Universidad del Valle

Facultad de ingeniería

Ingeniería en sistemas

Cristian David Pacheco Torres
2227437
Juan Sebastian Molina Cuellar
2224491

November 10, 2023

Taller 5: Multiplicación de matrices en paralelo:

Contents

1 Introducción

1.1 Preliminares

Con el proposito de implementar diferentes algoritmos de multiplicación de matrices, tanto secuenciales, recursivos y paralelos, se nos otorgó a través del *Taller 5: Multiplicación de matrices en paralelo*, la definicion matemática de las siguientes operaciones:

- Transpuesta de una matriz.

$$T : \mathbb{R}^{m \times n} \to \mathbb{R}^{n \times m}$$
$$T(A) = [a_{ji}]_{n \times m} \tag{1}$$

  Se denota como : $\mathbf{A}^T$

- Producto punto de vectores.

$$\cdot : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$$
$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i \; v_i \tag{2}$$

  Donde "$\cdot$" denota la operacion binaria entre dos vectores $\in \mathbb{R}^n$.

- Multiplicación de matrices.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot \; B_{kj} \tag{3}$$

- Suma de matrices.

$$+ : \mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n} \to \mathbb{R}^{m \times \; n}$$
$$A \; + \; B \; = \; C$$
$$C_{ij} = A_{ij} + B_{ij} \tag{4}$$

  Donde "+" representa la operación de adición entre dos matrices $\in \mathbb{R}^{m \times n}$. A y B son las matrices a sumar. C es la matriz resultante de la suma.

- Resta de matrices.

$$- : \mathbb{R}^{m \times n} \times \mathbb{R}^{m \times n} \to \mathbb{R}^{m \times \; n}$$
$$A \; - \; B \; = \; C$$
$$C_{ij} = A_{ij} - B_{ij} \tag{5}$$

Definiciones de utilidad:

Sea una tarea computacional $T = (t, r)$, donde t es el tiempo de ejecución y r el resultado.
Además definase dos funciones sobre T como:

$$\rho(T) = \rho(t, r) = r \tag{6}$$

$$\phi(T) = \phi(t, r) = t \tag{7}$$

Definase una computación secuencial S:

$$S =< T_1, T_2, ...T_i, ..., T_{n-1}, T_n > \tag{8}$$

Donde i representa el orden de ejecución de la tarea $T_i \mid 0 \leq i \leq n$.

Definase una computación paralela P:

$$P = \{T_1, T_2, ...T_i, ..., T_{n-1}, T_n\} \tag{9}$$

Donde i identifica cada tarea que a posteriori se le extraerá el tiempo de ejecución y el resultado.

Sea $\phi(S)$ el tiempo de ejecución de una secuencia S:

$$\phi(S) = \sum_{1}^{n} \phi(T_i) \tag{10}$$

Sea $\phi(T)$ el tiempo de ejecución del conjunto de tareas P:

$$\phi(P) = \max(\phi_i(T_i)) \mid 1 \leq i \leq n \wedge \phi_i(T_i) \in P \tag{11}$$

1.2 Algoritmos de proporcionados de utilidad

En el listing 1, se definen dos tipos de datos esenciales: `Matriz` y `MatrizD`. Estos tipos representan matrices de enteros, donde `MatrizD` está diseñada para un procesamiento paralelo en base al tipo *ParVector* de *Scala*.

```
type Matriz = Vector[Vector[Int]]
type MatrizD = ParVector[ParVector[Int]]
```

Listing 1: Definiciones tipos de datos

```
def matrizAlAzar(long:Int, vals:Int) = {
    //Crea una matriz de enteros cuadrada de long x long,
    //con valores entre 0 y vals
    val v = Vector.fill(long, long){random.nextInt(vals)}
    v
}
```

Listing 2: matriz al azar

```
def vectorAlAzar(long:Int, vals:Int): Vector[Int] = {
    //Crea un vector de enteros de longitud long,
    //con valores aleatorios entre 0 y vals
    val v = Vector.fill(long){random.nextInt(vals)}
    v
}
```

Listing 3: vector al azar

```
def prodPunto(v1: Vector[Int], v2: Vector[Int]): Int = {
    //Calcula el producto punto entre dos vectores
    (v1 zip v2).map({case (i,j) => i*j}).sum
}
```

Listing 4: producto punto

```
def transpuesta(m: Matriz): Matriz = {
    //Calcula la transpuesta de una matriz
    val l =m.length
    Vector.tabulate(l,l)((i,j)=>m(j)(i))
}
```

Listing 5: transpuesta de una matriz

Los algoritmos anteriores fueron sugeridos en el *Taller*5 por parte del profesor, para la implementación de las funciones a desarrollar en este informe. Estos algoritmos fueron de utilidad para generar matrices aletorias y operaciones fundamentales entre vectores.

2 Informe del taller - secciones

2.1 Informe de corrección

multMatriz

```
def multMatriz(m1: Matriz, m2: Matriz): Matriz = {
    val l = m1.length
    val m = m2.length
    val v =Vector.tabulate(l,m)((i,j)=>prodPunto(m1(i),
       transpuesta(m2)(j)))
    v
}
```

Listing 6: mult matriz

Sea m_1 m_2, $\in \mathbb{R}^{n\times n}$, dos matrices a aplicar el operador binario definido en (3), entonces, por medio del método de sustitución se quiere demostrar que la implementación es equivalente de (3). Se tiene, además, i, l representa los tamaños de m_1, m_2, respectivmente; prodPunto definido (2) y la transformación transpuesta tranpose definida como (1); llamese la función transpose como $t()$, esto por simplicidad y ahorro de espacio; por tanto

$m_1 \cdot m_2 = \text{multMatriz}(m_1, m_2)$

$\rightarrow \text{Vector.tabulate}(l, n)((i, j) \Rightarrow \text{prodPunto}(m_1[i], \text{transpose}(m_2)[j]))$

$$\twoheadrightarrow \begin{bmatrix} m_1[0][0] \cdot \text{t}(m_2)[0][0] + m_1[0][1] \cdot \text{t}(m_2)[1][0] + \ldots + m_1[0][n-1] & \ldots \\ \vdots & \ddots & \vdots \\ \ldots & m_1[n-1][1] \cdot \text{t}(m_2)[1][n-1] + \ldots + m_1[l-1][n-1] \cdot \text{t}(m_2)[l-1][n-1] \end{bmatrix}$$

$= \sum_{k=0}^{n-1} m_{ik} \cdot t_{kj}^T$
$= m_1 \cdot m_2$
$\therefore \quad m_1 \cdot m_2 = \text{multMatriz}(m_1,\ m_2)$

multMatrizPar

```
def multMatrizPar(m1: Matriz, m2: Matriz): Matriz = {
    val l = m1.length
    val m = m2.length
    val parRows = for (k <- 0 until l )
        yield {
            (k,task(Vector.tabulate(1,m)((i,j)=>
                prodPunto(m1(k),transpuesta(m2)(j)))))
        }
    val v = parRows.map({case (i,j) =>
        (i,j.join())}).sortBy(_._1).map(_._2)
    v.reduce(_++_)
}
```

Listing 7: mult matriz paralela

Sea $m_1\ m_2 \in \mathbb{R}^{n\times n}$, dos matrices a aplicar el operador binario definido en (3) en forma paralela, es decir, en sentido en que se puede realizar los calculos de las partes unos independientes de otros y de forma simultanea, ya sea por un agente, o por varios, por medio de una tarea computacional $T = (t, r)$; entonces, por medio del método de sustitución se quiere demostrar que la implementación es equivalente de (3). Se tiene, además, i, l representa los tamaños de m_1, m_2, respectivamente; prodPunto definido (2) y la transformación transpuesta trapose definida como (1); llamese la funcón traspose como $t()$, esto por simplicidad y ahorro de espacio: por tanto

Definase $r_i = [\ \sum_{j=1}^{n} m_1[i][j] \cdot m_2[j][i]\ ]$ como la i-séptima fila de C, C_i. Luego,

$$m_1 \cdot m_2 = \begin{bmatrix} r_1 \\ \vdots \\ r_i \\ \vdots \\ r_n \end{bmatrix}$$

esto es, se tiene n matrices $r_i \in \mathbb{R}^{1\times n}$.
Si se define un objecto agendamiento $A = (i, T)$ que representa un tarea T relacionado con un entero $i\ |1 \leq i \leq n$, asociado con los correspondientes indices i's de r_i, posiblemente azaroso pero culminada en un tiempo finito $\phi(T_i)$
Si para las n $r_i \in \mathbb{R}^{1\times n}$, procedemos como en la subsesion 2.1, se tiene
$r_i = \text{multMatriz}(m_1[i], m_2[1..n][i])$
$\rightarrow \text{Vector.tabulate}(1, n)((i, j) \Rightarrow \text{prodPunto}(m_i[1], \text{transpose}(m_2)[j]))$
$\twoheadrightarrow r_i = [\ \sum_{k=1}^{n} m_{ik} \cdot t_{kj}^T]$
$\rightarrow r_i = m_1[i] \cdot m_2$
$\therefore \quad m_1[i] \cdot m_2 = \text{multMatriz}(m_1[i], m_2)$
Se sabe que
$A = (i, T_i)$ representa cada cómputo, los cuales son linealmente independientes, también

$\rho(T_i) = r_i$ representa el resultado de la tarea T_i, r_i, con i siendo la i-ésima fila de m_1. Por tanto, si se combinan los n resultados r_i por el orden impuesto por i, primer elemento de la tupla de asignación A_i, en el sentido de formar una matriz a partir de la n filas, se llega

$$\therefore \quad m_1 \cdot m_2 = \begin{bmatrix} \rho(T_1) \\ \vdots \\ \rho(T_i) \\ \vdots \\ \rho(T_n) \end{bmatrix} \equiv \begin{bmatrix} r_1 \\ \vdots \\ r_i \\ \vdots \\ r_n \end{bmatrix}$$

multMatrizRec

```
def multMatrizRec(m1:Matriz, m2:Matriz): Matriz ={
    //recibe m1 y m2 matrices cuadradas de la misma dimension,
       potencia de 2
    //y devuelve la multiplicacion de las 2 matrices
    val n = m1.length
    if(n == 1) {
        Vector(Vector(m1(0)(0)*m2(0)(0)))
    }
    else {
        val l = n/2
        val (m1_11, m1_12, m1_21, m1_22) =
            (subMatriz(m1,0,0,l),subMatriz(m1,0,l,l),
             subMatriz(m1,l,0,l),subMatriz(m1,l,l,l))
        val (m2_11, m2_12, m2_21, m2_22) =
            (subMatriz(m2,0,0,l),subMatriz(m2,0,l,l),
             subMatriz(m2,l,0,l),subMatriz(m2,l,l,l))

        val c_11 = sumMatriz(multMatrizRec(m1_11,m2_11),
           multMatrizRec(m1_12,m2_21))
        val c_12 = sumMatriz(multMatrizRec(m1_11,m2_12),
           multMatrizRec(m1_12,m2_22))
        val c_21 = sumMatriz(multMatrizRec(m1_21,m2_11),
           multMatrizRec(m1_22,m2_21))
        val c_22 = sumMatriz(multMatrizRec(m1_21,m2_12),
           multMatrizRec(m1_22,m2_22))

        Vector.tabulate(n,n)((i,j)=>
            if(i<l && j<l) c_11(i)(j)
            else if(i<l && j>=l) c_12(i)(j-l)
            else if(i>=l && j<l) c_21(i-l)(j)
            else c_22(i-l)(j-l))
    }
}
```

Listing 8: mult matriz recursiva

Sea dos matrices, m_1 y $m_2 \in \mathbb{R}^{n \times n}$, y la multiplicación entre estas definido por (3), tal que $|m_1|, |m_2| = 2^k \mid k \in N^+$, se quiere probar que la función implementada en *Scala*

multMatrizRec son equivalentes.

- **Caso base (n = 1)**: Si la matriz $m \in \mathbb{R}^{1\times 1}$ (el caso base), el algoritmo realiza una multiplicación de dos escalares, y retorna el resultado envuelto en una matriz $m_r \in \mathbb{R}^{1\times 1}$.

$$\text{Si } n = 1 \quad \rightarrow \quad \big[m_1[0][0] \cdot m_2[0][0]\big]$$

- **Caso recursivo** $n \geq 2$: Si $|m_1| \wedge |m_2| = k \geq 2$, se divide cada matriz, m_1 y m_2, en 4 submatrices de igual tamaño $m111$, $m112$, $m121$, $m122$, $m211$, $m212$, $m221$, y $m222$. Éste es paso *divide* en la estraégia *divide y conquistaras*.
  Luego, después de que cada llamada recursiva haya retornado, se computan la suma de los productos respectivas submatrices (La definición de esta operación esta propuesta en el especificación del laboratorio, entonces se tomará como axioma, y se utilizará su resultado):

$$\begin{aligned}
c_{11} &= \text{multMatrizRec}(m111, m211) + \text{multMatrizRec}(m112, m221)\\
c_{12} &= \text{multMatrizRec}(m111, m212) + \text{multMatrizRec}(m112, m222)\\
c_{21} &= \text{multMatrizRec}(m121, m211) + \text{multMatrizRec}(m122, m221)\\
c_{22} &= \text{multMatrizRec}(m121, m212) + \text{multMatrizRec}(m122, m222)
\end{aligned}$$

  Para el paso *conquistaras* en la estraégia *divide y conquistaras*, las matrices c_{11}, c_{12}, c_{21}, and c_{22} se combinan para formar una matriz $m_r \in \mathbb{R}^{2k\times 2k}$:

$$m_r[i][j] = \begin{cases} c_{11}[i][j] & \text{si } i < l \wedge j < l \\ c_{12}[i][j-l] & \text{si } i < l \wedge j \geq l \\ c_{21}[i-l][j] & \text{si } i \geq l \wedge j < l \\ c_{22}[i-l][j-l] & \text{si } i \geq l \wedge j \geq l \end{cases}$$

  Donde cada combinación de subíndices determinados por la condiciones guardas en los casos, determina el valor $m_r[i][j]$, y representa la región del cuadrante donde va a ser dispuesta cada una de las cuatro $c_{11}[i][j]$, $c_{12}[i][j-l]$, $c_{21}[i-l][j]$, $c_{22}[i-l][j-l]$ en el paso de la combinación para dar un matriz $|m_r| = k^2$

- **Invariantes**:
  - **Sobre el tamaño**: Las matrices m_1, m_2 cumple al comienzo de cada paso recursivo

$$m_1, m_2 \ \in \mathbb{R}^{n-k\times n-k} \wedge |m_1| = |m_2| = 2^{n-k}, \quad k \in \mathbb{N}^+, \quad 0 \leq k \leq n$$

    Nótese que

$$\lim_{k\to n}(n-k) = 0$$

– **Sobre la división en submatrices**: En cada llamada recursiva, divide una matriz de entrada en cuatro submatrices: $m111$, $m112$, $m121$, $m122$, $m211$, $m212$, $m221$, y $m222$. Esas submatrices representa $\frac{1}{4}$ de la matriz original y corresponde a una de las cuatro regiones dados por $k, k/2$ sobre el tamaño de sus filas y columnas.

$$\text{Invariante: } m1 = \begin{bmatrix} m111 & m112 \\ m121 & m122 \end{bmatrix}, \quad m2 = \begin{bmatrix} m211 & m212 \\ m221 & m222 \end{bmatrix}$$

y $|m_{\{1,2\}}| = \frac{k-n}{2}$

– **Sobre la multiplicación**: Las llamadas recursivas a multMatrizRec son hechas sobre submatrices, y la tupla de resultados (c_{11}, c_{12}, c_{21}, c_{22}) son combinados de forma bien definida para retorna una matriz m_r como resultado de la recursión. La hipótesis inductiva asume que los producto de matrices se realiza de forma correcta en cada k-*esimo* paso.

$$\text{Invariante: } c_{ij} = \text{multMatrizRec}(m1ij, m2ij)$$

– **Sobre la combinación**: La combinación de los productos de las submatrices forman la matriz resultado m_r, la posición esta definida de acuerdo a las guardas sobre los indices i, j y corresponde a una región "*cartesiana*". La hipótesis innductiva asegura que cada submatriz es colocada en el cuadrate pertinente dentro de la matriz resultante.

$$\text{Invariante: } m_r[i][j] = \begin{cases} c_{11}[i][j] & \text{si } i < l \wedge j < l \\ c_{12}[i][j-l] & \text{si } i < l \wedge j \geq l \\ c_{21}[i-l][j] & \text{si } i \geq l \wedge j < l \\ c_{22}[i-l][j-l] & \text{si } i \geq l \wedge j \geq l \end{cases}$$

– **Sobre el caso base**: Se alcanza cuando $|m_{\{1,2\}}| = 1$. Por tanto, el algoritmo asegura tratar la matriz más pequeña posible para $n \geq 1$.

$$\text{Invariante: } n = 1$$

multMatrizRecPar

```
def multMatrizRecPar(m1:Matriz, m2:Matriz): Matriz ={
    //recibe m1 y m2 matrices cuadradas de la misma dimension,
       potencia de 2
    //y devuelve la multiplicacion de las 2 matrices,
       paralelizando tareas
    val n = m1.length
    if(n == 1) {
        Vector(Vector(m1(0)(0)*m2(0)(0)))
    }
    else {
        val l = n/2
        val (m1_11, m1_12, m1_21, m1_22) =
```

```
            (subMatriz(m1,0,0,l),subMatriz(m1,0,l,l),
             subMatriz(m1,l,0,l),subMatriz(m1,l,l,l))
        val (m2_11, m2_12, m2_21, m2_22) =
            (subMatriz(m2,0,0,l),subMatriz(m2,0,l,l),
             subMatriz(m2,l,0,l),subMatriz(m2,l,l,l))
        val (c_11, c_12, c_21, c_22) = parallel(
            sumMatriz(multMatrizRec(m1_11,m2_11),
                      multMatrizRec(m1_12,m2_21)),
            sumMatriz(multMatrizRec(m1_11,m2_12),
                      multMatrizRec(m1_12,m2_22)),
            sumMatriz(multMatrizRec(m1_21,m2_11),
                      multMatrizRec(m1_22,m2_21)),
            sumMatriz(multMatrizRec(m1_21,m2_12),
                      multMatrizRec(m1_22,m2_22)))
        Vector.tabulate(n,n)((i,j)=>
            if(i<l && j<l) c_11(i)(j)
            else if(i<l && j>=l) c_12(i)(j-l)
            else if(i>=l && j<l) c_21(i-l)(j)
            else c_22(i-l)(j-l))
    }
}
```

Listing 9: mult matriz recursiva paralela

La función `multMatrizRecPar` en Scala realiza la multiplicación de matrices de manera paralela y es equivalente a `multMatrizRec`, siguiendo los siguientes puntos clave:

1. **Estructura y Pasos Básicos**: La función sigue la misma estructura lógica del algoritmo de multiplicación de matrices recursivo, que incluye:
   - El caso base cuando $n = 1$, donde se realiza una multiplicación directa de escalares.
   - El caso recursivo para $n \geq 2$, que implica dividir cada matriz en 4 submatrices y calcular las sumas de los productos de estas submatrices.
2. **Paralelización**: La principal diferencia en `multMatrizRecPar` es la paralelización de las operaciones. Utiliza `parallel` para ejecutar operaciones de forma concurrente, específicamente:
   - Las multiplicaciones y sumas de submatrices `m1_ij` y `m2_ij` para calcular `c_11`, `c_12`, `c_21`, `c_22`.
3. **Mantenimiento de la Correctitud**: A pesar de la paralelización, la función conserva la lógica del método de multiplicación matricial y no altera el resultado final de las operaciones matemáticas.
4. **Optimización y Eficiencia**: La paralelización mejora la eficiencia del algoritmo, especialmente para matrices grandes, reduciendo el tiempo total de ejecución y aprovechando mejor los recursos del sistema.
   De manera formal, a partir de (10) y (11) se hace una comparación entre los tiempos de ejecución de un algoritmo implementado de forma secuencial y un algoritmo implementado de forma paralela. La idea es análoga al teoría de complejidad de

funciones. Supongase que se tiene dos funciones que multiplica matrices, una de forma secuencial, cuya operaciones están dadas por $S =< T_1, T_2, ...T_i, ..., T_{n-1}, T_n >$ n tareas T secuenciales y otra de forma paralela $P = \{T_1, T_2, ...T_i, ..., T_{n-1}, T_n\}$ de n tareas T paralelas. Entonces,

$$\exists\ k_0 \mid k_0 \in N^+ \ \wedge\ n\ \geq\ k_0 \rightarrow\ \phi(P) \leq \phi(S) = \max(\phi_i(T_i)) \leq \sum_{1}^{n} \phi(T_i)$$

es decir, existe un número k_0, en este caso, un $k_0 = 2^k$ (por ser matrices cuadradas) tal que el tiempo de computo total pasa de ser la suma del tiempo de todas las tareas, en la ejecución secuencial, a ser igual al mayor tiempo de ejecución de una tarea del conjunto, en la forma paralela. Éste número k_0 depende de la instancia de ejecución, de los recursos disponibles, y, en este laboratorio, se le denominó *umbral*, cuyo valor determina el tamaño de la matriz en el cual el computo es más eficiente si se realiza una paralelización.

multStrassen

```
def multStrassen(m1:Matriz, m2:Matriz): Matriz ={
    //recibe m1 y m2 matrices cuadradas de la misma dimension, potencia de 2
    //y devuelve la multiplicacion de las 2 matrices
    val n = m1.length
    if(n == 1) {
        Vector(Vector(m1(0)(0)*m2(0)(0)))
    }
    else {
        val l = n/2
        val (s1, s2, s3, s4, s5, s6, s7, s8, s9, s10) = (
            restaMatriz(subMatriz(m2,0,l,l),subMatriz(m1,l,l,l)),
            sumMatriz(subMatriz(m1,0,0,l), subMatriz(m1,0,l,l)),
            sumMatriz(subMatriz(m1,l,0,l), subMatriz(m1,l,l,l)),
            restaMatriz(subMatriz(m2,l,0,l), subMatriz(m2,0,0,l)),
            sumMatriz(subMatriz(m1,0,0,l), subMatriz(m1,l,l,l)),
            sumMatriz(subMatriz(m2,0,0,l), subMatriz(m2,l,l,l)),
            restaMatriz(subMatriz(m1,0,l,l), subMatriz(m1,l,l,l)),
            sumMatriz(subMatriz(m2,l,0,l), subMatriz(m2,l,l,l)),
            restaMatriz(subMatriz(m1,0,0,l), subMatriz(m1,l,0,l)),
            sumMatriz(subMatriz(m2,0,0,l), subMatriz(m2,0,l,l))
        )
        val (p1, p2, p3, p4, p5, p6, p7) = (
            multStrassen(subMatriz(m1,0,0,l), s1),
            multStrassen(s2, subMatriz(m2,l,l,l)),
            multStrassen(s3, subMatriz(m2,0,0,l)),
            multStrassen(subMatriz(m1,l,l,l), s4),
            multStrassen(s5, s6),
```

```
            multStrassen(s7, s8),
            multStrassen(s9, s10)
        )
        val (c_11, c_12, c_21, c_22) = (
            restaMatriz(sumMatriz(p5, p4), sumMatriz(p6, p2)),
            sumMatriz(p1, p2),
            sumMatriz(p3, p4),
            restaMatriz(sumMatriz(p1, p5), restaMatriz(p3, p7))
        )
        Vector.tabulate(n,n)((i,j)=>
            if(i<l && j<l) c_11(i)(j)
            else if(i<l && j>=l) c_12(i)(j-l)
            else if(i>=l && j<l) c_21(i-l)(j)
            else c_22(i-l)(j-l))
    }
}
```

Listing 10: mult Strassen

Multiplicación de matrices usando el método de Strassen

Sea $m1$ y $m2$ matrices cuadradas de la misma dimensión, potencia de 2, y la función *multStrassen* implementada en Scala. Se quiere demostrar que la función *multStrassen* realiza la multiplicación de las matrices $m1$ y $m2$ utilizando el método de Strassen.

- **Caso base** $n = 1$: Si la matriz $m \in \mathbb{R}^{1\times1}$ (el caso base), el algoritmo realiza una multiplicación de dos escalares y retorna el resultado envuelto en una matriz $m_r \in \mathbb{R}^{1\times1}$

$$\text{Si } n = 1 \quad \rightarrow \quad \left[m1[0][0] \cdot m2[0][0]\right]$$

- **Caso recursivo** $n \geq 2$: Si $|m1| \wedge |m2| = k \geq 2$, se divide cada matriz, $m1$ y $m2$, en 4 submatrices de igual tamaño: $m111$, $m112$, $m121$, $m122$, $m211$, $m212$, $m221$, y $m222$. Este es el paso *divide* en la estrategia *divide y venceras*. Se genera díez matrices s_i como resultado de operaciones aritméticas entre las ocho submatrices generadas a partir de las matrices m_1 y m_2 de entrada:

$$\begin{aligned}
s1 &= m2_{21} - m2_{11} \\
s2 &= m1_{11} + m1_{12} \\
s3 &= m1_{21} + m1_{22} \\
s4 &= m2_{12} - m2_{22} \\
s5 &= m1_{11} + m1_{22} \\
s6 &= m2_{11} + m2_{22} \\
s7 &= m1_{12} - m1_{22} \\
s8 &= m2_{21} + m2_{22} \\
s9 &= m1_{11} - m1_{21} \\
s10 &= m2_{11} + m2_{12}
\end{aligned}$$

Luego, se realizan siete llamadas recursivas (como se definió en el documento proporcionado como guía de laboratorio) a *multStrassen* para calcular productos intermedios de forma recursiva:

$$
\begin{aligned}
p1 &= \text{multStrassen}(m1_{11}, s1) \\
p2 &= \text{multStrassen}(s2, m2_{22}) \\
p3 &= \text{multStrassen}(s3, m2_{11}) \\
p4 &= \text{multStrassen}(m1_{22}, s4) \\
p5 &= \text{multStrassen}(s5, s6) \\
p6 &= \text{multStrassen}(s7, s8) \\
p7 &= \text{multStrassen}(s9, s10)
\end{aligned}
$$

Finalmente, se combinan los resultados intermedios para formar la matriz resultado m_r:

$$
\begin{aligned}
c_{11} &= p5 + p4 - p2 + p6 \\
c_{12} &= p1 + p2 \\
c_{21} &= p3 + p4 \\
c_{22} &= p5 + p1 - p3 - p7
\end{aligned}
$$

- **Invariantes**:
  - **Sobre el tamaño**: Las matrices $m1, m2$ cumplen al comienzo de cada paso recursivo

    $$m1, m2 \ \in \mathbb{R}^{n-k \times n-k} \wedge |m1| = |m2| = 2^{n-k}, \quad k \in \mathbb{N}^+, \quad 0 \leq k \leq n$$

    Nótese que

    $$\lim_{k \to n}(n-k) = 0$$

  - **Sobre la división en submatrices**: En cada llamada recursiva, se divide una matriz de entrada en cuatro submatrices: $m111$, $m112$, $m121$, $m122$, $m211$, $m212$, $m221$ y $m222$. Esas submatrices representan $\frac{1}{4}$ de la matriz original y corresponden a una de las cuatro regiones dadas por $k, k/2$ sobre el tamaño de sus filas y columnas.

    $$\text{Invariante: } m1 = \begin{bmatrix} m111 & m112 \\ m121 & m122 \end{bmatrix}, \quad m2 = \begin{bmatrix} m211 & m212 \\ m221 & m222 \end{bmatrix}$$

    y $|m_{\{1,2\}}| = \frac{k-n}{2}$
  - **Sobre la multiplicación**: Las llamadas recursivas a multStrassen se hacen sobre submatrices, y la tupla de resultados $(s1, s2, \ldots, p7)$ se combinan de forma bien definida para retornar una matriz m_r como resultado de la recursión. La hipótesis inductiva asume que los productos de matrices se realizan de forma correcta en cada k-ésimo paso.

    $$\text{Invariante: } c_{ij} = \text{multStrassen}(m1ij, m2ij)$$

– **Sobre la combinación**: La combinación de los productos de las submatrices forma la matriz resultado m_r, la posición está definida de acuerdo a las guardas sobre los índices i, j y corresponde a una región "*cartesiana*". La hipótesis inductiva asegura que cada submatriz se coloca en el cuadrante pertinente dentro de la matriz resultante.

$$\text{Invariante: } m_r[i][j] = \begin{cases} c_{11}[i][j] & \text{si } i < l \wedge j < l \\ c_{12}[i][j-l] & \text{si } i < l \wedge j \geq l \\ c_{21}[i-l][j] & \text{si } i \geq l \wedge j < l \\ c_{22}[i-l][j-l] & \text{si } i \geq l \wedge j \geq l \end{cases}$$

– **Sobre el caso base**: Se alcanza cuando $|m_{\{1,2\}}| = 1$. Por tanto, el algoritmo asegura tratar la matriz más pequeña posible para $n \geq 1$.

$$\text{Invariante: } n = 1$$

multStrassenPar

```
1   def multStrassenPar(m1:Matriz, m2:Matriz): Matriz ={
2       //recibe m1 y m2 matrices cuadradas de la misma dimension
            , potencia de 2
3       //y devuelve la multiplicacion de las 2 matrices
4       val n = m1.length
5
6       /*if(umbral <= n){
7           multMatrizRec(m1,m2)
8       }*/
9       if(n == 1) {
10          Vector(Vector(m1(0)(0)*m2(0)(0)))
11      }
12      else {
13          val l = n/2
14          val (s1, s2, s3, s4, s5, s6, s7, s8, s9, s10) = (
15              restaMatriz(subMatriz(m2,0,1,1),
16                          subMatriz(m1,1,1,1)),
17              sumMatriz(subMatriz(m1,0,0,1),
18                        subMatriz(m1,0,1,1)),
19              sumMatriz(subMatriz(m1,1,0,1),
20                        subMatriz(m1,1,1,1)),
21              restaMatriz(subMatriz(m2,1,0,1),
22                          subMatriz(m2,0,0,1)),
23              sumMatriz(subMatriz(m1,0,0,1),
24                        subMatriz(m1,1,1,1)),
25              sumMatriz(subMatriz(m2,0,0,1),
26                        subMatriz(m2,1,1,1)),
27              restaMatriz(subMatriz(m1,0,1,1),
28                          subMatriz(m1,1,1,1)),
29              sumMatriz(subMatriz(m2,1,0,1),
```

```
                        subMatriz(m2,1,1,1)),
                restaMatriz(subMatriz(m1,0,0,1),
                            subMatriz(m1,1,0,1)),
                sumMatriz(subMatriz(m2,0,0,1),
                        subMatriz(m2,0,1,1))
            )
            val (p1, p2, p3, p4, p5, p6, p7) = (
                task(multStrassenPar(subMatriz(m1,0,0,1), s1)),
                task(multStrassenPar(s2, subMatriz(m2,1,1,1))),
                task(multStrassenPar(s3, subMatriz(m2,0,0,1))),
                task(multStrassenPar(subMatriz(m1,1,1,1), s4)),
                task(multStrassenPar(s5, s6)),
                task(multStrassenPar(s7, s8)),
                task(multStrassenPar(s9, s10))
            )
            val (c_11, c_12, c_21, c_22) = (
                restaMatriz(sumMatriz(p5.join(), p4.join()),
                   sumMatriz(p6.join(), p2.join())),
                sumMatriz(p1.join(),p2.join()),
                sumMatriz(p3.join(),p4.join()),
                restaMatriz(sumMatriz(p1.join(), p5.join()),
                   restaMatriz(p3.join(), p7.join()))
            )
            Vector.tabulate(n,n)((i,j)=>
                if(i<l && j<l) c_11(i)(j)
                else if(i<l && j>=l) c_12(i)(j-l)
                else if(i>=l && j<l) c_21(i-l)(j)
                else c_22(i-l)(j-l))
        }
    }
```

Listing 11: mult Strassen paralela

La función `multStrassenPar` en Scala realiza la multiplicación de matrices utilizando el método de Strassen de manera paralela, siguiendo los siguientes puntos clave:

1. **Estructura y Pasos Básicos**: La función sigue la misma estructura lógica del algoritmo de Strassen, incluyendo:
   - El caso base cuando $n = 1$, donde se realiza una multiplicación directa de escalares.
   - La división de las matrices en submatrices y el cálculo de las matrices intermedias $s1$ a $s10$.
   - El cálculo de los productos $p1$ a $p7$ usando llamadas recursivas a `multStrassenPar`.
   - La combinación de estos productos para formar la matriz resultante m_r.
2. **Paralelización**: Se introduce la paralelización mediante la función `task`, que inicia operaciones en hilos separados. Las operaciones clave paralelizadas son:
   - El cálculo de $p1$ a $p7$, permitiendo ejecuciones simultáneas y mejorando la eficiencia en sistemas con múltiples núcleos.

- El uso de `join` en variables como `p1.join()`, asegurando que los resultados estén disponibles para los pasos subsiguientes.

3. **Mantenimiento de la Correctitud**: La paralelización no altera el orden de las operaciones matemáticas necesarias para el método de Strassen, manteniendo la correctitud del algoritmo.

4. **Optimización y Eficiencia**: La paralelización está diseñada para mejorar la eficiencia del algoritmo, especialmente para matrices grandes, reduciendo el tiempo de ejecución en comparación con la versión secuencial. Nuevamente, como se argumento en la seción $multMatrizRecPar$, de (10) y (11) se hace una comparación entre los tiempos de ejecución de un algoritmo en forma secuencial y en forma paralela. Supongase una operación $m_1 \cdot m_2$ cuya ejecución $S\ S =< T_1, T_2, ...T_i, ..., T_{n-1}, T_n >$ representa n tareas T secuenciales y otra ejecución paralela $P\ P = \{T_1, T_2, ...T_i, ..., T_{n-1}, T_n\}$ de n tareas T paralelas. De nuevo,

$$\exists\ k_0 \mid k_0 \in N^+ \ \wedge\ n\ \geq\ k_0 \rightarrow\ \phi(P) \leq \phi(S) = \max(\phi_i(T_i)) \leq \sum_{1}^{n} \phi(T_i)$$

Existirá, por lo tanto, un número de tamaño de matriz, k_0, de modo que la ejecución en forma paralela tomará un tiempo de computor menor en la determinación del resultado $m_1 \cdot m_2$. De nuevo, éste es el umbral(en nuestro caso, 2^6) buscado para condicionar el código y hacerlo más eficiente en las diferentes condiciones de ejecución.

2.2 Informe de desempeño de las funciones secuenciales y de las funciones paralelas

2.2.1 Metodología de Generación de Matrices de Prueba

Para la metodologia de la Generación de las matrices de prueba usamos los siguientes algoritmos creados por nosotros:

```
def compareAlgmMRPmSP(m1: Matriz, m2: Matriz): (Double,Double,
    Double) ={
  compararAlgoritmos(multMatrizRecPar, multStrassenPar)(m1, m2)
}
```

Listing 12: compareAlgo

Define una función `compareAlgmMRPmSP`, la cual compara dos algoritmos de multiplicación de matrices paralelos: `multMatrizRecPar` y `multStrassenPar`. La función toma dos matrices, `m1` y `m2`, como argumentos y devuelve una tupla de tres elementos `(Double, Double, Double)` que representa las métricas de comparación de estos algoritmos.
Para hacer la comparativa utilizamos lo siguiente:

```
def automaticBenchmark(a:(Matriz, Matriz)=>(Double,Double, Double
    ), b: Vector[Vector[Vector[Int]]]): List[(Double,Double,Double
    )] ={
  (for {i <- 0 until b.length by 2} yield {
    a(b(i), b(i+1))
```

```
    }).toList
  }
```

Listing 13: autoBench

La función `automaticBenchmark` en Scala está diseñada para realizar evaluaciones automáticas de rendimiento sobre un algoritmo de multiplicación de matrices. La función toma como argumentos una función de multiplicación de matrices `a`, que devuelve una tupla con tres valores `Double`, y un vector de vectores de vectores de enteros `b`, que representa pares de matrices. La función itera sobre estos pares y aplica la función `a` a cada par, recolectando los resultados en una lista.

Inicializacion de datos:
Puesto que queriamos que fueran las mismas matrices para todos y pensando en que una iteracion en un for podría cambiar eso usamos el siguiente vector:

```
    val values =Vector(
        Vector(matrizAlAzar(pow(2, 0).toInt, 2)
        ,matrizAlAzar(pow(2, 0).toInt, 2)
        ,matrizAlAzar(pow(2, 1).toInt, 2)
        ,matrizAlAzar(pow(2, 1).toInt, 2)
        ,matrizAlAzar(pow(2, 2).toInt, 2)
        ,matrizAlAzar(pow(2, 2).toInt, 2)
        ,matrizAlAzar(pow(2, 3).toInt, 2)
        ,matrizAlAzar(pow(2, 3).toInt, 2)
        ,matrizAlAzar(pow(2, 4).toInt, 2)
        ,matrizAlAzar(pow(2, 4).toInt, 2)
        ,matrizAlAzar(pow(2, 5).toInt, 2)
        ,matrizAlAzar(pow(2, 5).toInt, 2)
        ,matrizAlAzar(pow(2, 6).toInt, 2)
        ,matrizAlAzar(pow(2, 6).toInt, 2)
        ,matrizAlAzar(pow(2, 7).toInt, 2)
        ,matrizAlAzar(pow(2, 7).toInt, 2)
        ,matrizAlAzar(pow(2, 8).toInt, 2)
        ,matrizAlAzar(pow(2, 8).toInt, 2))
```

Listing 14: Vector

2.2.2 Resultados de Multiplicación de Matrices y análisis de Resultados

Tabla comparativa y análisis de desempeño.

Matrix size	mulMatriz(ms)	multMatrizPar(ms)	speedup
2^0	0.0208	0.0126	1.65
2^1	0.054	0.0899	0.60
2^2	0.1658	0.2217	0.75
2^3	0.6989	1.003	0.70
2^4	2.2913	4.7724	0.48
2^5	2.8232	5.416	0.52
2^6	22.1308	36.1349	0.61
2^7	172.5227	249.1038	0.69
2^8	1391.6189	1747.6074	0.80
2^9	11624.2295	13095.9631	0.88
2^{10}	92733.1786	92184.1319	1.005

Table 1: mulMatriz vs multMatrizPar

La tabla proporcionada muestra los tiempos de ejecución de dos implementaciones de multiplicación de matrices, una secuencial (`mulMatriz`) y otra paralela (`multMatrizPar`), para diferentes tamaños de matrices, expresados como potencias de 2. Además, se presenta el *speedup* de la versión paralela en comparación con la secuencial. A continuación, se analizan diversos aspectos de estas implementaciones:

1. **¿Cuál de las implementaciones es más rápida?**
   Para matrices de tamaño pequeño (hasta 2^6), la versión secuencial (`mulMatriz`) muestra un mejor rendimiento. Sin embargo, a partir de matrices de tamaño 2^7, la implementación paralela (`multMatrizPar`) empieza a ser más eficiente, superando ligeramente a la versión secuencial en tamaños de 2^{10}.

2. **¿De qué depende que la aceleración sea mejor?**
   La aceleración se ve influenciada principalmente por el tamaño de la matriz. Para tamaños pequeños, el *overhead* de la paralelización puede no compensar sus beneficios. A medida que el tamaño aumenta, los beneficios del paralelismo superan este *overhead*, resultando en un mejor *speedup*.

3. **¿En qué casos es mejor usar la versión secuencial o paralela de cada algoritmo?**

   - **Versión Secuencial (`mulMatriz`):** Ideal para matrices pequeñas (hasta 2^6), donde el *overhead* del paralelismo no se justifica.
   - **Versión Paralela (`multMatrizPar`):** Recomendada para matrices de gran tamaño (a partir de 2^7), donde el coste del paralelismo se ve compensado por la eficiencia en el procesamiento de grandes volúmenes de datos.

Matrix size	mulMatrizRecPar(ms)	multStrassen(ms)	speedup
n^0	0.0882	0.0248	3.56
n^1	0.2109	0.1308	1.61
n^2	0.3769	0.4245	0.89
n^3	0.9086	1.5833	0.57
n^4	0.8598	4.6825	0.18
n^5	17.6741	30.5887	0.58
n^6	51.2801	232.0896	0.22
n^7	389.73	1692.9611	0.23
n^8	3114.4097	5199.9472	0.60

Table 2: mulMatrizRecPar vs multStrassen

Analizamos la eficiencia de dos implementaciones de algoritmos para multiplicar matrices: `mulMatrizRecPar` (una versión paralela del algoritmo de multiplicación de matrices recursiva) y `multStrassen` (el algoritmo de Strassen):

1. **¿Cuál de las Implementaciones es Más Rápida?**
   - Para tamaños de matriz pequeños (n^0, n^1), `multStrassen` es más rápido.
   - A medida que el tamaño de la matriz aumenta, `mulMatrizRecPar` comienza a superar a `multStrassen`, especialmente en tamaños muy grandes (n^5, n^6, n^7, n^8).
2. **¿De Qué Depende que la Aceleración Sea Mejor?**
   - La aceleración depende del tamaño de la matriz, la complejidad del algoritmo, los recursos de hardware disponibles y la sobrecarga de la paralelización.
   - Para matrices más grandes, la paralelización en `mulMatrizRecPar` se vuelve más efectiva.
3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
   - **Versión Secuencial (`multStrassen`):** Preferible para matrices más pequeñas y sistemas con recursos limitados de procesamiento paralelo.
   - **Versión Paralela (`mulMatrizRecPar`):** Beneficiosa para matrices grandes y sistemas con múltiples núcleos de procesador.

La elección entre `mulMatrizRecPar` y `multStrassen` depende del tamaño de la matriz y los recursos de hardware disponibles. Para matrices pequeñas, `multStrassen` es generalmente preferible, mientras que para matrices grandes, `mulMatrizRecPar` suele ofrecer un mejor rendimiento debido a su capacidad para paralelizar tareas.

Matrix size	mulMatrizRec(ms)	multStrassenPar(ms)	speedup
n^0	0.0541	0.035	1.55
n^1	0.033	0.2113	0.16
n^2	0.1192	0.33	0.36
n^3	0.5392	2.0168	0.27
n^4	1.1699	5.0879	0.23
n^5	8.4393	11.5936	0.73
n^6	67.8253	64.9989	1.04
n^7	567.7783	451.8261	1.26

Table 3: mulMatrizRec vs multStrassenPar

Analizando la eficiencia de dos implementaciones de algoritmos para multiplicar matrices: `mulMatrizRec` (versión secuencial) y `multStrassenPar` (versión paralela del algoritmo de Strassen):

1. **¿Cuál de las Implementaciones es Más Rápida?**
   - Para matrices muy pequeñas (n^0), `multStrassenPar` es más rápido.
   - Para tamaños de matriz intermedios (n^1, n^2, n^3, n^4), `mulMatrizRec` es significativamente más rápido.
   - Para matrices grandes (n^5, n^6, n^7), los tiempos se vuelven más competitivos, con `multStrassenPar` siendo ligeramente más rápido en los tamaños más grandes.
2. **¿De Qué Depende que la Aceleración Sea Mejor?**
   - La aceleración depende del tamaño de la matriz, la complejidad del algoritmo, los recursos de hardware y la eficiencia de la paralelización.
   - `mulMatrizRec` es más eficiente para tamaños de matriz pequeños a intermedios.
   - `multStrassenPar` muestra una mejora en matrices más grandes, donde la paralelización compensa la mayor complejidad del algoritmo de Strassen.
3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
   - **Versión Secuencial (`mulMatrizRec`):** Preferible para tamaños de matriz pequeños a intermedios.
   - **Versión Paralela (`multStrassenPar`):** Más adecuada para matrices grandes.

La elección entre `mulMatrizRec` y `multStrassenPar` depende del tamaño de la matriz. Para matrices más pequeñas, la versión secuencial es generalmente más rápida, mientras que para matrices más grandes, la versión paralela del algoritmo de Strassen puede ofrecer un mejor rendimiento.

Matrix size	multMatrizRec(ms)	multStrassen(ms)	Speedup
n^0	0.0195	0.0129	1.51
n^1	0.0584	0.0403	1.45
n^2	0.0926	0.3343	0.28
n^3	0.6009	0.9823	0.61
n^4	1.2007	2.0959	0.57
n^5	5.8948	10.1365	0.58
n^6	49.869	70.9033	0.70
n^7	415.2096	529.4741	0.78
n^8	3099.9929	3523.037	0.88
n^9	25379.6906	25235.8456	1.01

Table 4: multMatrizRec vs multStrassen

Analizamos la eficiencia de dos implementaciones de algoritmos para multiplicar matrices: `multMatrizRec` y `multStrassen`, basándonos en la tabla de rendimiento proporcionada:

1. **¿Cuál de las Implementaciones es Más Rápida?**
   - Para tamaños de matriz más pequeños (n^0, n^1), `multStrassen` es más rápido.
   - A medida que el tamaño de la matriz aumenta, la diferencia de velocidad se reduce y para tamaños muy grandes (n^8, n^9), ambos algoritmos muestran tiempos de ejecución comparables.
2. **¿De Qué Depende que la Aceleración Sea Mejor?**
   - La aceleración depende del tamaño de la matriz y de la complejidad inherente de cada algoritmo.
   - `multMatrizRec` es más eficiente para tamaños de matriz grandes, mientras que `multStrassen` es más efectivo para matrices más pequeñas.
3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
   - **Versión Secuencial (`multMatrizRec`):** Preferible para matrices grandes, donde su rendimiento es relativamente mejor.
   - **Versión Paralela (`multStrassen`):** Más adecuada para matrices más pequeñas, donde ofrece un mejor rendimiento.

La elección entre `multMatrizRec` y `multStrassen` debe basarse en el tamaño de la matriz, con `multStrassen` siendo preferible para matrices más pequeñas y `multMatrizRec` para matrices más grandes.

Matrix size	multMatrizRec(ms)	multMatrizRecPar(ms)	Speedup
n^0	0.0535	0.0495	1.08
n^1	0.0632	0.1054	0.60
n^2	0.1286	0.1518	0.85
n^3	0.403	0.4911	0.82
n^4	5.5323	3.9688	1.39
n^5	8.5952	4.7139	1.82
n^6	67.7342	37.6179	1.80
n^7	551.341	300.7862	1.83
n^8	4441.5969	2399.4469	1.85

Table 5: multMatrizRec vs multMatrizRecPar

Procedemos a analizar la eficiencia de dos implementaciones de algoritmos para multiplicar matrices: `multMatrizRec` (versión secuencial) y `multMatrizRecPar` (versión paralela):

1. **¿Cuál de las Implementaciones es Más Rápida?**
   - Para tamaños de matriz más pequeños (n^0, n^1, n^2, n^3), `multMatrizRec` es más rápido o comparable.
   - A medida que el tamaño de la matriz aumenta (n^4 a n^8), `multMatrizRecPar` se vuelve significativamente más rápido.
2. **¿De Qué Depende que la Aceleración Sea Mejor?**
   - La aceleración depende del tamaño de la matriz y de la eficacia de la paralelización.
   - Para matrices más grandes, la paralelización en `multMatrizRecPar` ofrece una ventaja significativa.
3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
   - **Versión Secuencial (`multMatrizRec`):** Preferible para tamaños de matriz más pequeños, donde la sobrecarga de la paralelización no se justifica.
   - **Versión Paralela (`multMatrizRecPar`):** Más adecuada para matrices grandes, donde la paralelización mejora el rendimiento.

Matrix size	multMatriz(ms)	multMatrizPar(ms)	Speedup
n^0	0.157074	0.508098	0.31
n^1	0.058667	0.259741	0.23
n^2	0.06202	0.304369	0.20
n^3	0.241862	0.186058	1.30
n^4	1.196248	0.536174	2.23
n^5	15.896863	2.565981	6.20
n^6	234.5659	21.393817	10.96
n^7	3373.980202	374.861453	9.00
n^8	52778.116937	10691.328672	4.94

Table 6: multMatriz vs multMatrizPar

Analizamos la eficiencia de dos implementaciones de algoritmos para multiplicar matrices: `multMatriz` y `multMatrizPar`, basándonos en la tabla de rendimiento proporcionada:

1. **¿Cuál de las Implementaciones es Más Rápida?**
   - Para tamaños de matriz pequeños (n^0, n^1, n^2), `multMatriz` es más rápida.
   - A medida que el tamaño de la matriz aumenta (n^3 en adelante), `multMatrizPar` se vuelve significativamente más rápida.
2. **¿De Qué Depende que la Aceleración Sea Mejor?**
   - La aceleración depende del tamaño de la matriz, la eficiencia de la paralelización y la complejidad computacional del algoritmo.
   - El speedup incrementa con el aumento del tamaño de la matriz, favoreciendo a `multMatrizPar` en matrices grandes.
3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
   - **Versión Secuencial (`multMatriz`):** Preferible para tamaños de matriz pequeños.
   - **Versión Paralela (`multMatrizPar`):** Más adecuada para matrices grandes, donde la paralelización mejora significativamente el rendimiento.

Matrix size	multMatriz(ms)	multMatrizRec(ms)	Speedup
n^0	0.197233	0.124877	1.58
n^1	0.098057	0.02626	3.73
n^2	0.072356	0.128369	0.56
n^3	0.128857	0.153372	0.84
n^4	1.26574	0.863941	1.47
n^5	15.580685	6.217719	2.51
n^6	233.389835	57.153132	4.08
n^7	3756.499596	466.273799	8.06
n^8	74007.560303	3649.418623	20.28

Table 7: multMatriz vs multMatrizRec

Analizamos la eficiencia de dos implementaciones de algoritmos para multiplicar matrices: `multMatriz` y `multMatrizRec`, basándonos en la tabla de rendimiento proporcionada:

1. **¿Cuál de las Implementaciones es Más Rápida?**
   - Para tamaños de matriz pequeños (n^0 y n^1), `multMatrizRec` es más rápida.
   - A medida que el tamaño de la matriz aumenta, `multMatrizRec` sigue siendo más eficiente, con un speedup significativo en tamaños de matriz grandes (n^5 a n^8).
2. **¿De Qué Depende que la Aceleración Sea Mejor?**

- La aceleración depende del tamaño de la matriz, la eficiencia de la implementación del algoritmo y las optimizaciones específicas de cada versión.
- El speedup incrementa con el aumento del tamaño de la matriz, favoreciendo a `multMatrizRec`.

3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
   - **Versión Secuencial (`multMatriz`):** Podría ser preferible para tamaños de matriz extremadamente pequeños.
   - **Versión Paralela (`multMatrizRec`):** Claramente más adecuada para todos los tamaños de matriz, especialmente para matrices grandes.

Matrix size	multMatriz(ms)	multMatrizRecPar(ms)	Speedup
n^0	0.213017	0.054686	3.90
n^1	0.05727	0.261487	0.22
n^2	0.093867	0.193461	0.49
n^3	0.138426	0.931548	0.15
n^4	1.035612	0.553565	1.87
n^5	15.601226	4.07065	3.83
n^6	260.689531	30.724248	8.48
n^7	4699.357314	223.874599	20.99
n^8	94123.683973	1796.040497	52.41

Table 8: multMatriz vs multMatrizRecPar

1. **¿Cuál de las Implementaciones es Más Rápida?**
   - Para el tamaño de matriz más pequeño (n^0), `multMatrizRecPar` es más rápida.
   - Para tamaños de matriz intermedios (n^1, n^2, n^3), `multMatriz` es más eficiente.
   - A medida que el tamaño de la matriz aumenta (n^4 en adelante), `multMatrizRecPar` se vuelve significativamente más rápida, con un aumento considerable en el speedup.

2. **¿De Qué Depende que la Aceleración Sea Mejor?**
   - La aceleración depende del tamaño de la matriz, la eficiencia de la paralelización y la complejidad computacional del algoritmo.
   - Para matrices más grandes, la paralelización en `multMatrizRecPar` ofrece una ventaja significativa, como se refleja en el aumento del speedup.

3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
   - **Versión Secuencial (`multMatriz`):** Preferible para tamaños de matriz pequeños a intermedios, donde la sobrecarga de la paralelización no se justifica.
   - **Versión Paralela (`multMatrizRecPar`):** Más adecuada para matrices grandes, donde la paralelización mejora el rendimiento de manera significativa.

Matrix size	multMatriz(ms)	multStrassenPar(ms)	Speedup
n^0	0.0271	0.0113	2.40
n^1	0.05	0.0886	0.56
n^2	0.0298	0.1949	0.15
n^3	0.1785	0.687	0.26
n^4	0.9352	2.2692	0.41
n^5	10.9968	4.8675	2.26
n^6	199.8272	38.1879	5.23
n^7	3721.3193	240.8355	15.45
n^8	77156.0959	1724.9942	44.73

Table 9: multMatriz vs multStrassenPar

1. **¿Cuál de las Implementaciones es Más Rápida?**
   - Para el tamaño de matriz más pequeño (n^0), `multStrassenPar` es más rápida.
   - A medida que el tamaño de la matriz aumenta, `multMatriz` se vuelve menos eficiente en comparación con `multStrassenPar`, especialmente en tamaños de matriz grandes (n^5 en adelante), donde `multStrassenPar` muestra un speedup significativo.
2. **¿De Qué Depende que la Aceleración Sea Mejor?**
   - La aceleración depende del tamaño de la matriz y de la eficacia de la paralelización en `multStrassenPar`.
   - La optimización y la menor complejidad computacional del algoritmo de Strassen en su versión paralela contribuyen a un mejor speedup para matrices más grandes.
3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
   - **Versión Secuencial (`multMatriz`):** Puede ser más adecuada para tamaños de matriz extremadamente pequeños.
   - **Versión Paralela (`multStrassenPar`):** Claramente más adecuada para matrices más grandes, aprovechando la eficiencia del algoritmo de Strassen y la paralelización.

Matrix size	multStrassen(ms)	multMatrizPar(ms)	Speedup
n^0	0.0187	0.0708	0.26
n^1	0.0472	0.0979	0.48
n^2	0.1763	0.1098	1.61
n^3	1.4824	0.1596	9.29
n^4	1.3893	0.2277	6.10
n^5	10.0845	2.1428	4.71
n^6	71.4526	39.3428	1.82
n^7	524.8294	660.7993	0.79
n^8	3715.7102	13099.9935	0.28

Table 10: multStrassen vs multMatrizPar

1. **¿Cuál de las Implementaciones es Más Rápida?**

   - Para tamaños de matriz más pequeños (n^0, n^1), `multStrassen` es más rápida.
   - A medida que el tamaño de la matriz aumenta (n^2 en adelante) hasta n^6, `multMatrizPar` se vuelve más eficiente, mostrando un mejor speedup, (n^6 en adelante) el algoritmo *Strassen* comienza a mostrar un mejor tiempo.

2. **¿De Qué Depende que la Aceleración Sea Mejor?**

   - La aceleración depende del tamaño de la matriz, la complejidad del algoritmo y la eficacia de la paralelización.
   - `multMatrizPar` muestra un mejor rendimiento en matrices "intermedias o pequeñas", pero al final "Strassen" presenta mejores resultados para las matrices grandes.

3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**

   - **Versión Paralela (`multMatrizPar`):** Preferible para tamaños de matriz más pequeños, donde la paralelización puede ser aprovechada para mejorar el rendimiento.
   - **Versión Secuencial (`multStrassen`):** Más adecuada para matrices grandes, donde su eficiencia y menor complejidad computacional sobresalen.

Matrix size	multStrassen(ms)	multStrassenPar(ms)	Speedup
n^0	0.094566	0.018089	5.23
n^1	0.164617	0.266935	0.62
n^2	0.082203	0.486237	0.17
n^3	0.551889	0.492383	1.12
n^4	2.002918	2.132335	0.94
n^5	11.121369	9.672645	1.15
n^6	83.202037	56.999489	1.46
n^7	591.982072	397.681511	1.49
n^8	4214.366236	2770.456839	1.52

Table 11: multStrassen vs multStrassenPar

1. **¿Cuál de las Implementaciones es Más Rápida?**

   - Para el tamaño de matriz más pequeño (n^0), `multStrassenPar` es más rápida.
   - Para tamaños de matriz intermedios (n^1, n^2), `multStrassen` es más eficiente.
   - A medida que el tamaño de la matriz aumenta (n^3 en adelante), `multStrassenPar` muestra un mejor rendimiento, especialmente en tamaños de matriz grandes (n^6, n^7, n^8).

2. **¿De Qué Depende que la Aceleración Sea Mejor?**

   - La aceleración depende del tamaño de la matriz, la complejidad del algoritmo y la eficacia de la paralelización.
   - En tamaños de matriz más grandes, las ventajas de la paralelización en `multStrassenPar` se vuelven más evidentes, mejorando la eficiencia.

3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**

   - **Versión Secuencial (`multStrassen`):** Más adecuada para tamaños de matriz pequeños y medianos, donde la paralelización no ofrece una ventaja significativa.
   - **Versión Paralela (`multStrassenPar`):** Mejor para matrices más grandes, donde el speedup debido a la paralelización supera a la versión secuencial.

Matrix size	multMatrizRec(ms)	multMatrizPar(ms)	Speedup
n^0	0.069562	0.198001	0.35
n^1	0.046654	0.318967	0.15
n^2	0.501463	0.312541	1.60
n^3	0.171252	0.516339	0.33
n^4	0.914017	0.475552	1.92
n^5	6.9931	2.383554	2.93
n^6	52.731112	44.138359	1.19
n^7	459.95615	668.513945	0.69
n^8	3509.126971	13464.049988	0.26

Table 12: multMatrizRec vs multMatrizPar

1. **¿Cuál de las Implementaciones es Más Rápida?**

   - Para tamaños de matriz más pequeños (n^0, n^1), `multMatrizRec` es más rápida.
   - A medida que el tamaño de la matriz aumenta, `multMatrizPar` muestra un mejor rendimiento en ciertos tamaños (n^2, n^4, n^5), pero no de forma consistente.
   - En tamaños de matriz muy grandes (n^7, n^8), `multMatrizPar` es menos eficiente que `multMatrizRec`.

2. **¿De Qué Depende que la Aceleración Sea Mejor?**

   - La aceleración depende del tamaño de la matriz, la complejidad del algoritmo y la eficacia de la paralelización.
   - El speedup es variable y parece depender de cómo cada algoritmo gestiona diferentes tamaños de matrices y de la sobrecarga asociada a la paralelización.

3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**

   - **Versión Secuencial (`multMatrizRec`):** Más adecuada para tamaños de matriz muy pequeños y muy grandes.
   - **Versión Paralela (`multMatrizPar`):** Puede ofrecer ventajas en tamaños de matriz intermedios, pero su eficacia varía.

Matrix size	multStrassenPar(ms)	multMatrizPar(ms)	Speedup
n^0	0.0183	0.0677	0.27
n^1	0.068	0.0845	0.80
n^2	0.1879	0.1407	1.34
n^3	0.6789	0.2121	3.20
n^4	2.3105	0.4184	5.52
n^5	5.0187	2.1041	2.39
n^6	35.2597	41.5603	0.85
n^7	254.7461	671.1146	0.38
n^8	1759.3622	13238.1516	0.13

Table 13: multStrassenPar vs multMatrizPar

1. **¿Cuál de las Implementaciones es Más Rápida?**

   - Para tamaños de matriz más pequeños (n^0, n^1), `multStrassenPar` y `multMatrizPar` muestran un rendimiento similar, con `multStrassenPar` siendo ligeramente más eficiente.
   - A medida que el tamaño de la matriz aumenta, `multStrassenPar` muestra un mejor rendimiento, especialmente en tamaños de matriz grandes (n^6 en adelante).

2. **¿De Qué Depende que la Aceleración Sea Mejor?**

   - La aceleración depende del tamaño de la matriz, la complejidad del algoritmo y la eficacia de la paralelización.
   - `multStrassenPar` parece ser más eficiente en la gestión de recursos y paralelización para matrices de mayor tamaño.

3. **¿Cuándo Usar la Versión Paralela de Cada Algoritmo?**

   - **multStrassenPar:** Más adecuado para tamaños de matriz intermedios y grandes, donde su algoritmo optimizado ofrece un mejor rendimiento.
   - **multMatrizPar:** Puede ser más eficiente en tamaños de matriz pequeños a intermedios, pero su rendimiento disminuye en comparación con `multStrassenPar` en matrices más grandes.

Matrix size	multMatrizRecPar(ms)	multMatrizPar(ms)	Speedup
n^0	0.0702	0.0311	2.26
n^1	0.0424	0.1279	0.33
n^2	0.0463	0.1546	0.30
n^3	0.2225	0.7962	0.28
n^4	1.8466	2.8966	0.64
n^5	19.7595	14.4537	1.37
n^6	325.1826	104.5296	3.11
n^7	5908.247	727.7273	8.12
n^8	114646.5989	5121.9016	22.38

Table 14: multMatrizRecPar vs multMatrizPar

1. **¿Cuál de las Implementaciones es Más Rápida?**

   - Para el tamaño de matriz más pequeño (n^0), `multMatrizRecPar` es más rápida.
   - A medida que el tamaño de la matriz aumenta, `multMatrizPar` muestra un mejor rendimiento, especialmente en tamaños de matriz grandes (n^5 en adelante), donde el speedup es más notable.

2. **¿De Qué Depende que la Aceleración Sea Mejor?**

   - La aceleración depende del tamaño de la matriz, la complejidad del algoritmo y la eficacia de la paralelización.
   - En tamaños de matriz más grandes, las ventajas de la paralelización en `multMatrizPar` se vuelven más evidentes, mejorando la eficiencia.

3. **¿Cuándo Usar la Versión Paralela de Cada Algoritmo?**

   - **multMatrizRecPar:** Más adecuado para tamaños de matriz pequeños y medianos, donde su algoritmo optimizado ofrece un mejor rendimiento.
   - **multMatrizPar:** Mejor para matrices más grandes, donde la paralelización puede ser aprovechada para mejorar el rendimiento.

2.2.3 Analisis comparativo paralelismo de tareas (secuencial vs secuencial ∧ paralela vs paralela):

Matrix size	multMatriz (ms)	multMatrizRec (ms)	multStrassen (ms)
n^0	0.208	0.195	00.94566
n^1	0.54	0.584	01.64617
n^2	1.658	0.926	00.82203
n^3	6.989	6.009	05.51889
n^4	22.913	12.007	20.02918
n^5	28.232	58.948	111.21369
n^6	221.308	498.69	832.02037
n^7	1725.227	4152.096	5919.82072
n^8	13916.189	30999.929	42143.66236
n^9	116242.295	253796.906	252358.456

Table 15: resultados segun la multiplicación por metodología secuencial

1. **¿Cuál de las Implementaciones es Más Rápida?**

   - Para matrices pequeñas (hasta n^3), las implementaciones tienen un rendimiento similar, aunque `multStrassen` comienza a mostrar un rendimiento ligeramente inferior en n^2 y n^3.
   - En el rango de n^4 a n^6, `multMatrizRec` muestra una eficiencia notablemente mejor.
   - Para matrices más grandes (n^7, n^8 y n^9), `multMatriz` es significativamente más rápida. `multStrassen` es la más lenta en estos tamaños.

2. **¿De Qué Depende que la Aceleración Sea Mejor?**

- La eficiencia de cada algoritmo depende del tamaño de la matriz y de las optimizaciones específicas de cada uno.
- `multStrassen` es teóricamente más eficiente para matrices grandes, pero su complejidad adicional puede no ser práctica para ciertos tamaños.

3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
    - `multMatriz`**:** Mejor para matrices más grandes.
    - `multMatrizRec`**:** Eficaz para matrices de tamaño intermedio.
    - `multStrassen`**:** A pesar de su promesa teórica, menos eficiente en la práctica para los tamaños probados.

Matrix size	multMatrizPar (ms)	multMatrizRecPar (ms)	multStrassenPar (ms)
n^0	0.126	0.882	00.183
n^1	0.899	2.109	00.68
n^2	2.217	3.769	01.879
n^3	10.03	9.086	06.789
n^4	47.724	8.598	23.105
n^5	54.16	176.741	50.187
n^6	361.349	512.801	352.597
n^7	2491.038	3897.3	2547.461
n^8	17476.07	31144.097	17593.622

Table 16: resultados segun la multiplicación por metodología paralela

1. **¿Cuál de las Implementaciones es Más Rápida?**
    - Para matrices pequeñas (n^0, n^1), `multStrassenPar` muestra el mejor rendimiento.
    - A medida que el tamaño de la matriz aumenta, `multStrassenPar` sigue siendo eficiente, especialmente en tamaños más grandes (n^4 en adelante).
    - `multMatrizRecPar` y `multMatrizPar` muestran un rendimiento competitivo en tamaños de matriz más grandes, aunque generalmente son superados por `multStrassenPar`.

2. **¿De Qué Depende que la Aceleración Sea Mejor?**
    - La aceleración depende del tamaño de la matriz y de la eficacia de la paralelización en cada implementación.
    - `multStrassenPar`, con su enfoque más complejo pero teóricamente más eficiente, parece beneficiarse más de la paralelización en tamaños de matriz grandes.

3. **¿Cuándo Usar la Versión Paralela de Cada Algoritmo?**
    - **multStrassenPar:** Recomendado para todos los tamaños de matriz, especialmente eficaz para matrices más grandes.
    - **multMatrizRecPar y multMatrizPar:** Pueden ser consideradas para tamaños de matriz pequeños a medianos, aunque su rendimiento puede ser menos predecible.

2.2.4 Paralelismo de datos:

Vector size	prodPunto(ms)	prodPuntoParD(ms)	speedup
2^0	0.0437	0.2868	0.1524
2^1	0.0129	0.3295	0.0392
2^2	0.0152	0.5742	0.0265
2^3	0.0127	0.3947	0.0322
2^4	0.0302	0.6982	0.0433
2^5	0.0292	0.8443	0.0346
2^6	0.0309	0.6806	0.0454
2^7	0.106	1.2756	0.0831
2^8	0.0517	1.3694	0.0378
2^9	0.0905	0.7633	0.1186
2^{10}	0.4051	1.5263	0.2654

Table 17: Comparativa entre prodPunto y prodPuntoParD

1. **¿Cuál de las Implementaciones es Más Rápida?**
   - Para tamaños de vector más pequeños (hasta 2^7), `prodPunto` es más rápido que `prodPuntoParD`.
   - A medida que el tamaño del vector aumenta (desde 2^8 en adelante), `prodPuntoParD` comienza a mostrar un mejor rendimiento en comparación con `prodPunto`, como lo indica el aumento en el valor de speedup.
2. **¿De Qué Depende que la Aceleración Sea Mejor?**
   - La aceleración depende del tamaño del vector y de la eficacia de la paralelización.
   - Para vectores más grandes, la paralelización en `prodPuntoParD` parece ofrecer una ventaja significativa, como se refleja en el incremento del speedup.
3. **¿Cuándo Usar la Versión Secuencial o Paralela de Cada Algoritmo?**
   - `prodPunto`: Mejor para vectores de tamaño pequeño a mediano, donde la sobrecarga de la paralelización no se justifica.
   - `prodPuntoParD`: Más adecuada para vectores de mayor tamaño, donde la paralelización mejora significativamente el rendimiento.

Sin embargo a pesar de los cambios en el speedup, podemos concluir que para, por lo menos estos datos, la paralelización no es eficiente para el paralelismo de datos, aunque como vimos anteriormente en el paralelismo de tareas si lo es.

3 Conclusiones

Tras analizar las diferentes variantes de las funciones de multiplicación de matrices y sus resultados, llegamos a las siguientes conclusiones:

1. **Eficiencia en Diferentes Tamaños de Matrices:** Las funciones de multiplicación de matrices varían en eficiencia según el tamaño de las matrices. Implementaciones estándar, como `multMatriz`, son eficientes en matrices grandes, mientras que variantes como `multMatrizRec`, `multMatrizRecPar`, y `multStrassenPar` muestran un mejor rendimiento en tamaños específicos. `multStrassenPar` es especialmente eficiente en matrices grandes debido a su optimización algorítmica y paralelización.

2. **Importancia de la Paralelización:** La paralelización es crucial para mejorar el rendimiento en matrices de tamaño considerable. Las versiones paralelas generalmente superan a las secuenciales en matrices grandes, aprovechando los recursos de hardware modernos.

3. **Trade-offs entre Complejidad y Rendimiento:** Algoritmos como Strassen ofrecen una complejidad teórica reducida, pero la sobrecarga adicional puede no justificar su uso en todos los casos. Esto se refleja en el rendimiento variable de `multStrassen` y `multStrassenPar`.

4. **Elección del Algoritmo Apropiado:** La elección adecuada del algoritmo depende del tamaño de la matriz y del entorno computacional. Mientras que las implementaciones secuenciales pueden ser adecuadas para matrices pequeñas, las versiones paralelas son preferibles para matrices más grandes en entornos que soportan paralelización eficiente.