquí observamos el primer llamado a pila de la función Parallel:

```

parallel:50, package$ (common)
multStrassenPar:230, Taller3 (taller)
$anonfun$new$1:48, MultStrassenParTest (taller)

```

Figura 128: Debug MultStrassenPar

Llamadas Recursivas paralelas A continuación, se generan las siete llamadas recursivas de la multiplicación de Strassen.

- `multStrassenPar(A11, B12 - B22) → multStrassenPar(1, -1)`
- `multStrassenPar(A11 + A12, B22) → multStrassenPar(3, 1)`
- `multStrassenPar(A21 + A22, B11) → multStrassenPar(7, 1)`
- `multStrassenPar(A22, B21 - B11) → multStrassenPar(4, -1)`
- `multStrassenPar(A11 + A22, B11 + B22) → multStrassenPar(5, 2)`
- `multStrassenPar(A12 - A22, B21 + B22) → multStrassenPar(-2, 2)`
- `multStrassenPar(A11 - A21, B11 + B12) → multStrassenPar(-2, 1)`

Cada una de estas llamadas recursivas es evaluada por separado, alcanzando finalmente el caso base con matrices de tamaño 1×1 .

Se combinan los resultados de las submatrices obtenidas y se reconstruye. Esto continúa hasta resolver todas las multiplicaciones necesarias, utilizando los resultados de las submatrices para finalmente obtener las submatrices de la matriz resultante C . a continuación podemos observarlo:

```
// Combinar los resultados de las submatrices
val c11 = sumMatriz(restaMatriz(sumMatriz(p5, p4), p2), p6)  c11: size = 1
val c12 = sumMatriz(p1, p2)  c12: size = 1
val c21 = sumMatriz(p3, p4)  c21: size = 1
val c22 = restaMatriz(restaMatriz(sumMatriz(p5, p1), p3), p7)  c22: size = 1

// Reconstruir la matriz resultante
Vector.tabulate(n, n) { (i, j) =>  j: 0    i: 1
  if (i < half && j < half) c11(i)(j)  c11: size = 1
  else if (i < half) c12(i)(j - half)  j: 0    c12: size = 1    i: 1    half: 1
```

Figura 129: Debug MultStrassenPar

Una vez resueltas todas las multiplicaciones, los resultados se combinan según las fórmulas dadas anteriormente para obtener las submatrices del resultado final. Por último entramos a los condicionales, en donde se evalúa uno a uno si cumple o no la condición.

```
Vector.tabulate(n, n) { (i, j) =>  j: 0    i: 1
  if (i < half && j < half) c11(i)(j)  c11: size = 1
  else if (i < half) c12(i)(j - half)  c12: size = 1
  else if (j < half) c21(i - half)(j)  j: 0    c21: size = 1    i: 1    half: 1
```

Figura 130: Debug MultStrassenPar

```
Vector.tabulate(n, n) { (i, j) =>  j: 1    i: 1
  if (i < half && j < half) c11(i)(j)
  else if (i < half) c12(i)(j - half)
  else if (j < half) c21(i - half)(j)
  else c22(i - half)(j - half)  j: 1    i: 1    half: 1
```

Figura 131: Debug MultStrassenPar

Seguimos la ejecución hasta que se cumplan todas las situaciones y nos aparezca un mensaje en donde efectivamente el 'assert' se cumple sin problemas.

```
✓ Tests passed: 1 of 1 test - 49 ms
```

Figura 132: Debug MultStrassenPar

Conclusión El análisis muestra cómo la recursión en el algoritmo de Strassen divide el problema de multiplicar matrices grandes en subproblemas más pequeños. Cada subproblema se resuelve recursivamente hasta llegar al caso base, y luego se combinan los resultados intermedios para obtener la matriz final. Este enfoque mejora la eficiencia de la multiplicación de matrices y es especialmente útil cuando las matrices son de gran tamaño.

2. Informe de Paralización

2.1. Multiplicación Vectores (paralela)

```
..... Iniciando benchmarking .....  
----- Benchmarking: Producto Punto de Vectores -----  
----- | Iteración 1: Tamaño Vector = 2 | -----  
Unable to create a system terminal  
  
Tiempo Secuencial: 0,0276 ms  
Tiempo Paralelo: 0,5307 ms  
Aceleración: 0,0520  
  
----- | Iteración 2: Tamaño Vector = 4 | -----  
  
Tiempo Secuencial: 0,0151 ms  
Tiempo Paralelo: 0,4257 ms  
Aceleración: 0,0355  
  
----- | Iteración 3: Tamaño Vector = 8 | -----  
  
Tiempo Secuencial: 0,0226 ms  
Tiempo Paralelo: 0,3992 ms  
Aceleración: 0,0566  
  
----- | Iteración 4: Tamaño Vector = 16 | -----  
  
Tiempo Secuencial: 0,0171 ms  
Tiempo Paralelo: 0,3646 ms  
Aceleración: 0,0469  
  
----- | Iteración 5: Tamaño Vector = 32 | -----  
  
Tiempo Secuencial: 0,0182 ms  
Tiempo Paralelo: 0,4127 ms  
Aceleración: 0,0441
```

Figura 133: Prueba Vector

```
----- | Iteración 6: Tamaño Vector = 64 | -----  
  
Tiempo Secuencial: 0,0245 ms  
Tiempo Paralelo: 0,7099 ms  
Aceleración: 0,0345  
  
----- | Iteración 7: Tamaño Vector = 128 | -----  
  
Tiempo Secuencial: 0,0321 ms  
Tiempo Paralelo: 0,8125 ms  
Aceleración: 0,0395  
  
----- | Iteración 8: Tamaño Vector = 256 | -----  
  
Tiempo Secuencial: 0,0834 ms  
Tiempo Paralelo: 0,7271 ms  
Aceleración: 0,1147  
  
----- | Iteración 9: Tamaño Vector = 512 | -----  
  
Tiempo Secuencial: 0,0574 ms  
Tiempo Paralelo: 0,8702 ms  
Aceleración: 0,0660  
  
----- | Iteración 10: Tamaño Vector = 1024 | -----  
  
Tiempo Secuencial: 0,1191 ms  
Tiempo Paralelo: 1,1496 ms  
Aceleración: 0,1036
```

Figura 134: Pruebas vector

```

..... Iniciando benchmarking .....

----- Benchmarking: Producto Punto de Vectores -----

----- | Iteración 1: Tamaño Vector = 2 | -----
Unable to create a system terminal

Tiempo Secuencial: 0,0608 ms
Tiempo Paralelo: 0,4771 ms
Aceleración: 0,1274

----- | Iteración 2: Tamaño Vector = 4 | -----

Tiempo Secuencial: 0,0202 ms
Tiempo Paralelo: 0,4378 ms
Aceleración: 0,0461

----- | Iteración 3: Tamaño Vector = 8 | -----

Tiempo Secuencial: 0,0146 ms
Tiempo Paralelo: 0,4071 ms
Aceleración: 0,0359

----- | Iteración 4: Tamaño Vector = 16 | -----

Tiempo Secuencial: 0,0315 ms
Tiempo Paralelo: 0,5211 ms
Aceleración: 0,0604

----- | Iteración 5: Tamaño Vector = 32 | -----

Tiempo Secuencial: 0,0171 ms
Tiempo Paralelo: 0,4748 ms
Aceleración: 0,0360

```

Figura 135: Prueba vector

```

----- | Iteración 6: Tamaño Vector = 64 | -----

Tiempo Secuencial: 0,0577 ms
Tiempo Paralelo: 0,4246 ms
Aceleración: 0,1359

----- | Iteración 7: Tamaño Vector = 128 | -----

Tiempo Secuencial: 0,0435 ms
Tiempo Paralelo: 0,5728 ms
Aceleración: 0,0759

----- | Iteración 8: Tamaño Vector = 256 | -----

Tiempo Secuencial: 0,0394 ms
Tiempo Paralelo: 0,4830 ms
Aceleración: 0,0816

----- | Iteración 9: Tamaño Vector = 512 | -----

Tiempo Secuencial: 0,0602 ms
Tiempo Paralelo: 0,5241 ms
Aceleración: 0,1149

----- | Iteración 10: Tamaño Vector = 1024 | -----

```

Figura 136: Prueba vectores

```

..... Iniciando benchmarking .....

----- Benchmarking: Producto Punto de Vectores -----

----- | Iteración 1: Tamaño Vector = 2 | -----
Unable to create a system terminal

Tiempo Secuencial: 0,0292 ms
Tiempo Paralelo: 0,4576 ms
Aceleración: 0,0638

----- | Iteración 2: Tamaño Vector = 4 | -----

Tiempo Secuencial: 0,0197 ms
Tiempo Paralelo: 0,5788 ms
Aceleración: 0,0340

----- | Iteración 3: Tamaño Vector = 8 | -----

Tiempo Secuencial: 0,0237 ms
Tiempo Paralelo: 0,3577 ms
Aceleración: 0,0663

----- | Iteración 4: Tamaño Vector = 16 | -----

Tiempo Secuencial: 0,0135 ms
Tiempo Paralelo: 0,4918 ms
Aceleración: 0,0275

----- | Iteración 5: Tamaño Vector = 32 | -----

Tiempo Secuencial: 0,1150 ms
Tiempo Paralelo: 0,4803 ms
Aceleración: 0,2394

```

Figura 137: Prueba Vector

```

----- | Iteración 6: Tamaño Vector = 64 | -----

Tiempo Secuencial: 0,0380 ms
Tiempo Paralelo: 0,3789 ms
Aceleración: 0,1003

----- | Iteración 7: Tamaño Vector = 128 | -----

Tiempo Secuencial: 0,0519 ms
Tiempo Paralelo: 0,3578 ms
Aceleración: 0,1451

----- | Iteración 8: Tamaño Vector = 256 | -----

Tiempo Secuencial: 0,0355 ms
Tiempo Paralelo: 0,3829 ms
Aceleración: 0,0927

----- | Iteración 9: Tamaño Vector = 512 | -----

Tiempo Secuencial: 0,0800 ms
Tiempo Paralelo: 0,5977 ms
Aceleración: 0,1338

----- | Iteración 10: Tamaño Vector = 1024 | -----

Tiempo Secuencial: 0,1186 ms
Tiempo Paralelo: 3,1330 ms
Aceleración: 0,0379

```

Figura 138: vector

```

..... Iniciando benchmarking .....

----- Benchmarking: Producto Punto de Vectores -----

----- | Iteración 2: Tamaño Vector = 4 | -----
Unable to create a system terminal

Tiempo Secuencial: 0,0420 ms
Tiempo Paralelo: 0,6318 ms
Aceleración: 0,0665

----- | Iteración 3: Tamaño Vector = 8 | -----

Tiempo Secuencial: 0,0157 ms
Tiempo Paralelo: 0,4633 ms
Aceleración: 0,0339

----- | Iteración 4: Tamaño Vector = 16 | -----

Tiempo Secuencial: 0,0165 ms
Tiempo Paralelo: 0,4161 ms
Aceleración: 0,0397

----- | Iteración 5: Tamaño Vector = 32 | -----

Tiempo Secuencial: 0,0355 ms
Tiempo Paralelo: 0,4981 ms
Aceleración: 0,0713

----- | Iteración 6: Tamaño Vector = 64 | -----

Tiempo Secuencial: 0,0319 ms
Tiempo Paralelo: 0,6758 ms
Aceleración: 0,0472

```

Figura 139: Vector

El cálculo del producto punto entre dos vectores de enteros es una operación clave en muchas aplicaciones numéricas. La implementación paralela de esta operación busca mejorar su desempeño aprovechando la paralelización inherente en las operaciones de suma y multiplicación.

La versión secuencial del algoritmo utiliza una colección estándar de Scala (`Vector`), donde las operaciones se realizan de manera iterativa y secuencial. La versión paralela, por otro lado, utiliza `ParVector`, lo que permite dividir el trabajo en tareas concurrentes distribuidas entre los núcleos disponibles.

1. Beneficios:

- La versión paralela (`prodPuntoParD`) acelera el cálculo para vectores grandes, reduciendo significativamente el tiempo total de ejecución.
- Permite aprovechar al máximo los recursos de hardware en sistemas multicore.

2. Limitaciones:

- Para vectores pequeños, el costo adicional de crear y gestionar tareas paralelas puede superar el beneficio de la paralelización.
- Requiere sistemas con múltiples núcleos para observar mejoras notables, y el desempeño puede depender del balance de carga.

1. Comparación de tiempos según el tamaño de los vectores

A través del benchmarking proporcionado con el paquete `Benchmark`, se evalúan comparativamente las dos funciones, probándolas con vectores de tamaños crecientes:

Comparación para vectores de tamaño n

- Para vectores pequeños ($n \leq 5000$): La versión secuencial (`prodPunto`) es más eficiente debido a que no incurre en sobrecostos relacionados con la paralelización.
- Para vectores grandes ($n > 5000$): La versión paralela (`prodPuntoParD`) supera a la secuencial al dividir el trabajo entre múltiples núcleos, logrando una aceleración significativa.

Ejemplo de resultados obtenidos con `compararProdPunto` para distintos tamaños de vectores:

De los resultados se deduce que la versión paralela tiene ventajas a partir de tamaños intermedios, pero es importante determinar un umbral (por ejemplo, $n > 5000$) para aprovecharla eficientemente.

2. Uso de `prodPuntoParD` en la multiplicación de matrices

La multiplicación de matrices se basa en múltiples cálculos de producto punto entre filas y columnas. Incorporar `prodPuntoParD` puede mejorar el desempeño al paralelizar estas operaciones. Sin embargo, hay factores importantes a considerar:

1. Ventajas:

- Cada cálculo de producto punto puede ejecutarse en paralelo, maximizando el uso de los recursos del sistema.
- La mejora es especialmente notable para matrices grandes donde las filas y columnas tienen dimensiones considerables.

2. Limitaciones:

- La sobrecarga de gestionar tareas paralelas para matrices pequeñas puede anular los beneficios de la paralelización.
- Para matrices densas y grandes, se necesita memoria adicional para gestionar las tareas concurrentes.

3. Ley de Amdahl en la práctica

La aceleración máxima que puede lograrse al paralelizar el cálculo del producto punto entre filas y columnas de matrices está limitada por la proporción del trabajo que puede ejecutarse en paralelo (P) y el número de núcleos disponibles (N):

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

En el caso de la multiplicación de matrices, P puede ser alto (debido a la independencia de los productos punto), pero los costos de sincronización entre las tareas y el manejo de subprocesos limitan el escalamiento.

Conclusión

- Para vectores pequeños, la implementación secuencial (`prodPunto`) es preferible por su menor sobrecarga.
- Para vectores grandes, la implementación paralela (`prodPuntoParD`) ofrece beneficios claros en términos de desempeño.
- En el caso de la multiplicación de matrices, el uso de `prodPuntoParD` es práctico para matrices grandes, pero es crucial establecer un umbral para evitar paralelizar cálculos pequeños innecesariamente.
- La metodología es escalable y puede integrarse con otros algoritmos como el de Strassen para reducir aún más la complejidad computacional.

En resumen, `prodPuntoParD` es una herramienta eficiente para aprovechar hardware paralelo, pero su uso debe estar condicionado al tamaño de los vectores o matrices para maximizar el rendimiento.

2.2. Multiplicación de matrices (paralela)

```
----- Benchmarking: Multiplicación de Matrices Secuencial y Paralela -----  
----- | Iteración 1: Tamaño Matriz = [2 x 2] | -----  
Unable to create a system terminal  
  
Tiempo Secuencial: 0,0793 ms  
Tiempo Paralelo: 0,1468 ms  
Aceleración: 0,5402  
  
----- | Iteración 2: Tamaño Matriz = [4 x 4] | -----  
  
Tiempo Secuencial: 0,1903 ms  
Tiempo Paralelo: 0,3388 ms  
Aceleración: 0,5617  
  
----- | Iteración 3: Tamaño Matriz = [8 x 8] | -----  
  
Tiempo Secuencial: 0,0853 ms  
Tiempo Paralelo: 0,7563 ms  
Aceleración: 0,1128  
  
----- | Iteración 4: Tamaño Matriz = [16 x 16] | -----  
  
Tiempo Secuencial: 0,0953 ms  
Tiempo Paralelo: 1,4811 ms  
Aceleración: 0,0643  
  
----- | Iteración 5: Tamaño Matriz = [32 x 32] | -----  
  
Tiempo Secuencial: 0,7566 ms  
Tiempo Paralelo: 10,9597 ms  
Aceleración: 0,0690
```

```
Tiempo Paralelo: 0,1408 ms  
Aceleración: 0,5639  
  
----- | Iteración 2: Tamaño Matriz = [4 x 4] | -----  
  
Tiempo Secuencial: 0,0521 ms  
Tiempo Paralelo: 0,2391 ms  
Aceleración: 0,2179  
  
----- | Iteración 3: Tamaño Matriz = [8 x 8] | -----  
  
Tiempo Secuencial: 0,0871 ms  
Tiempo Paralelo: 0,8168 ms  
Aceleración: 0,1066  
  
----- | Iteración 4: Tamaño Matriz = [16 x 16] | -----  
  
Tiempo Secuencial: 0,1247 ms  
Tiempo Paralelo: 1,5326 ms  
Aceleración: 0,0814  
  
----- | Iteración 5: Tamaño Matriz = [32 x 32] | -----  
  
Tiempo Secuencial: 1,4880 ms  
Tiempo Paralelo: 7,5969 ms  
Aceleración: 0,1959  
  
----- | Iteración 6: Tamaño Matriz = [64 x 64] | -----  
  
Tiempo Secuencial: 6,3989 ms  
Tiempo Paralelo: 48,6236 ms  
Aceleración: 0,1316
```

```

----- Benchmarking: Multiplicación de Matrices Secuencial y Paralela -----

----- | Iteración 1: Tamaño Matriz = [2 x 2] | -----
Unable to create a system terminal

Tiempo Secuencial: 0,0946 ms
Tiempo Paralelo: 0,1448 ms
Aceleración: 0,6533

----- | Iteración 2: Tamaño Matriz = [4 x 4] | -----

Tiempo Secuencial: 0,0618 ms
Tiempo Paralelo: 0,2206 ms
Aceleración: 0,2801

----- | Iteración 3: Tamaño Matriz = [8 x 8] | -----

Tiempo Secuencial: 0,1379 ms
Tiempo Paralelo: 0,5330 ms
Aceleración: 0,2587

----- | Iteración 4: Tamaño Matriz = [16 x 16] | -----

Tiempo Secuencial: 0,1362 ms
Tiempo Paralelo: 1,6266 ms
Aceleración: 0,0837

----- | Iteración 5: Tamaño Matriz = [32 x 32] | -----

Tiempo Secuencial: 0,7514 ms
Tiempo Paralelo: 12,1100 ms
Aceleración: 0,0620

```

El cálculo de la multiplicación de matrices es una operación fundamental en muchas aplicaciones de álgebra lineal y procesamiento de datos. Implementar una versión paralela permite aprovechar los recursos de hardware modernos para acelerar el cálculo, especialmente en matrices grandes.

La versión secuencial del algoritmo utiliza la transposición de la segunda matriz (m_2) para calcular eficientemente los productos punto entre las filas de la primera matriz (m_1) y las columnas de la transpuesta. La versión paralela, en cambio, divide este cálculo en tareas concurrentes para que los productos punto se calculen simultáneamente.

1. Beneficios:

- La versión paralela (`multMatrizPar`) acelera significativamente la multiplicación para matrices grandes.
- Reduce el tiempo de cálculo al permitir que múltiples filas y columnas se procesen en paralelo.

2. Limitaciones:

- Para matrices pequeñas, el costo de crear y sincronizar tareas paralelas puede ser mayor que los beneficios obtenidos.
- El escalado depende de la cantidad de núcleos disponibles y del balance de carga entre las tareas.

1. Comparación de tiempos según el tamaño de las matrices

Se realizó una evaluación comparativa entre la versión secuencial (`multMatriz`) y la versión paralela (`multMatrizPar`) utilizando matrices de diferentes tamaños.

De estos resultados se deduce que la versión paralela es especialmente efectiva para matrices grandes ($n > 50$), mientras que para matrices pequeñas ($n \leq 50$), los costos adicionales de paralelización hacen que la versión secuencial sea más adecuada.

2. Ventajas y desventajas de usar paralelismo en la multiplicación de matrices

La multiplicación de matrices se puede paralelizar eficientemente porque cada elemento de la matriz resultado depende únicamente de productos punto individuales que son independientes entre sí. Sin embargo, existen factores a considerar:

1. Ventajas:

- Permite una aceleración significativa en sistemas multicore al distribuir los cálculos entre los núcleos disponibles.
- Es adecuada para matrices grandes donde la cantidad de operaciones es elevada.

2. Limitaciones:

- Para matrices pequeñas, los costos asociados con la creación y gestión de tareas paralelas pueden ser mayores que el tiempo de ejecución de la versión secuencial.
- La implementación paralela puede requerir más memoria y recursos para gestionar las tareas concurrentes.

3. Ley de Amdahl en la práctica

La aceleración máxima que puede lograrse al paralelizar la multiplicación de matrices está limitada por la fracción del trabajo que puede ejecutarse en paralelo (P) y el número de núcleos disponibles (N):

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

En este caso, P es alto porque las operaciones para calcular cada elemento de la matriz resultado son independientes. Sin embargo, los costos de sincronización y comunicación entre tareas pueden reducir la aceleración teórica.

Conclusión

- Para matrices pequeñas, la implementación secuencial (`multMatriz`) es más eficiente por su simplicidad y baja sobrecarga.
- Para matrices grandes, la versión paralela (`multMatrizPar`) ofrece una mejora significativa en el tiempo de ejecución al aprovechar la concurrencia.
- Es importante establecer un umbral para decidir cuándo usar paralelismo, evitando sobrecostos innecesarios para cálculos pequeños.
- Este enfoque es escalable y se puede combinar con algoritmos optimizados como el de Strassen para matrices densas o grandes.

En resumen, la versión paralela de la multiplicación de matrices demuestra ser una herramienta poderosa para sistemas multicore, logrando aceleraciones significativas para matrices grandes. Sin embargo, su uso debe estar condicionado por el tamaño de las matrices y la disponibilidad de recursos para maximizar su eficiencia.

2.3. Multiplicación recursiva de matrices (paralela)

```
..... Iniciando benchmarking .....  
  
----- Benchmarking: Multiplicación Recursiva de Matrices Secuencial y Paralela -----  
  
----- | Iteración 1: Tamaño Matriz = [2 x 2] | -----  
Unable to create a system terminal  
  
Tiempo Secuencial: 0,1947 ms  
Tiempo Paralelo: 0,0391 ms  
Aceleración: 4,9795  
  
----- | Iteración 2: Tamaño Matriz = [4 x 4] | -----  
  
Tiempo Secuencial: 0,1608 ms  
Tiempo Paralelo: 0,1251 ms  
Aceleración: 1,2854  
  
----- | Iteración 3: Tamaño Matriz = [8 x 8] | -----  
  
Tiempo Secuencial: 0,5025 ms  
Tiempo Paralelo: 0,4184 ms  
Aceleración: 1,2010  
  
----- | Iteración 4: Tamaño Matriz = [16 x 16] | -----  
  
Tiempo Secuencial: 1,1069 ms  
Tiempo Paralelo: 2,2484 ms  
Aceleración: 0,4923  
  
----- | Iteración 5: Tamaño Matriz = [32 x 32] | -----  
  
Tiempo Secuencial: 9,3234 ms  
Tiempo Paralelo: 8,2348 ms  
Aceleración: 1,1322
```

```
----- Benchmarking: Multiplicación Recursiva de Matrices Secuencial y Paralela -----  
  
----- | Iteración 1: Tamaño Matriz = [2 x 2] | -----  
Unable to create a system terminal  
  
Tiempo Secuencial: 0,2018 ms  
Tiempo Paralelo: 0,0417 ms  
Aceleración: 4,8393  
  
----- | Iteración 2: Tamaño Matriz = [4 x 4] | -----  
  
Tiempo Secuencial: 0,3298 ms  
Tiempo Paralelo: 0,2815 ms  
Aceleración: 1,1716  
  
----- | Iteración 3: Tamaño Matriz = [8 x 8] | -----  
  
Tiempo Secuencial: 0,9012 ms  
Tiempo Paralelo: 0,3628 ms  
Aceleración: 2,4840  
  
----- | Iteración 4: Tamaño Matriz = [16 x 16] | -----  
  
Tiempo Secuencial: 0,9818 ms  
Tiempo Paralelo: 0,9782 ms  
Aceleración: 1,0037  
  
----- | Iteración 5: Tamaño Matriz = [32 x 32] | -----  
  
Tiempo Secuencial: 8,4808 ms  
Tiempo Paralelo: 7,8036 ms  
Aceleración: 1,0868
```

```
----- Benchmarking: Multiplicación Recursiva de Matrices Secuencial y Paralela -----  
  
----- | Iteración 1: Tamaño Matriz = [2 x 2] | -----  
Unable to create a system terminal  
  
Tiempo Secuencial: 0,1800 ms  
Tiempo Paralelo: 0,0536 ms  
Aceleración: 3,3582  
  
----- | Iteración 2: Tamaño Matriz = [4 x 4] | -----  
  
Tiempo Secuencial: 0,1460 ms  
Tiempo Paralelo: 0,1472 ms  
Aceleración: 0,9918  
  
----- | Iteración 3: Tamaño Matriz = [8 x 8] | -----  
  
Tiempo Secuencial: 0,3129 ms  
Tiempo Paralelo: 0,4237 ms  
Aceleración: 0,7385  
  
----- | Iteración 4: Tamaño Matriz = [16 x 16] | -----  
  
Tiempo Secuencial: 3,4637 ms  
Tiempo Paralelo: 1,0377 ms  
Aceleración: 3,3379  
  
----- | Iteración 5: Tamaño Matriz = [32 x 32] | -----  
  
Tiempo Secuencial: 9,8148 ms  
Tiempo Paralelo: 10,4992 ms  
Aceleración: 0,9348
```

```

----- | Iteración 6: Tamaño Matriz = [64 x 64] | -----
Unable to create a system terminal

> Task :app:run
Hello, World!

..... Iniciando benchmarking .....

----- Benchmarking: Multiplicación Recursiva de Matrices Secuencial y Paralela -----

----- | Iteración 6: Tamaño Matriz = [64 x 64] | -----
Unable to create a system terminal

Tiempo Secuencial: 73,1741 ms
Tiempo Paralelo: 72,6837 ms
Aceleración: 1,0067

----- | Iteración 7: Tamaño Matriz = [128 x 128] | -----

Tiempo Secuencial: 73,1741 ms
Tiempo Paralelo: 72,6837 ms
Aceleración: 1,0067

----- | Iteración 7: Tamaño Matriz = [128 x 128] | -----

Tiempo Secuencial: 630,5161 ms
Tiempo Paralelo: 434,2346 ms
Aceleración: 1,4520

----- | Iteración 8: Tamaño Matriz = [256 x 256] | -----

Tiempo Secuencial: 630,5161 ms
Tiempo Paralelo: 434,2346 ms
Aceleración: 1,4520

```

El algoritmo recursivo para la multiplicación de matrices divide el problema en partes más pequeñas hasta alcanzar el caso base, donde las submatrices tienen tamaño 1×1 . La ventaja clave de la estructura divide y vencerás asegura que el algoritmo sea más manejable para matrices grandes y se puede combinar con otros métodos como la paralelización. Sin embargo, la desventaja principal en la versión secuencial es que la profundidad de la recursión puede impactar negativamente el rendimiento cuando el tamaño de las matrices es grande.

La versión paralela del algoritmo utiliza estrategias de paralelización con tareas concurrentes, lo que permite calcular diferentes submatrices ($C_{11}, C_{12}, C_{21}, C_{22}$) de manera simultánea.

1. Beneficios:

- Acelera significativamente el cálculo, especialmente para matrices grandes.
- Permite aprovechar múltiples núcleos de procesamiento.

2. Limitaciones:

- Cuando el tamaño de las matrices es pequeño ($n \leq 8$), el costo de crear y gestionar tareas paralelas puede ser mayor que el beneficio obtenido.

3. Ley de Amdahl en la práctica

La aceleración máxima alcanzada por la paralelización depende de la fracción del algoritmo que puede ejecutarse en paralelo (P) y el número de núcleos disponibles (N):

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

En este caso, la paralelización es efectiva porque las operaciones para calcular $C_{11}, C_{12}, C_{21}, C_{22}$ son independientes.

- Comparando las versiones secuencial y paralela, la versión secuencial es simple de implementar y funciona bien para matrices pequeñas, pero el tiempo de ejecución crece significativamente con matrices grandes debido a la naturaleza recursiva. La versión paralela es más eficiente para matrices grandes, pero introduce sobrecostos en la creación y sincronización de tareas, requiriendo un umbral para evitar paralelizar tareas pequeñas innecesariamente.

- Este algoritmo es adecuado para matrices cuadradas de dimensiones $2^k \times 2^k$. Si las matrices no cumplen con estas dimensiones, es necesario rellenarlas con ceros antes de aplicar el algoritmo. La metodología es escalable y puede combinarse con algoritmos optimizados como el de Strassen para reducir aún más la complejidad computacional.

En resumen, el algoritmo recursivo, especialmente en su versión paralela, demuestra ser una herramienta poderosa para la multiplicación de matrices grandes. Aunque su implementación es más compleja, la aceleración lograda en comparación con la versión secuencial justifica su uso en sistemas con recursos paralelos disponibles. Sin embargo, es crucial balancear el umbral de paralelización y tener en cuenta los costos asociados para garantizar un rendimiento óptimo.

2.4. Función algoritmo de Strassen (*paralela*)

```
..... Iniciando benchmarking .....
----- Benchmarking: Multiplicación de Matrices con Algoritmo de Strassen -----
----- | Iteración 1: Tamaño Matriz = [2 x 2] | -----
Unable to create a system terminal

Tiempo Secuencial: 0,1586 ms
Tiempo Paralelo: 0,1695 ms
Aceleración: 0,9357

----- | Iteración 2: Tamaño Matriz = [4 x 4] | -----
Tiempo Secuencial: 0,0964 ms
Tiempo Paralelo: 0,3322 ms
Aceleración: 0,2902

----- | Iteración 3: Tamaño Matriz = [8 x 8] | -----
Tiempo Secuencial: 0,4281 ms
Tiempo Paralelo: 0,4284 ms
Aceleración: 0,9993

----- | Iteración 4: Tamaño Matriz = [16 x 16] | -----
Tiempo Secuencial: 1,9373 ms
Tiempo Paralelo: 1,5670 ms
Aceleración: 1,2363

----- | Iteración 5: Tamaño Matriz = [32 x 32] | -----
Tiempo Secuencial: 11,2790 ms
Tiempo Paralelo: 8,9517 ms
Aceleración: 1,2600
```

```
..... Iniciando benchmarking .....
----- Benchmarking: Multiplicación de Matrices con Algoritmo de Strassen -----
----- | Iteración 1: Tamaño Matriz = [2 x 2] | -----
Unable to create a system terminal

Tiempo Secuencial: 0,8770 ms
Tiempo Paralelo: 0,1898 ms
Aceleración: 4,6207

----- | Iteración 2: Tamaño Matriz = [4 x 4] | -----
Tiempo Secuencial: 0,6871 ms
Tiempo Paralelo: 0,1845 ms
Aceleración: 3,7241

----- | Iteración 3: Tamaño Matriz = [8 x 8] | -----
Tiempo Secuencial: 0,4215 ms
Tiempo Paralelo: 0,5266 ms
Aceleración: 0,8004

----- | Iteración 4: Tamaño Matriz = [16 x 16] | -----
Tiempo Secuencial: 5,4058 ms
Tiempo Paralelo: 2,4596 ms
Aceleración: 2,1978

----- | Iteración 5: Tamaño Matriz = [32 x 32] | -----
Tiempo Secuencial: 11,0211 ms
Tiempo Paralelo: 9,0612 ms
Aceleración: 1,2163
```

```

----- Benchmarking: Multiplicación de Matrices con Algoritmo de Strassen -----
----- | Iteración 1: Tamaño Matriz = [2 x 2] | -----
Unable to create a system terminal

Tiempo Secuencial: 0,0815 ms
Tiempo Paralelo: 0,1711 ms
Aceleración: 0,4763

----- | Iteración 2: Tamaño Matriz = [4 x 4] | -----

Tiempo Secuencial: 0,1766 ms
Tiempo Paralelo: 0,1441 ms
Aceleración: 1,2255

----- | Iteración 3: Tamaño Matriz = [8 x 8] | -----

Tiempo Secuencial: 0,4073 ms
Tiempo Paralelo: 0,5524 ms
Aceleración: 0,7373

----- | Iteración 4: Tamaño Matriz = [16 x 16] | -----

Tiempo Secuencial: 2,8215 ms
Tiempo Paralelo: 3,2136 ms
Aceleración: 0,8780

----- | Iteración 5: Tamaño Matriz = [32 x 32] | -----

Tiempo Secuencial: 11,7506 ms
Tiempo Paralelo: 7,0751 ms
Aceleración: 1,6608

Tiempo Secuencial: 11,7506 ms
Tiempo Paralelo: 7,0751 ms
Aceleración: 1,6608

```

```

----- Benchmarking: Multiplicación de Matrices con Algoritmo de Strassen -----
----- | Iteración 6: Tamaño Matriz = [64 x 64] | -----
Unable to create a system terminal

> Task :app:run
Hello, World!

..... Iniciando benchmarking .....

----- Benchmarking: Multiplicación de Matrices con Algoritmo de Strassen -----
----- | Iteración 6: Tamaño Matriz = [64 x 64] | -----
Unable to create a system terminal

Tiempo Secuencial: 79,2098 ms
Tiempo Paralelo: 59,4211 ms
Aceleración: 1,3330

----- | Iteración 7: Tamaño Matriz = [128 x 128] | -----

Tiempo Secuencial: 79,2098 ms
Tiempo Paralelo: 59,4211 ms
Aceleración: 1,3330

----- | Iteración 7: Tamaño Matriz = [128 x 128] | -----

Tiempo Secuencial: 569,5738 ms
Tiempo Paralelo: 504,6739 ms
Aceleración: 1,1286

----- | Iteración 8: Tamaño Matriz = [256 x 256] | -----

Tiempo Secuencial: 569,5738 ms
Tiempo Paralelo: 504,6739 ms
Aceleración: 1,1286

```

```

----- | Iteración 6: Tamaño Matriz = [64 x 64] | -----
Unable to create a system terminal

> Task :app:run
Hello, World!

..... Iniciando benchmarking .....

----- Benchmarking: Multiplicación Recursiva de Matrices Secuencial y Paralela -----

----- | Iteración 6: Tamaño Matriz = [64 x 64] | -----
Unable to create a system terminal

Tiempo Secuencial: 73,1741 ms
Tiempo Paralelo: 72,6837 ms
Aceleración: 1,0067

----- | Iteración 7: Tamaño Matriz = [128 x 128] | -----

Tiempo Secuencial: 73,1741 ms
Tiempo Paralelo: 72,6837 ms
Aceleración: 1,0067

----- | Iteración 7: Tamaño Matriz = [128 x 128] | -----

Tiempo Secuencial: 630,5161 ms
Tiempo Paralelo: 434,2346 ms
Aceleración: 1,4520

----- | Iteración 8: Tamaño Matriz = [256 x 256] | -----

Tiempo Secuencial: 630,5161 ms
Tiempo Paralelo: 434,2346 ms
Aceleración: 1,4520

```

La estrategia utilizada para paralelizar la función `multStrassenPar` sigue el algoritmo de Strassen, que es un método para multiplicar matrices de manera eficiente, dividiéndolas en submatrices más pequeñas y reduciendo el número de multiplicaciones necesarias en comparación con la multiplicación de matrices tradicional.

La paralelización en este caso se realiza dividiendo las 7 multiplicaciones necesarias (**p1** a **p7**) en dos grupos:

1. **Primer grupo:** Las multiplicaciones **p1**, **p2**, **p3**, y **p4** se ejecutan en paralelo usando el operador `parallel`. Estas multiplicaciones corresponden a las operaciones principales del algoritmo de Strassen, donde las submatrices se combinan de diversas formas para obtener los productos.
2. **Segundo grupo:** En este caso, se realiza un segundo nivel de paralelización. La multiplicación **p5** se calcula en paralelo con el cálculo de **p6** y **p7** en un segundo nivel de paralelización usando otro `parallel`. Al hacerlo, se reduce la cantidad de operaciones secuenciales, acelerando el proceso.

La estrategia de paralelización se basa en la ley de Amdahl, que establece que el rendimiento de una paralelización está limitado por la fracción del programa que no puede ser paralelizada (la parte secuencial). En este caso, las multiplicaciones **p1** a **p7** son las operaciones que pueden ser paralelizadas, pero el proceso de reconstrucción de la matriz final (las operaciones de suma y resta entre las submatrices resultantes) es secuencial. Aunque se mejora el rendimiento significativamente con la paralelización de las multiplicaciones, la mejora total está limitada por la parte secuencial de la función.

Ganancias de acuerdo a la ley de Amdahl: La ley de Amdahl se expresa como:

$$S_{\text{total}} = \frac{1}{(1 - p) + \frac{p}{N}}$$

donde:

- S_{total} es la mejora total del sistema.
- p es la fracción del programa que puede ser paralelizada.
- N es el número de procesadores disponibles.

En este caso, p sería la fracción del tiempo de ejecución dedicado a las multiplicaciones **p1** a **p7**, mientras que el resto de las operaciones (principalmente la reconstrucción de la matriz) es secuencial.

Si se usa una gran cantidad de procesadores (por ejemplo, N muy alto), el rendimiento puede mejorar significativamente, pero la mejora total sigue estando limitada por la fracción secuencial del

programa (la reconstrucción de la matriz final). Por lo tanto, a pesar de la paralelización, el tiempo de ejecución no se reduce de manera proporcional a N debido a esta parte secuencial.

Explicación de los Resultados: El conjunto de pruebas desarrollado para la función `multStrassenPar` tiene como objetivo evaluar su correcto funcionamiento al multiplicar matrices cuadradas de diferentes tamaños y valores de entrada.

Estas pruebas garantizan que la función `multStrassenPar`:

- Funcione correctamente para matrices de diferentes tamaños.
- Respete la precisión en los resultados.
- Maneje de forma adecuada la división recursiva de matrices grandes.
- Asegure la eficiencia del cálculo a través de la paralelización, especialmente en matrices de gran tamaño.

3. Informe de Corrección

3.1. Funciones para Multiplicación *Estándar* de Matrices

3.1.1. Función para multiplicación de matrices *secuencial*

Se implementó la función `multMatriz` que tiene como objetivo realizar la multiplicación de dos matrices A y B de dimensiones compatibles. La implementación utiliza un enfoque basado en producto punto y transposición para calcular la matriz resultante. Los parámetros de entrada son:

- $m1$: Matriz de entrada de dimensión $m \times n$.
- $m2$: Matriz de entrada de dimensión $n \times p$.

La implementación se muestra a continuación:

```
1 def multMatriz(m1: Matriz, m2: Matriz): Matriz = {  
2     val m2T = transpuesta(m2)  
3     val n = m1.length  
4     Vector.tabulate(n, n) { (i, j) =>  
5         prodPunto(m1(i), m2T(j))  
6     }  
7 }
```

Se asume que las matrices de entrada son cuadradas y de dimensiones compatibles para la multiplicación, garantizando que el número de columnas de $m1$ sea igual al número de filas de $m2$. **Demostración** Queremos demostrar que para toda matriz A de dimensión $m \times n$ y B de dimensión $n \times p$, la función `multMatriz` retorna la matriz producto esperada C . Formalmente: $[\forall m1 \in \mathbb{M}^{m \times n}, \forall m2 \in \mathbb{M}^{n \times p} : \text{multMatriz}(m1, m2) = C]$ donde C es la matriz producto de dimensión $m \times p$ que cumple $c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$.

Caso base: Matrices de dimensión 1×1 Cuando las matrices son 1×1 , la multiplicación se reduce a un simple producto de elementos: $[A = [a_{11}], B = [b_{11}]]$ El resultado será: $[C = A \times B = [a_{11} \cdot b_{11}]]$ La función `prodPunto` calcula correctamente este producto para matrices unitarias.

Caso inductivo: Matrices de dimensión $k + 1$ Asumimos como hipótesis de inducción (HI) que la función es correcta para matrices de dimensión $k \times k$. Queremos demostrar que `multMatriz` genera correctamente la matriz producto para matrices de dimensión $(k + 1) \times (k + 1)$. La implementación utiliza: $[C = \text{Vector.tabulate}(k+1, k+1)((i, j) \mapsto \text{prodPunto}(m1(i), \text{transpuesta}(m2)(j)))]$

La función `transpuesta` garantiza acceso eficiente a las columnas de $m2$.

`prodPunto` calcula el producto escalar entre filas y columnas.

`Vector.tabulate` genera la matriz completa de manera eficiente.

Por construcción, cada elemento c_{ij} se calcula como $\sum_{k=1}^n a_{ik} \cdot b_{kj}$, cumpliendo la definición matemática de multiplicación matricial.

Conclusión Por inducción, `multMatriz` implementa correctamente la multiplicación de matrices para dimensiones 1×1 y $k \times k$, generalizándose a matrices de cualquier dimensión cuadrada.

Análisis de complejidad La función `multMatriz` tiene las siguientes características de complejidad:

- **Complejidad temporal:** $O(n^3)$, debido a los tres bucles anidados implícitos en `Vector.tabulate` y `prodPunto`.
- **Complejidad espacial:** $O(n^2)$ para almacenar la matriz resultante y las matrices intermedias.

3.1.2. Función para multiplicación de matrices *paralela*

Se implementó la función `multMatrizPar` que realiza la multiplicación de dos matrices A y B utilizando procesamiento paralelo. Esta implementación mejora el rendimiento de la multiplicación matricial mediante la creación de tareas concurrentes para calcular elementos de la matriz resultado. Los parámetros de entrada son:

- $m1$: Matriz de entrada de dimensión $n \times n$.
- $m2$: Matriz de entrada de dimensión $n \times n$.

La implementación se muestra a continuación:

```
1  def multMatrizPar(m1: Matriz, m2: Matriz): Matriz = {  
2      val n = m1.length  
3      val m2T = transpuesta(m2)  
4      Vector.tabulate(n, m2T(0).length) { (i, j) =>  
5          val taskElement = task {  
6              prodPunto(m1(i), m2T(j))  
7          }  
8          taskElement.join()  
9      }  
10 }
```

Se asume que las matrices de entrada son cuadradas y de dimensiones compatibles, permitiendo la creación de tareas paralelas para cada elemento de la matriz resultante. **Demostración** Queremos demostrar que la función `multMatrizPar` genera la matriz producto C de manera correcta, preservando la semántica de la multiplicación matricial secuencial. Formalmente: $[\forall m1 \in \mathbb{M}n \times n, \forall m2 \in \mathbb{M}n \times n : \text{multMatrizPar}(m1, m2) = C, \text{donde } C \text{ es la matriz producto que cumple } c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}]$.

Caso base: Matrices de dimensión 1×1 Para matrices unitarias: $[A = [a_{11}], B = [b_{11}]]$ El resultado será: $[C = A \times B = [a_{11} \cdot b_{11}]]$

Caso inductivo: Matrices de dimensión $k + 1$ Asumimos como hipótesis de inducción (HI) que la función es correcta para matrices de dimensión $k \times k$. Demostraremos la corrección para matrices de dimensión $(k + 1) \times (k + 1)$. La implementación utiliza: $[C = \text{Vector.tabulate}(k+1, k+1)((i, j) \mapsto \text{taskprodPunto}(m1(i), \text{transpuesta}(m2)(j)).join()))]$ *Características de la implementación:*

Cada elemento de la matriz resultado se calcula en una tarea independiente.

`task` crea una tarea concurrente para cada elemento.

`join()` espera la finalización de la tarea, garantizando sincronización.

`transpuesta` permite acceso eficiente a columnas de $m2$.

La estrategia paralela preserva la semántica del cálculo: $c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$

Consideraciones de Concurrencia

- Las tareas se ejecutan potencialmente en paralelo.
- `join()` asegura que todos los cálculos completen antes de generar la matriz final.
- No hay condiciones de carrera, ya que cada tarea calcula un elemento único e independiente.

Conclusión `multMatrizPar` implementa correctamente la multiplicación de matrices con paralelismo, manteniendo la semántica matemática de la multiplicación y potencialmente mejorando el rendimiento mediante la ejecución concurrente de tareas.

Análisis de complejidad

■ Complejidad temporal:

- Peor caso: $O(n^3)$ (limitado por la secuencialidad de `prodPunto`)
- Potencial mejora con paralelismo dependiendo de núcleos disponibles

■ Complejidad espacial: $O(n^2)$ para la matriz resultante

■ Overhead de concurrencia: Costos adicionales de creación y sincronización de tareas

3.2. Funciones para Multiplicación *Recursiva* de Matrices

3.2.1. Función para extraer submatrices

Se implemento la función `subMatriz(m,i,j,l)` que tiene como objetivo extraer una submatriz de dimensión $l \times l$ de una matriz A de dimensión $n \times n$, comenzando en la posición (i, j) de la matriz original. Los parámetros de entrada son:

- m : Matriz de entrada de dimensión $n \times n$.
- i : Fila inicial de la submatriz a extraer.
- j : Columna inicial de la submatriz a extraer.
- l : Dimensión de la submatriz a extraer ($l \times l$).

La implementación se muestra a continuación:

```
1 def subMatriz(m: Matriz, i: Int, j: Int, l: Int): Matriz = {  
2   Vector.tabulate(l, l)((x, y) => m(i + x)(j + y))  
3 }
```

Se asume que los índices (i, j) y la dimensión l están definidos de tal forma que $1 \leq i, j \leq n$ y $i + l \leq n$, $j + l \leq n$, lo cual garantiza que la submatriz se encuentra completamente contenida en A .

Demostración

Queremos demostrar que para toda matriz A de dimensión $n \times n$, posición (i, j) y dimensión l , la función `subMatriz` retorna la submatriz esperada de A . Formalmente:

$$\forall m \in \mathbb{M}_{n \times n}, \forall i, j, l \mid (1 \leq i, j \leq n \wedge i + l \leq n \wedge j + l \leq n) : \text{subMatriz}(m, i, j, l) = S,$$

donde S es la submatriz de m de dimensión $l \times l$ que comienza en la posición (i, j) .

Caso base: $l = 1$ Cuando $l = 1$, la submatriz debe contener solo un elemento A_{ij} . La implementación de `subMatriz` genera un vector de 1×1 con:

$$\text{Vector.tabulate}(1, 1)((x, y) \Rightarrow m(i + x)(j + y))$$

- $x = 0, y = 0$ (únicos valores posibles dado que $l = 1$).
- Entonces, el único valor es $m(i + 0)(j + 0) = m(i)(j)$, que corresponde a A_{ij} .

Por lo tanto, `subMatriz(m, i, j, 1)` retorna $[A_{ij}]$, que es correcto.

Caso inductivo: $l = k + 1$, $k \geq 1$ Asumimos como hipótesis de inducción (HI) que la función es correcta para $l = k$. Es decir, `subMatriz(m, i, j, k)` retorna la submatriz S_k de dimensión $k \times k$ que comienza en (i, j) .

Queremos demostrar que `subMatriz(m, i, j, k + 1)` genera la submatriz S_{k+1} de dimensión $(k + 1) \times (k + 1)$. La implementación genera:

$$S_{k+1} = \text{Vector.tabulate}(k + 1, k + 1)((x, y) \Rightarrow m(i + x)(j + y)).$$

- Para $x, y < k$, los valores generados coinciden con los de `subMatriz(m, i, j, k)` debido a la misma fórmula.
- Para $x = k$ o $y = k$, los nuevos valores corresponden a los elementos adicionales necesarios para extender S_k a S_{k+1} .

Por construcción, S_{k+1} contiene correctamente todos los elementos en el rango $(i \leq r < i + k + 1, j \leq c < j + k + 1)$.

Conclusión Por inducción, `subMatriz` es correcta para toda dimensión $l \geq 1$ siempre que los índices especificados sean válidos.

Análisis de complejidad La función `subMatriz` utiliza `Vector.tabulate` para generar una nueva matriz de $l \times l$, con un costo de $O(l^2)$ debido a la evaluación de l^2 elementos. La extracción de cada elemento $m(i + x)(j + y)$ tiene costo $O(1)$, dado que el acceso a elementos en una matriz representada como `Vector` en Scala es constante.

3.2.2. Función para sumar matrices

La función `sumMatriz` está diseñada para realizar la suma de dos matrices cuadradas m_1 y m_2 , ambas de dimensiones $n \times n$. La función utiliza `Vector.tabulate` para generar una nueva matriz m_3 , cuyos elementos se obtienen sumando los elementos correspondientes de m_1 y m_2 . El código es el siguiente:

```
1 def sumMatriz(m1: Matriz, m2: Matriz): Matriz = {
2   Vector.tabulate(m1.length, m1.length)((i, j) => m1(i)(j) + m2(i)(j))
3 }
```

Vamos a demostrar que la función `sumMatriz` es correcta, es decir, que para toda entrada válida m_1 y m_2 de dimensiones $n \times n$, la salida m_3 cumple:

$$\forall i, j \in \{0, \dots, n - 1\}, \quad m_3[i][j] = m_1[i][j] + m_2[i][j].$$

Caso base: $n = 1$ Si las matrices m_1 y m_2 son de dimensión 1×1 , entonces:

$$m_1 = [a], \quad m_2 = [b],$$

la función `sumMatriz` devuelve:

$$m = \text{Vector.tabulate}(1, 1)((i, j) \Rightarrow m_1(i)(j) + m_2(i)(j)) = [a + b].$$

Esto coincide con la definición esperada de la suma de matrices, ya que:

$$m[0][0] = m_1[0][0] + m_2[0][0].$$

Por lo tanto, el caso base es correcto.

Caso inductivo: $n \times n$ Supongamos que la función `sumMatriz` es correcta para matrices de dimensión $k \times k$ (hipótesis inductiva). Ahora probaremos que es correcta para matrices de dimensión $(k+1) \times (k+1)$.

Sea m_1 y m_2 matrices de dimensión $(k+1) \times (k+1)$. La construcción de m_3 se realiza de la siguiente manera:

$$m_3[i][j] = m_1[i][j] + m_2[i][j], \quad \forall i, j \in \{0, \dots, k\}.$$

El operador `Vector.tabulate` evalúa cada par de índices (i, j) dentro del rango $0 \leq i, j \leq k$, y para cada par, suma los elementos correspondientes de m_1 y m_2 .

Por la hipótesis inductiva, sabemos que la propiedad se cumple para cualquier submatriz de dimensión $k \times k$. Dado que la construcción de m_3 sigue el mismo principio para la fila y columna adicionales, la propiedad también se cumple para la dimensión $(k+1) \times (k+1)$.

Conclusión Por inducción, se concluye que la función `sumMatriz` produce correctamente una matriz m_3 tal que $m_3[i][j] = m_1[i][j] + m_2[i][j]$ para cualquier par de matrices cuadradas m_1 y m_2 de dimensión $n \times n$.

3.2.3. Función para multiplicar matrices recursivas

La multiplicación de matrices es un problema fundamental en matemáticas y ciencias de la computación. Dados dos matrices cuadradas A y B de tamaño $n \times n$, el producto $C = A \cdot B$ es otra matriz cuadrada $n \times n$, donde cada elemento C_{ij} se define como:

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

- A_{ik} : Elemento de la fila i y columna k de A .
- B_{kj} : Elemento de la fila k y columna j de B .
- C_{ij} : Resultado del producto punto entre la fila i de A y la columna j de B .

El algoritmo que se analiza divide las matrices en submatrices más pequeñas y utiliza recursión para resolver subproblemas, aplicando la técnica de divide y vencerás. Esto evita la sobrecarga de la recursión y el paralelismo para matrices pequeñas.

Estructura del Algoritmo

El algoritmo base implementado consta de dos versiones:

- **Versión Secuencial:** Utiliza recursión para resolver cada subproblema de manera individual.
- **Versión Paralela:** Divide el cálculo en múltiples tareas independientes que se ejecutan simultáneamente.

Ambas versiones tienen una estructura similar, pero con diferencias clave en la ejecución de los subproblemas.

Función para multiplicar matrices recursivamente (*secuencial*)

La función `multMatrizRec` es una implementación recursiva de la multiplicación de matrices cuadradas basada en el enfoque de **divide y vencerás**, utilizando propiedades matemáticas de la multiplicación de matrices y descomposición en submatrices.

```
1 def multMatrizRec(m1: Matriz, m2: Matriz): Matriz = {
2   val n = m1.length // Tamaño de las matrices
3   if (n == 1) {
4     // Caso base: multiplicación de un solo elemento
5     Vector(Vector(m1(0)(0) * m2(0)(0)))
6   } else {
7     val half = n / 2
8
9     // División de las matrices en submatrices
10    val (a11, a12, a21, a22) = (
11      subMatriz(m1, 0, 0, half),
12      subMatriz(m1, 0, half, half),
13      subMatriz(m1, half, 0, half),
14      subMatriz(m1, half, half, half)
15    )
16    val (b11, b12, b21, b22) = (
17      subMatriz(m2, 0, 0, half),
18      subMatriz(m2, 0, half, half),
19      subMatriz(m2, half, 0, half),
20      subMatriz(m2, half, half, half)
21    )
22
23    // Calcular las submatrices de C
24    val c11 = sumMatriz(multMatrizRec(a11, b11), multMatrizRec(a12, b21))
25    val c12 = sumMatriz(multMatrizRec(a11, b12), multMatrizRec(a12, b22))
26    val c21 = sumMatriz(multMatrizRec(a21, b11), multMatrizRec(a22, b21))
27    val c22 = sumMatriz(multMatrizRec(a21, b12), multMatrizRec(a22, b22))
28
29    // Construcción de la matriz resultante
30    Vector.tabulate(n, n) { (i, j) =>
31      if (i < half && j < half) c11(i)(j)
32      else if (i < half) c12(i)(j - half)
33      else if (j < half) c21(i - half)(j)
34      else c22(i - half)(j - half)
35    }
36  }
37 }
```

Paso Base: $n = 1$ Cuando el tamaño de las matrices es $n = 1$, estas contienen un único elemento. El algoritmo realiza directamente el cálculo del producto:

$$\blacksquare C = A \cdot B.$$

Por ejemplo: Si $A = [2]$ y $B = [3]$, el cálculo es:

$$\blacksquare C = [2 \cdot 3] = [6].$$

Esto garantiza que el algoritmo tiene un punto de terminación claro y cumple con la definición matemática de la multiplicación de matrices.

$$\blacksquare \text{Dividir y Resolver Subproblemas: } n > 1$$

Cuando $n > 1$, las matrices A y B se dividen en cuatro submatrices cuadradas de tamaño $\frac{n}{2} \times \frac{n}{2}$:

$$\blacksquare \text{Para matrices cuadradas grandes } (n > 1), \text{ este problema puede abordarse dividiendo } A \text{ y } B \text{ en submatrices cuadradas de tamaño } \frac{n}{2} \times \frac{n}{2}. \text{ Si:}$$

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

entonces:

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

donde:

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

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

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

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

Cada submatriz se extrae utilizando la función `subMatriz`, que selecciona un rango específico de filas y columnas. Por ejemplo, si A y B tienen tamaño 4×4 :

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Las submatrices serían:

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}, \quad A_{12} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix}, \quad \text{etc.}$$

Cálculo de las Submatrices de C Cada submatriz de C se calcula mediante la combinación de submatrices de A y B . Por ejemplo:

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

Este cálculo se realiza recursivamente, llamando a `multMatrizRec` para resolver los productos parciales.

Ejemplo: Si $A_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}$ y $B_{11} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, entonces:

$$C_{11} = A_{11} \cdot B_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}$$

Ensamblaje de la Matriz Final Las submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ se combinan utilizando `Vector.tabulate`, que asegura que cada elemento de C se coloca en su posición correcta.

Por ejemplo, para $n = 4$, C será:

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

Caso Inductivo: Corrección Matemática: Hipótesis de Inducción Supongamos que el algoritmo es correcto para matrices de tamaño $\frac{n}{2}$. Es decir, para cualquier par de matrices A y B de tamaño $\frac{n}{2} \times \frac{n}{2}$, el algoritmo produce una matriz C tal que:

$$C = A \cdot B$$

Para matrices de tamaño n :

- A y B se dividen correctamente en submatrices como se vio anteriormente en el caso base.
- Las submatrices C_{ij} se calculan recursivamente según la hipótesis inductiva.

Propiedad Distributiva de las Matrices

Se realiza aplicando la propiedad distributiva de las matrices:

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

Donde cada cuadrante de C es calculado como:

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

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

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

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

- La combinación de las submatrices garantiza que C es el producto correcto de A y B , ya que respeta las propiedades distributivas de la multiplicación matricial.

Por lo tanto, el algoritmo es correcto por inducción.

- Si llamamos $T(n)$ al tiempo para multiplicar matrices de tamaño $n \times n$, la recurrencia se expresa como:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

Esto se debe a:

- 8 llamadas recursivas a $T\left(\frac{n}{2}\right)$ (un cálculo para cada combinación de submatrices).
- $O(n^2)$ operaciones para sumar matrices de tamaño $n \times n$.

Resolviendo esta recurrencia, se obtiene $T(n) = O(n^3)$, igual a la multiplicación clásica. Sin embargo, algoritmos optimizados como Strassen reducen este costo.

Función para multiplicar matrices recursivamente (*paralela*)

El algoritmo `multMatrizRecParl` es una extensión del algoritmo recursivo para la multiplicación de matrices con una mejora clave: utiliza **paralelismo** para acelerar el cálculo de las submatrices intermedias.

```

1 def multMatrizRecPar(m1: Matriz, m2: Matriz, umbral: Int = 64): Matriz = {
2   val n = m1.length
3   if (n <= umbral) {
4     // Caso base: usar la versi n secuencial
5     multMatrizRec(m1, m2)
6   } else {
7     // Dividir las matrices en submatrices de tama o n/2
8     val half = n / 2
9
10    val (a11, a12, a21, a22) = (
11      subMatriz(m1, 0, 0, half),
12      subMatriz(m1, 0, half, half),
13      subMatriz(m1, half, 0, half),
14      subMatriz(m1, half, half, half)
15    )
16    val (b11, b12, b21, b22) = (
17      subMatriz(m2, 0, 0, half),
18      subMatriz(m2, 0, half, half),
19      subMatriz(m2, half, 0, half),
20      subMatriz(m2, half, half, half)
21    )
22
23    // Paralelizar el c lculo de las submatrices
24    val (c11, c12, c21, c22) = parallel(
25      sumMatriz(multMatrizRecPar(a11, b11, umbral), multMatrizRecPar(a12, b21
26      , umbral)),
27      sumMatriz(multMatrizRecPar(a11, b12, umbral), multMatrizRecPar(a12, b22
28      , umbral)),
29      sumMatriz(multMatrizRecPar(a21, b11, umbral), multMatrizRecPar(a22, b21
30      , umbral)),
31      sumMatriz(multMatrizRecPar(a21, b12, umbral), multMatrizRecPar(a22, b22
32      , umbral))
33    )
34
35    // Combinar las submatrices en la matriz resultante usando Vector.
36    tabulate
37    Vector.tabulate(n, n) { (i, j) =>
38      // Se combinan las submatrices en la matriz resultante usando Vector.
39      tabulate
40      if (i < half && j < half) c11(i)(j)
41      else if (i < half) c12(i)(j - half)
42      else if (j < half) c21(i - half)(j)
43      else c22(i - half)(j - half)
44    }
45  }
46 }

```

1. Umbral para paralelización

- Si $n \leq \text{umbral}$:

La función utiliza la versión secuencial `multMatrizRec`. Esto evita la creación innecesaria de tareas paralelas para matrices pequeñas.

2. Paralelización

La función paralela divide el cálculo de $C_{11}, C_{12}, C_{21}, C_{22}$ en tareas independientes. Estas tareas se ejecutan simultáneamente para aprovechar múltiples núcleos de procesamiento. Por ejemplo, para matrices de tamaño 4×4 :

- El cálculo de los cuadrantes $C_{11}, C_{12}, C_{21}, C_{22}$ se paraleliza utilizando la operación `parallel`. Esto significa que los productos de submatrices y sus sumas se calculan simultáneamente en lugar de secuencialmente.

Por ejemplo: Calcular simultáneamente:

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

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

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

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

Esto reduce el tiempo efectivo de cálculo, aprovechando múltiples núcleos de procesamiento.

- Una vez calculadas, se ensamblan en C .

Este enfoque reduce significativamente el tiempo de ejecución para matrices grandes. La Ley de Amdahl establece la ganancia teórica máxima de la paralelización:

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

donde P es la fracción del programa paralelizable y N es el número de núcleos.

3. Combinar resultados

Las submatrices calculadas en paralelo se ensamblan para formar C .

Hipótesis de Inducción

Supongamos que el algoritmo funciona correctamente para matrices de tamaño $\frac{n}{2}$. Esto significa que para cualquier par de matrices A y B de tamaño $\frac{n}{2} \times \frac{n}{2}$, el algoritmo produce una matriz C tal que:

$$C = A \cdot B \quad (\text{es decir, } C_{ij} \text{ se calcula correctamente para todas las posiciones } i, j).$$

Queremos demostrar que el algoritmo también es correcto para matrices de tamaño $n \times n$.

Caso Base

Cuando el tamaño de la matriz es $n = 1$, las matrices A y B contienen un único elemento, y el producto se calcula directamente.

En este caso, el algoritmo realiza:

$$C = A \cdot B = [A_{11} \cdot B_{11}].$$

Esto es claramente correcto, ya que sigue la definición de la multiplicación matricial:

$$C_{11} = A_{11} \cdot B_{11}.$$

Por lo tanto, el caso base es válido.

Paso Inductivo

Para el caso general ($n > 1$), el algoritmo divide las matrices A y B en cuatro submatrices cuadradas de tamaño $\frac{n}{2} \times \frac{n}{2}$:

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

El objetivo es calcular la matriz C , que también se divide en cuatro submatrices:

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

Cada submatriz de C se calcula como:

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

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

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

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

Aplicando la Hipótesis de Inducción

Por la hipótesis inductiva, sabemos que el algoritmo calcula correctamente las multiplicaciones de matrices de tamaño $\frac{n}{2}$. Por lo tanto, las operaciones recursivas para calcular:

$$A_{11} \cdot B_{11}, A_{12} \cdot B_{21}, A_{11} \cdot B_{12}, \text{ etc.},$$

serán correctas.

Combinando las Submatrices

La suma de las submatrices (por ejemplo, $A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$) también es correcta, ya que opera directamente sobre las matrices de tamaño reducido.

El ensamblaje de C a partir de $C_{11}, C_{12}, C_{21}, C_{22}$ se realiza de manera directa utilizando `Vector.tabulate`, que asegura que cada elemento de C se coloca en la posición correcta:

```
1 Vector.tabulate(n, n) { (i, j) =>
2   if (i < half && j < half) c11(i)(j)
3   else if (i < half) c12(i)(j - half)
4   else if (j < half) c21(i - half)(j)
5   else c22(i - half)(j - half)
6 }
```

Esto garantiza que los elementos de las submatrices calculadas se ubiquen de manera adecuada en C .

Aplicación a la Versión Paralela

En la versión paralela, el cálculo de las submatrices ($C_{11}, C_{12}, C_{21}, C_{22}$) se realiza de forma simultánea mediante la función `parallel`:

```
1 val (c11, c12, c21, c22) = parallel(
2   sumMatriz(multMatrizRecPar(a11, b11, umbral), multMatrizRecPar(a12, b21,
3     umbral)),
4   sumMatriz(multMatrizRecPar(a11, b12, umbral), multMatrizRecPar(a12, b22,
5     umbral)),
6   sumMatriz(multMatrizRecPar(a21, b11, umbral), multMatrizRecPar(a22, b21,
7     umbral)),
8   sumMatriz(multMatrizRecPar(a21, b12, umbral), multMatrizRecPar(a22, b22,
9     umbral))
10 )
```

Aunque los cálculos son paralelos, el proceso lógico no cambia:

- Cada submatriz se calcula correctamente gracias a la hipótesis inductiva.
- La paralelización no altera la estructura de los datos, solo distribuye las tareas.

3.3. Funciones para Multiplicación de matrices usando el algoritmo de Strassen

3.3.1. Función para restar matrices

La función `restaMatriz` toma dos matrices m_1 y m_2 de dimensiones $n \times n$ y devuelve una nueva matriz m de las mismas dimensiones, donde cada elemento $m[i][j]$ es la diferencia entre los elementos correspondientes de m_1 y m_2 . Su implementación es la siguiente:

```
1 def restaMatriz(m1: Matriz, m2: Matriz): Matriz = {  
2   Vector.tabulate(m1.length, m1.length)((i, j) => m1(i)(j) - m2(i)(j))  
3 }
```

Queremos demostrar que la función `restaMatriz` es correcta, es decir, que para toda entrada válida m_1 y m_2 , produce una matriz m tal que:

$$\forall i, j \quad m[i][j] = m_1[i][j] - m_2[i][j].$$

Caso Base: Matrices de dimensión 1×1 Si m_1 y m_2 son matrices de dimensión 1×1 :

$$m_1 = [a], \quad m_2 = [b],$$

la función `restaMatriz` devuelve:

$$m = \text{Vector.tabulate}(1, 1)((i, j) \Rightarrow m_1(i)(j) - m_2(i)(j)) = [a - b].$$

Esto coincide con la definición esperada de la resta de matrices, ya que:

$$m[0][0] = m_1[0][0] - m_2[0][0].$$

Por lo tanto, el caso base es correcto.

Caso Inductivo: Matrices de dimensión $k \times k$ Supongamos que la función `restaMatriz` es correcta para matrices de dimensión $k \times k$, es decir, que:

$$\forall i, j < k \quad m[i][j] = m_1[i][j] - m_2[i][j].$$

Queremos demostrar que también es correcta para matrices de dimensión $(k+1) \times (k+1)$.

Si m_1 y m_2 son matrices de dimensión $(k+1) \times (k+1)$, la función `restaMatriz` calcula:

$$m[i][j] = m_1[i][j] - m_2[i][j] \quad \text{para } i, j \leq k.$$

Por la hipótesis de inducción, sabemos que esto es correcto para todos los elementos en las posiciones i, j con $i, j < k$. Para los nuevos elementos en la fila y columna adicionales ($i = k$ o $j = k$), la función simplemente calcula la diferencia entre los elementos correspondientes de m_1 y m_2 :

$$m[k][j] = m_1[k][j] - m_2[k][j], \quad m[i][k] = m_1[i][k] - m_2[i][k], \quad m[k][k] = m_1[k][k] - m_2[k][k].$$

Esto sigue siendo consistente con la definición de la resta de matrices.

Conclusión Por inducción estructural, hemos demostrado que la función `restaMatriz` es correcta para matrices de cualquier dimensión $n \times n$.

3.3.2. Función aplicando algoritmo de Strassen(*secuencial*)

Se implementó la función `multStrassen(m1, m2)` que tiene como objetivo realizar la multiplicación de dos matrices cuadradas de dimensiones iguales utilizando el algoritmo de Strassen. Los parámetros de entrada son:

- `m1`: Primera matriz de entrada, de dimensión $n \times n$.
- `m2`: Segunda matriz de entrada, de dimensión $n \times n$.

La implementación se muestra a continuación:

```
1 def multStrassen(m1: Matriz, m2: Matriz): Matriz = {
2   // Verificar que las matrices son cuadradas y tienen dimensiones iguales
3   val n = m1.length
4   require(n == m2.length && (n & (n - 1)) == 0)
5
6   if (n == 1) {
7     // Caso base: multiplicación de un único elemento
8     Vector(Vector(m1(0)(0) * m2(0)(0)))
9   } else {
10    // Dividir las matrices en submatrices de tamaño n/2
11    val half = n / 2
12
13    val (a11, a12, a21, a22) = (
14      subMatriz(m1, 0, 0, half),
15      subMatriz(m1, 0, half, half),
16      subMatriz(m1, half, 0, half),
17      subMatriz(m1, half, half, half)
18    )
19
20    val (b11, b12, b21, b22) = (
21      subMatriz(m2, 0, 0, half),
22      subMatriz(m2, 0, half, half),
23      subMatriz(m2, half, 0, half),
24      subMatriz(m2, half, half, half)
25    )
26
27    // Calcular las 7 multiplicaciones de Strassen
28    val p1 = multStrassen(a11, restaMatriz(b12, b22))
29    val p2 = multStrassen(sumMatriz(a11, a12), b22)
30    val p3 = multStrassen(sumMatriz(a21, a22), b11)
31    val p4 = multStrassen(a22, restaMatriz(b21, b11))
32    val p5 = multStrassen(sumMatriz(a11, a22), sumMatriz(b11, b22))
33    val p6 = multStrassen(restaMatriz(a12, a22), sumMatriz(b21, b22))
34    val p7 = multStrassen(restaMatriz(a11, a21), sumMatriz(b11, b12))
35
36    // Combinar los resultados para obtener las submatrices de la matriz
37    // resultante
38    val c11 = sumMatriz(restaMatriz(sumMatriz(p5, p4), p2), p6)
39    val c12 = sumMatriz(p1, p2)
40    val c21 = sumMatriz(p3, p4)
41    val c22 = restaMatriz(restaMatriz(sumMatriz(p5, p1), p3), p7)
42
43    // Construir la matriz resultante combinando las submatrices
44    Vector.tabulate(n, n) { (i, j) =>
45      if (i < half && j < half) c11(i)(j)
46      else if (i < half) c12(i)(j - half)
47      else if (j < half) c21(i - half)(j)
48      else c22(i - half)(j - half)
49    }
50  }
```

Listing 1: Implementación de `multStrassen`

Se asume que las matrices de entrada son cuadradas, de dimensiones iguales, y que n es una potencia de 2, lo que garantiza que el algoritmo puede dividir las matrices de manera

uniforme y de forma correcta.

Demostración

Queremos demostrar que para toda matriz $m1, m2 \in \mathbb{M}_{n \times n}$ con n una potencia de 2, la función `multStrassen` retorna la matriz producto $C = m1 \cdot m2$ calculada correctamente. Formalmente:

$$\forall m1, m2 \in \mathbb{M}_{n \times n}, \text{multStrassen}(m1, m2) = C,$$

donde C es el resultado de la multiplicación clásica de matrices.

Caso base: $n = 1$ Cuando $n = 1$, ambas matrices contienen un único elemento. La implementación realiza directamente la multiplicación:

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

Esto coincide con el producto clásico de matrices escalares.

Caso inductivo: $n = 2^k, k \geq 1$ Asumimos como hipótesis de inducción (HI) que la función es correcta para matrices de dimensión $2^{k-1} \times 2^{k-1}$.

Queremos demostrar que `multStrassen` genera la matriz correcta para $n = 2^k$. La implementación divide cada matriz en 4 submatrices de tamaño $n/2 \times n/2$ y calcula las 7 multiplicaciones necesarias:

$$\begin{aligned} P1 &= \text{multStrassen}(A11, B12 - B22), \\ P2 &= \text{multStrassen}(A11 + A12, B22), \\ &\text{etc.} \end{aligned}$$

Luego, combina estas submatrices para obtener C :

$$C11 = P5 + P4 - P2 + P6, \quad C12 = P1 + P2, \quad C21 = P3 + P4, \quad C22 = P5 + P1 - P3 - P7.$$

Por construcción, C cumple con las reglas de multiplicación de matrices. La hipótesis de inducción garantiza la corrección en subniveles.

Conclusión Por inducción, `multStrassen` es correcta para toda matriz cuadrada de dimensiones iguales, donde n es una potencia de 2.

Análisis de complejidad El algoritmo `multStrassen` divide las matrices en submatrices de tamaño $n/2 \times n/2$ y realiza 7 llamadas recursivas. Por lo tanto, la complejidad se puede describir mediante la relación de recurrencia:

$$T(n) = 7T(n/2) + O(n^2),$$

donde $O(n^2)$ corresponde al costo de sumar y restar matrices. Resolviendo esta recurrencia, obtenemos $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$, lo que mejora el $O(n^3)$ de la multiplicación clásica.

3.3.3. Función aplicando algoritmo de Strassen (*paralela*)

La función `multStrassenPar(m1, m2)` extiende el algoritmo de Strassen para aprovechar la paralelización y mejorar el rendimiento en sistemas con múltiples núcleos de procesamiento. Sus parámetros de entrada son:

- `m1`: Primera matriz de entrada, de dimensión $n \times n$.

- m2: Segunda matriz de entrada, de dimensión $n \times n$.

La implementación se detalla a continuación:

```

1 def multStrassenPar(m1: Matriz, m2: Matriz): Matriz = {
2   // Verificar que las matrices son cuadradas y tienen dimensiones iguales
3   val n = m1.length
4   require(n == m2.length && (n & (n - 1)) == 0)
5
6   if (n == 1) {
7     // Caso base: multiplicación de un único elemento
8     Vector(Vector(m1(0)(0) * m2(0)(0)))
9   } else {
10    // Dividir las matrices en submatrices de tamaño n/2
11    val half = n / 2
12
13    val (a11, a12, a21, a22) = (
14      subMatriz(m1, 0, 0, half),
15      subMatriz(m1, 0, half, half),
16      subMatriz(m1, half, 0, half),
17      subMatriz(m1, half, half, half)
18    )
19
20    val (b11, b12, b21, b22) = (
21      subMatriz(m2, 0, 0, half),
22      subMatriz(m2, 0, half, half),
23      subMatriz(m2, half, 0, half),
24      subMatriz(m2, half, half, half)
25    )
26
27    // Paralelizar las 7 multiplicaciones de Strassen en dos grupos
28    val (p1, p2, p3, p4) = parallel(
29      multStrassenPar(a11, restaMatriz(b12, b22)),
30      multStrassenPar(sumMatriz(a11, a12), b22),
31      multStrassenPar(sumMatriz(a21, a22), b11),
32      multStrassenPar(a22, restaMatriz(b21, b11))
33    )
34
35    val (p5, temp) = parallel(
36      multStrassenPar(sumMatriz(a11, a22), sumMatriz(b11, b22)),
37      parallel(
38        multStrassenPar(restaMatriz(a12, a22), sumMatriz(b21, b22)),
39        multStrassenPar(restaMatriz(a11, a21), sumMatriz(b11, b12))
40      )
41    )
42
43    val (p6, p7) = temp
44
45    // Combinar los resultados de las submatrices
46    val c11 = sumMatriz(restaMatriz(sumMatriz(p5, p4), p2), p6)
47    val c12 = sumMatriz(p1, p2)
48    val c21 = sumMatriz(p3, p4)
49    val c22 = restaMatriz(restaMatriz(sumMatriz(p5, p1), p3), p7)
50
51    // Reconstruir la matriz resultante
52    Vector.tabulate(n, n) { (i, j) =>
53      if (i < half && j < half) c11(i)(j)
54      else if (i < half) c12(i)(j - half)
55      else if (j < half) c21(i - half)(j)
56      else c22(i - half)(j - half)
57    }
58  }
59 }

```

Se asume que las matrices de entrada son cuadradas, de dimensiones iguales, y que n es una potencia de 2. Esto garantiza que las divisiones en submatrices sean uniformes. Además, se utiliza la función `parallel` para ejecutar simultáneamente las operaciones independientes, dividiendo las multiplicaciones de Strassen en dos grupos.

Demostración

Queremos demostrar que para toda matriz $m1, m2 \in \mathbb{M}_{n \times n}$ con n una potencia de 2, la función `multStrassenPar` retorna la matriz producto $C = m1 \cdot m2$. Formalmente:

$$\forall m1, m2 \in \mathbb{M}_{n \times n}, \text{multStrassenPar}(m1, m2) = C,$$

donde C es el resultado de la multiplicación clásica de matrices.

Caso base: $n = 1$ Cuando $n = 1$, ambas matrices contienen un único elemento. La implementación realiza directamente la multiplicación:

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

Esto coincide con el producto clásico de matrices escalares.

Caso inductivo: $n = 2^k, k \geq 1$ Asumimos como hipótesis de inducción (HI) que la función es correcta para matrices de dimensión $2^{k-1} \times 2^{k-1}$.

Para $n = 2^k$, la función divide cada matriz en 4 submatrices de tamaño $n/2 \times n/2$. Las 7 multiplicaciones de Strassen se dividen en dos grupos para ejecutarse en paralelo:

$$\begin{aligned} (p1, p2, p3, p4) = & \text{parallel}(\text{multStrassenPar}(A11, B12 - B22), \\ & \text{multStrassenPar}(A11 + A12, B22), \\ & \text{multStrassenPar}(A21 + A22, B11), \\ & \text{multStrassenPar}(A22, B21 - B11)). \end{aligned}$$

Se utiliza un segundo nivel de paralelización para calcular $p5, p6, p7$. Luego, se combinan los resultados para obtener las submatrices $C11, C12, C21, C22$ de la matriz final C . Por construcción, C cumple con las reglas de multiplicación de matrices, y la hipótesis de inducción garantiza la corrección en subniveles.

Conclusión Por inducción, `multStrassenPar` es correcta para toda matriz cuadrada de dimensiones iguales, donde n es una potencia de 2.

Análisis de complejidad El uso de `parallel` no modifica la relación de recurrencia para el costo secuencial:

$$T(n) = 7T(n/2) + O(n^2),$$

resolviendo, obtenemos $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$. Sin embargo, la paralelización permite reducir el tiempo real de ejecución en sistemas con suficientes núcleos de procesamiento, disminuyendo el factor constante asociado al cálculo de las multiplicaciones.

4. Conclusiones

- **Implementación secuencial vs. paralela:** El grado de pertenencia en los conjuntos difusos es crucial porque facilita la categorización de los elementos, ofreciendo mayor flexibilidad y precisión en la representación de la realidad, esto permite combinar y manipular funciones de pertenencia de manera más efectiva. Se implementaron varias versiones de algoritmos de multiplicación de matrices, incluyendo el algoritmo estándar, una versión recursiva, y el algoritmo de Strassen, tanto en versiones secuenciales como paralelas. Las pruebas demostraron que las versiones paralelas tienden a ser más rápidas, especialmente en matrices de mayor tamaño.
- **Desempeño y utilidad:** La versión paralela del producto punto nos mostró mejoras en el tiempo de ejecución, haciendo más eficientes las operaciones con vectores grandes. Esta técnica puede ser muy útil en aplicaciones de cálculo intensivo donde se manejan grandes cantidades de datos.

- **Condiciones de prueba:** Las pruebas se realizaron utilizando matrices cuadradas con entradas binarias (0 y 1) y dimensiones que son potencias de 2, lo que permitió una comparación directa y uniforme de los resultados.
- **Desempeño y eficiencia:** Al analizar el desempeño, se observó que la versión paralela del algoritmo de Strassen ofreció la mayor mejora en términos de tiempo de ejecución en comparación con las versiones secuenciales y otras versiones paralelas.

En resumen el taller realizado nos demostró que la paralelización de tareas y datos es una estrategia eficaz para acelerar el procesamiento de multiplicación de matrices y el producto punto entre vectores. Observamos que las implementaciones paralelas no solo reducen el tiempo de cálculo, sino que también optimizan el uso de recursos, resultando en una mejora considerable del desempeño en comparación con las versiones secuenciales tradicionales.