Informe del Taller 3: Programación Concurrente

Fundamentos de Programación Funcional y Concurrente

Integrantes del grupo:

Juan Esteban Franco López - 2259475

Andrés Narváez - 2259545

Alejandro Garzón Mayorga - 2266088

Andres Felipe Chaparro - 2266252

Fecha: diciembre de 2024

Github: https://github.com/Andres111203/Taller3_PFC/tree/master

---

Introducción

Este taller se centra en implementar y optimizar algoritmos de multiplicación de matrices cuadradas, tanto como en versiones secuenciales como paralelas. Además, se analiza el producto punto entre vectores utilizando paralelización de datos.

El objetivo principal es comprender las ventajas de la paralelización en problemas computacionalmente intensivos, medir su desempeño y evaluar las implementaciones a través de pruebas exhaustivas.

El informe está estructurado en tres secciones principales:

- Informe de procesos: se describen las implementaciones realizadas, incluyendo la pila de llamadas generada por los algoritmos recursivos.
- Informe de paralelización: se detalla la estrategia de paralelización empleada y se analizan los resultados obtenidos mediante benchmarking.
- Informe de corrección: se argumenta la corrección de los algoritmos y se presentan casos de prueba.

Descripción del Problema

El objetivo de este taller es implementar y optimizar los siguientes algoritmos:

1. Multiplicación de matrices:
    - Versión secuencial estándar.
    - Versión paralela estándar.
    - Versión recursiva secuencial.
    - Versión recursiva paralela.
    - Algoritmo de Strassen secuencial.
    - Algoritmo de Strassen paralelizado.
2. Producto punto de vectores:
    - Versión secuencial.
    - Versión paralela con colecciones paralelas (ParVector).

Se trabajará con matrices cuadradas y vectores de diferentes tamaños.

Informe de Paralelización

Estrategia de paralelización: Se utilizó la abstracción task para dividir las operaciones en subprocesos y ejecutar las multiplicaciones de submatrices en paralelo. La sincronización de resultados se realizó con join.

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
0.05	0.10	0.50	
1.00	0.60	1.67	
50.00	20.00	2.50	
200.00	80.00	2.50	

Análisis:

1. La paralelización mejora significativamente el tiempo de ejecución en matrices grandes.
2. En matrices pequeñas, el costo de sincronización supera los beneficios.

2 Función subMatriz

2.1 Descripción

La función subMatriz toma como entrada una matriz (Matriz), junto con los índices de inicio (i, j) y la longitud (l) de la submatriz que se desea extraer. Esta función devuelve una submatriz que se obtiene a partir de la matriz original, comenzando desde la fila i y la columna j, y de tamaño l × l.

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:

```
package taller


class SumMatriz() {
    def sumMatriz(m1: Matriz, m2: Matriz): Matriz = {
        // recibe m1 y m2 matrices cuadradas de la misma dimension , potencia,→ de 2
        val n = m1.length
        require(m1.length == m2.length && m1(0).length == m2(0).length)
        val resultado = Vector.tabulate(n, n)((i, j) => m1(i)(j) + m2(i)(j))
        // y devuelve la matriz resultante de la suma de las 2 matrices
        resultado
    }
}
```

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)=

$$\begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

```
val obj = new SumMatriz()

val m1 = Vector(
    Vector(1, 2),
    Vector(3, 4)
)
val m2 = Vector(
    Vector(5, 6),
    Vector(7, 8)
)

val result = obj.sumMatriz(m1, m2)

test("SumMatriz Test #1") {
    assert(result(0)(0) == 6)
}

test("SumMatriz Test #2") {
    assert(result(0)(1) == 8)
}

test("SumMatriz Test #3") {
    assert(result(1)(0) == 10)
}

test("SumMatriz Test #4") {
    assert(result(1)(1) == 12)
}

test("SumMatriz Test #5") {
    assert(result.length == m1.length)
    assert(result(0).length == m1(0).length)
}
```

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))`