INFORME PFC || TALLER 4

Kevin Steven Ramírez Torres 2259371, Juan David Rojas 2259673, Juan Camilo Diaz 2259583.

Informe - corrección de las funciones implementadas

- **matrizAlAzar**: Esta función genera una matriz cuadrada de tamaño especificado por el parámetro `long`, el cual contiene números enteros aleatorios en el rango de 0 a `vals - 1`. Utiliza la clase `Random` para generar los valores aleatorios y la función `Vector.fill` para crear y llenar la matriz bidimensional. La matriz resultante se devuelve como un vector, representando así una estructura bidimensional.

- **transpuesta**: Esta función recibe una matriz `m` y devuelve su transpuesta. Primero, calcula la longitud de la matriz y luego utiliza el método `tabulate` de la clase `Vector` para generar una nueva matriz bidimensional. En esta nueva matriz, los elementos en la posición `(i, j)` son los mismos que los elementos en la posición `(j, i)` de la matriz original. En esencia, la función intercambia las filas y columnas de la matriz de entrada, proporcionando así la matriz transpuesta como resultado.

- **prodPunto**: Esta función toma dos vectores de enteros, `v1` y `v2`, y realiza el producto punto entre ellos. Para lograr esto, primero utiliza la función `zip` para combinar los elementos correspondientes de ambos vectores en pares. Luego, con `map`, multiplica cada par de elementos y finalmente, con `sum`, suma todos los productos obtenidos. El resultado final es un único valor entero que representa el producto punto entre los dos vectores.

- **subMatriz**: Esta función toma como entrada una matriz `m` y tres parámetros enteros: `i`, `j` y `l`. Esta función devuelve una submatriz de tamaño `l x l` que comienza en la posición `(i, j)` de la matriz original. Utiliza la función `Vector.tabulate` para generar una nueva matriz de tamaño `l x l`, donde cada elemento en la posición `(f, c)` de la nueva matriz es obtenido de la posición `(i + f, j + c)` en la matriz original `m`.

- **restaMatriz**: Esta función toma dos matrices `m1` y `m2` como argumentos y devuelve una nueva matriz que representa la resta de las dos matrices de entrada. La longitud de las matrices se determina como `n`, y luego se utiliza la función `Vector.tabulate` para generar una nueva matriz de tamaño `n` por `n`. En cada posición `(i, j)` de la nueva matriz, se calcula el resultado restando el elemento correspondiente de `m2` de la matriz `m1`.

- **sumMatriz**: Esta función toma dos matrices bidimensionales `m1` y `m2` como entrada y devuelve una nueva matriz resultante de sumar elemento por elemento las correspondientes posiciones de ambas matrices. Utiliza la función `Vector.tabulate` para generar una nueva

matriz del mismo tamaño que las matrices de entrada, donde cada elemento en la posición `(i, j)` es la suma de los elementos correspondientes en las matrices originales. La variable `n` representa la longitud de las filas o columnas de las matrices cuadradas.

- **mulMatriz**: Esta función realiza la multiplicación de dos matrices representadas por los argumentos `m1` y `m2`. Primero, se calcula la matriz transpuesta de `m2` y se almacena en `m2t`. Luego, se obtiene la longitud de las filas de `m1` (y asume que ambas matrices son cuadradas). A continuación, se utiliza `Vector.tabulate` para generar una nueva matriz del mismo tamaño que las matrices de entrada. En cada posición `(i, j)` de esta matriz resultante, se calcula el producto punto entre la fila `i` de `m1` y la columna `j` de la matriz transpuesta `m2t` utilizando la función `prodPunto`. Finalmente, la matriz resultante se devuelve como resultado de la función.

- **multMatrizParalelo**: Esta función realiza la multiplicación de dos matrices `m1` y `m2` de manera paralela. En primer lugar, transpone la matriz `m2` para facilitar el acceso a sus columnas. Luego, divide la matriz `m1` en dos partes, `m1a` y `m1b`. Posteriormente, crea dos tareas paralelas utilizando la función `task`, donde cada tarea calcula el producto punto entre las filas de la matriz `m1` y las columnas de la matriz transpuesta `m2`. Finalmente, se unen los resultados de las tareas superiores e inferiores, generadas por las matrices `m1a` y `m1b`, respectivamente, para obtener la matriz resultante de la multiplicación. Es importante destacar que esta implementación aprovecha la paralelización para mejorar el rendimiento en la multiplicación de matrices.

- **multMatrizRec**: La función toma dos matrices cuadradas como entrada, `m1` y `m2`, representadas como vectores de vectores. La base de la recursión se alcanza cuando ambas matrices son de tamaño 1x1. En este caso, se realiza la multiplicación de los elementos correspondientes y se devuelve una nueva matriz de 1x1 como resultado. En caso contrario, se dividen ambas matrices en cuatro submatrices más pequeñas y se realiza la multiplicación de matrices recursivamente en estas submatrices. Luego, se combinan las submatrices resultantes para formar la matriz final. La función utiliza operaciones como la suma de matrices (`sumMatriz`) y la obtención de submatrices (`subMatriz`) como subrutinas auxiliares. La matriz resultante se representa como un vector de vectores.

- **multMatrizRecPar**: La función toma dos matrices cuadradas como entrada (m1 y m2) y devuelve su producto como una nueva matriz cuadrada. La base del algoritmo es dividir cada matriz en cuatro submatrices más pequeñas, luego calcular recursivamente el producto de estas submatrices utilizando llamadas recursivas a la misma función. La multiplicación se realiza de manera paralela para mejorar el rendimiento. El caso base ocurre cuando las matrices son de tamaño 1x1, y en ese caso, se realiza la multiplicación directa. En cada nivel de recursión, se combinan los resultados intermedios para formar la matriz resultante. La función utiliza operaciones sobre vectores y matrices para realizar las operaciones de suma y multiplicación de manera eficiente.

- **multStrassen**: Esta función implementa el algoritmo de multiplicación de matrices de Strassen, un método eficiente para multiplicar matrices cuadradas. La función toma dos matrices cuadradas como entrada, `m1` y `m2`, representadas como vectores de vectores. La función verifica si el tamaño de las matrices es 1x1, en cuyo caso realiza una multiplicación

trivial y devuelve la matriz resultante. Si las matrices son de tamaño mayor a 1x1, divide cada matriz en cuatro submatrices, realiza siete productos recursivos (denominados p1 a p7) utilizando estas submatrices, y luego combina los resultados para formar la matriz resultante. El algoritmo utiliza operaciones de suma y resta de matrices, así como llamadas recursivas para lograr una multiplicación de matrices más eficiente en términos de complejidad computacional. Finalmente, se construye y devuelve la matriz resultante de acuerdo con la estructura de bloques de la matriz original.

- **multStrassenPar**: Esta función implementa el algoritmo de multiplicación de matrices de Strassen de manera paralela utilizando programación funcional. El algoritmo de Strassen es un método de división y conquista eficiente para la multiplicación de matrices. La función toma dos matrices `m1` y `m2` como entrada y devuelve su producto. Primero, verifica si las matrices son de tamaño 1, en cuyo caso realiza una multiplicación simple. Si las matrices son de tamaño mayor que 1, divide cada matriz en cuatro submatrices, realiza cálculos intermedios (p1 a p7) de manera recursiva utilizando llamadas paralelizadas a la función de multiplicación de Strassen, y luego combina estos resultados para formar la matriz resultante según las fórmulas del algoritmo de Strassen. La implementación utiliza la biblioteca `scala.concurrent` para paralelizar las operaciones y mejorar el rendimiento.

- **vectorAlAzar**: La función vectorAlAzar genera un vector de enteros aleatorios. Toma dos parámetros como entrada: long, que representa la longitud del vector que se va a generar, y vals, que indica el rango máximo de los valores aleatorios que se pueden generar (excluyendo este valor).

- **vectorAlAzarpa**: La función vectorAlAzarPar genera un vector paralelo de enteros aleatorios. Utiliza ParVector.fill para crear el vector, donde cada elemento es un número entero aleatorio generado dentro del rango especificado por vals. La utilización de ParVector permite potencialmente realizar las operaciones de generación de elementos de manera concurrente, mejorando así el rendimiento en situaciones de cómputo intensivo.

- **prodPuntoParD**: La función prodPuntoParD calcula de manera paralela el producto punto entre dos vectores v1 y v2 de tipo ParVector[Int]. Utiliza zip para combinar los elementos correspondientes, map para multiplicar los pares de elementos y sum para obtener la suma de los resultados, devolviendo un valor entero que representa el producto punto de los vectores.

¿por qué están bien implementadas las funciones solicitadas?

Las funciones creadas para la solución de las operaciones de matrices fueron comprobadas mediante un software online llamado https://matrixcalc.org/es/ en el cual realizamos las operaciones como la transpuesta, suma, resta etc. Y una vez se compró la primera forma se usó como forma de comprobación para las siguientes como la forma recursiva y strassen.

Informe - desempeño de las funciones secuenciales y paralelas

Para la realización de los ejemplos de prueba usamos la función matrizAlAzar previamente mencionada y una función para la comparación de los tiempos la cual recibía como parámetros las dos funciones en este caso las funciones que realizaban las multiplicaciones de matrices y otros dos parámetros que eran las matrices.

Tabla I

Tabla de comparación de tiempos de multiplicación de matrices estándar y estándar paralelo

Tamaño matriz	mulMatriz tiempo	multMatrizParalelo tiempo	aceleración
2x2	0,3577	0,251399	1,42283779967303
4x4	0,112901	0,2982	0,37860831656606303
8x8	0,2604	0,592601	0,439418765746261
16x16	1.4764802621275355	1.4764802621275355	1.4764802621275355
32x32	1.4764802621275355	1.4764802621275355	1.4764802621275355
64x64	1.4764802621275355	1.4764802621275355	1.4764802621275355
128x128	1.4764802621275355	1.4764802621275355	1.4764802621275355
256x256	1.4764802621275355	1.4764802621275355	1.4764802621275355
512x512	5694,3832	3802.771	1.4974299530526556
1024x1024	80409.4926	48310.7698	1.6644216793250104

Tabla II

Tabla de comparación de tiempos de multiplicación de matrices recursiva y recursiva paralelo

Tamaño matriz	mulMatrizrec tiempo	multMatrizrecParalelo tiempo	aceleración
2x2	0.4754	0.2369	2.0067539046010974
4x4	1.0258	0.688399	1.4901241866998645
8x8	0.5136	2.9096	0.1765191091558977
16x16	2.0546	2.9968	0.6855979711692473
32x32	18.161199	16.9638	1.070585540975489
64x64	146.7442	98.270699	1.493265047397292
128x128	1270.9628	797.8517	1.5929812520296691
256x256	10243.393	7734.1101	1.324443648662307
512x512	81557.6877	56162.2064	1.4521809759240512

Tabla III

Tabla de comparación de tiempos de multiplicación de matrices métodos Strassen y Strassen paralelo

Tamaño matriz	multStrassen tiempo	multStrassenPar tiempo	aceleración
2x2	0.4069	0.4399	0.9249829506706069
4x4	0.392101	0.5698	0.6881379431379431
8x8	1.845499	1.045	1.7660277511961724
16x16	4.000501	4.9535	0.8076109821338447
32x32	26.0445	23.1852	1.1233243620930595
64x64	206.311099	115.853201	1.7807975715750832
128x128	1963.3321	936.3318	2.096833729239998
256x256	8627.0378	6841.9445	1.2609043817879553
512x512	58198.3663	48136.5471	1.209026608807178

Tabla IV

Tabla de comparación de tiempos de implementaciones de producto punto de vectores

Tamaño vector	Producto punto	Producto punto paralelo	aceleración
10	0.1075	2.0993	0.05120754537226695
100	0.4418	1.3689	0.322740886843451
1000	0.2801	1.636	0.17121026894865526
10000	2.3788	2.7507	0.8647980514050968
100000	8.7636	8.3697	1.0470626187318541
1000000	123.6488	58.69	2.1068120633838814
10000000	2206.1327	3509.3916	0.6286367984695695

Análisis comparativo de las diferentes soluciones

Tabla V

Tabla de comparación de tiempos de multiplicación de matrices secuenciales

Tamaño de matriz	Multmatriz	Multimatrizrec	multStrassen
2x2	0,3577	0.4754	0.4069
4x4	0,112901	1.0258	0.392101
8x8	0,2604	0.5136	1.845499
16x16	1.4764802621275355	2.0546	4.000501
32x32	1.4764802621275355	18.161199	26.0445
64x64	1.4764802621275355	146.7442	206.311099
128x128	1.4764802621275355	1270.9628	1963.3321
256x256	1.4764802621275355	10243.393	8627.0378
512x512	5694,3832	81557.6877	58198.3663

Tabla VI

Tabla de comparación de tiempos de multiplicación de matrices paralelas

Tamaño de matriz	multMatrizParalelo	multMatrizrecParalelo	multStrassenPar tiempo
2x2	0,251399	0.2369	0.4399
4x4	0,2982	0.688399	0.5698
8x8	0,592601	2.9096	1.045
16x16	1.4764802621275355	2.9968	4.9535
32x32	1.4764802621275355	16.9638	23.1852
64x64	1.4764802621275355	98.270699	115.853201
128x128	1.4764802621275355	797.8517	936.3318
256x256	1.4764802621275355	7734.1101	6841.9445
512x512	3802.771	56162.2064	48136.5471

¿Las paralizaciones sirvieron?

Efectivamente las paralelizaciones sirvieron mejorando los tiempos en la gran mayoría de casos, sin embargo, en matrices pequeñas no son tan eficientes o incluso peores que las formas secuenciales.

¿Es realmente más eficiente el algoritmo de Strassen?

Es un poco más eficiente que la forma recursiva pero peor que la secuencial

¿No se puede concluir nada al respecto?

Se concluye que la forma mas eficiente es la mulMatrizParalelo esto quedo demostrado en todas las pruebas de comparación se ve una gran diferencia de tiempo. En las pruebas realizadas se apreciaba una gran velocidad a la hora de ejecutar esta funcion

Evaluación comparativa

¿Cuál de las implementaciones es más rápida?
De las implementaciones secuenciales la más rápida fue la multiplicación de matrices normal y de las implementaciones paralelas también fue la más rápida.

¿De qué depende que la aceleración sea mejor?
Para que la aceleración sea mejor el tiempo de la paralelizada debe ser menor que el de la secuencial

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

La versión secuencial es mejor en matrices de tamaños pequeñas hasta 16x16 se ven unos resultados mejores, sin embargo, la paralelización tiene una gran mejora en los tiempos después de las matrices de tamaño 16x16

¿Será práctico construir versiones de los algoritmos de multiplicación de matrices del enunciado, para la colección ParV ector y usar prodPuntoParD en lugar de prodPunto?

El producto punto no resulto ser eficiente hacerlo de forma paralela, salvo en una situacion especifica y es cuando se pone un valor de 1000000 como tamaño del vector, en ese caso si se muestra una mayor eficiencia sin embargo en el resto de las pruebas resulto ser mas eficiente la forma sin paralelizar. Los resultados de esta prueba están en la tabla IV