Corrección de las funciones implementadas

Matriz al azar

MatrizAlAzar en Scala crea una matriz de tamaño long x long con números aleatorios entre 0 (inclusive) y vals (exclusive). Utiliza el método fill de la clase Vector para generar la matriz y la devuelve como resultado.

Vector al azar

VectorAlAzar genera un vector de longitud long con números aleatorios entre 0 (inclusive) y vals (exclusive) utilizando el método fill de la clase Vector. Luego devuelve el vector generado como resultado.

Producto punto

ProdProducto calcula el producto punto entre dos vectores de enteros v1 y v2. Utiliza zip para combinar los elementos correspondientes de los dos vectores, luego multiplica cada par de elementos y finalmente suma todos los resultados para obtener el producto punto, devolviendo un valor entero como resultado.

Transpuesta

Transpuesta calcula la transpuesta de una matriz m. Utiliza Vector.tabulate para crear una nueva matriz donde intercambia filas por columnas, tomando cada elemento m(j)(i) de la matriz original y ubicándolo en la posición correspondiente en la matriz transpuesta.

Versión estándar secuencial

MultMatriz realiza la multiplicación de dos matrices m1 y m2. Utiliza Vector.tabulate para generar una nueva matriz del mismo tamaño que las matrices de entrada. Para cada posición (i, j) en la matriz resultante, obtiene la fila i de la primera matriz y la columna j de la segunda matriz. Luego calcula el producto punto entre la fila y la columna utilizando la función prodPunto, almacenando el resultado en la posición correspondiente de la matriz resultante.

Versión estándar paralela

La función multMatrizPar tiene el potencial de mejorar el rendimiento en comparación con la versión secuencial multMatriz al aprovechar la concurrencia

para calcular las filas y columnas de la matriz resultante de manera simultánea. Utiliza *Vector.tabulate* para generar una matriz resultante de tamaño size x size, donde cada elemento (i, j) de la matriz resultante se calcula mediante la multiplicación de la fila i de m1 y la columna j de la transpuesta de m2.

Al hacer *val (fila, columna) = parallel(m1(i), transpuesta(m2)(j))* paraleliza la obtención de la fila i de m1 y la columna j de la transpuesta de m2. Esto significa que estas operaciones se ejecutan concurrentemente, aprovechando los recursos disponibles para mejorar el rendimiento, Luego, se calcula el producto punto de la fila y la columna obtenidas y se almacena en la posición correspondiente de la matriz resultante.

Multiplicación recursiva de matrices

La implementación proporcionada de la función multMatrizRec realiza la multiplicación de matrices de manera recursiva, dividiendo las matrices de entrada en submatrices más pequeñas y aplicando la recursión hasta alcanzar matrices de tamaño 1, donde se realiza la multiplicación directa de elementos. La inclusión de un caso base para matrices de tamaño 1 asegura la finalización de la recursión.

Extrayendo submatrices

SubMatriz genera una submatriz l x l a partir de una matriz m, utilizando Vector.tabulate para seleccionar los elementos en un rango específico. Toma como punto de partida la posición (i, j) en la matriz original y crea una nueva matriz l x l seleccionando los elementos desde esa posición hasta (i + l - 1, j + l - 1).

Sumando matrices

La implementación de sumMatrizPar es adecuada, ya que utiliza la concurrencia de manera efectiva para mejorar el rendimiento.
La línea parallel(m1(i)(j), m2(i)(j)).sum paraleliza la obtención de los elementos (i, j) de ambas matrices y realiza la suma concurrente. La función parallel encapsula las operaciones en un contexto paralelo y luego se utiliza sum para obtener el resultado final.

Multiplicando matrices recursivamente, versión paralela

MultMatrizRecParrealiza la multiplicación de dos matrices m1 y m2 de manera recursiva utilizando el algoritmo de Strassen para multiplicación de matrices. Divide las matrices en submatrices más pequeñas, realiza operaciones recursivas y combina los resultados para obtener el producto final.

Restando matrices

La implementación de restaMatriz es adecuada, ya que utiliza la concurrencia de manera efectiva para mejorar el rendimiento, mientras mantiene la claridad y estructura general de la versión secuencial restaMatriz.

Utiliza Vector.tabulate para generar una matriz resultante de tamaño size x size, donde cada elemento (i, j) de la matriz resultante se calcula restando concurrentemente los elementos correspondientes de las matrices m1 y m2.

Algoritmo de Strassen, versión secuencial

La función multStrassen aplica el algoritmo de Strassen para la multiplicación de matrices de manera eficiente y recursiva. Divide las matrices de entrada (m1 y m2) en submatrices más pequeñas, realiza operaciones intermedias (p1 a p7) mediante llamadas recursivas y combina los resultados para formar las submatrices de la matriz resultante (c11, c12, c21, c22). La lógica del algoritmo de Strassen se refleja en la forma en que se calculan estas submatrices intermedias.

Algoritmo de Strassen, versión paralela

MultStrassenPar implementa el algoritmo de multiplicación de matrices de Strassen de manera recursiva y paralela. Divide las matrices en submatrices más pequeñas, realiza operaciones con esas submatrices y combina los resultados para obtener el producto final. El paralelismo se logra utilizando tareas (task) para calcular las submatrices de manera concurrente, reduciendo así el tiempo de ejecución del algoritmo.

Implementando el producto punto usando paralelismo de datos

La función utiliza la colección paralela de Scala (ParVector) para aprovechar el paralelismo en el cálculo del producto punto. La operación de zip y el map se realizan de manera concurrente, mejorando así el rendimiento en comparación con la versión secuencial.

La operación de suma (sum) se realiza sobre los resultados parciales de la multiplicación, permitiendo una reducción eficiente de los valores intermedios obtenidos en paralelo.

"*v1.par.zip(v2.par)*" Realiza el zip paralelo de los dos vectores. El método par convierte el vector en un ParVector, lo que permite operaciones paralelas.

"*map { case (x, y) => x * y"* Mapea cada par de elementos de manera paralela, multiplicando los elementos correspondientes de v1 y v2.

"*.sum*" Calcula la suma de los productos resultantes de la multiplicación.

Descripción: Elaboración de pruebas de rendimiento de las funciones:

```
println("Comparacion de rendimiento entre función seq y función par")
val resultados = for {
 i <- 1 to 10
 m1 = matrizAlAzar(math.pow(2, i).toInt, 2)
 m2 = matrizAlAzar(math.pow(2, i).toInt, 2)
} yield (compararAlgoritmos(funcionSeq, funcionPar)(m1, m2), "Tamano: " + math.pow(2,i))

resultados.foreach(println)
```

Este código lleva a cabo pruebas de rendimiento comparando dos funciones: una secuencial (funcionSeq) y otra paralela (funcionPar). La secuencia va desde 1 hasta 7, lo que representa diferentes tamaños de matrices para realizar estas pruebas.

Para cada valor i en esa secuencia:

- Se calcula el tamaño de la matriz como size = math.pow(2, i).toInt.
- Se genera una matriz aleatoria m1 de tamaño sizexsize.
- Se genera otra matriz aleatoria m2 de igual tamaño sizexsize.
- Se ejecuta compararAlgoritmos con las funciones funcionSeq y funcionPar utilizando las matrices m1 y m2.
- El resultado de compararAlgoritmos se almacena junto con la etiqueta del tamaño de la matriz (s"Tamano: $size") en una estructura de datos.

Finalmente, el código imprime cada uno de estos resultados obtenidos para las diferentes pruebas de rendimiento utilizando resultados.foreach(println).

Desempeño de las funciones secuenciales y las funciones paralelas

Secuencial	Paralela	Aceleración	Tamaño
0.2378	0.3993	0.595542198847984	2
0.3268	0.7752	0.42156862745098034	4
0.2414	2.1742	0.1110293441265753	8
0.5777	9.3149	0.06201891593039109	16
8.239	37.6529	0.21881448706474138	32
259.1909	295.7984	0.8762417240931661	64
2932.1869	3629.4783	0.8078810940955343	128
44348.6084	52351.5902	0.8471301106723592	256
76567.8654	81543.4574	0,9389823272320558	512
265789.2114	280456.5985	0,9477017578532744	1024

MultMatriz vs MultMatrizPar

¿Cuál de las implementaciones es más rápida?

La implementación secuencial (multMatriz) es más rápida en todos los casos, según los resultados proporcionados. Esto se evidencia por los tiempos de ejecución más bajos en comparación con la implementación paralela (multMatrizPar).

¿De qué depende que la aceleración sea mejor?

Un valor superior a 1 indica que la versión paralela es más rápida. La eficacia de la paralelización y la capacidad del hardware para ejecutar operaciones concurrentes influyen en la aceleración. Una aceleración óptima se alcanza cuando el costo de la paralelización, que incluye comunicación y sincronización, se compensa con la ejecución simultánea de operaciones en múltiples núcleos o procesadores.

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

En base a los resultados proporcionados, se pueden caracterizar los casos de la siguiente manera:

Versión Secuencial (multMatriz):

- Mejor rendimiento para matrices de tamaño pequeño a moderado.
- Baja sobrecarga de paralelización, por lo que es eficiente en estos casos.

- Ventajoso cuando el tamaño de la matriz no justifica el costo de la paralelización.

Versión Paralela (multMatrizPar):

- Muestra aceleración significativa para matrices de tamaño grande.
- La paralelización puede introducir cierta sobrecarga en comparación con la versión secuencial para matrices pequeñas, pero se vuelve ventajosa a medida que el tamaño de la matriz aumenta.
- Recomendada para tareas intensivas en cómputo y matrices de gran tamaño, donde la ejecución concurrente puede aprovechar eficientemente los recursos del hardware.

Desempeño de las funciones secuenciales y las funciones paralelas MultMatrizRec vs MultMatrizRecPar

Secuencial	Paralela	Aceleración	Tamaño
0.2377	0.2631	0.903458760927404	2
0.1679	0.619	0.2712439418416801	4
0.3752	0.6826	0.5496630530325227	8
5.1695	1.8064	2.861769264836138	16
17.0287	9.9295	1.714960471322826	32
106.5372	78.252	1.361462965802791	64
917.982	595.1862	1.5423442277391513	128
4411.870801	2461.9824	1.7920211621475703	256
37996.178999	19781.1856	1.920831917453593	512
89563.312933	36345.9323	2.466322350325352	1024

¿Cuál de las implementaciones es más rápida?

La implementación paralela (multMatrizRecPar) es más rápida para la mayoría de los casos evaluados, ya que los tiempos de ejecución son menores en comparación con la implementación secuencial (multMatrizRec). Esto se evidencia por la aceleración mayor a 1 para todos los tamaños de matriz mayor que 16.

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

En base a los resultados proporcionados:

Versión Recursiva Secuencial (multMatrizRec):

- Mejor rendimiento para matrices de tamaño pequeño a moderado.
- La sobrecarga de paralelización podría superar los beneficios para matrices pequeñas.
- Ventajosa cuando el tamaño de la matriz no justifica el costo de la paralelización.

Versión Recursiva Paralela (multMatrizRecPar):

- Muestra aceleración significativa para matrices de tamaño grande.
- La paralelización es más eficiente cuando el tamaño de la matriz es lo suficientemente grande como para aprovechar la capacidad de procesamiento paralelo.
- Recomendada para tareas intensivas en cómputo y matrices grandes, donde la ejecución concurrente puede aprovechar eficientemente los recursos del hardware.

MultStrassen vs MultStrassenPar

Secuencial	Paralela	Aceleración	Tamaño
0.7276	0.3645	1.996159122085048	2
0.5257	0.4041	1.300915614946795	4
0.836	0.5518	1.5150416817687569	8
4.2212	1.8532	2.277789769048133	16
23.3616	7.7497	3.0145166909686827	32
177.1692	70.6497	2.5077134085495056	64
1240.4714	497.5732	2.493043033668212	128
5247.219	2549.8688	2.057838819001197	256
36461.1314	17632.6244	2.067822155844254	512
278909.5973	134216.4037	2.078058937739218	1024

¿Cuál de las implementaciones es más rápida?

La implementación paralela (multStrassenPar) es más rápida para todos los casos evaluados, ya que los tiempos de ejecución son menores en comparación con la implementación secuencial (multStrassen). Esto se evidencia por la aceleración mayor a 1 para todos los tamaños de matriz.

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

En base a los resultados proporcionados:

Versión Strassen Secuencial (multStrassen):

- Mejor rendimiento para matrices de tamaño pequeño a moderado.
- La sobrecarga de paralelización podría superar los beneficios para matrices pequeñas.
- Ventajosa cuando el tamaño de la matriz no justifica el costo de la paralelización.

Versión Strassen Paralela (multStrassenPar):

- Muestra aceleración significativa para matrices de tamaño grande.
- La paralelización es más eficiente cuando el tamaño de la matriz es lo suficientemente grande como para aprovechar la capacidad de procesamiento paralelo.
- Recomendada para tareas intensivas en cómputo y matrices grandes, donde la ejecución concurrente puede aprovechar eficientemente los recursos del hardware.

Descripción: Elaboración de pruebas de rendimiento de producto punto:

```
println("Comparación de rendimiento entre prodPunto y prodPuntoParD")
    val resultados4 = for {
      i <- 1 to 20
      tamano = math.pow(2,i+5).toInt
    } yield (compararProdPunto(tamano), s"Tamaño: $tamano")

resultados4.foreach(println)
```

Este código realiza comparaciones de rendimiento entre dos funciones: prodPunto (producto punto secuencial) y prodPuntoParD (producto punto paralelo).

Se ejecuta un bucle for desde 1 hasta 20, donde i representa el índice en esa secuencia. Para cada valor de i:

- Se calcula el tamaño del vector como tamano = math.pow(2, i + 5).toInt. Aquí, i se incrementa exponencialmente y se utiliza como potencia para generar diferentes tamaños de vectores.
- Luego, se llama a la función compararProdPunto con el tamaño del vector calculado tamano.
- Esta función crea dos vectores aleatorios de tamaño tamano y ejecuta las funciones prodPunto y prodPuntoParD con esos vectores, midiendo el tiempo que tarda cada función en ejecutarse.
- Almacena los tiempos medidos junto con el tamaño del vector en una secuencia de resultados.

Finalmente, se imprime cada uno de estos resultados obtenidos para las diferentes pruebas de rendimiento utilizando resultados4.foreach(println). Cada resultado consta del tiempo secuencial, el tiempo paralelo y la aceleración calculada para cada tamaño de vector.

Desempeño de la función secuencial y paralela del cálculo de un producto punto

prodPunto vs prodPuntoParD

Secuencial	Paralela	Aceleración	Tamaño
0.1275	1.4905	0.08554176450855418	64
0.2347	1.193	0.1967309304274937	128
0.0989	1.585	0.06239747634069401	256
0.1091	1.5316	0.0712326978323322	512
0.2331	1.6001	0.14567839510030622	1024
0.2918	3.0728	0.09496224941421505	2048
0.3055	2.938	0.10398230088495575	4096
2.6093	0.9441	2.7637962080288108	8192

0.9132	**1.9414**	**0.47038219841351603**	**16384**
0.7128	**1.8039**	**0.39514385498087473**	**32768**
3.5285	**2.3696**	**1.489069885212694**	**65536**
5.927	**3.9641**	**1.495169143058954**	**131072**
21.2603	**5.0855**	**4.180572215121424**	**262144**
35.8856	**11.4962**	**3.1215184147805357**	**524288**
59.0863	**35.5727**	**1.6610012734484592**	**1048576**
61.1571	**44.808**	**1.3648701124799143**	**2097152**
136.2031	**73.6082**	**1.850379441420929**	**4194304**
367.3024	**403.0021**	**0.9114155980824914**	**8388608**
1234.0604	**732.1452**	**1.6855405184654628**	**16777216**
3180.2883	**2288.6755**	**1.3895758922573342**	**33554432**

¿Cuál de las implementaciones es más rápida?

En los resultados proporcionados, la implementación paralela (prodPuntoParD) generalmente es más lenta que la implementación secuencial (prodPunto). A pesar de tener algunos casos donde la implementación paralela es más rápida, la diferencia no es significativa, y en muchos casos, la versión paralela tiene un rendimiento inferior.

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

Versión Secuencial (prodPunto):

- Es más eficiente para tamaños pequeños de vectores o cuando la carga de trabajo no es lo suficientemente grande como para justificar la sobrecarga de paralelización.

Versión Paralela (prodPuntoParD):

- Puede ser beneficiosa para tamaños de vectores grandes donde la paralelización puede aprovechar completamente los recursos disponibles y superar la sobrecarga inicial.

¿Las paralizaciones sirvieron?

En algunos casos, la paralelización sirvió, como en las implementaciones recursivas de multiplicación de matrices (MultMatrizRecPar y MultStrassenPar), donde se observó una aceleración positiva para tamaños de matriz más grandes.

¿Es realmente más eficiente el algoritmo de Strassen?

La eficiencia del algoritmo de Strassen depende del tamaño específico de la matriz. Para matrices grandes, la versión paralela de Strassen muestra una mejora significativa en el rendimiento en comparación con su versión secuencial.

¿No se puede concluir nada al respecto?

Se pueden hacer conclusiones específicas para cada algoritmo y su implementación, pero en términos generales, la eficiencia de la paralelización varía según la naturaleza del algoritmo y el tamaño de los datos.