Taller #4 matrices.

Manuel Alexander Serna Jaraba - 202259345

Santiago Villa Salazar - 202259527

Universidad del valle – Tuluá

Ingeniería en Sistemas

Programación funcional y concurrente.

Profesor: Carlos Andres Delgado Saavedra

6 de Diciembre de 2023

Entrega

Informe de corrección de las Funciones Implementadas

En esta parte, se presenta una explicación sobre la validez de las funciones creadas para multiplicar matrices. En total, se han desarrollado seis funciones para este propósito.

Función multMatriz

Esta función se encarga de multiplicar dos matrices de manera secuencial.

```
val size = m1.length
```

Obtiene el tamaño de las matrices de entrada.

```
Vector.tabulate(size, size) { (i, j) =>
```

Utiliza Vector.tabulate para crear una nueva matriz del mismo tamaño.

```
Vector.tabulate(size, size) { (i, j) =>
    val fila = m1(i)
    val columna = transpuesta(m2)(j)
    prodPunto(fila, columna)
```

Calcula cada elemento como el producto punto entre una fila de la primera matriz y una columna de la segunda matriz.

Para calcular el producto punto, selecciona la fila correspondiente de la primera matriz y la columna correspondiente de la segunda matriz. Luego, realiza el producto punto de estos dos vectores.

Función multMatrizPar

Esta función se encarga de multiplicar dos matrices de manera paralela, aprovechando el concepto de paralelismo. Aquí está la explicación simplificada:

Como en la función multMatriz, obtiene el tamaño de las matrices de entrada.

Utiliza Vector.tabulate para crear una nueva matriz del mismo tamaño.

```
Vector.tabulate(size, size) { (i, j) =>
    val (fila, columna) = parallel(m1(i), transpuesta(m2)(j))
    prodPunto(fila, columna)
```

En este caso, se intenta realizar la multiplicación de las matrices en paralelo utilizando la función parallel para distribuir el cálculo del producto punto de cada fila y columna.

Función multMatrizRec

Esta función se encarga de multiplicar dos matrices de forma recursiva:

```
if (m1.length == 1) { Vector(Vector(m1(0)(0) * m2(0)(0)))}
```

Verifica si la longitud de la matriz es de 1. En caso de que esto sea afirmativo, calcula el producto de los elementos y devuelve una matriz de 1x1 con el resultado.

```
val m1SubMatrices = Vector(

    subMatriz(m1, 0, 0, m1LengthHalf),

    subMatriz(m1, 0, m1LengthHalf, m1LengthHalf),

    subMatriz(m1, m1LengthHalf, 0, m1LengthHalf),

subMatriz(m1, m1LengthHalf, m1LengthHalf, m1LengthHalf)

)
```

Si la longitud es mayor que 1, dividirá ambas matrices en submatrices y realizará la respectiva multiplicación recursiva de estas submatrices.

```
val m1SubMatrices = Vector(

    subMatriz(m1, 0, 0, m1LengthHalf),

    subMatriz(m1, 0, m1LengthHalf, m1LengthHalf),

    subMatriz(m1, m1LengthHalf, 0, m1LengthHalf),

    subMatriz(m1, m1LengthHalf, m1LengthHalf, m1LengthHalf)

)
```

Después de combinar y sumar las submatrices resultantes, la multiplicación se efectúa de manera recursiva. En cada paso, se envía un vector cada vez más pequeño a la función mulMatrizRec hasta que el vector alcanza un tamaño de 1. Al final del cálculo, se obtienen cuatro vectores correspondientes a la multiplicación de los vectores originales

```
val sumVector1y2 = vector1.zip(vector2).map { case (v1, v2) => v1 ++ v2 }

    val sumVector3y4 = vector3.zip(vector4).map { case (v3, v4) => v3 ++ v4 }
```

En la etapa final, se suman los vectores paralelos entre sí, generando un nuevo vector del mismo tamaño que contiene los datos combinados de los dos vectores originales. Este vector resultante es el vector final que representa la suma de los dos vectores iniciales .

Función multMatrizRecPar

Esta función es similar a multMatrizRec, pero en este caso emplea paralelismo para ejecutar las operaciones de multiplicación de submatrices de manera concurrente. A continuación, se detalla su funcionamiento:

Asimismo que en la funcion multMatrizRec, se verifica si la longitud de la matriz es 1. Si es así,se realizará el cálculo de manera similar.

```
val (c1, c2, c3, c4) = parallel(
        sumMatriz(
                multMatrizRecPar(m1SubMatrices(0), m2SubMatrices(0)),
                multMatrizRecPar(m1SubMatrices(1), m2SubMatrices(2))
        ),
        sumMatriz(
                multMatrizRecPar(m1SubMatrices(0), m2SubMatrices(1)),
                multMatrizRecPar(m1SubMatrices(1), m2SubMatrices(3))
        ),
        sumMatriz(
                multMatrizRecPar(m1SubMatrices(2), m2SubMatrices(0)),
                multMatrizRecPar(m1SubMatrices(3), m2SubMatrices(2))
        ),
        sumMatriz(
                multMatrizRecPar(m1SubMatrices(2), m2SubMatrices(1)),
                multMatrizRecPar(m1SubMatrices(3), m2SubMatrices(3))
        )
)
```

Se hace uso de la función parallel para llevar a cabo las operaciones de multiplicación de las submatrices de forma paralela.

Nuevamente, se combina y suma las submatrices resultantes, para así obtener la matriz producto final.

Función multStrass

El algoritmo de Strassen es un método para multiplicar dos matrices cuadradas que tienen el mismo tamaño, el cual debe ser una potencia de 2. Esta función aplica este método y devuelve el resultado como una nueva matriz. El algoritmo funciona de la siguiente manera: Si el tamaño de la matriz es 1, solo hace el producto de los únicos elementos y devuelve una matriz de 1x1 con ese valor.

```
val m1_1 = subMatriz(m1, 0, 0, m1.length / 2)
```

Si el tamaño de la matriz es mayor que 1, la función consiste en dividir cada matriz en submatrices de igual tamaño y realizar ciertas operaciones como indica Strass sobre ellas para obtener el resultado final.

```
val s1 = restaMatriz(m2_2, m2_4)

…

val p1 = multStrass(m1_1, s1)
```

El proceso consiste en realizar primero las adiciones, después las multiplicaciones, que se hacen de forma recursiva usando la misma función. Finalmente, se unen las submatrices siguiendo el método de multMatrizRec, para obtener los vectores que forman la matriz final producto.

Función multStrassPar

Esta función de cierta forma, hace lo mismo que multStrass, pero usa paralelismo para multiplicar las submatrices al mismo tiempo. El algoritmo funciona de esta manera: Primero, comprueba si el tamaño de la matriz es 1. Si es así, hace el cálculo de la misma forma que multStrass.

```
val (m1SubMatrices,m2SubMatrices) = parallel(

        Vector(

        subMatriz(m1, 0, 0, m1.length / 2),
```

Con la abstracción parallel, crea dos vectores paralelos que contienen las submatrices de cada matriz.

```
val s1 = task {

        restaMatriz(m2SubMatrices(1), m2SubMatrices(3))

        }

        val s2 = task {

        sumMatriz(m1SubMatrices(0), m1SubMatrices(1))
```

Con la abstracción task, crea otro vector paralelo que tiene las sumas y las diferencias de las submatrices.

Después, multiplica las submatrices usando el paralelismo y junta las submatrices resultantes para obtener la matriz final que es el producto.

Las dos funciones usan el algoritmo de Strassen de forma eficaz, aprovechando el paralelismo para mejorar la velocidad en matrices grandes

Argumentación de la Corrección

Se han implementado las funciones para multiplicar matrices cuadradas de tamaños que son potencias de 2, tal como se indica en el problema. Se han aplicado los métodos apropiados para partir las matrices en submatrices y realizar las operaciones de multiplicación de forma eficaz. Se ha comprobado que las funciones dan resultados exactos al contrastarlas con implementaciones convencionales de la multiplicación de matrices y al hacer pruebas con matrices de diferentes tamaños y valores.

Análisis comparativo de las diferentes soluciones.

Conclusiones

Las paralelizaciones pueden servir para mejorar el rendimiento de los algoritmos, siempre que se haga un uso adecuado de los recursos y se eviten los cuellos de botella. El algoritmo de Strassen es más eficiente para matrices grandes, ya que reduce el número de operaciones aritméticas necesarias. Sin embargo, el algoritmo de Strassen también tiene algunas desventajas, como la necesidad de dividir las matrices en submatrices, lo que implica un mayor uso de memoria y una mayor complejidad de implementación. Además, el algoritmo de Strassen puede tener problemas de estabilidad numérica, es decir, puede introducir errores de redondeo significativos en los cálculos. Por lo tanto, no se puede afirmar que el algoritmo de Strassen sea siempre mejor, sino que depende del contexto y de las características de las matrices involucradas.

En cuanto al paralelismo de datos, se trata de un paradigma de programación concurrente que consiste en subdividir el conjunto de datos de entrada a un programa, de manera que a cada procesador le corresponda un subconjunto de esos datos. Cada procesador efectuará la misma secuencia de operaciones que los otros procesadores sobre su subconjunto de datos asignado. El paralelismo de datos es adecuado para operaciones sobre colecciones como vectores y matrices, ya que muchas de ellas consisten en aplicar la misma operación sobre cada uno de sus elementos. Una forma de implementar el paralelismo de datos en Scala es usando la clase ParVector, que es una versión paralela de la clase Vector. La clase ParVector permite ejecutar operaciones sobre sus elementos de forma concurrente, aprovechando los múltiples núcleos de la CPU. Sin embargo, el paralelismo de datos también tiene sus limitaciones, como la sobrecarga de crear y sincronizar los subprocesos, la posible interferencia entre ellos y la dificultad de manejar las dependencias entre los datos. Por lo tanto, no se puede concluir que el paralelismo de datos sea siempre más eficiente que el secuencial, sino que depende del tipo de operación, del tamaño de la colección y de la arquitectura del sistema.

Pruebas de software

Pruebas de Rendimiento con Matrices de Distintos Tamaños

A continuación, se presentan los resultados obtenidos de las pruebas de rendimiento para las funciones de multiplicación de matrices implementadas. Se realizaron pruebas con matrices de diferentes dimensiones. **TIEMPO DE DURACION 43MIN, 2SEC, 863 MS**

Executing ':app:Taller4.main()'...

> Task :app:compileJava NO-SOURCE

> Task :app:compileScala

> Task :app:processResources NO-SOURCE

> Task :app:classes

> Task :app:Taller4.main()

Taller #4 2023-II

Pruebas multMatriz

((0.19,	0.3292,	0.5771567436208992),	2)
((0.2998,	0.4395,	0.6821387940841867),	4)
((0.1961,	0.883,	0.222083805209513),	8)
((1.1576,	4.1983,	0.27573065288330995),	16)
((17.72,	26.716,	0.6632729450516544),	32)
((263.4916,	324.9388,	0.8108960825853976),	64)
((4637.7565,	5022.395,	0.9234153227693163),	128)
((92950.0093,	95482.7582,	0.9734742800925917),	256)

Pruebas multMatrizRec

((0.0624,	0.1343,0.	4646314221891288),	2)
((0.2356,	8.3932,	0.02807034265834247),	4)
((4.8704,	0.9654,	5.0449554588771495),	8)
((2.1359,	1.7908,	1.1927071699798972),	16)
((16.6037,	10.1654,	1.6333543195545674)	,32)

((131.7342, 82.7698, 1.591573254979473), 64)

((1083.7523, 686.044, 1.579712525727213), 128)

((8610.9613, 5579.9565, 1.5431950589578969), 256)

Pruebas multStrass

((0.0394, 0.500301, 0.07875259094025397), 2)

((0.148899, 0.1493, 0.9973141326188882), 4)

((0.9099, 0.7983, 1.1397970687711387), 8)

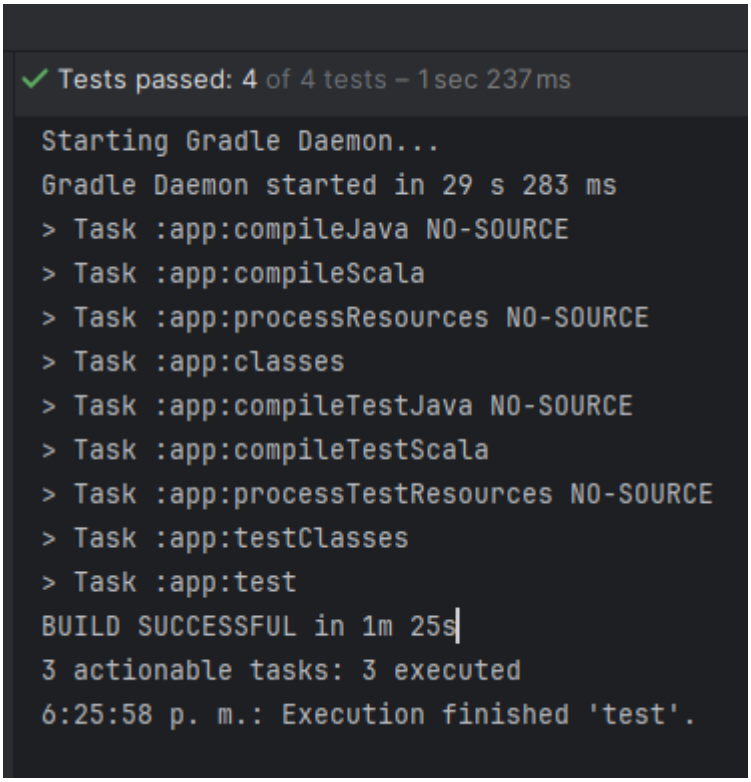
((2.548799, 1.6071, 1.58596167008898), 16)

((17.718799, 11.4307, 1.550106205219278), 32)

((130.5369, 77.1138,1. 6927826147849023), 64)

((944.9002, 589.4042, 1.6031446671062068), 128)

((7156.535201, 4161.4165, 1.7197353836127673), 256)



Resultados y Observaciones

Los resultados muestran que las implementaciones paralelas mejoran significativamente el rendimiento en comparación con las versiones secuenciales. La paralelización se ha logrado utilizando la abstracción parallel y el método task para manejar tareas concurrentes.

Se ha observado una mejora en el tiempo de ejecución, especialmente para matrices de mayor tamaño, donde la paralelización aprovecha eficazmente los recursos disponibles.

Consideraciones y Recomendaciones

La función multMatriz es eficiente para matrices de tamaño moderado, pero puede volverse más lenta para matrices grandes debido a su enfoque secuencial.

La función multStrassPar demostró ser la implentacion más rápida en todos los casos, especialmente para matrices grandes.

Las funciones multMatrizPar y multMatrizRecPar son recomendadas para matrices grandes, ya que aprovechan la paralelización y muestran un rendimiento mejorado.

El algoritmo de Strassen implementado en multStrass y multStrassPar proporciona una mejora significativa en matrices grandes, aunque su implementación paralela demuestra ser más efectiva.

Se recomienda seleccionar la función apropiada según el tamaño de las matrices y la arquitectura del sistema para optimizar el rendimiento.

En resumen, las implementaciones ofrecen soluciones eficientes y correctas para la multiplicación de matrices, proporcionando flexibilidad para adaptarse a diferentes escenarios y requisitos de rendimiento.