INFORME PFC || TALLER 4

Tina María Torres, Juan José Gallego, Marlon Astudillo, Juan José Hernandez.

informe - corrección de las funciones implementadas

¿Por qué están bien implementadas las funciones solicitadas?

-Tener en cuenta que se obvia información si su función anterior de base ya posee esa implementación-

---

Los test…

se ideó un sistema "Test" para comprobar la veracidad de cada resultado en cada función (estándar, recursiva, strassen) en sus dos versiones (secuencial y paralela), evaluando hasta tamaño 16x16 porque la finalidad de los test es mostrar que arroja resultados correctos; dejando así, los resultados de tiempo para el otro algoritmo.

El sistema consta de 15 matrices estáticas, entre ellas un par de matrices binarias; las cuales están organizadas así:

- Matrices: un par de matrices y una tercera que contiene el resultado de multiplicar ambas matrices.
- Sistema assert: se comprueba que la matriz que contiene el resultado de multiplicar ambas matrices es igual al resultado que arroja la función a probar con los argumentos del par de matrices en cuestión, por ejemplo: assert(matrizresultado1_2 == Taller4.mulMatriz(matriz1, matriz2)).
- Total deL test: al final contamos con 30 test para comprobar cada algoritmo suficientes veces.

Preliminares

- **MatrizAlAzar -** *esta es una función para generar matrices aleatorias (generar matrices para sus pruebas)*

1. Tipo de Retorno Adecuado:
   - La función devuelve un tipo `Vector[Vector[Int]]`, que refleja correctamente la representación de una matriz como un vector de vectores.

2. Generación Aleatoria de Valores:
   - Utiliza la clase `Random` para generar números aleatorios, asegurando que los valores de la matriz sean aleatorios en función de la semilla actual.

3. Uso de `Vector.fill`:
   - La función utiliza el método `Vector.fill` para crear una matriz cuadrada de tamaño `long x long` y llenarla con valores aleatorios generados dentro del rango especificado.

4. Parámetros Bien Definidos:
   - La función toma dos parámetros: `long`, que representa la longitud de la matriz cuadrada, y `vals`, que representa el rango de valores aleatorios que pueden ser generados. Ambos parámetros son necesarios y proporcionan flexibilidad para crear matrices de diferentes tamaños y con valores en un rango específico.

5. Concisión y Claridad:
   - La implementación es concisa y clara, utilizando funciones de la biblioteca estándar de Scala. La combinación de `Random` y `Vector.fill` simplifica la generación de matrices aleatorias.

La función cumple con su propósito de generar matrices aleatorias de manera eficiente y representarlas adecuadamente como vectores de vectores.

1.1. Multiplicación estándar de matrices

- **ProdPunto -** *realiza el cálculo del producto punto entre dos vectores `v1` y `v2` este es un algoritmo estándar de ProdPunto (sin restricción en el tamaño de vectores)*

1. Tipo de Retorno Adecuado:
   - La función devuelve un tipo `Int`, que es el resultado del producto punto entre los dos vectores.

2. Operación de Producto Punto:
   - Utiliza la función `zip` para combinar los elementos correspondientes de los dos vectores `v1` y `v2`. Luego, utiliza `map` para multiplicar cada par de elementos y `sum` para calcular la suma de los resultados. Esto refleja correctamente la operación de producto punto entre dos vectores.

3. Desestructuración de Tuplas:
   - La función utiliza la desestructuración de tuplas en el `case (i, j) => i * j` dentro de `map`, lo que mejora la claridad y legibilidad del código.

4. Concisión y Claridad:
   - La implementación es concisa y clara, utilizando funciones de orden superior de Scala (`zip`, `map`, y `sum`) que expresan de manera declarativa la operación de producto punto.

La función `prodPunto` está correctamente implementada y cumple con su propósito de calcular el producto punto entre dos vectores.

- **transpuesta -** *realiza la operación de transposición de una matriz representada, esta recibe una matriz (B) y devuelve su transpuesta (BT).*

1. Uso de `Vector.tabulate`:
   - Utiliza el método `Vector.tabulate` para crear una nueva matriz con las dimensiones transpuestas. El índice `(i, j)` se intercambia para obtener la transpuesta.

2. Manejo de Dimensiones:
   - La función toma la longitud `l` de la matriz original `m` y utiliza esta información para crear la nueva matriz transpuesta con las dimensiones correctas.

3. Concisión y Claridad:
   - La implementación es concisa y clara, utilizando funciones de la biblioteca estándar de Scala (`Vector.tabulate`). La expresión `m(j)(i)` dentro de la función de tabulación refleja la correcta transposición de los elementos.

La función `transpuesta` cumple con su propósito de devolver la transpuesta de la matriz de entrada.

1.1.1. Versión estándar secuencial

- **mulMatriz -** *realiza la multiplicación de dos matrices (el algoritmo estándar de multiplicación de matrices.)*

1. Uso de `transpuesta`:
   - Utiliza la función `transpuesta` para obtener la transpuesta de la segunda matriz `m2`. Esto es esencial para la correcta multiplicación de matrices, ya que permite que los elementos de las filas de la primera matriz coincidan con los elementos de las columnas de la segunda matriz.

2. Uso de `Vector.tabulate` y `prodPunto`:
   - Utiliza el método `Vector.tabulate` para crear una nueva matriz resultante. La expresión `prodPunto(m1(i), m2t(j))` dentro de la función de tabulación refleja correctamente la multiplicación de las filas de la primera matriz por las columnas de la segunda matriz.

La función `mulMatriz` está correctamente implementada y cumple con su propósito de realizar la multiplicación de dos matrices.

1.1.2. Versión estándar paralela

- **mulMatrizPar -** *una versión paralelizada de la multiplicación de matrices.*

1. Uso de `task` y `join`:
   - Utiliza la función `task` para crear tareas paralelas y `join` para esperar a que finalicen las tareas. La multiplicación de matrices se realiza en paralelo para las submatrices `a` y `b`.

2. Concisión y Claridad:
   - La implementación es concisa y clara, utilizando funciones de la biblioteca estándar de Scala (`transpuesta`, `splitAt`, `task`, y `join`). La estructura de la función refleja de manera efectiva la estrategia de paralelización.

3. Manejo de Paralelismo:
   - Divide la tarea en tareas paralelas para procesar submatrices independientes, lo que podría mejorar el rendimiento al aprovechar los recursos del sistema en tareas concurrentes.

Está correctamente implementada y cumple con su propósito de realizar una multiplicación de matrices en paralelo.

1.2.1. Extrayendo submatrices

- **SubMatriz -** *Se encarga de extraer una submatriz de la matriz original `m` a partir de una posición de inicio `(i, j)` y con una longitud `l`.*

1. Uso de `Vector.tabulate`:
   - Utiliza el método `Vector.tabulate` para crear una nueva matriz que representa la submatriz. La expresión `(f, c) => m(i + f)(j + c)` dentro de la función de tabulación genera los elementos de la submatriz correctamente.

2. Manejo de Índices:
   - La función utiliza los índices `i` y `j` para determinar la posición de inicio de la submatriz y `l` para la longitud de la submatriz en ambas dimensiones. Esto garantiza que la submatriz se extraiga correctamente de la matriz original.

1.2.2. Sumando matrices

- **sumMatriz -** *se encarga de realizar la suma de dos matrices `m1` y `m2`.*

1. Uso de `Vector.tabulate`:
   - Utiliza el método `Vector.tabulate` para crear una nueva matriz que representa la suma de las dos matrices originales. La expresión `(i, j) =>

m1(i)(j) + m2(i)(j)` dentro de la función de tabulación realiza la suma de los elementos correspondientes correctamente.

2. Manejo de Dimensiones:
   - La función utiliza la longitud `n` de las matrices originales para crear la nueva matriz resultante con las dimensiones correctas.

1.2.3. Multiplicando matrices recursivamente // versión secuencial

- **multMatrizRec -** *Implementa la multiplicación de matrices utilizando el enfoque de dividir y conquistar.*

1. Manejo de Caso Base:
   - La función verifica si la longitud de las matrices es igual a 1 (`n == 1`). En caso verdadero, se trata de matrices de 1x1 y se realiza la multiplicación directa, evitando la recursión adicional. Esto asegura un caso base bien definido.

2. Dividir y Conquistar:
   - Divide las matrices originales en submatrices más pequeñas (`a11`, `a12`, `a21`, `a22`, `b11`, `b12`, `b21`, `b22`). Luego, realiza llamadas recursivas a `multMatrizRec` para calcular las submatrices intermedias (`c11`, `c12`, `c21`, `c22`).

3. Combinación de Submatrices:
   - Combina las submatrices intermedias para obtener la matriz resultante (`Vector.tabulate(n, n)`). La lógica de combinación está basada en la posición de las submatrices dentro de la matriz resultante.

4. SubMatriz - SumMatriz:
   - Utiliza SubMatriz y SumMatriz en su implementación para los diferentes tipos de submatrices.

1.2.4. Multiplicando matrices recursivamente // versión paralela

- **mulMatrizRecPar** - *es una implementación de la multiplicación de matrices utilizando el enfoque de dividir y conquistar y paralelización.*

1**.** Abstracción Parallel
  - Utiliza la función `parallel` para ejecutar de manera paralela las llamadas a `multMatrizRecPar`. Esto puede mejorar el rendimiento al aprovechar múltiples núcleos del procesador.

2. Respetando el umbral
  - Solo se paraleliza los llamados recursivos, es decir, se respeta el umbral sin paralelizar.

1.3.1. Restando matrices

- **restaMatriz -** *Restar matrices de la misma dimensión*

1. Manejo de Dimensiones:
  - Utiliza la longitud `n` de una de las matrices (se asume que ambas matrices tienen la misma dimensión) para asegurarse de que las matrices tengan el mismo tamaño y, por lo tanto, puedan ser restadas.

2. Uso de `Vector.tabulate`:
  - Utiliza el método `Vector.tabulate` para generar una nueva matriz que represente la resta de las matrices originales. Este enfoque es idóneo para crear matrices en Scala.

3. Operación de Resta:
  - Realiza la operación de resta correctamente, restando cada elemento correspondiente de las matrices originales y colocando el resultado en la posición adecuada de la nueva matriz.

1.3.2. Algoritmo de Strassen // versión secuencial

- **multStrassen -** *un conocido algoritmo más eficiente para la multiplicación de matrices denominado el algoritmo de Strassen*

1. Manejo del caso base:

La función maneja correctamente el caso base cuando la dimensión de las matrices es 1. En este caso, simplemente multiplica los elementos y devuelve una matriz de 1x1.

2. División de las matrices en submatrices:
La función divide las matrices de entrada en submatrices más pequeñas, como se espera en el algoritmo de Strassen. Las submatrices a11, a12, a21, y a22 representan las subdivisiones de la matriz m1, y b11, b12, b21, y b22 representan las subdivisiones de la matriz m2.

3. Recursividad:
La función realiza llamadas recursivas para multiplicar las submatrices de acuerdo con la lógica del algoritmo de Strassen.

5. `subMatriz`, `sumMatriz`, y `restaMatriz`:
Es importante señalar que la correcta implementación y eficiencia del algoritmo de Strassen dependen en gran medida de la implementación de las funciones auxiliares como `subMatriz`, `sumMatriz`, y `restaMatriz`, las cuales ya se encuentran implementadas.

1.3.3. Algoritmo de Strassen //versión paralela

- **multStrassenPar -** *algoritmo de multiplicación de matrices, pero en su versión paralela, sigue el enfoque de paralelización utilizando el modelo de concurrencia de tareas (`task`).*

1. Llamadas Recursivas:
Se realizan llamadas recursivas de manera paralela utilizando `task` para las operaciones de multiplicación de las submatrices, lo cual es consistente con el enfoque de paralelización del algoritmo de Strassen.

2. Cálculos de Submatrices Resultantes:
Las submatrices resultantes `c11`, `c12`, `c21`, y `c22` se calculan correctamente utilizando las fórmulas del algoritmo de Strassen.

3. Construcción de la Matriz Resultante:
La construcción de la matriz resultante se realiza correctamente, combinando las submatrices resultantes de acuerdo con las reglas del algoritmo de Strassen.

4. Uso de `join`:
El uso de `join` está en línea con la lógica del modelo de concurrencia de tareas que se utiliza para realizar las operaciones de manera paralela.

informe - desempeño de las funciones secuenciales//paralelas

Los test…

se ideó un sistema “Test” para comprobar la veracidad de cada resultado en cada función (estándar, recursiva, strassen) en sus dos versiones (secuencial y paralela), evaluando hasta tamaño 16x16 porque la finalidad de los test es mostrar que arroja resultados correctos; dejando así, los resultados de tiempo para el otro algoritmo.

El sistema consta de 15 matrices estáticas, entre ellas un par de matrices binarias; las cuales están organizadas así:

- Matrices: un par de matrices y una tercera que contiene el resultado de multiplicar ambas matrices.
- Sistema assert: se comprueba que la matriz que contiene el resultado de multiplicar ambas matrices es igual al resultado que arroja la función a probar con los argumentos del par de matrices en cuestion, por ejemplo: assert(matrizresultado1_2 == Taller4.mulMatriz(matriz1, matriz2))
- total, de los test: al final contamos con 30 test para comprobar cada algoritmo suficientes veces.

Las pruebas se generaron por medio de una función de matrices aleatorias la cual se le dio por orden que sus valores fueran de hasta 8 *(para tener un aproximado a la complejidad de los cálculos simples)* y con el método de comparar algoritmos en cada llamado se efectuó con las mismas dos matrices tanto para la secuencial como para la concurrente o paralela (debido a la naturaleza del algoritmo de comparación). También, el resultado mostrado en las tablas es derivado de varias pruebas y el promedio de todas ellas en cada caso

MULMATRIZ

Tamaño de la matriz	**Tiempo - mulMatriz**	**Tiempo - multMatrizPar**	**aceleración** ***(paralelo con respecto al secuencial)***
2x2	0.3266	0.7071	0.46188657898458496
4x4	0.2019	0.4692	0.430306905370844
8x8	0.475099	0.624901	0.7602788281663815
16x16	0.978499	0,8917	1.089887664
32x32	3.9956	2.6866	1.4942143772161
64x64	19.2017	11.190099	1.715954434629
128x128	127.862801	76.7009	1.6670312995023526
256x256	1327.5266	741.933901	1.7912500979812092
512x512	9599.6177	6529.982799	1.8700831526646723
1024x1024	74369.2662,	41045.371001	1.8118794978899841

MULMATRIZREC

Tamaño de la matriz	**Tiempo - multMatrizRec**	**Tiempo - multMatrizRecPar**	**aceleración** ***(paralelo con respecto al secuencial)***
2x2	0.1481	0.11159186	0.753497346840328
4x4	0.1176	0.1363	0.8628026412325751
8x8	0.6063	0.4214	1.4387755102040816
16x16	2.5014	1.9711	1.2690375932220586
32x32	6.9713	4.084	1.7069784524975515
64x64	48.3757	25.424	1.902757237256136
128x128	391.7189	204.3917	1.916510797649807
256x256	3243.9664	1632.1425	1.987550964453165
512x512	26971.6072	13773.9292	1.9581636298812977
1024x1024	191564.555	100814.272	1.9001729735250183

MULMATRIZSTRASSEN

Tamaño de la matriz	**Tiempo de la funcion multStrassen**	**Tiempo de la funcion multStrassenPar**	**aceleracion** ***(paralelo con respecto al secuencial)***
2x2	0.0783	0.1126	0.62166962
4x4	0.1136	0.2118	0.5363550519357886
8x8	0.6715	0.634	1.059148264984227
16x16	2.6881	1.5706	1.7115115242582453
32x32	6.9268	3.5926	1.9280743751043812
64x64	52.005	27.2645	1.9074254066643437
128x128	386.8074	190.4424	2.0310991669922243
256x256	2774.7921	1292.803	2.1463379184608944
512x512	19373.2635	9549.492	2.0287218943164724
1024x1024	133632.1097	55219.8347	2.421957110297

análisis comparativo de las diferentes soluciones

VERSIÓN SECUENCIAL VS VERSION PARALELA

mulMatriz vs multMatrizPar

En las matrices más pequeñas, hasta tamaño 8, se ve que es más rápida la versión secuencial, debido a que son más simples los cálculos que la paralelización misma, llevando así a retardarse por tener que administrar las tareas.
a partir de tamaño 16 se nota como la curva de aceleración de la paralela sobrepasa la versión secuencial siendo más eficiente separar el problema en varios hilos, llegando a tener, en promedio, 1.8 de aceleración. para

este caso se utilizaron 2 tareas, podríamos intuir así, que utilizar 4 hilos podría mejorar el rendimiento de la función paralela.

multMatrizRec vs multMatrizRecPar

en este caso, se percibe que la velocidad de ejecución de la función paralela supera a la secuencial desde tamaño 8, pero no solo eso, también podemos evidenciar que la diferencia de velocidades desde tamaño 2 no es muy grande, teniendo como aceleración 1,75 y 1,8 en los casos de tamaño 2 y 4 respectivamente. teniendo como aceleración al final de la tabla con el máximo tamaño de en promedio 1,95. Concluyendo así, que paralelizar funciones recursivas tiene mucha ganancia en términos de eficiencia a la hora de programar aun desde cálculos simples.

multStrassen vs multStrassenPar

Estas dos funciones se comportaron según lo esperado, llegando la versión paralela a ser mejor desde tamaño 8, demostrando nuevamente que el paradigma concurrente supera en eficiencia al secuencial a medida que la complejidad de los cálculos aumenta, llegando a tener un promedio de aceleración de 2.4, siendo la que más supero en aceleración con respecto a la otra. concluyendo así que la función que mayor aprovecho la paralelización fue el algoritmo de Strassen.

Comparación entre secuenciales

entre las secuenciales, el algoritmo de strassen ganó en velocidad en los tamaños más pequeños, pero rápidamente fue superado por el algoritmo estándar y luego por el algoritmo recursiv. podrías intuir que esto ocurre porque el dividir tantas veces la matriz para luego empezar a hacer los cálculos le tomaría casi el mismo tiempo que se tarda el algoritmo estándar en solo hacer las multiplicaciones, teniendo así unos resultados finales en que la versión estándar retorna el mismo resultado en casi la mitad de tiempo.

comparación entre paralelas
en este caso, en las matrices más pequeñas empezó ganando en velocidad el algoritmo recursivo, pero en la resolución de matrices más grandes termino, específicamente desde tamaño 64, supera en velocidad el algoritmo paralelo de la multiplicación estándar de matrices. Podemos concluir que para matrices pequeñas es mejor el algoritmo recursivo y en matrices de tamaños grandes es mejor utilizar el sistema estándar.

P R E G U N T A S

1.4 Evaluación comparativa - *Benchmarking*

- ¿Cuál de las implementaciones es más rápida?
  para matrices de tamaños grandes como por ejemplo tamaño 512 es mejor utilizar el algoritmo estándar paralelo y para matrices de tamaño 2 o 4 es mejor el algoritmo multStrassen
- ¿De qué depende que la aceleración sea mejor?
  la aceleración depende que tanta ventaja tiene el sistema paralelo ante el secuencial
- ¿Puede caracterizar los casos en que es mejor usar la versión secuencial/paralela de cada algoritmo de multiplicación de matrices?

  mulMatrizRec
  Tamaño pequeño (2x2, 4x4):
  La versión secuencial tiende a ser más rápida para matrices pequeñas debido a la sobrecarga de la paralelización.
  Mejor uso de la versión paralela:
  Tamaño mediano (8x8, 16x16):
  La versión paralela muestra una aceleración significativa para matrices medianas, siendo preferible en estos casos.
  Tamaño grande (32x32 en adelante):
  La versión paralela sigue siendo favorable para matrices grandes, donde la paralelización ofrece mejoras considerables.

  mulMatrizStrassen

Mejor uso de la versión secuencial:

Tamaño pequeño (2x2, 4x4):

La versión secuencial tiende a ser más rápida para matrices pequeñas debido a la sobrecarga de la paralelización.

Mejor uso de la versión paralela:

Tamaño mediano a grande (8x8 en adelante):

La versión paralela muestra una aceleración significativa para matrices de tamaño mediano a grande, siendo preferible en estos casos.

mulMatriz

Pequeñas matrices (2x2, 4x4):

En general, la versión paralela tiene una aceleración menor en matrices pequeñas. Esto puede deberse a la sobrecarga de la paralelización y la comunicación entre los hilos paralelos.

Matrices medianas (8x8, 16x16):

La versión paralela comienza a mostrar una aceleración significativa, siendo más rápida que la secuencial. Esto puede ser beneficioso para tamaños de matrices medianos.

Matrices grandes (32x32 en adelante):

La versión paralela demuestra una aceleración considerable en comparación con la secuencial. A medida que el tamaño de la matriz aumenta, la paralelización se vuelve más eficiente.