Taller 5: Multiplicación De Matrices En Paralelo

Angie Natalia Córdoba Collazos 2124366 - 3743
Wilson Andrés Mosquera Zapata 2182116 - 3743

Noviembre 2023

Informe del taller

Informe de Correcciones:

- **Función multMatriz:**

```
Java
 // 1.1.1. Producto cruz (Versión estándar secuencial).
  def multMatriz(m1: Matriz, m2: Matriz): Matriz = {
      val longitud = m1.length
      val m2Transpuesta = transpuesta(m2)
      Vector.tabulate(longitud, longitud)((i, j) => prodPunto(m1(i),
m2Transpuesta(j)))
  }
```

Para esta argumentación tendremos en cuenta lo siguiente:

- Sea A una matriz de dimensiones n×n, y B una matriz de dimensiones n×n.
- Denotemos la entrada en la fila i y columna j de la matriz C resultante de la multiplicación de A por B como C_{ij}
- La transpuesta de una matriz B la denotamos como M^{T} .
- La función prodPunto(v1,v2) devuelve el producto punto de dos vectores v1 y v2.
- La función transpuesta(M) devuelve la transpuesta de la matriz M.

Considerando esto, recordemos que la multiplicación de matrices se define como:

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

Sabemos que **multMatriz** devuelve correctamente la matriz resultante de la multiplicación de dos matrices A y B, es decir, C=A×B.

➔ **Base de inducción:** Para n=1, la matriz A y B son matrices 1×1. En este caso, la multiplicación es simplemente el producto de sus únicas entradas, y la función multMatriz devuelve correctamente el resultado.

➔ **Hipotesis de inducción:** Supongamos que la función multMatriz es correcta para matrices de tamaño n×n.

➔ **Paso inductivo:** Queremos demostrar que la función es correcta para matrices de tamaño (n+1)×(n+1). Consideremos las matrices A y B de tamaño (n+1)×(n+1). Podemos expresarlas de la siguiente manera:

$$A = \begin{bmatrix} A' & a \\ a^T & a_{n+1,n+1} \end{bmatrix} B = \begin{bmatrix} B' & b \\ b^T & b_{n+1,n+1} \end{bmatrix}$$

Donde A′ y B′ son matrices de tamaño n×n, a y b son vectores de tamaño n, y $a_{n+1,n+1}$ y $b_{n+1,n+1}$ son escalares.
La multiplicación C = A × B resulta en una matriz de tamaño (n + 1) × (n + 1) con siguientes entradas:

$$c_{ij} = \sum_{k=1}^{n+1} A_{ik} \cdot B_{kj}$$

Para $i, j \leq n$, la entrada C_{ij} es calculada correctamente por la hipótesis de inducción. Para $i, j = n + 1,$ las entradas son:

$$C_{n+1,n+1} = \quad c_{ij} = \sum_{k=1}^{n+1} A_{(n+1)k} \cdot B_{k(n+1)}$$

Esto involucra las entradas $a, - \ b^{\mathrm{T}}, \ a_{n+1,n+1}, \ y \ b_{n+1,n+1}$. La función **multMatriz** correctamente calcula estas entradas usando las funciones auxiliares **prodPunto** y **transpuesta**.

Por inducción, la función **multMatriz** es correcta para matrices de cualquier tamaño n×n.

- **Función multMatrizPar:**

```java
// 1.1.2. Producto cruz (Versión estándar paralela).
 def multMatrizPar(m1: Matriz, m2: Matriz): Matriz = {
     // Obtener el número de filas de la matriz m1.
     val filasM1 = m1.length
     // Obtener el número de filas de la matriz m2.
     val filasM2 = m2.length
     // Calcular la matriz transpuesta de la matriz m2.
     val m2Transpuesta = transpuesta(m2)

     // Verificar si el número de filas en m1 es mayor o igual a 4
para aplicar paralelismo.
     if (filasM1 >= 4) {
     // Dividir m1 en dos partes iguales, m1Derecha y m1Izquierda.
```

```
      val mitad = filasM1 / 2
      val (m1Derecha, m1Izquierda) = m1.splitAt(mitad)

      // Crear tareas paralelas para multiplicar m1Derecha y
m1Izquierda por m2.
      val resultado1 = task {
      multMatrizPar(m1Derecha, m2)
      }
      val resultado2 = task {
      multMatrizPar(m1Izquierda, m2)
      }

      // Obtener los resultados de las tareas paralelas.
      val resultadoDerecha = resultado1.join()
      val resultadoIzquierda = resultado2.join()

      // Concatenar los resultados de las matrices multiplicadas.
      resultadoDerecha ++ resultadoIzquierda
      } else {
      // Calcular la matriz resultante en serie utilizando producto
punto entre filas de m1 y columnas de m2 transpuesta.
      Vector.tabulate(filasM1, filasM2) { (i, j) =>
      prodPunto(m1(i), m2Transpuesta(j))
      }
      }
  }
```

Vamos a analizar formalmente la corrección de la función **multMatrizPar**. La función realiza la multiplicación de dos matrices cuadradas de la misma dimensión, **m1** y **m2**, utilizando paralelismo. La estrategia de paralelismo se aplica dividiendo recursivamente las matrices hasta que el número de filas de la matriz sea menor que 4, momento en el cual se realiza la multiplicación de manera secuencial.

Para esta argumentación tendremos en cuenta lo siguiente:

- Sea m1 una matriz de tamaño n×n, y m2 su matriz transpuesta de tamaño n×n.
- Sea prodPunto(v1,v2) la función que calcula el producto punto entre dos vectores de la misma longitud.
- La matriz resultante C de la multiplicación m1×m2 se denota como C=m1×m2.

➔ **Caso Base :** si *filasM1* es menor que 4, entonces la función realiza la multiplicación de manera secuencial, y el resultado es correcto.

$$c_{ij} = \sum_{k=1}^{n+1} m1(i)_k \times m2(j)_k$$

➔ **Caso Recursivo:**

- si *filasM1* es mayor o igual a 4, entonces la función divide m1m1 en dos partes iguales, *m1Derecha* y *m1Izquierda,* y realiza llamadas recursivas para multiplicar estas submatrices por m2.

- La concatenación de los resultados de las submatrices es correcta, ya que corresponde a la multiplicación total de m1×m2.

$$c = \begin{pmatrix} C_{derecha} \\ C_{izquierda} \end{pmatrix}$$

- **Función multMatrizRec:**

```java
// 1.2.3 Multiplicando matrices recursivamente (Versión secuencial).
  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 l = m1.length
      val mitad = l/2

      if (l == 1) {
      // Caso base: multiplicación de matrices de 1x1
      Vector.tabulate(l, l)((i, j) => prodPunto(m1(i), m2(j)))
      } else {
      // Realizo subdivisiones una por una
      val A11 = subMatriz(m1, 0, 0, mitad)
      val A12 = subMatriz(m1, 0, mitad, mitad)
      val A21 = subMatriz(m1, mitad, 0, mitad)
      val A22 = subMatriz(m1, mitad, mitad, mitad)

      val B11 = subMatriz(m2, 0, 0, mitad)
      val B12 = subMatriz(m2, 0, mitad, mitad)
      val B21 = subMatriz(m2, mitad, 0, mitad)
      val B22 = subMatriz(m2, mitad, mitad, mitad)
```

```
      // Multiplico las subdivisiones y guardo resultados
      val C11 = sumMatriz(multMatrizRec(A11, B11), multMatrizRec(A12,
B21))
      val C12 = sumMatriz(multMatrizRec(A11, B12), multMatrizRec(A12,
B22))
      val C21 = sumMatriz(multMatrizRec(A21, B11), multMatrizRec(A22,
B21))
      val C22 = sumMatriz(multMatrizRec(A21, B12), multMatrizRec(A22,
B22))

      // Uno los resultados en dos partes
      val part1 = C11.zip(C12).map { case (row1, row2) => row1 ++
row2 }
      val part2 = C21.zip(C22).map { case (row1, row2) => row1 ++
row2 }

      // Uno las dos mitades para obtener el resultado final
      part1 ++ part2
      }
  }
```

➔ **Caso Base :** Cuando la dimensión de las matrices es 1x1, la función realiza correctamente la multiplicación utilizando el producto punto. Sea A y B matrices 1x1, entonces:

$$multMatrizRec(A, B) = [A_{11} \cdot B_{11}]$$

Esto cumple con el resultado esperado de la multiplicación de matrices de 1x1.

➔ **Hipótesis de inducción:** Supongamos que la función es correcta para matrices de dimensión n × n y n > 1.

➔ **Paso de inducción:** Queremos demostrar que la función es correcta para matrices de dimensión 2n×2n. Para ello, dividimos las matrices A y B en submatrices de tamaño n×n como sigue:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

La función se ejecuta recursivamente en estas submatrices y calcula las submatrices C_{11}, C_{12}, C_{21}, C_{22} como sigue:

$$C_{11} = multMatrizRec(A_{11}, B_{11}) + multMatrizRec(A_{12}, B_{21})$$

$$C_{12} = multMatrizRec(A_{11}, B_{12}) + multMatrizRec(A_{12}, B_{22})$$

$$C_{21} = multMatrizRec(A_{21}, B_{11}) + multMatrizRec(A_{22}, B_{21})$$

$$C_{22} = multMatrizRec(A_{21}, B_{12}) + multMatrizRec(A_{22}, B_{22})$$

Luego, se combinan estas submatrices para formar la matriz resultante C:

$$C = \begin{bmatrix} C_{11} & A_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Esto demuestra que la función es correcta para matrices de dimensión 2n × 2n.

Así, tomando el principio de la inducción, podemos concluir que **multMatrizRec** es correcta para matrices de cualquier dimensión potencia de 2.

- **Función multRecPar:**

```java
// 1.2.4 Multiplicando matrices recursivamente (Versión paralela).
  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 l = m1.length
      val mitad = l/2
      val umbral = 100

      // Caso base: multiplicación de matrices de 1x1
      if (l == 1) {
      return Vector.tabulate(l, l)((i, j) => prodPunto(m1(i), m2(j)))
      }
```

```
      // Establezco un umbral
      if (m1.length <= umbral) {
      return multMatrizRec(m1, m2)
      }

      // Realizo subdivisiones una por una
      val A11 = subMatriz(m1, 0, 0, mitad)
      val A12 = subMatriz(m1, 0, mitad, mitad)
      val A21 = subMatriz(m1, mitad, 0, mitad)
      val A22 = subMatriz(m1, mitad, mitad, mitad)

      val B11 = subMatriz(m2, 0, 0, mitad)
      val B12 = subMatriz(m2, 0, mitad, mitad)
      val B21 = subMatriz(m2, mitad, 0, mitad)
      val B22 = subMatriz(m2, mitad, mitad, mitad)

      // Multiplico las subdivisiones y guardo resultados en tareas
(task)
      val tC11 = task(sumMatriz(multMatrizRecPar(A11, B11),
multMatrizRecPar(A12, B21)))
      val tC12 = task(sumMatriz(multMatrizRecPar(A11, B12),
multMatrizRecPar(A12, B22)))
      val tC21 = task(sumMatriz(multMatrizRecPar(A21, B11),
multMatrizRecPar(A22, B21)))
      val tC22 = task(sumMatriz(multMatrizRecPar(A21, B12),
multMatrizRecPar(A22, B22)))

      // Hago join a los tasks
      val C11 = tC11.join()
      val C12 = tC12.join()
      val C21 = tC21.join()
      val C22 = tC22.join()

      // Uno los resultados en dos partes
      val part1 = C11.zip(C12).map { case (row1, row2) => row1 ++
row2 }
      val part2 = C21.zip(C22).map { case (row1, row2) => row1 ++
row2 }

      // Uno las dos mitades para obtener el resultado final
      part1 ++ part2
  }
```

Para esta argumentación tendremos en cuenta lo siguiente:

- Sea M_1 una matriz de m1.
- Sea M_2 una matriz de m1.
- Sea M_r una matriz resultante.
- Sea $M_r{}^{\mathrm{P}}$ la matriz resultante calculada por **multMatrizRecPar**.
- Sea l la dimensión de las matrices cuadradas (potencia de 2).
- Sea $l/2$ la mitad de la dimensión.
- Sea (i, j) una posición de la matriz.

➔ **Hipótesis de inducción:** Para 1 > 1, asumamos que **multMatrizRecPar** es correcta para matrices de dimensión: $\frac{1}{2} \times \frac{1}{2}$.

➔ **Caso Base** ($l = 1$): Para ($l = 1$), la función **multMatrizRecPar** realiza la multiplicación de matrices de 1 × 1 directamente y devuelve el resultado, lo cual es correcto trivialmente.

➔ **Paso inductivo:** Supongamos que la función multMatrizRecPar es correcta para matrices de dimensión $\frac{1}{2} \times \frac{1}{2}$. Queremos demostrar que es correcta para matrices de dimensión $l \times l$.

La función divide las matrices M_1 y M_2 en submatrices de tamaño $\frac{1}{2} \times \frac{1}{2}$. Luego, realiza la multiplicación de estas submatrices recursivamente y suma los resultados de manera paralela.

Sea $C^{\mathrm{P}}{}_{11}$ el resultado de multiplicar las submatrices correspondientes en la esquina superior izquierda de $M^{\mathrm{P}}{}_r$. Similarmente definimos $C^{\mathrm{P}}{}_{12}$, $C^{\mathrm{P}}{}_{21}$, $C^{\mathrm{P}}{}_{22}$.

Caso $C^{\mathrm{P}}{}_{11}$, Caso $C^{\mathrm{P}}{}_{12}$, $C^{\mathrm{P}}{}_{21}$ y $Caso$ $C^{\mathrm{P}}{}_{22}$:

$$C^{\mathrm{P}}{}_{11} = sumMatriz(multMatrizRecPar(A_{11}, B_{11}), multMatrizRecPar(A_{12}, B_{21}))$$

Por nuestra hipótesis de inducción, sabemos que $multMatrizRecPar(A_{11}, B_{11})$ y $multMatrizRecPar(A_{12}, B_{21})$ son correctas para dimensiones $\frac{1}{2} \times \frac{1}{2}$. Además, la función sumMatriz realiza la suma de matrices correctamente. Por lo tanto, $C^{\mathrm{P}}{}_{11}$ está calculada correctamente. En el caso $C^{\mathrm{P}}{}_{12}$, es similar. Utilizando las submatrices

A_{11}, B_{12}, A_{12} y B_{22}. Por último, los casos C^{P}_{21} y C^{P}_{22} son análogos a los casos anteriores.

Finalmente, la función **multMatrizRecPar** une estas submatrices correctamente para obtener la matriz resultante M^{P}_{r}. Puesto que la función está utilizando la correcta aplicación de la multiplicación de matrices en dimensiones más pequeñas (por hipótesis de inducción) y las operaciones de suma son correctas, podemos concluir que la función **multMatrizRecPar** es correcta para matrices de dimensión $l \times l$.

- **Función subMatriz:**

```
Java
 / 1.2.1. Extrayendo submatrices.
  def subMatriz(m:Matriz, i:Int, j:Int, l:Int):Matriz = {
      // Dada m, matriz cuadrada de NxN, 1<=i , j<=N, i+n<=N, j+n<=N,
      // devuelve la submatriz de nxn correspondiente a m[i..i+(n-1),
j..j+(n-1)]
      Vector.tabulate(l, l)((x, y) => m(i+x)(y+j))
  }
```

La función devuelve la submatriz de tamaño lxl correspondiente a m[i..i+(l-1), j..j+(l-1)]. Podemos argumentar su corrección utilizando inducción estructural sobre l. La base inductiva sería l = 1, donde la submatriz resultante es simplemente el elemento en la posición (i, j) de la matriz original.

- ➔ **Base Inductiva:** $subMatriz(m, i, j, 1) = [m(i, j)]$
- ➔ **Paso Inductivo:** Supongamos que la función es correcta para l = k. Queremos demostrar que también es correcta para l = k + 1. La submatriz resultante sería:

$$subMatriz(m, i, j, k{+}1) = \begin{bmatrix} m(i,j) & m(i,j+1) & \dots & m(i,j+k) \\ m(i+1,j) & m(i+1,j+1) & \dots & m(i+1,j+k) \\ \dots & \dots & \dots & \dots \\ m(i+k,j) & m(i+k,j+1) & \dots & m(i+k,j+k) \end{bmatrix}$$

- **Función sumMatriz:**

```
Java
 // 1.2.2. Sumando matrices.
  def sumMatriz(m1:Matriz, m2:Matriz):Matriz = {
      // recibe m1 y m2 matrices cuadradas de la misma dimension,
potencia de 2
      // y de vuelve la matriz resultante de la suma de las 2
matrices
      val l = m1.length
      Vector.tabulate(l, l)((i, j) => m1(i)(j) + m2(i)(j))
  }
```

La función devuelve la matriz resultante de la suma de dos matrices cuadradas de la misma dimensión, que es una operación bien definida para matrices del mismo tamaño. Podemos argumentar su corrección utilizando inducción estructural sobre la longitud de las matrices.

➔ Base Inductiva: Para matrices de tamaño 1x1, la función devuelve la matriz resultante de sumar los elementos correspondientes, lo cual es correcto.

➔ Paso Inductivo: Supongamos que la función es correcta para matrices de tamaño kxk. Queremos demostrar que también es correcta para matrices de tamaño (k+1)x(k+1). La suma de matrices sería:

$$sumMatriz(m1, m2) = \begin{bmatrix} m_1(0,0) + m_2(0,0) & m_1(0,1) + m \\ m_1(1,0) + m_2(1,0) & m_1(1,1) + m \\ \vdots & \vdots \\ m_1(k,0) + m_2(k,0) & m_1(k,1) + m \end{bmatrix}$$

- **Función restaMatriz:**

```
Java
 // 1.3.1. Restando matrices.
  def restaMatriz(m1: Matriz, m2: Matriz): Matriz = {
    // recibe m1 y m2 matrices cuadradas de la misma dimension,
potencia de 2
    // y devuelve la matriz resultante de la resta de las 2 matrices
    val l = m1.length
    Vector.tabulate(l, l)((i, j) => m1(i)(j) - m2(i)(j))
  }
```

La función restaMatriz, es similar a la argumentación de sumMatriz

- **Función multStrassen:**

Java

```
// 1.3.2. Algoritmo de Strassen (versión secuencial)
  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 usando el
algoritmo de Strassen
    val l = m1.length
    val mitad = l / 2

    // Caso base: Matrices de 1x1
    if (l == 1) {
      // Multiplicación simple
      return Vector.tabulate(l, l)((i, j) => prodPunto(m1(i), m2(j)))
    } else {

      // Subdivisión de las matrices de entrada
      val A11 = subMatriz(m1, 0, 0, mitad)
      val A12 = subMatriz(m1, 0, mitad, mitad)
      val A21 = subMatriz(m1, mitad, 0, mitad)
      val A22 = subMatriz(m1, mitad, mitad, mitad)

      val B11 = subMatriz(m2, 0, 0, mitad)
      val B12 = subMatriz(m2, 0, mitad, mitad)
      val B21 = subMatriz(m2, mitad, 0, mitad)
      val B22 = subMatriz(m2, mitad, mitad, mitad)

      // Cálculos de las matrices S y P
      val S1 = restaMatriz(B12, B22)
      val S2 = sumMatriz(A11, A12)
      val S3 = sumMatriz(A21, A22)
      val S4 = restaMatriz(B21, B11)
      val S5 = sumMatriz(A11, A22)
      val S6 = sumMatriz(B11, B22)
      val S7 = restaMatriz(A12, A22)
      val S8 = sumMatriz(B21, B22)
      val S9 = restaMatriz(A11, A21)
```

```
      val S10 = sumMatriz(B11, B12)

      val P1 = multStrassen(A11, S1)
      val P2 = multStrassen(S2, B22)
      val P3 = multStrassen(S3, B11)
      val P4 = multStrassen(A22, S4)
      val P5 = multStrassen(S5, S6)
      val P6 = multStrassen(S7, S8)
      val P7 = multStrassen(S9, S10)

      // Cálculos de las matrices C
      val C11 = restaMatriz(sumMatriz(sumMatriz(P5, P4), P6), P2)
      val C12 = sumMatriz(P1, P2)
      val C21 = sumMatriz(P3, P4)
      val C22 = restaMatriz(restaMatriz(sumMatriz(P5, P1), P3), P7)

      // Combinación de resultados en dos partes
      val parte1 = C11.zip(C12).map { case (fila1, fila2) => fila1 ++
fila2 }
      val parte2 = C21.zip(C22).map { case (fila1, fila2) => fila1 ++
fila2 }

      // Uno las dos mitades para obtener el resultado final
      parte1 ++ parte2
    }
  }
```

La función multStrassen en Scala implementa el algoritmo de Strassen para multiplicar matrices. Para demostrar su corrección, mostraremos que para cualquier par de matrices m1 y m2, el resultado de la función es equivalente a la multiplicación estándar de matrices.

➔ **Base Inductiva:**

Para matrices de tamaño 1x1, la función multStrassen realiza una multiplicación simple, lo cual es trivialmente correcto. Por lo tanto:

$\forall a, b \in R\ multStrassen(Matriz(a), Matriz(b)) = Matriz(a \cdot b)$

➔ **Hipotesis de induccion:**

Supongamos que la función es correcta para matrices de tamaño *(k-1) × (k-1)* donde *k>1*. Es decir para cualquier M1 y M2 de tamaño *(k-1) × (k-1)* se cumple:

$multStrassen(M1, M2) = M1 \cdot M2$

➔ **Paso inductivo:**

Queremos demostrar que la función es correcta para matrices de tamaño k×k. Sea M1 y M2 de tamaño k×k. Entonces, según la función ***multStrassen***, se realiza la siguiente descomposición:

M1= $\begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix}$ **, M2=** $\begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix}$

Donde $A\ ij$ *y* $B\ ij$ son submatrices de tamaño $(k/2)\times(k/2)$.

Ahora, según la hipótesis de inducción, podemos afirmar que:

$\forall i, j \in \{1, 2\},\ multStrassen(A\ ij\ , B\ ij\ ) = A\ ij\ \bullet\ B\ ij$

Luego, la función realiza operaciones adicionales para combinar estas submatrices y obtener el resultado final. Por lo tanto, para cualquier par de matrices de tamaño k×k, se cumple:

$multStrassen(M\ 1\ , M\ 2\ ) = M\ 1\ \bullet\ M\ 2$

- **Función multStrassenPar:**

```java
// 1.3.2. Algoritmo de Strassen (versión paralela)
  def multStrassenPar(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 usando el
algoritmo de Strassen en paralelo
    val l = m1.length
    val mitad = l / 2
    val umbral = 128
```

```
// Caso base: Matrices de 1x1
if (l == 1) {
  // Multiplicación simple
  return Vector.tabulate(l, l)((i, j) => prodPunto(m1(i), m2(j)))
} else {
  // Establezco un umbral
  if (l <= umbral) {
    return multStrassen(m1, m2)
  } else {
    // Subdivisión de las matrices de entrada
    val A11 = subMatriz(m1, 0, 0, mitad)
    val A12 = subMatriz(m1, 0, mitad, mitad)
    val A21 = subMatriz(m1, mitad, 0, mitad)
    val A22 = subMatriz(m1, mitad, mitad, mitad)

    val B11 = subMatriz(m2, 0, 0, mitad)
    val B12 = subMatriz(m2, 0, mitad, mitad)
    val B21 = subMatriz(m2, mitad, 0, mitad)
    val B22 = subMatriz(m2, mitad, mitad, mitad)

    // Cálculos de las matrices S
    val S1 = restaMatriz(B12, B22)
    val S2 = sumMatriz(A11, A12)
    val S3 = sumMatriz(A21, A22)
    val S4 = restaMatriz(B21, B11)
    val S5 = sumMatriz(A11, A22)
    val S6 = sumMatriz(B11, B22)
    val S7 = restaMatriz(A12, A22)
    val S8 = sumMatriz(B21, B22)
    val S9 = restaMatriz(A11, A21)
    val S10 = sumMatriz(B11, B12)

    // Cálculos de las matrices P (en paralelo usando task)
    val tP1 = task(multStrassen(A11, S1))
    val tP2 = task(multStrassen(S2, B22))
    val tP3 = task(multStrassen(S3, B11))
    val tP4 = task(multStrassen(A22, S4))
    val tP5 = task(multStrassen(S5, S6))
    val tP6 = task(multStrassen(S7, S8))
    val tP7 = task(multStrassen(S9, S10))

    // Realizo el join de las matrices P
```

```
        val P1 = tP1.join()
        val P2 = tP2.join()
        val P3 = tP3.join()
        val P4 = tP4.join()
        val P5 = tP5.join()
        val P6 = tP6.join()
        val P7 = tP7.join()

        // Cálculos de las matrices C
        val C11 = restaMatriz(sumMatriz(sumMatriz(P5, P4), P6), P2)
        val C12 = sumMatriz(P1, P2)
        val C21 = sumMatriz(P3, P4)
        val C22 = restaMatriz(restaMatriz(sumMatriz(P5, P1), P3), P7)

        // Combinación de resultados en dos partes
        val part1 = Vector.tabulate(C11.length)(i => C11(i) ++
C12(i))
        val part2 = Vector.tabulate(C21.length)(i => C21(i) ++
C22(i))

        // Uno las dos mitades para obtener el resultado final.
        part1 ++ part2
      }
    }
  }
```

En esta función no se va a tocar a fondo la demostración ya que esta es una versión paralela de la funcion multStrassen por lo que su demostración resulta ser muy similar a esta, tan solo teniendo algunas modificaciones.
La paralelización de datos en esta funcion se realiza en la fase de cálculos de las matrices P mediante el uso de tareas **(task)**. La idea principal era dividir el cálculo de las matrices P en tareas más pequeñas y realizar estas tareas de manera concurrente.
Aquí cada tarea *tP1, tP2, ..., tP7* representa el cálculo de una de las matrices P de forma independiente, luego se realiza un *join* y se combinan los resultados. En esta función también se definió un umbral que define hasta qué punto realizar esta función de acuerdo con su rendimiento.

Informe de Desempeño de las Funciones Secuenciales y Paralelas:

En cuanto a los tiempos de ejecución de las funciones de este taller se tienen los siguientes:

Tamaño Matriz	Tiempo de Ejecucion [ms]					
[nxn]	multMatriz	multMatrizPar	multMatrizRec	multMatrizRecPar	multStrassen	multStrassenPar
2	0.0272	0.0142	0.0472	0.0959	0.0329	0.0268
8	0.0624	0.1694	0.3898	0.3839	0.2563	0.2471
64	3.4076	1.7352	66.561	66.7165	55.0367	54.6584
128	28.2687	11.5593	562.0714	390.3337	390.3337	289.0243
256	217.5566	100.0271	4668.3829	3592.1897	2848.095	2114.4789
512	1.750,25	959,99	37.076,15	29407,77	20168,04	14408,05

Estos resultados se obtuvieron a partir de evaluar cada función en el mismo par de matrices de dimensión nxn de dos elementos de 0´s y 1´s con ayuda de la función *probarAlgoritmo*, la cual fue definida en el paquete Benchmark. A continuación se realizará un análisis comparativo de todos los datos suministrados en la tabla anterior tomando como referencia los algoritmos secuenciales con su versión paralela para así sacar conclusiones.

Para la multiplicación estándar de matrices se tiene lo siguiente:

multMatriz vs multMatrizPar			
Tamaño Matriz	Tiempo de Ejecucion [ms]		
[nxn]	multMatriz	multMatrizPar	Aceleración
2	0.0272	0.0142	1.9154
8	0.0624	0.1694	0.3683
64	3.4076	1.7352	1.9638
128	28.2687	11.5593	2.4455
256	217.5566	100.0271	2.1749
512	1.750,25	959,99	1.8231

Al analizar estos resultados, se puede observar que se obtuvo una aceleración considerable al usar la versión paralela por lo que en este caso se puede concluir que la paralelización si sirvió, cabe mencionar que solo hubo un caso (matriz 8x8) en el cual la paralelización no obtuvo la aceleración deseada, consideramos que es debido a causas alternas a esto.

Para la multiplicación recursiva de matrices se tiene lo siguiente:

multMatrizRec vs multMatrizRecPar			
Tamaño Matriz	**Tiempo de Ejecucion [ms]**		
[nxn]	**multMatrizRec**	**multMatrizRecPar**	**Aceleración**
2	0.0472	0.0959	1.2276
8	0.3898	0.3839	1.0372
64	66.561	66.7165	1.0069
128	562.0714	390.3337	1.3540
256	4668.3829	3592.1897	1.3469
512	37.076,15	29407,77	1.3997

Como en el caso anterior, se obtuvo una aceleración importante al utilizar la versión paralela, es por ello que podemos concluir que en este caso la paralelización sí sirvió como se esperaba.

Para la multiplicación Strassen de matrices se tiene lo siguiente:

multStrassen vs multStrassenPar			
Tamaño Matriz	**Tiempo de Ejecucion [ms]**		
[nxn]	**multStrassen**	**multStrassenPar**	**Aceleración**
2	0.0329	0.0268	0.4921
8	0.2563	0.2471	1.0153
64	55.0367	54.6584	0.9976
128	390.3337	289.0243	1.4399
256	2848.095	2114.4789	1.2995
512	20168,04	14408,05	1.2607

Como se puede observar el tiempo de ejecución de multStrassenPar resultó ser mucho menor en la mayoría de los casos , por lo que resultó ser más eficiente en su versión paralela.

De acuerdo con lo observado y analizado anteriormente, se puede concluir que el algoritmo más eficiente de todos es multMatrizPar para matrices mayores o iguales a 8x8 y multMatriz para matrices menores a 8x8.

Informe de análisis comparativo de las diferentes soluciones:

- **Paralelización de tareas:** En general, vemos que las paralelizaciones de tareas sirvieron para reducir los tiempos de ejecución de los algoritmos. Sin embargo, el grado de mejora varió según el algoritmo y el tamaño de la matriz. Por ejemplo, en el caso del algoritmo multMatriz, la paralelización de tareas redujo el tiempo de ejecución en un factor de hasta 2,5. En el caso del algoritmo multMatrizRec, la paralelización de tareas redujo el tiempo de ejecución en un factor de hasta 4. En el caso del algoritmo multStrassen, la paralelización de tareas redujo el tiempo de ejecución en un factor de hasta 2.

- **Algoritmo de Strassen:** Por lo que pudimos notar el algoritmo de Strassen es un algoritmo más eficiente que el algoritmo tradicional. Sin embargo, la mejora de esta eficiencia depende del tamaño de la matriz. Por lo que para matrices pequeñas, el algoritmo tradicional es más eficiente que el algoritmo de Strassen. Esto se debe a que el costo de la sobrecarga de la paralelización es mayor que la mejora en la eficiencia del algoritmo de Strassen. Para matrices grandes, el algoritmo de Strassen es más eficiente que el algoritmo tradicional. Esto se debe a que el costo de la sobrecarga de la paralelización se reduce y la mejora en la eficiencia del algoritmo de Strassen es significativa.

Así podemos concluir que la implementación más rápida dependerá del tamaño de la matriz. Sabiendo esto decimos que para matrices pequeñas, la implementación más rápida es multMatriz. Esto se debe a que el costo de la sobrecarga de la paralelización es mayor que la mejora en la eficiencia de las otras implementaciones. Para matrices grandes, la implementación más rápida es multStrassenPar. Esto se debe a que el algoritmo de Strassen es más eficiente que los algoritmos tradicionales y la paralelización de tareas reduce significativamente el tiempo de ejecución.