Taller 5: Multiplicación de matrices en paralelo

Juan Francisco Díaz Frias Julián Ernesto Puyo

Abril 2024

1. Ejercicios de programación

En este taller, se implementarán diferentes algoritmos de multiplicación de matrices, y se analizarán sus desempeños en sus versiones secuencial y paralela, y se concluirá sobre la aceleración lograda usando **paralelización de tareas**. También se implementarán algoritmos para hallar el producto punto entre dos vectores, usando paralelización de datos y se analizará su utilidad. Para efectos de simpleza de las soluciones y facilidad de los análisis, supondremos que las matrices a multiplicar serán cuadradas, sus entradas sólo serán 0's y 1's y el número de filas y columnas serán una potencia de 2. Se podrá probar entonces las implementaciones con matrices de dimensiones $1 \times 1, 2 \times 2, 4 \times 4, 8 \times 8, \ldots, 128 \times 128, 256 \times 256, \ldots$. Se trabajará con **matrices inmutables**. Para el producto punto entre vectores, no tendremos restricciones sobre el tamaño de los vectores.

El taller, en o correspondiente a **paralelización de tareas** se desarrollará en cuatro pasos:

- Primero, se implementará el algoritmo estándar de multiplicación de matrices y una versión paralela de él.
- En segundo lugar, se implementará una versión recursiva de la multiplicación de matrices y su respectiva versión en paralelo.
- En tercer lugar, se implementará un conocido algoritmo de multiplicación de matrices denominado el *algoritmo de Strassen* y su versión paralela.
- Luego se harán las pruebas de los algoritmos sobre diferentes matrices de diferentes tamaños y se analizará cuándo las versiones paralelas se comportan mejor que la secuenciales.

El taller, en o correspondiente a **paralelización de datos** se desarrollará en dos pasos:

- Primero, se implementará el algoritmo estándar de producto punto de vectores y una versión paralela de él.
- Luego se harán las pruebas de los algoritmos sobre diferentes vectores de diferentes tamaños y se analizará cuándo las versiones paralelas se comportan mejor que la secuenciales.

El propósito de este taller, es que al finalizar, usted:

- haya entendido cómo funciona la multiplicación de matrices usando los métodos estándar, recursivo y de Strassen,
- sea capaz de reconocer problemas fuertemente paralelos,
- conozca cómo medir el desempeño de un algoritmo paralelo,
- vea cómo juntar cálculos paralelos en lotes,
- y, comprenda cómo, en la práctica, los patrones de acceso a la memoria pueden afectar el desempeño paralelo de los algoritmos.

Preliminares

Antes de empezar a resolver el taller, es importante definir algunos tipos de datos y funciones auxiliares básicas para este ejercicio.

Primero se define el tipo $Matriz$ para denotar los valores que se van a operar:

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

Usaremos la colección $Vector$ porque es similar a los arreglos, pero es inmutable. Por defecto, los índices de un vector de longitud n son $0, 1, 2, \ldots, (n-1)$.

Para generar matrices para sus pruebas, se provee la función `matrizAlAzar` que dados una dimensión de la matriz ($long$) y un límite ($vals$) para los valores que va a contener la matriz, devuelve una matriz de dimensión $long \times long$ con entradas enteras en el intervalo $[0, vals)$.

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

Por ejemplo, al invocar esta función se generan los siguientes resultados:

```
scala> matrizAlAzar(2,2)
val res35: Vector[Vector[Int]] = Vector(Vector(1, 0), Vector(0, 1))

scala> matrizAlAzar(4,2)
val res36: Vector[Vector[Int]] = Vector(Vector(0, 1, 0, 1), Vector(0, 0, 0, 1),
Vector(1, 0, 0, 1), Vector(1, 0, 1, 1))
```

```
scala> matrizAlAzar(8,2)
val res37: Vector[Vector[Int]] = Vector(Vector(0, 1, 1, 0, 1, 1, 0, 1), Vector(1, 0, 1, 0, 1, 1, 1, 0),
Vector(1, 1, 1, 1, 0, 0, 1, 0), Vector(0, 1, 1, 0, 1, 1, 0, 1), Vector(1, 1, 1, 1, 1, 1, 0, 0),
Vector(1, 0, 1, 0, 1, 0, 0, 1), Vector(0, 1, 0, 0, 1, 0, 0, 0), Vector(1, 1, 0, 0, 1, 1, 0, 0))
```

Por su parte, para generar vectores para sus pruebas se provee la función `vectorAlAzar` que dados una dimensión del vector (*long*) y un límite (*vals*) para los valores que va a contener el vector, devuelve un vector de dimensión *long* con entradas enteras en el intervalo $[0, vals)$.

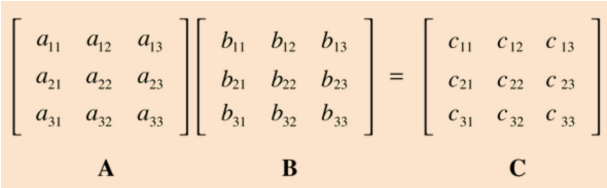
```
def vectorAlAzar(long:Int, vals:Int): Vector[Int] = {
  //Crea un vector de enteros de longitud  long,
  // con valores aleatorios entre 0 y vals
  val v = Vector.fill(long){random.nextInt(vals)}
  v
}
```

Antes de continuar definiendo funciones, recordaremos cómo se multiplican matrices (cuadradas en nuestro caso). Dadas dos matrices de dimensión $n \times n$, A y B, la multiplicación da como resultado una matriz C, también de dimensión $n \times n$ tal que

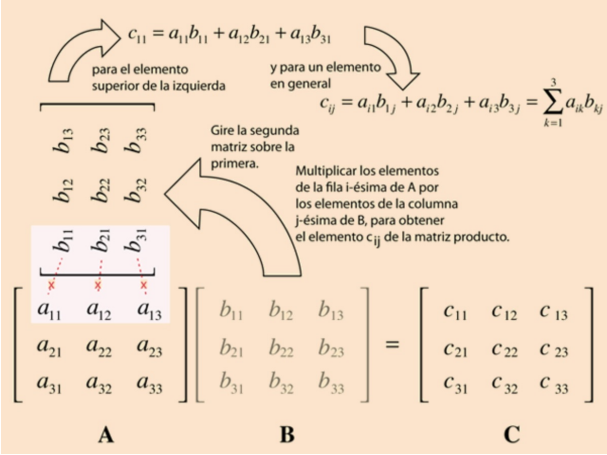
$$C_{ij} = A_{i[1..n]} \cdot B_{[1..n]j},$$

es decir la entrada (i, j) de C contiene el producto punto de la fila i de A con la columna j de B.

Gráficamente (imágenes tomadas de `http://hyperphysics.phy-astr.gsu.edu/hbasees/Math/matrix.html`)en el caso de 3×3 se puede ver así:



donde



1.1. Multiplicación estándar de matrices

Para implementar el método visto en la gráfica anterior, le puede ser útil usar las funciones `prodPunto` y `transpuesta` siguientes:

```
def prodPunto(v1:Vector[Int],v2:Vector[Int]):Int ={
  (v1 zip v2).map({case (i,j)=> (i*j)}).sum
}

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

La función `prodPunto` recibe dos vectores de la misma longitud, y devuelve el resultado del producto punto entre esos dos vectores. Nótese que C_{ij} es el producto punto de la fila i de A con la columna j de B.

La función `transpuesta` recibe una matriz (B) y devuelve su transpuesta (B^T). Recuerde que la transpuesta de una matriz de dimensión $n \times n$ es otra matriz de dimensión $n \times n$ cuyas filas (columnas) corresponden a las columnas (filas) de la matriz original. Formalmente:

$$B^T_{ij} = B_{ji}.$$

1.1.1. Versión estándar secuencial

Implemente la función `multMatriz` que reciba dos matrices cuadradas (A, B) de la misma dimensión y devuelva la matriz (C) correspondiente a la multiplicación de las matrices de entrada (AB). Use las dos funciones definidas previamente, `prodPunto` y `transpuesta`.

```
def multMatriz(m1:Matriz, m2:Matriz):Matriz = { ...
}
```

[**Ayuda:**] Use la función *Vector.tabulate*

1.1.2. Versión estándar paralela

Implemente la función `multMatrizPar` que reciba dos matrices cuadradas (A, B) de la misma dimensión y devuelva la matriz (C) correspondiente a la multiplicación de las matrices de entrada (AB), modificando ligeramente la función `multMatriz` para paralelizar algunas operaciones (usando las abstracciones `parallel` o `task`).

1.2. Multiplicación recursiva de matrices

En esta sección vamos a abordar al problema de multiplicación de matrices con un enfoque de *dividir y conquistar* que nos llevará diseñar a un algoritmo recursivo.

En este ejercicio es donde la suposición que la dimensión de la matriz es de la forma $n \times n, n = 2^k$ es importante para simplificar los cálculos.

Dados A y B matrices de dimensión $n \times n$ y $C = AB$, suponga que descomponemos cada matriz en 4 submatrices como muestra la figura:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Nótese que la ecuación $C = AB$ se expresaría así:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

donde:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

Cada una de las cuatro ecuaciones conlleva dos productos de matrices de dimensión $n/2 \times n/2$ (los cuales se hacen recursivamente con el mismo algoritmo que estamos diseñando para matrices de dimensión $n \times n$) y una suma de dos matrices de dimensión $n/2 \times n/2$.

1.2.1. Extrayendo submatrices

Para poder implementar la idea esbozada anteriormente, necesitamos poder extraer las submatrices de una matriz.

Implemente la función `subMatriz` que dadas una matriz (A) de dimensión $n \times n$, una posición ((i, j))de la matriz y una dimensión deseada (l) de la submatriz, devuelva la submatriz de A de dimensión $l \times l$ que comienza en la esquina A_{ij} de la matriz. Puede suponer que los índices especifican efectivamente una submatriz de A, es decir, que $1 \leq i, j \leq n \wedge i + l \leq n \wedge j + l \leq n$.

```
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)]
  ...
}
```

Por ejemplo, si deseamos extraer la submatriz A_{11} debemos invocar esta función con los índices $i = 0, j = 0, l = n/2$, pero si queremos la submatriz A_{22} debemos invocar esta función con los índices $i = n/2, j = n/2, l = n/2$.

[**Ayuda:**] Use la función *Vector.tabulate*

1.2.2. Sumando matrices

También hace falta, poder sumar matrices de la misma dimensión.

Implemente la función `sumMatriz` que dadas dos matrices (A, B) de dimensión $n \times n$, devuelve la matriz $A + B$ de dimensión $n \times n$.

```
def sumMatriz(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 suma de las 2 matrices
  ...
}
```

[**Ayuda:**] Use la función $Vector.tabulate$

1.2.3. Multiplicando matrices recursivamente, versión secuencial

Implemente la función `multMatrizRec` que dadas dos matrices (A, B) de dimensión $n \times n$, devuelve la matriz AB de dimensión $n \times n$, calculada usando la idea expresada anteriormente. Use las funciones `subMatriz` y `sumMatriz` en su implementación.

```
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
  ...
}
```

[**Ayuda:**] Use también la función $Vector.tabulate$

1.2.4. Multiplicando matrices recursivamente, versión paralela

Implemente la función `multMatrizRecPar` que dadas dos matrices (A, B) de dimensión $n \times n$, devuelve la matriz AB de dimensión $n \times n$, calculada usando la idea expresada anteriormente, pero usando paralelización de tareas (utilizando las abstracciones `parallel` o `task`). Use las funciones `subMatriz` y `sumMatriz` en su implementación.

```
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
  ...
}
```

[**Ayuda:**] Recuerde definir un umbral a partir del cual no se paraleliza nada. Use también la función $Vector.tabulate$

1.3. Multiplicación de matrices usando el algoritmo de Strassen

En 1969, Strassen publicó un algoritmo para multiplicación de matrices basado en la misma idea del algoritmo recursivo anterior, pero en lugar de hacer 8 multiplicaciones recursivas de matrices y 4 sumas, como en

$$
\begin{aligned}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{aligned}
$$

se hacen sólo 7 multiplicaciones recursivas de matrices y 18 sumas. Eso da lugar a un algoritmo, en teoría, más eficiente que el algoritmo anterior.

El algoritmo de Strassen está basado en:

1. El cálculo de las siguientes 10 matrices, a partir de sumas de las submatrices $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}$:

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

2. Posteriormente, el cálculo de las siguientes 7 multiplicaciones de matrices:

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 \\
P_2 &= S_2 \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 \\
P_5 &= S_5 \cdot S_6 \\
P_6 &= S_7 \cdot S_8 \\
P_7 &= S_9 \cdot S_{10}
\end{aligned}
$$

3. Y, por último el cálculo de $C_{11}, C_{12}, C_{21}, C_{22}$ a partir de las siguientes sumas de matrices:

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

Para entender por qué funcionan esas multiplicaciones desarrolle las sumas especificadas teniendo en cuenta que:

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \,, \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \,, \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \,, \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \,, \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \,, \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \,, \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \,.
\end{aligned}$$

1.3.1. Restando matrices

Ahora hace falta, poder restar matrices de la misma dimensión.

Implemente la función `restaMatriz` que dadas dos matrices (A, B) de dimensión $n \times n$, devuelve la matriz $A - B$ de dimensión $n \times n$.

```
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
  ...
}
```

[**Ayuda:**] Use la función *Vector.tabulate*

1.3.2. Algoritmo de Strassen, versión secuencial

Implemente la función `multStrassen` que dadas dos matrices (A, B) de dimensión $n \times n$, devuelve la matriz AB de dimensión $n \times n$, calculada usando el algoritmo de Strassen implementado secuencialmente. Use las funciones `subMatriz` , `sumMatriz` y `restaMatriz` en su implementación.

```
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
  ...
}
```

[**Ayuda:**] Use también la función *Vector.tabulate*

1.3.3. Algoritmo de Strassen, versión paralela

Implemente la función `multStrassenPar` que dadas dos matrices (A, B) de dimensión $n \times n$, devuelve la matriz AB de dimensión $n \times n$, calculada usando el algoritmo de Strassen implementado de forma paralela, usando paralelización de tareas (utilizando las abstracciones `parallel` o `task`). Use las funciones `subMatriz` , `sumMatriz` y `restaMatriz` en su implementación.

```
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
  ...
}
```

[**Ayuda:**] Use también la función $Vector.tabulate$

1.4. Evaluación comparativa

Para realizar el *benchmarking* de sus soluciones entregamos con este taller el paquete *Benchmark* para que usted pruebe las implementaciones de sus diferentes algoritmos de multiplicación de matrices.

Para usarlo usted simplemente importa el paquete *Benchmark* e invoca la función `compararAlgoritmos`, con dos implementaciones de la multiplicación de matrices que quiera comparar. Por ejemplo:

```
scala> compararAlgoritmos(multMatrizRec, multMatrizRecPar)(m1,m2)
val res13: (Double, Double, Double) = (80.672455,37.427045,2.1554588399912418)
```

La comparación da como resultado una tripleta de dobles: el primero y el segundo corresponden al tiempo en milisegundos que tomaron los algoritmos comparados en el orden en que se pasaron como argumentos. Y el tercero es la aceleración del segundo algoritmo con respecto al primero. Para los mismos algoritmos, usted puede pasar varias matrices de diferentes tamaños. Y puede entonces usarlos para recolectar los diferentes resultados y luego analizarlos. Por ejemplo:

```
for {
  i <- 1 to 10
  m1=matrizAlAzar(math.pow(2,i).toInt, 2)
  m2=matrizAlAzar(math.pow(2,i).toInt, 2)
} yield (compararAlgoritmos(multMatrizRec, multMatrizRecPar)(m1,m2),
  math.pow(2,i).toInt)
```

recolectaría datos de multiplicaciones de matrices de dimensiones $n \times n$ con $n \in \{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$.

- ¿Cuál de las implementaciones es más rápida?
- ¿De qué depende que la aceleración sea mejor?
- ¿Puede caracterizar los casos en que es mejor usar la versión secuencial/paralela de cada algoritmo de multiplicación de matrices?

1.5. Implementando el producto punto usando paralelismo de datos

Una manera de mejorar aún más el desempeño de los algoritmos anteriores, es tratando de mejorar el desempeño de algoritmos básicos de sus implementaciones. En este caso vamos a estudiar cómo mejorar el desempeño del algoritmo de cálculo del producto punto entre dos vectores de enteros.

Para ello simplemente se usará la colección paralela de vectores que provee Scala, $ParVector$, y se implementará la función `prodPuntoParD` tal cual se implementó para la colección $Vect$:

```
def prodPuntoParD(v1:ParVector[Int],v2:ParVector[Int]):Int ={
  (v1 zip v2).map({case (i,j)=> (i*j)}).sum
}
```

Su tarea en esta sección es (1) evaluar comparativamente las dos funciones `prodPunto` y `prodPuntoParD` e indicar para qué tamaños de vectores es mejor usar la primera y para qué tamaños de vectores es mejor usar la segunda y, (2) responder a la pregunta siguiente: ¿Será práctico construir versiones de los algoritmos de multiplicación de matrices del enunciado, para la colección $ParVector$ y usar `prodPuntoParD` en lugar de `prodPunto`? ¿Por qué si o por qué no?

Para realizar el *benchmarking* entregamos con este taller el paquete *Benchmark* para que usted pruebe las implementaciones de los dos algoritmos de cálculo secuencial y paralelo del producto punto.

Para usarlo usted simplemente importa el paquete *Benchmark* e invoca la función `compararProdPunto`, con el tamaño de vector con que quiere hacer la prueba. Por ejemplo:

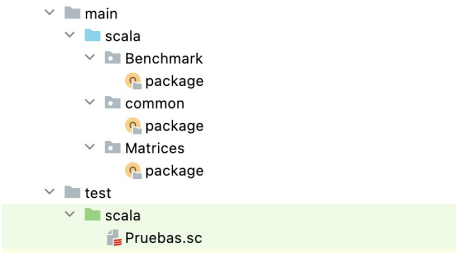
```
scala> compararProdPunto(100)
val res2: (Double, Double, Double) = (0.051141,0.64772,0.07895541283270549)
```

La comparación da como resultado una tripleta de dobles: el primero y el segundo corresponden al tiempo en milisegundos que tomaron los algoritmos secuencial y paralelo para el cálculo del producto punto, en ese orden. Y el tercero es la aceleración del segundo algoritmo con respecto al primero. Usted puede probarlos con vectores de diferentes tamaños. Y puede entonces recolectar los diferentes resultados y luego analizarlos.

2. Entrega

2.1. Paquete *Matrices* y *worksheet* de pruebas

Usted deberá entregar dos archivos `package.scala` y `pruebas.sc` los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación:



Las 6 funciones correspondientes a cada ejercicio, `multMatriz`, `multMatrizPar`, `multMatrizRec`, `multMatrizRecPar`, `multStrassen`, y `multStrassenPar` y las 3 funciones auxiliares `subMatriz`, `sumMatriz` y `restaMatriz` deben ser implementadas en un paquete de Scala denominado `Matrices`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
import common._
import scala.util.Random
package object Matrices {
  val random = new Random()

  type Matriz = Vector[Vector[Int]]

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

  def transpuesta(m:Matriz):Matriz = {
    val l=m.length
    Vector.tabulate(l,l)((i,j)=>m(j)(i))
  }

  def prodPunto(v1:Vector[Int],v2:Vector[Int]):Int ={
    (v1 zip v2).map({case (i,j)=> (i*j)}).sum
  }

    def prodPuntoParD(v1:ParVector[Int],v2:ParVector[Int]):Int ={ // A ser usada en el punto 1.5
    (v1 zip v2).map({case (i,j)=> (i*j)}).sum
  }

// Ejercicio 1.1.1
  def multMatriz(m1:Matriz, m2:Matriz):Matriz = {
...   }

// Ejercicio 1.1.2
  def multMatrizPar(m1:Matriz, m2:Matriz):Matriz = {
...   }

// Ejercicio 1.2.1
  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)]
...
  }

// Ejercicio 1.2.2
  def sumMatriz(m1:Matriz, m2:Matriz): Matriz ={
    // recibe m1 y m2 matrices cuadradas del mismo tamanio, potencia de 2
    //  y devuelve la suma de las 2 matrices
...
  }

// Ejercicio 1.2.3
  def multMatrizRec(m1:Matriz, m2:Matriz): Matriz ={
    // recibe m1 y m2 matrices cuadradas del mismo tamanioo, potencia de 2
    // y devuelve la multiplicacion de las 2 matrices
...   }

// Ejercicio 1.2.4
  def multMatrizRecPar(m1:Matriz, m2:Matriz): Matriz ={
    // recibe m1 y m2 matrices cuadradas del mismo tamanio, potencia de 2
```

```
    // y devuelve la multiplicacion de las 2 matrices, en paralelo
...  }


// Ejercicio 1.3.1
  def restaMatriz(m1:Matriz, m2:Matriz): Matriz ={
    // recibe m1 y m2 matrices cuadradas del mismo tamanio, potencia de 2
    //  y devuelve la resta de las 2 matrices
...  }

// Ejercicio 1.3.2
  def multStrassen(m1:Matriz, m2:Matriz): Matriz ={
    // recibe m1 y m2 matrices cuadradas del mismo tamanio, potencia de 2
    // y devuelve la multiplicacion de las 2 matrices usando el algoritmo de Strassen
...  }

// Ejercicio 1.3.3
  def multStrassenPar(m1:Matriz, m2:Matriz): Matriz ={
    // recibe m1 y m2 matrices cuadradas del mismo tamanio, potencia de 2
    // y devuelve la multiplicacion en paralelo de las 2 matrices usando el algoritmo de Strassen
...}
```

Dicho paquete será usado en un *worksheet* de Scala con casos de prueba. Estos casos de prueba deben venir en un archivo denominado `pruebas.sc` . Un ejemplo de un tal archivo es el siguiente:

```
import Matrices._

val m1= matrizAlAzar(16,2)
val m2= matrizAlAzar(16,2)


multMatriz(m1,m2)
multMatrizPar(m1,m2)

multMatrizRec(m1,m2)
multMatrizRecPar(m1,m2)

multStrassen(m1,m2)
multStrassenPar(m1,m2)
...
```

2.2. Informe del taller - secciones

Todo taller debe venir acompañado de un informe en formato pdf. El informe debe contener la información explícita solicitada en el enunciado.

Para este caso, el informe del taller debe contener al menos tres secciones: informe de corrección de las funciones implementadas, informe de desempeño de las funciones secuenciales y de las funciones paralelas, y análisis comparativo de las diferentes soluciones. El primero, como se ha venido haciendo en los otros talleres, corresponde a una argumentación del por qué están bien implementadas las funciones solicitadas. El segundo y el tercero, se pueden hacer usando las funciones `matrizAlAzar` provista en el paquete *Matrices* y `compararAlgoritmos` provista en el paquete *Benchmark*.

2.2.1. Informe de corrección

Es muy importante reflexionar sobre la corrección del código entregado. Para ello se deberá argumentar sobre la corrección de las funciones entregadas.

Para cada función de multiplicación de matrices, `multMatriz`, `multMatrizPar`, `multMatrizRec`,

`multMatrizRecPar`, `multStrassen`, y `multStrassenPar` argumente lo más formalmente posible por qué es correcta. Utilice inducción o inducción estructural donde lo vea pertinente. Estas argumentaciones las consigna en esta sección del informe.

2.2.2. Informe de desempeño de las funciones secuenciales y de las funciones paralelas

Su informe debe presentar una tabla con los resultados de correr los diferentes algoritmos de multiplicación de matrices para las mismas matrices. Es importante que genere diversos ejemplos con matrices de la misma dimensión, y hacerlo para muchas dimensiones (recuerde que las dimensiones deben ser potencias de 2). No olvide describir cómo generó los ejemplos de pruebas.

De igual manera su informe debe presentar una tabla con los resultados de correr las dos implementaciones de producto punto de vectores (usando las colecciones $Vector$ y $ParVector$, para los mismos vectores). En este caso los vectores pueden tener las dimensiones que quieran. Pueden crecer más rápido que de potencias de a dos (por ejemplo, por potencias de 10).

2.2.3. Análisis comparativo de las diferentes soluciones

Su informe debe presentar diferentes análisis basado en los resultados.

Para el caso de paralelismo de tareas, analizar cada versión secuencial versus su versión paralela correspondiente. Pero también analizar las diferentes versiones secuenciales entre ellas mismas, así como las versiones paralelas. ¿Las paralelizaciones sirvieron? ¿Es realmente más eficiente el algoritmo de Strassen? ¿No se puede concluir nada al respecto?

Igualmente, para el caso de paralelismo de datos, analizar la versión sobre la colección $Vector$ versus su versión paralela sobre $ParVector$.

2.3. Fecha y medio de entrega

Todo lo anterior, es decir la plantilla diligenciada del taller y el `Informe del taller`, debe ser entregado vía el campus virtual, a más tardar a las 10 a.m. del jueves 9 de mayo de 2024, en un archivo comprimido que traiga estos dos elementos.

Cualquier indicación adicional será informada vía el foro del campus asociado a *programación concurrente*.

3. Evaluación

Cada taller será evaluado tanto por el profesor del curso con ayuda del monitor, como por 2 compañeros que hayan entregado el taller. A este tipo de evaluación se le conoce como *Coevaluación*.

El objetivo de la coevaluación es lograr aprendizajes a través de:

- La lectura de lo que otros compañeros hicieron ante el mismo reto. Esto permite contrastar las soluciones propias con las de otros, y aprender de ellas, o compartir mejores maneras de hacer algo con otros.
- Retroalimentar a los compañeros que fueron asignados para evaluar. Al escribir la percepción que tenemos sobre el trabajo del otro, podemos aprender de cómo lo hicieron, y dar indicaciones al otro sobre otra formas de hacer lo mismo o, incluso, felicitarlo por la solución que presenta.
- La lectura de las retroalimentaciones de mis compañeros o del profesor/monitor.

La calificación de cada taller corresponderá entonces:

- en un 80 % a la calificación ponderada que reciba del profesor/monitor, vía una rúbrica de evaluación (pesa 5 veces lo que pesa la de otro estudiante) y de los tres compañeros asignados para evaluarlo.
- en un 20 % a la calificación que el sistema hará del trabajo de evaluación asignado. El Sistema tiene un método inteligente para estimar esa calificación, a partir de las evaluaciones realizadas por cada estudiante y por el profesor/monitor.