Taller #4 **Programación Concurrente**

Carlos Andrés Delgado

@Authors
Carlos Camacho Castaño – 2160331

Juan José Hernández Arenas - 2259500

Santiago Reyes Rodríguez - 2259738

Taller 4 - Multiplicación de matrices en paralelo

1.1 Multiplicación estándar de matrices

1.1.1. Versión estándar secuencial

Se implementó función multMatriz que reciba dos matrices cuadradas (A, B) de la misma dimensión y devuelva la matriz (C) correspondiente a la multiplicación de las matrices de entrada (AB).

Para esta función se emplean las siguientes funciones:

```
def transpuesta(m: Matriz): Matriz =

def prodPunto(v1: Vector[Int], v2: Vector[Int]): Int =

Y la funcion principal:

def multMatriz(m1: Matriz, m2: Matriz): Matriz =
```

Esta función obtiene dos matrices cuadradas y devuelve una matriz correspondiente a la multiplicación de estas, de la siguiente manera:

val m2t = transpuesta(m2): Se calcula la transpuesta de la matriz m2 utilizando la función transpuesta. m2t es la matriz transpuesta de m2. Vector.tabulate(m1.length, m2t.length) { (i, j) => ... }: Se utiliza Vector.tabulate para generar una nueva matriz con las dimensiones adecuadas. La longitud de esta nueva matriz será igual al número de filas de m1 y al número de filas de la transpuesta de m2.

En el bloque de código { (i, j) => ... }, para cada posición (i, j) de la nueva matriz: prodPunto(m1(i), m2t(j)): Se calcula el producto punto entre la fila i de la matriz m1 y la fila j de la transpuesta de m2. Esto se logra utilizando la función prodPunto.

1.1.2. Versión estándar paralela

Se implementó la función multMatrizPar que reciba dos matrices cuadradas (A, B) de la misma dimensión y devuelva la matriz (C) correspondiente a la multiplicación de las matrices de entrada (AB),

modificando ligeramente la función multMatriz para paralelizar algunas operaciones (usando las abstracciones parallel o task).

```
def multMatrizPar(m1: Matriz, m2: Matriz): Matriz
```

Esta función es la paralelización de la función multMatriz y funciona de la siguiente manera:

val m2t = transpuesta(m2): Se calcula la transpuesta de la matriz m2 utilizando la función transpuesta de la misma manera que en la función anterior. val (m1a, m1b) = m1.splitAt(m1.length / 2): Se divide la matriz m1 en dos partes iguales, m1a y m1b.

val top = task(m1a.map(row => m2t.map(col => prodPunto(row, col)))): Se crea una tarea paralela para calcular la multiplicación de la submatriz m1a con la transpuesta de m2. val bot = task(m1b.map(row => m2t.map(col => prodPunto(row, col)))): Similar a la línea anterior, crea una tarea paralela para calcular la multiplicación de la submatriz m1b con la transpuesta de m2. top.join() ++ bot.join(): Espera a que ambas tareas paralelas se completen y luego concatena los resultados obtenidos. La función join() bloquea hasta que la tarea asociada se complete y devuelve el resultado.

1.2 Multiplicación recursiva de matrices

1.2.1 Extrayendo submatrices

Se implementó la función subMatriz que dadas una matriz (A) de dimensión n × n, una posición ((i, j)) de la matriz y una dimensión deseada (l) de la submatriz, devuelva la submatriz de A de dimensión l × l que comienza en la esquina A_{ij} de la matriz. Puede suponer que los índices especifican efectivamente una submatriz de A, es decir, que $1 \leq i, j \leq n \land i + l \leq n \land j + l \leq n$.

```
def subMatriz(m: Matriz, i: Int, j: Int, l: Int): Matriz =
```

La función subMatriz calcula la submatriz de una matriz cuadrada m, junto con las coordenadas (i, j) que especifican la esquina superior izquierda de una submatriz, y el tamaño l de la submatriz así:

m: Es la matriz de entrada desde la cual se extraerá la submatriz. i y j: Son las coordenadas que especifican la esquina superior izquierda de la submatriz en la matriz original. l: Es el tamaño de la submatriz, y se asume que la submatriz será cuadrada de tamaño l x l.

La función utiliza Vector.tabulate(l, l) para generar una nueva matriz de tamaño l x l. En cada posición (row, col) de esta nueva matriz, se asigna el valor correspondiente de la matriz original m usando las coordenadas (i + row, j + col).

1.2.2 Sumando matrices

Implemente la función sumMatriz que dadas dos matrices (A, B) de dimensión n × n, devuelve la matriz A + B de dimensión n × n.

```
def sumMatriz(m1: Matriz, m2: Matriz): Matriz
```

Esta función recibe las matrices m1 y m2 (matrices cuadradas de la misma dimensión, potencia de 2) y devuelve la matriz resultante de la suma de estas, de la siguiente manera:

La función utiliza Vector.tabulate para generar una nueva matriz con las mismas dimensiones que la matriz de entrada m1. La función de tabulación se aplica en cada posición (i, j) de la nueva matriz, donde i representa la fila y j la columna. Dentro del bloque de la función de tabulación, se suma el elemento correspondiente en la posición (i, j) de las matrices m1 y m2. La suma se realiza utilizando el operador +.

1.2.3. Multiplicando matrices recursivamente, versión secuencial

Se implementó la función multMatrizRec que dadas dos matrices (A, B) de dimensión n × n, devuelve la matriz AB de dimensión n × n, calculada usando la idea expresada anteriormente. Use las funciones subMatriz y sumMatriz en su implementación.

```
def multMatrizRec(m1: Matriz, m2: Matriz): Matriz =
```

Esta función recibe las matrices m1 y m2 (matrices cuadradas de la misma dimensión, potencia de 2) y devuelve la multiplicación de estas, así:

m1 y m2: Son las dos matrices de entrada que se multiplicarán. n: Es la dimensión de las matrices de entrada. La función calcula n como el tamaño de una fila (o columna) de la matriz m1. La función utiliza una condición base para el caso en que la dimensión n de las matrices llega a 1. En ese caso, realiza la multiplicación de los elementos correspondientes y devuelve una nueva matriz de 1x1 con el resultado.

En caso contrario (cuando n es mayor que 1), la función divide cada una de las matrices m1 y m2 en cuatro submatrices más pequeñas y realiza operaciones recursivas para calcular las submatrices. Finalmente, se construye la matriz resultante c concatenando las submatrices calculadas en el orden correcto.

1.2.4. Multiplicando matrices recursivamente, versión paralela

Implemente la función multMatrizRecPar que dadas dos matrices (A, B) de dimensión n×n, devuelve la matriz AB de dimensión n×n, calculada usando la idea expresada anteriormente, pero usando paralelización de tareas (utilizando las abstracciones parallel o task). Use las funciones subMatriz y sumMatriz en su implementación.

```
def multMatrizRecPar(m1: Matriz, m2: Matriz, threshold: Int): Matriz =
```

Esta función es la paralelización de la función multMatrizRec, y funciona así:

m1 y m2: Son las dos matrices de entrada que se multiplicarán. threshold: Es un umbral que determina cuándo se debe utilizar la multiplicación de matrices estándar en lugar de la versión paralela. Si la dimensión de las matrices es menor o igual a threshold, se utiliza la función multMatrizRec (la versión no paralela) para la multiplicación de matrices.

La función comienza dividiendo ambas matrices de entrada en submatrices más pequeñas. Luego, realiza operaciones recursivas y paralelas para calcular las submatrices. Estas operaciones son esencialmente llamadas a multMatrizRecPar con submatrices específicas y luego realizan operaciones de suma y resta. Finalmente, se construye la matriz resultante C concatenando las submatrices en el orden correcto.

1.3 Multiplicación de matrices usando el algoritmo de Strassen

1.3.1. Restando matrices

Implemente la función restaMatriz que dadas dos matrices (A, B) de dimensión n×n, devuelve la matriz1 A − B de dimensión n × n.

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

Esta función recibe las matrices m1 y m2 (matrices cuadradas de la misma dimensión, potencia de 2) y devuelve la matriz resultante de la resta de estas, de la siguiente manera:

El funcionamiento es el mismo que el de sumMatriz, la función utiliza Vector.tabulate para generar una nueva matriz con las mismas dimensiones que la matriz de entrada m1. La función de tabulación se aplica en cada posición (i, j) de la nueva matriz, donde i representa la fila y j la columna. Dentro del bloque de la función de tabulación, se suma el elemento correspondiente en la posición (i, j) de las matrices m1 y m2. Aquí el uníco cambio, la resta se realiza utilizando el operador -.

1.3.2. Algoritmo de Strassen, versión secuencial

Implemente la función multStrassen que dadas dos matrices (A, B) de dimensión n×n, devuelve la matriz AB de dimensión n×n, calculada usando el algoritmo de Strassen implementado secuencialmente. Use las funciones subMatriz , sum Matriz y restaMatriz en su implementación.

```
def multStrassen(m1: Matriz, m2: Matriz): Matriz =
```

Esta función recibe las matrices m1 y m2 (matrices cuadradas de la misma dimensión, potencia de 2) y devuelve la multiplicación de las 2 matrices usando el algoritmo de Strassen:

La primera parte de la función es igual a la multiplicación recursiva, m1 y m2: Son las dos matrices de entrada que se multiplicarán. n: Es la dimensión de las matrices de entrada. La función calcula n como el tamaño de una fila (o columna) de la matriz m1. La función utiliza una condición base para el caso en que la dimensión n de las matrices llega a 1. En ese caso, realiza la multiplicación de los elementos correspondientes y devuelve una nueva matriz de 1x1 con el resultado.

En caso contrario (cuando n es mayor que 1), la función divide cada una de las matrices m1 y m2 en cuatro submatrices más pequeñas y realiza operaciones recursivas para calcular las submatrices. Luego, se calculan las submatrices y finalmente, se construye la matriz resultante c concatenando las submatrices en el orden correcto utilizando Vector.tabulate.

1.3.3. Algoritmo de Strassen, versión paralela

Implemente la función multStrassenPar que dadas dos matrices (A, B) de dimensión n × n, devuelve la matriz AB de dimensión n × n, calculada usando el algoritmo de Strassen implementado de forma paralela, usando paralelización de tareas (utilizando las abstracciones parallel o task). Use las funciones subMatriz , sumMatriz y restaMatriz en su implementación.

```
def multStrassenPar(m1: Matriz, m2: Matriz): Matriz =
```

Esta función es la paralelización de la función multStrassen:

m1 y m2: Son las dos matrices de entrada que se multiplicarán. n: Es la dimensión de las matrices de entrada. La función calcula n como el tamaño de una fila (o columna) de la matriz m1. umbral: Es un umbral que determina cuándo se debe utilizar la multiplicación de matrices estándar en lugar de la versión paralela. Si la dimensión de las matrices es menor o igual a umbral, se utiliza la función multStrassen (la versión no paralela) para la multiplicación de matrices.

La función comienza dividiendo ambas matrices de entrada en submatrices más pequeñas y realiza operaciones recursivas y paralelas para calcular las submatrices. Estas operaciones son esencialmente llamadas a multStrassenPar con submatrices específicas y luego realizan operaciones de suma y resta. Finalmente, se construye la matriz resultante concatenando las submatrices en el orden correcto.

1.4. Evaluación comparativa

Para realizar el benchmarking de las soluciones se usó el paquete Benchmark brindado para este taller para las implementaciones de los diferentes algoritmos de multiplicación de matrices.

Benchmarking:

Se implementó el objeto Benchmark importado desde el paquete *Benchmark*

```
object Benchmark
```

Después se proporcionó la función medirTiempo la cual mide el tiempo de ejecución en milisegundos de la siguiente manera:

```
def medirTiempo(funcion: => Unit): Double =
```

val inicio = System.nanoTime(): Obtiene el tiempo actual en nanosegundos antes de ejecutar la función. funcion: Ejecuta la función proporcionada como argumento.

val fin = System.nanoTime(): Obtiene el tiempo actual en nanosegundos después de ejecutar la función. (fin - inicio) / 1e6: Calcula la diferencia entre los tiempos de inicio y fin, y lo divide por 1e6 para obtener el tiempo en milisegundos.

Se invoca la función compararAlgoritmos que nos permitirá comparar el rendimiento entre dos algoritmos en específico teniendo en cuenta su tiempo de ejecución y así definir que implementación (secuencial o paralela) es más eficiente:

```
def compararAlgoritmos(algoritmo1: (Matriz, Matriz) => Matriz, algoritmo2: (Matriz, Matriz) => Matriz)(m1: Matriz, m2: Matriz): (Double, Double, Double) =
```

algoritmo1 y algoritmo2 son tomados como dos matrices de entrada y devuelven una matriz como resultado.

(m1: Matriz, m2: Matriz): Son las dos matrices que se utilizarán como entrada para los algoritmos.

val tiempoAlgoritmo1 = medirTiempo(algoritmo1(m1, m2)) y val tiempoAlgoritmo2 = medirTiempo(algoritmo2(m1, m2)): Miden el tiempo de ejecución del algoritmo1 y 2 utilizando las matrices de entrada m1 y m2.

val relacionRendimiento = tiempoAlgoritmo1 / tiempoAlgoritmo2: Calcula la relación de rendimiento dividiendo el tiempo de ejecución del algoritmo1 entre el tiempo de ejecución del algoritmo2. Una relación mayor que 1 indica que el algoritmo1 es más lento, mientras que una relación menor que 1 indica que el algoritmo2 es más lento.

(tiempoAlgoritmo1, tiempoAlgoritmo2, relacionRendimiento): Devuelve una tupla que contiene el tiempo de ejecución del algoritmo1, el tiempo de ejecución del algoritmo2 y la relación de rendimiento.

De esta manera obtenemos el tiempo de ejecución de cada una de las implementaciones (secuenciales y paraleles) y además tenemos la aceleración del segundo algoritmo con respecto al primero. Para los mismos algoritmos, podemos pasar varias matrices de diferentes tamaños.

Para probarlo con matrices de diferentes tamaños y realizar comparaciones de rendimientos entre las implementaciones se creó el objeto main ubicado en:

app/src/main/scala/taller4/main.scala

Para este apartado emplearemos la siguiente función llamada matrizAlAzar:

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

La función matrizAlAzar crea una matriz cuadrada de enteros con dimensiones long x long y valores aleatorios entre 0 (inclusive) y vals (no inclusivo).

En la multiplicación de producto punto se comparan los tiempos de ejecución de la operación de producto punto para vectores de diferentes longitudes potencias de 10. La cual se evidencia más adelante en el documento.

En él se implementa la clase main:

```
  def main(args: Array[String]): Unit =
```

Esta función cuenta con un calentador para asegurarse de que las mediciones de tiempo siguientes sean más precisas. Se crea un rango de números y se convierte en un array para realizar operaciones, ayudando a que la JVM realice algunas optimizaciones antes de medir tiempos.

Después se realiza la multiplicación de matrices secuenciales y paralelas, se generan matrices aleatorias de tamaños que son potencias de 2 hasta una matriz de tamaño 1024x1024. Luego, se mide el tiempo de ejecución para las funciones de multiplicación de matrices multMatrizRec (recursiva) y

multMatrizRecPar (recursiva paralela). La función compararAlgoritmos2 mide el tiempo de ejecución de ambas funciones y calcula la aceleración.

Comparación multMatrizRec y multMatrizRecPar			
Matriz	Tiempo secuencial	Paralelo	Aceleración
2*2	0,1075	0,1861	0,57764642665234
4*4	0,1075	0,227899	0,471700182975791
8*8	0,706401	0,566	1,248058303886920
16*16	1,141201	0,9307	1,22617492210164
32*32	8,4226	10,3088	0,817030110197113
64*64	65,336101	62,3774	1,04743225911948
128*128	496,7216	454,3886	1,09316474929168
256*256	3.882,3452	3.525,2591	1,10129357583957
512*512	30.798,9578	27.592,6123	1,11620304250786
1024*1024	281.470,8946	289.057,4631	0,973754116504594

Después se implementa la multiplicación de matrices normales y paralelas, similar al bloque anterior, se mide el tiempo de ejecución para las funciones de multiplicación de matrices multMatriz (normal) y multMatrizPar (paralela).

Comparación multMatriz y multMatrizPar			
Matriz	**Tiempo secuencial**	**Tiempo paralelo**	**aceleración**
2x2	0,3266	0,7071	0,461886578984584
4x4	0,2019	0,4692	0,430306905370844
8x8	0,475099	0,624901	0,760278828166381
16x16	0,978499	0,8917	1,089887664
32x32	3,9956	2,6866	1,4942143772161
64x64	19,2017	8,190099	2,34450157440099
128x128	127,862801	76,7009	1,66703129950235
256x256	1.327,5266	732,933901	1,8112500979812
512x512	9.599,6177	6.529,982799	1,47008315266467
1024x1024	74.369,2662	41.045,371001	1,81187949789

Por último, tenemos la multiplicación de los algoritmos de Strassen y Strassen paralelo, aquí, se mide el tiempo de ejecución para las funciones de multiplicación de matrices multStrassen (Strassen) y multStrassenPar (Strassen paralela).

Comparación MultStrassen y MultStrassenPar			
Matriz	Tiempo secuencial	Paralelo	Aceleración
2*2	0,2448	0,3185	0,768602825745682
4*4	0,3777	0,4698	0,803959131545338
8*8	1,2085	0,6855	1,76294675419401
16*16	3,1674	2,9772	1,06388553002821
32*32	19,9033	20,7374	0,959777985668405
64*64	150,022	158,7077	0,945272346584318
128*128	1.201,1175	1.137,1086	1,05629092946795
256*256	7.669,0607	7.677,8407	0,998856449313932
512*512	51.439,7965	53.601,6254	0,959668594303485
1024*1024	220.127,8376	205.451,1894	1,07143618025703

¿Cuál de las implementaciones es más rápida?

Analizando cada versión con la aceleración, se da por concluido que es más rápida la versión paralela, dando un ejemplo con la matriz de 1024*1024, en esta, la multiplicación de matrices y la multiplicación de matrices por el método Strassen tienen como versión más rápida por el lado paralelo y en la multiplicación de matrices recursiva es más rápida la versión secuencial. Analizando todas las tablas, la tabla de multiplicación de matrices por la forma normal es más rápida que las demás formas, ya que las versiones paralelas y secuenciales tienen un tiempo muchísimo más rápido que las otras.

¿De qué depende que la aceleración sea mejor?

La aceleración depende de una combinación de factores que incluyen el tamaño de la matriz, el hardware, la eficiencia de la implementación paralela y las condiciones de carga del sistema, ya que, en algunos casos es posible que la paralelización no proporcione una aceleración significativa debido a factores como la sobrecarga de comunicación entre los procesos o hilos.

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

Es mejor utilizar la versión paralela en la multiplicación de matrices y en la multiplicación de matrices recursivas, ya que encontramos más valores en los que es más rápida la versión paralela. Pero en la multiplicación por el método Strassen, es mejor la versión secuencial. Por ejemplo, en la multiplicación de matrices por forma normal, en la matriz de 64*64 es muchísimo mejor utilizar la versión paralela, ya que, tiene una aceleración del 2,3. En el método Strassen es mejor utilizar la versión paralela para matrices 8*8 porque tiene una aceleración de 1,7 y en la matriz 2*2 es mejor utilizar la versión secuencial. En la multiplicación de matrices por la forma recursiva, en la matriz 8*8 es mejor utilizar la versión paralela porque tiene una aceleración de 1,2 y en la 4*4 es mejor la versión secuencial.

1.5 Implementando el producto punto usando paralelismo de datos

Una manera de mejorar aún más el desempeño de los algoritmos anteriores es tratando de mejorar el desempeño de algoritmos básicos de sus implementaciones. En este caso vamos a estudiar cómo mejorar el desempeño del algoritmo de cálculo del producto punto entre dos vectores de enteros.

Se crea la función prodPuntoPar que utiliza ParVector proveído por Scala

```
def prodPuntoParD(v1: ParVector[Int], v2: ParVector[Int]): Int =
```

v1 y v2: Son dos ParVector que representan el primer y el segundo vector de enteros. (v1 zip v2): Realiza la operación de zip (emparejamiento) entre los dos ParVectors, creando un nuevo ParVector de tuplas (Int, Int), donde cada tupla contiene elementos correspondientes de v1 y v2. .map { case (i, j) => i * j }: Mapea sobre las tuplas resultantes y realiza la multiplicación elemento por elemento. Cada elemento de la tupla se multiplica entre sí, y el resultado es un ParVector de los productos correspondientes. sum: Calcula la suma de todos los elementos en el ParVector resultante de las multiplicaciones, dando como resultado el producto punto total. Retorno: La función devuelve el resultado del producto punto, que es un valor entero (Int).

Y se utiliza la función ya proporcionada para calcular el producto punto usando la colección Vector:

```
def prodPunto(v1: Vector[Int], v2: Vector[Int]): Int =
```

Para probarlo con vectores de diferentes tamaños y realizar comparaciones de rendimientos entre las implementaciones se creó el objeto main ubicado en:

app/src/main/scala/taller4/main.scala

```
print("Multiplicacion de productoPunto: \n")
```

Se realizan tres llamadas a la función compararProdPunto con diferentes tamaños de vectores (100, 1000 y 10000).

Se generan dos vectores aleatorios v1 y v2 de tamaño “tamaño” utilizando la función vectorAlAzar. Se mide el tiempo de ejecución de la operación de producto punto secuencial con obj.prodPunto(v1, v2). Se mide el tiempo de ejecución de la operación de producto punto paralela con obj.prodPuntoParD(v1.par, v2.par). Se calcula la relación de rendimiento dividiendo el tiempo secuencial entre el tiempo paralelo. La función devuelve una tupla con los tiempos secuencial y paralelo, así como la relación de rendimiento.

Comparación ProdPunto y ProdPuntoPar			
Tamaño	Tiempo secuencial	Paralelo	Aceleración
100	0,0597	1,4026	0,0425638100670183
1000	0,6088	3,1337	0,194275138015764
10000	1,6299	2,6724	0,609901212393354

Comparando las funciones **prodPunto y ProdPuntoPar,** se logra ver que en los tamaños 100, 1000 y 10000, es mejor utilizar la función secuencial, ya que tienen como aceleración un número menor que el 1, dando a entender que la versión paralela es más lenta. En nuestra opinión, no sería práctico construir un algoritmo como el de la matriz, implementado vectores, ya que, para los vectores, se ejecuta más rápido el algoritmo que tenemos implementado. Es mejor no usar prodPuntoParD en lugar de prodPunto, ya que en la ejecución logró ver que es más veloz el algoritmo de prodPunto.

A través de la descripción detallada y la evaluación comparativa, se han extraído varias conclusiones significativas:

Rendimiento de algoritmos:

Se demostró que la paralelización puede ofrecer mejoras significativas en el rendimiento, especialmente en la multiplicación de matrices estándar (siendo su versión paralela la más eficiente de todos) y la versión recursiva paralela. Sin embargo, la elección de la implementación (secuencial o paralela) depende del tamaño de las matrices y del algoritmo específico.

Importancia del tamaño de las matrices:

La eficiencia de las implementaciones paralelas se destaca en matrices de mayor tamaño, donde la sobrecarga de la paralelización se ve compensada por la capacidad de realizar operaciones simultáneas. Para matrices pequeñas, la versión secuencial puede ser más eficiente. No obstante, para el algoritmo Recursivo Paralelo, esta diferencia no fue tan significativa, lo cual puede deberse a la complejidad computacional que posee la función tanto en versión secuencial como paralelo.

Comparación entre implementaciones secuenciales:

En matrices pequeñas como la 2x2 y la 4x4 la multMatrizRec es más eficiente, pero a medida que las matrices van aumentando su dimensión la más eficiente en cuanto a tiempo de ejecución es la multMatriz, con diferencias bastantes considerables respecto a las otras implementaciones, con esto se puede dedudcir que el algoritmo de Strassen en las implementaciones secuenciales no destaca mucho por su eficiencia.

Mismo caso que en las versiones paralelas, donde en matrices más pequeñas la implementación recursiva es más eficiente y a medida que aumentan su dimensión la implementación estándar se vuelve la más eficiente.

Las paralelizaciones si sirvieron de mucho, ya que, con ellas pudimos comparar la eficiencia de cada algoritmo para la multiplicación de matrices. No es más eficiente el algoritmo de Strassen, porque al comparar cual es más eficiente, el algoritmo para la multiplicación de matrices por el método estándar es más eficiente entre todos. Podemos concluir que, para la mayoría de las matrices, es más eficiente multiplicarlas por la versión paralela, se resuelve con más velocidad.

Test’s de pruebas:

Se implementaron en total 9 test para comprobación de matrices resultantes para cada uno de los algoritmos, además de productoPunto y porductoPuntoPar.

Los test consisten en definir una matriz de entrada llamada matriz1. val matriz2: Matriz = ...: Define otra matriz de entrada llamada matriz2. Invocación de la Función y Aserción: assert(V== obj.multMatriz(matriz1, matriz2)): Invoca la función multMatriz con las matrices de entrada y compara el resultado con una matriz esperada. La aserción assert verifica si el resultado de la multiplicación es igual a la matriz esperada.

Para el caso del productoPunto y productoPuntoPar se definen los vectores de entrada, posteriormente se define un vector de entrada llamado vector1. val vector2: Vector[Int] = . Después se define otro vector de entrada llamado vector2. Invocación de la Función y Aserción: assert(== obj.prodPunto(vector1, vector2)): Invoca la función prodPunto con los vectores de entrada y compara el resultado con un valor esperado. La aserción assert verifica si el resultado del producto punto es igual al valor esperado. En la paralelización es igual, solo que en el momento de definir los vectores se definen vectores paralelos, además de invocar la función prodPuntoPar en lugar de prodPunto.