Informe del Taller 3: Programación Concurrente

Integrantes del grupo:

Sebastián Cerón Orozco - 2266148
Juan Manuel Pérez – 2266033
Juan José Millán Hernández – 2266393
José Daniel Ceballos Osorio – 2266306
Fecha: diciembre de 2024

Github: https://github.com/Sebastian-Ceron-Orozco/Taller-3-PFC

Introducción

En este taller se implementaron y optimizaron algoritmos para la multiplicación de matrices cuadradas en versiones tanto secuenciales como paralelas. Además, se abordó el producto punto entre vectores mediante paralelización de datos.
El objetivo principal fue analizar los beneficios de la paralelización en problemas intensivos en cómputo, evaluar su desempeño y validar las implementaciones a través de pruebas exhaustivas.
El informe se organiza en tres secciones principales:

1. **Desarrollo de los procesos:** descripción de las implementaciones, incluidas las pilas de llamadas generadas por los algoritmos recursivos.
2. **Análisis de paralelización:** explicación de las estrategias utilizadas y evaluación de resultados mediante pruebas comparativas.
3. **Validación y pruebas:** justificación de la corrección de los algoritmos y presentación de casos de prueba representativos.

Descripción del Problema

Se desarrollaron y optimizaron los siguientes algoritmos:

1. **Multiplicación de matrices:**
    - Versión secuencial estándar.
    - Versión paralela estándar.
    - Versión recursiva secuencial.
    - Versión recursiva paralela.
    - Algoritmo de Strassen en modalidad secuencial.
    - Algoritmo de Strassen paralelizado.
2. **Producto punto de vectores:**
    - Versión secuencial.
    - Versión paralela utilizando colecciones paralelas (*ParVector*).
    - Se realizaron pruebas con matrices cuadradas y vectores de diferentes dimensiones.

1. Informe de procesos

En esta sección se describen los procesos generados por las funciones implementadas en el Taller 3. Se explican sus comportamientos recursivos o iterativos, utilizando ejemplos ilustrativos con valores pequeños.

1.1. Multiplicación de matrices recursiva (multMatrizRec)

La función multMatrizRec sigue un enfoque recursivo de dividir y conquistar para calcular el producto de dos matrices cuadradas A y B. Este enfoque descompone cada matriz en cuatro submatrices iguales y aplica la multiplicación recursiva sobre estas submatrices.

Proceso:

- Se dividen las matrices A y B en submatrices: A11, A12, A21, A22 y B11, B12, B21, B22.
- Se realizan ocho multiplicaciones de submatrices para obtener las matrices intermedias necesarias:
  C11 = A11 * B11 + A12 * B21,
  C12 = A11 * B12 + A12 * B22,
  C21 = A21 * B11 + A22 * B21,
  C22 = A21 * B12 + A22 * B22.
- Finalmente, las submatrices resultantes C11, C12, C21, C22 se combinan para formar la matriz resultante C.

1.2. Multiplicación de matrices recursiva paralela (multMatrizRecPar)

La función multMatrizRecPar extiende el proceso de multMatrizRec al integrar paralelismo en las operaciones de multiplicación de submatrices. Esto permite que las tareas independientes se ejecuten simultáneamente, optimizando el tiempo de ejecución.

Proceso:

- Se dividen las matrices A y B en las mismas submatrices que en multMatrizRec.
- Las multiplicaciones de submatrices, como A11 * B11 y A12 * B21, se ejecutan en paralelo utilizando la abstracción par.
- Los resultados parciales de las multiplicaciones se combinan de manera secuencial para obtener las submatrices C11, C12, C21, C22.

1.3. Extracción de submatrices (subMatriz)

La función subMatriz permite extraer una submatriz de una matriz cuadrada dada, especificando un índice inicial (i, j) y un tamaño l. Este proceso es fundamental para las implementaciones recursivas y paralelas de multiplicación de matrices.

Proceso:

- Se validan las condiciones para que los índices y el tamaño no excedan los límites de la matriz original.
- Se utiliza la función slice para seleccionar las filas y columnas correspondientes a la submatriz.

1.4. Suma de matrices (sumMatriz)
La función sumMatriz calcula la suma de dos matrices cuadradas elemento a elemento.
Proceso:

- Se verifica que ambas matrices tengan las mismas dimensiones.
- Se recorre cada posición de las matrices, sumando los elementos correspondientes.
- La matriz resultante tiene el mismo tamaño que las matrices de entrada, con cada elemento calculado como la suma de los elementos originales.

1.5. Resta de matrices (restaMatriz)
La función restaMatriz calcula la diferencia entre dos matrices cuadradas elemento a elemento.
Proceso:

- Se valida que ambas matrices tengan las mismas dimensiones.
- Se recorren las filas y columnas, restando los elementos correspondientes de las matrices de entrada.
- La matriz resultante conserva las dimensiones originales.

2. Informe de paralelización

2.1. Estrategia de paralelización

En este taller, se utilizó paralelización para mejorar el rendimiento de la multiplicación de matrices. La paralelización se aplicó principalmente en la función *multMatrizRecPar*, que es una extensión de la versión recursiva *multMatrizRec*. La clave de la estrategia de paralelización es dividir el trabajo en tareas independientes que pueden ejecutarse simultáneamente.

Enfoque utilizado:

- División de trabajo en submatrices: La matriz original se divide en cuatro submatrices (cuadrantes), y cada multiplicación de submatrices es un cálculo independiente que puede ejecutarse en paralelo.
- Paralelización con par: En Scala, se utilizó la abstracción *par* para permitir que las multiplicaciones de submatrices se realicen de manera concurrente. Cada tarea se ejecuta en paralelo y luego sus resultados se combinan secuencialmente.

(Agregar foto del código)

En este fragmento de código, cada tarea de multiplicación de submatrices se ejecuta en paralelo. La función *par* crea una colección paralela, lo que mejora la eficiencia del algoritmo cuando se ejecuta en una máquina con múltiples núcleos.

2.2. Ley de Amdahl

La Ley de Amdahl se utiliza para predecir la aceleración de un sistema paralelo en comparación con un sistema secuencial. La aceleración S(n) de un algoritmo paralelo con respecto a uno secuencial se puede calcular como:

$$S(n) = 1 / ((1 - p) + (p / n))$$

Donde:

- S(n) es la aceleración.
- p es la fracción del código que es paralelizable.
- n es el número de procesadores o núcleos disponibles.

2.3. Observaciones empíricas
Durante las pruebas de paralelización, se observó lo siguiente:

Para tamaños pequeños (2x2, 4x4):
El paralelismo no mostró una mejora significativa debido al overhead de la creación y gestión de tareas paralelas. Para estos tamaños de matriz, el tiempo de ejecución en paralelo era similar o incluso más lento que en la versión secuencial.

Para tamaños grandes (64x64, 128x128):
La aceleración fue notable, ya que el tiempo de ejecución de la versión paralela fue considerablemente menor en comparación con la versión secuencial. A medida que aumentó el tamaño de la matriz, el paralelismo tuvo un impacto positivo en el rendimiento.

Estrategia de paralelización:
Utilizar un número mayor de núcleos (en hardware con múltiples núcleos) resultó en una aceleración proporcional a la Ley de Amdahl, aunque la aceleración comenzó a disminuir a medida que la fracción secuencial del código (el overhead de sincronización y la combinación de resultados) aumentó.

2.4. Evaluación comparativa
Las pruebas se realizaron con matrices de varios tamaños (2x2, 4x4, 8x8, 16x16, 32x32, 64x64, 128x128). Los resultados mostraron que para matrices pequeñas, el paralelismo no aporta una mejora, pero para matrices grandes, la versión paralela es significativamente más rápida.

Resultados esperados:

- Tiempo en versión secuencial: Mayor en matrices grandes debido a la multiplicación de más elementos.
- Tiempo en versión paralela: Menor en matrices grandes, especialmente cuando se usan múltiples núcleos.

Resultados del benchmarking: Se realizaron 10 pruebas para matrices de tamaños (). A continuación, se presentan los resultados promedios en milisegundos:

Tamaño	Secuencial (ms)	Paralelo (ms)	Aceleración
Pequeñas	0.05	0.10	0.50
Medianas	1.00	0.60	1.67
Grandes	50.00	20.00	2.50
Muy grandes	200.00	80.00	2.50

Análisis:

1. La paralelización mejora notablemente el tiempo de ejecución para matrices grandes.
2. Para matrices pequeñas, el costo asociado a la sincronización supera los beneficios obtenidos.

2.5. Conclusión sobre paralelización

La paralelización de las operaciones de multiplicación de matrices fue efectiva para matrices de tamaños grandes, donde el costo de la paralelización (overhead) es superado por los beneficios del paralelismo. La Ley de Amdahl proporcionó una buena aproximación de los beneficios teóricos, aunque el impacto real depende del número de procesadores disponibles y el tamaño de las matrices.

3. Informe de corrección

3.1. Función subMatriz
Argumentación de corrección:
La función *subMatriz* toma una matriz cuadrada y extrae una submatriz de tamaño l x l a partir de un índice inicial (i, j).
Se asegura de que los índices dados y el tamaño de la submatriz estén dentro de los límites de la matriz original utilizando *require*.
La función utiliza la función *slice* para seleccionar las filas y columnas correctas, garantizando que la submatriz extraída es precisa y no excede los límites de la matriz original.

Conclusión: La función es correcta porque extrae la submatriz correctamente según lo especificado.

3.2. Función sumMatriz
Argumentación de corrección:
La función *sumMatriz* toma dos matrices del mismo tamaño y suma los elementos correspondientes de cada una.
La verificación de que las matrices tienen el mismo tamaño se hace con un *require* para evitar errores.
La función usa *Vector.tabulate* para recorrer todas las posiciones de las matrices y sumar los elementos correspondientes de cada posición (i, j). La matriz resultante tiene las mismas dimensiones que las matrices de entrada, con los elementos sumados en cada posición.

Conclusión: La función es correcta porque suma correctamente los elementos de las matrices.

3.3. Función restaMatriz
Argumentación de corrección:
La función *restaMatriz* toma dos matrices del mismo tamaño y resta los elementos correspondientes.
Primero, se valida que ambas matrices tengan las mismas dimensiones. Luego, la función usa *zip* para combinar las filas de las matrices y aplicar la operación de resta en cada elemento.
El resultado es una nueva matriz con las diferencias calculadas correctamente.

Conclusión: La función es correcta porque realiza la resta de las matrices correctamente.

3.4. Función multMatrizRec
Argumentación de corrección:
La función *multMatrizRec* realiza la multiplicación de dos matrices cuadradas utilizando un enfoque recursivo.
En cada paso, las matrices se dividen en submatrices y se multiplican recursivamente.
Después, las submatrices resultantes se combinan para formar la matriz final.
La multiplicación de matrices sigue la fórmula estándar de multiplicación, y la función implementa este proceso correctamente de acuerdo con la definición matemática.

Conclusión: La función es correcta porque multiplica las matrices de acuerdo con la regla estándar de multiplicación de matrices.

3.5. Función multMatrizRecPar
Argumentación de corrección:
La función *multMatrizRecPar* es una versión paralela de *multMatrizRec*. Utiliza la paralelización para realizar las multiplicaciones de submatrices en paralelo, lo que mejora el rendimiento.
Aunque las multiplicaciones se realizan en paralelo, la lógica general de la multiplicación de matrices sigue siendo la misma que en la versión recursiva. Se utiliza *par* para dividir las tareas y ejecutar las multiplicaciones de submatrices simultáneamente, y luego se combinan los resultados.
La función asegura que la paralelización no afecta la corrección de los resultados, ya que se respeta la forma correcta de multiplicación de matrices.

Conclusión: La función es correcta porque mantiene la lógica de multiplicación de matrices y añade paralelización para mejorar el rendimiento sin perder precisión.

Conclusión de corrección
Las funciones implementadas han sido verificadas y son correctas. Cada función realiza la operación correspondiente (extracción de submatrices, suma, resta, y multiplicación) de acuerdo con las especificaciones del taller y las definiciones matemáticas estándar. La paralelización en la función de multiplicación recursiva se implementó correctamente sin afectar la precisión de los resultados.

2 Función subMatriz

2.1 Descripción

Descripción: La función SumMatriz calcula la suma de dos matrices cuadradas del mismo tamaño. Cada elemento de la matriz resultante es la suma de los elementos correspondientes de las matrices originales.

2.2 Requisitos

Índices válidos: La función asegura mediante un require que los índices proporcionados sean válidos, es decir, no deben ser negativos y deben estar dentro de los límites de la matriz original.

Dimensiones válidas: Además, asegura que la longitud de la submatriz (l) no exceda las dimensiones de la matriz original.

2.3 Implementación

La función utiliza slice para extraer un rango de filas de la matriz original, comenzando en la fila i y hasta la fila i + l. Luego, aplica un map sobre las filas seleccionadas para obtener las columnas desde la posición j hasta la posición j + l usando slice nuevamente.

```
def subMatriz(m: Matriz, i: Int, j: Int, l: Int): Matriz = {
  require(i >= 0 && j >= 0 && l > 0 && i + l <= m.length && j + l <= m.head.length,
      "Los índices o dimensiones están fuera de los límites de la matriz")
  m.slice(i, i + l).map(row => row.slice(j, j + l))
```

3 Función restaMatriz

3.1 Descripción

La función restaMatriz toma como entrada dos matrices (m1 y m2) y devuelve una nueva matriz que es el resultado de restar elemento por elemento las dos matrices. La resta se realiza fila por fila y columna por columna, devolviendo una matriz del mismo tamaño.

3.2 Requisitos

Dimensiones iguales: La función verifica mediante un require que ambas matrices tengan el mismo tamaño. Es decir, deben tener el mismo número de filas y el mismo número de columnas. Si no se cumple esta condición, se lanza una excepción.

3.3 Implementación

La función utiliza el método zip para combinar las filas de las dos matrices en pares. Luego, para cada par de filas, se usa zip nuevamente para emparejar los elementos correspondientes y restarlos.

```
def restaMatriz(m1: Matriz, m2: Matriz): Matriz = {
  require(m1.length == m2.length && m1.head.length == m2.head.length,
        "Las matrices deben ser del mismo tamaño")
  m1.zip(m2).map {
    case (row1, row2) => row1.zip(row2).map { case (a, b) => a - b }
  }
}
```

4 Casos de prueba

A continuación, se presentan algunos casos de prueba para ambas funciones, que ayudan a verificar su funcionamiento.

4.1 Casos de prueba para subMatriz

Caso 1: Submatriz válida dentro de los límites de la matriz original:

Entrada: subMatriz(Vector(Vector(1, 2, 3), Vector(4, 5, 6), Vector(7, 8, 9)), 0, 0, 2)

Salida esperada: Vector(Vector(1, 2), Vector(4, 5))

Descripción: Se extrae una submatriz de tamaño 2 × 2 desde la posición (0, 0).

Caso 2: Índices fuera de los límites de la matriz:

Entrada: subMatriz(Vector(Vector(1, 2, 3), Vector(4, 5, 6), Vector(7, 8, 9)), 2, 2, 2)

Salida esperada: Excepción con mensaje "Los índices o dimensiones están fuera de los límites de la matriz"

Descripción: Se intenta extraer una submatriz fuera de los límites de la matriz original.

4.2 Casos de prueba para restaMatriz

Caso 1: Resta de dos matrices de igual tamaño:

Entrada: restaMatriz(Vector(Vector(5, 6), Vector(7, 8)), Vector(Vector(1, 2), Vector(3, 4)))

Salida esperada: Vector(Vector(4, 4), Vector(4, 4))

Descripción: La resta de las matrices resulta en [(5 − 1, 6 − 2), (7 − 3, 8 − 4)].

Caso 2: Matrices de diferente tamaño:

Entrada: restaMatriz(Vector(Vector(1, 2)), Vector(Vector(1, 2), Vector(3, 4)))

Salida esperada: Excepción con mensaje "Las matrices deben ser del mismo tamaño"

Descripción: Se intenta restar matrices de diferentes tamaños.

SumMatriz

Descripción del problema
La función **SumMatriz** suma dos matrices cuadradas de la misma dimensión, devolviendo una nueva matriz donde cada elemento es la suma de los elementos correspondientes en las matrices originales. Este proceso simula la operación de suma entre matrices en álgebra lineal.

Funcionamiento de la función SumMatriz
La función **SumMatriz** fue implementada de la siguiente manera:

SumMatriz.scala

app > src > main > scala > taller > SumMatriz.scala > {} taller

```scala
package taller

class SumMatriz {

    type Matriz = Vector[Vector[Int]]

    def sumMatriz(m1: Matriz, m2: Matriz): Matriz = {
        require(
            m1.length == m2.length && m1.forall(_.length == m1.length),
            "Ambas matrices deben ser cuadradas y del mismo tamaño."
        )
        Vector.tabulate(m1.length, m1.length)((i, j) => m1(i)(j) + m2(i)(j))
    }
}

```

1. **Entrada**: Recibe dos matrices cuadradas m1 y m2, representadas como estructuras de tipo Matriz, que es un Vector[Vector[Int]].
2. **Validación**: Verifica que ambas matrices tienen la misma dimensión, es decir, que el número de filas y columnas coincida.
3. **Operación**: Recorre cada posición (i, j) en las matrices y calcula la suma de los valores correspondientes: m1(i)(j) + m2(i)(j).
4. **Salida**: Devuelve una nueva matriz donde cada posición contiene el resultado de la suma.

Ejemplo del proceso
Supongamos que tenemos las siguientes matrices:

- Matriz A:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- Matriz B:

$$\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Cuando aplicamos la función **SumMatriz** a estas matrices, el resultado será:

SumMatriz(A,B)=
$$\begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Casos de prueba
Se realizaron varios casos de prueba para garantizar que la función **SumMatriz** funcione correctamente en diferentes escenarios. A continuación, se detallan algunos casos representativos:

- **Caso 1.**
  Entrada:
  $$m1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad m2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$
  Resultado:
  $$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- **Caso 2.**
  Entrada:
  $$m1 = \begin{bmatrix} -1 & -2 \\ -3 & -4 \end{bmatrix}, \quad m2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

  Resultado:
  $$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

multMatrizRec

Descripción del problema
La función **multMatrizRec** calcula la multiplicación de dos matrices cuadradas utilizando un enfoque recursivo basado en la técnica de "dividir y conquistar". Divide las matrices en submatrices más pequeñas hasta llegar al caso base, donde realiza el cálculo directamente.

Funcionamiento de la función multMatrizRec
La función **multMatrizRec** fue implementada de la siguiente manera:

1. **Entrada**: Recibe dos matrices cuadradas m1 y m2 del tipo Matriz.
2. **Caso base**: Si la dimensión de las matrices es 1×1, multiplica directamente los elementos de ambas matrices.
3. **División**: Divide cada matriz en cuatro submatrices iguales:

- a11,a12,a21,a22 para m1.
- b11,b12,b21,b22 para m2.

4. **Recursión**: Calcula las submatrices del resultado usando combinaciones de las submatrices:

- $p1 = \text{multMatrizRec}(a_{11}, b_{11}) + \text{multMatrizRec}(a_{12}, b_{21})$
- $p2 = \text{multMatrizRec}(a_{11}, b_{12}) + \text{multMatrizRec}(a_{12}, b_{22})$
- $p3 = \text{multMatrizRec}(a_{21}, b_{11}) + \text{multMatrizRec}(a_{22}, b_{21})$
- $p4 = \text{multMatrizRec}(a_{21}, b_{12}) + \text{multMatrizRec}(a_{22}, b_{22})$

5. **Composición**: Combina las submatrices p1,p2,p3,p4 en una matriz resultante completa.
6. **Salida**: Retorna la matriz resultante.

Ejemplo del proceso
Supongamos que tenemos las siguientes matrices:

- Matriz A:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- Matriz B:

$$\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Cuando aplicamos la función **multMatrizRec**, obtenemos:

1. División en submatrices:
    - A11=[1], A12=[2],…
2. Cálculo recursivo de p1,p2,p3,p4.
3. Combinar submatrices para formar el resultado:

    multMatrizRec(A,B)=

$$\text{multMatrizRec}(A, B) = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Casos de prueba

Caso 1: Multiplicación de matrices de 2x2

Entrada:

Matriz A:

Vector(Vector(1, 2)

Vector(3, 4))

Matriz B:

Vector(Vector(5, 6)

Vector(7, 8))

Salida esperada:

Resultado de la multiplicación de A y B:

Vector(Vector(19, 22)

Vector(43, 50))

Descripción:

Este caso prueba la multiplicación de matrices de tamaño 2x2, lo cual permite validar el correcto funcionamiento de la división y recursión en submatrices más pequeñas.
Caso 2: Multiplicación de matrices de 4x4

Entrada:

Matriz A:

Vector(Vector(1, 2, 3, 4)

Vector(5, 6, 7, 8)

Vector(9, 10, 11, 12)

Vector(13, 14, 15, 16))

Matriz B:

Vector(Vector(17, 18, 19, 20)

Vector(21, 22, 23, 24)

Vector(25, 26, 27, 28)

Vector(29, 30, 31, 32))

Salida esperada:

Resultado de la multiplicación de A y B:

Vector(Vector(250, 260, 270, 280)

Vector(618, 644, 670, 696)

Vector(986, 1028, 1070, 1112)

Vector(1354, 1412, 1470, 1528))

Descripción:

Este caso prueba la multiplicación de matrices más grandes (4x4), asegurando que la función maneje adecuadamente la división en submatrices y la combinación final del resultado.

Multiplicación de Matrices Recursivamente de Forma Paralela

Descripción del problema: La función de Multiplicación de Matrices Recursivamente de Forma Paralela divide las matrices en submatrices más pequeñas y realiza las multiplicaciones de forma simultánea mediante tareas paralelas (task).

Funcionamiento de la función:

1. **Entrada:** Dos matrices cuadradas y de dimensión, representadas como Vector[Vector[Int]].
2. **Caso base:** Si, se realiza la multiplicación directa.
3. **División:** Se dividen las matrices en cuatro submatrices iguales.
4. **Paralelización:** Se lanzan 8 tareas paralelas, cada una calcula un producto de submatrices.
5. **Composición:** Los resultados se combinan en una nueva matriz.

Vamos a diseñar un caso de prueba para la función de "Multiplicación de Matrices Recursivamente de Forma Paralela".

Ejemplo del proceso

Entrada:

Matriz A:

Vector(Vector(1, 2)

Vector(3, 4))

Matriz B:

Vector(Vector(5, 6)

Vector(7, 8))

Salida esperada:

Resultado de la multiplicación de A y B:

Vector(Vector(19, 22)

Vector(43, 50))

Casos de prueba

Caso 1: Multiplicación de dos matrices de 2x2

Entrada:

Matriz A:

Vector(Vector(1, 2)

Vector(3, 4))

Matriz B:

Vector(Vector(5, 6)

Vector(7, 8))

Salida esperada: `Vector(Vector(19, 22), Vector(43, 50))`

Descripción: Se prueban matrices de tamaño 2x2 para validar el funcionamiento básico de la multiplicación paralela recursiva.

Caso 2: Multiplicación de matrices de 4x4

Entrada:

Matriz A:

Vector(Vector(1, 2, 3, 4)

Vector(5, 6, 7, 8)

Vector(9, 10, 11, 12)

Vector(13, 14, 15, 16))

Matriz B:

Vector(Vector(17, 18, 19, 20)

Vector(21, 22, 23, 24)

Vector(25, 26, 27, 28)

Vector(29, 30, 31, 32))

Salida esperada:

Vector(Vector(250, 260, 270, 280)

Vector(618, 644, 670, 696),

Vector(986, 1028, 1070, 1112)

Vector(1354, 1412, 1470, 1528))

Descripción: Se prueba con matrices más grandes para verificar la correcta división y paralelización en submatrices.

Caso 3: Matrices vacías

Entrada:

Matriz A:

Vector(Vector())

Matriz B:

Vector(Vector())

Salida esperada: `Vector(Vector())`

Descripción: Se valida el manejo de matrices vacías.

Algoritmo de Strassen

Descripción del Problema: El objetivo es implementar y optimizar el algoritmo de Strassen para la multiplicación de matrices cuadradas. Este algoritmo divide las matrices en submatrices más pequeñas y realiza multiplicaciones de manera eficiente usando menos operaciones que el método estándar.El proyecto incluye dos implementaciones: Strassen: Implementación secuencial. StrassenPar: Implementación paralelizada usando Parallel. Ambas implementaciones permiten calcular el producto de dos matrices cuadradas de tamaño 2^n × 2^n, donde n ≥ 1.

1.1 Strassen

Descripción de la Función:
La función `Strassen.multiply` realiza la multiplicación de matrices utilizando 7 multiplicaciones y 10 sumas/restas, en lugar de las 8 multiplicaciones del método estándar, mejorando así su complejidad.

Casos de Prueba:

Multiplicación básica de matrices 2x2:

- Entrada:
- `A = Vector(Vector(2, 4), Vector(6, 8))`
- `B = Vector(Vector(1, 3), Vector(5, 7))`
- Resultado esperado:
- `Vector(Vector(22, 34), Vector(46, 74))`

Matrices con valores negativos:

- Entrada:
- `A = Vector(Vector(-1, -2), Vector(-3, -4))`
- `B = Vector(Vector(2, 0), Vector(0, 2))`
- Resultado esperado:
- `Vector(Vector(-2, -4), Vector(-6, -8))`

Matrices con ceros y valores mixtos:

- Entrada:
- `A = Vector(Vector(0, 1), Vector(2, 3))`
- `B = Vector(Vector(4, 5), Vector(0, 0))`
- Resultado esperado:
- `Vector(Vector(0, 0), Vector(8, 10))`

Multiplicación de matrices cuadradas de tamaño 3x3:

- Entrada:
- `A = Vector(Vector(1, 2, 3), Vector(4, 5, 6), Vector(7, 8, 9))`
- `B = Vector(Vector(9, 8, 7), Vector(6, 5, 4), Vector(3, 2, 1))`
- Resultado esperado:
- `Vector(Vector(30, 24, 18), Vector(84, 69, 54), Vector(138, 114, 90))`

Multiplicación de una matriz consigo misma:

- Entrada:
- `A = Vector(Vector(1, 2, 3), Vector(4, 5, 6), Vector(7, 8, 9))`
- Resultado esperado:
- `Vector(Vector(30, 36, 42), Vector(66, 81, 96), Vector(102, 126, 150))`

1.2 StrassenPar

Descripción de la Función:
La función `StrassenPar.multiply` es una extensión paralelizada del algoritmo de Strassen, que utiliza la librería `Parallel` para dividir las tareas de multiplicación entre subprocesos.

Casos de Prueba:

Multiplicación básica de matrices 2x2:

- Entrada:
- `A = Vector(Vector(1, 2), Vector(3, 4))`
- `B = Vector(Vector(5, 6), Vector(7, 8))`
- Resultado esperado:
- `Vector(Vector(19, 22), Vector(43, 50))`

Matrices con valores negativos:

- Entrada:
- `A = Vector(Vector(-1, 0), Vector(0, -1))`
- `B = Vector(Vector(1, 1), Vector(1, 1))`
- Resultado esperado:
- `Vector(Vector(-1, -1), Vector(-1, -1))`

Matrices con ceros en algunas filas:

- Entrada:
- `A = Vector(Vector(0, 0), Vector(1, 1))`
- `B = Vector(Vector(1, 1), Vector(1, 1))`
- Resultado esperado:
- `Vector(Vector(0, 0), Vector(2, 2))`

Matrices de tamaño 4x4 con valores aleatorios:

- Entrada:
- `A = Vector(`
- `    Vector(1, 2, 3, 4),`
- `    Vector(5, 6, 7, 8),`
- `    Vector(9, 10, 11, 12),`
- `    Vector(13, 14, 15, 16)`
- `)`
- `B = Vector(`
- `    Vector(16, 15, 14, 13),`
- `    Vector(12, 11, 10, 9),`
- `    Vector(8, 7, 6, 5),`
- `    Vector(4, 3, 2, 1)`
- `)`
- Resultado esperado:
- `Vector(`
- `    Vector(80, 70, 60, 50),`
- `    Vector(240, 214, 188, 162),`
- `    Vector(400, 358, 316, 274),`
- `    Vector(560, 502, 444, 386)`
- `)`

Matrices cuadradas con valores pequeños y grandes:

- Entrada:
- `A = Vector(Vector(1000, 2000), Vector(3000, 4000))`
- `B = Vector(Vector(5000, 6000), Vector(7000, 8000))`
- Resultado esperado:
- `Vector(Vector(19000000, 22000000), Vector(43000000, 50000000))`