Programación Funcional y Concurrente

Alejandra Osorio Giraldo - 2266128
Luis Manuel Cardona Trochez - 2059942
Jose David Marmol Otero - 2266370

Carlos Andres Delgado Saavedra

Universidad del Valle
Sede Tuluá
2024

INFORME DE PROCESO

1) **Def matrizAlAzar:**

```
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
}
```

Esta función genera matrices cuadradas de tamaño aleatorio, rellenándolas con valores enteros aleatorios dentro de un rango especificado. Se utiliza para crear conjuntos de datos de prueba para evaluar el rendimiento de los diferentes algoritmos de multiplicación de matrices.

2) **Def prodEscalar:**

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

Calcula el producto escalar de dos vectores. Esta operación es fundamental en el álgebra lineal y se emplea en la multiplicación de matrices tradicional.

3) **Def transpuesta:**

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

Devuelve la transpuesta de una matriz, es decir, una nueva matriz donde las filas de la matriz original se convierten en columnas y viceversa. La transpuesta se utiliza comúnmente en operaciones con matrices, como la multiplicación.

4) **Def multMatriz:**

```
  def multMatriz(m1: Matriz, m2: Matriz): Matriz = {
    val m2t = transpuesta(m2)
    Vector.tabulate(m1.length, m2.length) { case (i, j) =>
prodEscalar(m1(i), m2t(j)) }
  }
```

Implementa el algoritmo estándar de multiplicación de matrices, utilizando el producto escalar de filas y columnas. Aunque es sencillo de entender, su complejidad computacional es relativamente alta para matrices grandes.

5) **Def multMatrizPar:**

```
  def multMatrizPar(m1: Matriz, m2: Matriz): Matriz = {
    val m2t = transpuesta(m2)
    val limite = 2
```

Es una versión paralela de Def multMatriz que aprovecha la capacidad de procesamiento de múltiples núcleos. Divide el cálculo en subtareas que se ejecutan en paralelo, lo que puede mejorar significativamente el rendimiento en sistemas multiprocesador.

6) **Def auxPar:**

```
    def auxPar(inf: Int, sup: Int): Matriz = {
      if (sup - inf < limite) Vector.tabulate(1, m2.length) { case (i,
j) => prodEscalar(m1(inf), m2t(j)) }
      else {
        val m = inf + (sup - inf) / 2
        val (x, y) = parallel(auxPar(inf, m), auxPar(m, sup))
        x ++ y
      }
    }
      auxPar(0, m1.length)
  }
```

Esta función es auxiliar y se utiliza dentro de la función multMatrizPar para paralelizar la multiplicación de matrices. Su objetivo principal es dividir el trabajo de multiplicar las matrices en subtareas más pequeñas y ejecutarlas en paralelo.

7) **Def subMatriz:**

```
  def subMatriz(m: Matriz, i: Int, j: Int, l: Int): Matriz =
```

```
  {
    Vector.tabulate(l, l)((a, b) => m(a + i)(b + j))
  }
```

Extrae una submatriz de una matriz más grande, especificando la fila y columna inicial, así como el tamaño de la submatriz. Esta función es esencial para los algoritmos de división y conquista, como el algoritmo de Strassen.

8) Def sumMatriz

```
  def sumMatriz(m1: Matriz, m2: Matriz): Matriz =
  {
    Vector.tabulate(m1.length, m1.length)((a, b) => m1(a)(b) +
m2(a)(b))
  }
```

Esta función se encarga de sumar dos matrices de igual tamaño elemento a elemento. Es decir, toma dos matrices como entrada y produce una nueva matriz donde cada elemento es la suma de los elementos correspondientes en las matrices de entrada.

9) Def multMatrizRec:

```
  def multMatrizRec(m1: Matriz, m2: Matriz): Matriz =
  {
    val n = m1.length

    if (n == 1)
    {
      Vector(Vector(m1(0)(0) * m2(0)(0)))
    }
    else
    {
      val m = n / 2

      val a11 = subMatriz(m1, 0, 0, m)
      val a12 = subMatriz(m1, 0, m, m)
      val a21 = subMatriz(m1, m, 0, m)
      val a22 = subMatriz(m1, m, m, m)

      val b11 = subMatriz(m2, 0, 0, m)
      val b12 = subMatriz(m2, 0, m, m)
      val b21 = subMatriz(m2, m, 0, m)
```

```
      val b22 = subMatriz(m2, m, m, m)

      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))

      c11.zip(c12).map { case (filaC11, filaC12) => filaC11 ++ filaC12
} ++ c21.zip(c22).map { case (filaC21, filaC22) => filaC21 ++ filaC22}
    }
  }
```

Implementa el algoritmo de multiplicación de matrices de forma recursiva, dividiendo las matrices en submatrices más pequeñas y resolviendo el problema de forma recursiva. Esta técnica es eficiente para matrices de tamaño potencia de dos.

10) Def multMatrizRecPar:

```
  def multMatrizRecPar(m1: Matriz, m2: Matriz): Matriz =
  {
    val n = m1.length

    if(n == 8)
    {
      multMatrizRec(m1, m2)
    }
    else
    {
      if (n == 1) {
        Vector(Vector(m1(0)(0) * m2(0)(0)))
      }
      else
      {
        val m = n / 2

        val a11 = subMatriz(m1, 0, 0, m)
        val a12 = subMatriz(m1, 0, m, m)
        val a21 = subMatriz(m1, m, 0, m)
```

```
        val a22 = subMatriz(m1, m, m, m)

        val b11 = subMatriz(m2, 0, 0, m)
        val b12 = subMatriz(m2, 0, m, m)
        val b21 = subMatriz(m2, m, 0, m)
        val b22 = subMatriz(m2, m, m, m)

        val c11 = task(sumMatriz(multMatrizRecPar(a11, b11),
multMatrizRecPar(a12, b21)))
        val c12 = task(sumMatriz(multMatrizRecPar(a11, b12),
multMatrizRecPar(a12, b22)))
        val c21 = task(sumMatriz(multMatrizRecPar(a21, b11),
multMatrizRecPar(a22, b21)))
        val c22 = task(sumMatriz(multMatrizRecPar(a21, b12),
multMatrizRecPar(a22, b22)))

        c11.join().zip(c12.join()).map { case (filaC11, filaC12) =>
filaC11 ++ filaC12 } ++ c21.join().zip(c22.join()).map { case (filaC21,
filaC22) => filaC21 ++ filaC22 }
      }
    }
  }
```

Utiliza la técnica de dividir y conquistar para paralelizar la multiplicación de matrices. Al dividir el problema en subproblemas más pequeños y ejecutarlos en paralelo, se puede lograr una aceleración significativa en el tiempo de ejecución, especialmente en sistemas con múltiples núcleos de procesamiento.

11) Def resMatriz:

```
  def resMatriz(m1: Matriz, m2: Matriz): Matriz = {
    Vector.tabulate(m1.length, m1.length)((a, b) => m1(a)(b) -
m2(a)(b))
  }
```

Esta función resta elemento a elemento dos matrices de igual tamaño. Es decir, toma dos matrices como entrada y produce una nueva matriz donde cada elemento es la resta de los elementos correspondientes en las matrices de entrada.

12) Def multStrassen:

```
def multStrassen(m1: Matriz, m2: Matriz): Matriz = {
  val n = m1.length

  if (n == 1) {
    Vector(Vector(m1(0)(0) * m2(0)(0)))
  }
  else {
    val m = n / 2

    val a11 = subMatriz(m1, 0, 0, m)
    val a12 = subMatriz(m1, 0, m, m)
    val a21 = subMatriz(m1, m, 0, m)
    val a22 = subMatriz(m1, m, m, m)

    val b11 = subMatriz(m2, 0, 0, m)
    val b12 = subMatriz(m2, 0, m, m)
    val b21 = subMatriz(m2, m, 0, m)
    val b22 = subMatriz(m2, m, m, m)

    val s1 = resMatriz(b12, b22)
    val s2 = sumMatriz(a11,a12)
    val s3 = sumMatriz(a21,a22)
    val s4 = resMatriz(b21,b11)
    val s5 = sumMatriz(a11,a22)
    val s6 = sumMatriz(b11,b22)
    val s7 = resMatriz(a12,a22)
    val s8 = sumMatriz(b21,b22)
    val s9 = resMatriz(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)

    val c11 = resMatriz(sumMatriz(p6,sumMatriz(p5,p4)),p2)
    val c12 = sumMatriz(p1,p2)
    val c21 = sumMatriz(p3,p4)
    val c22 = resMatriz(sumMatriz(p5,p1), sumMatriz(p3,p7))
```

```
      c11.zip(c12).map { case (filaC11, filaC12) => filaC11 ++ filaC12
} ++ c21.zip(c22).map { case (filaC21, filaC22) => filaC21 ++ filaC22}
    }
  }
```

Implementa el algoritmo de Strassen de manera secuencial. Divide recursivamente las matrices en submatrices más pequeñas y calcula productos intermedios de forma estratégica para reducir el número de multiplicaciones necesarias. El algoritmo de Strassen es una optimización del método tradicional de multiplicación de matrices, ofreciendo una mejora en la complejidad computacional, especialmente para matrices grandes. Sin embargo, al ser secuencial, se ejecuta en un solo núcleo del procesador.

13) Def multStraseenPar:

```
  def multStrassenPar(m1: Matriz, m2: Matriz): Matriz = {
    val n = m1.length

    if (n == 1) {
      Vector(Vector(m1(0)(0) * m2(0)(0)))
    }
    else {
      val m = n / 2

      val a11 = subMatriz(m1, 0, 0, m)
      val a12 = subMatriz(m1, 0, m, m)
      val a21 = subMatriz(m1, m, 0, m)
      val a22 = subMatriz(m1, m, m, m)

      val b11 = subMatriz(m2, 0, 0, m)
      val b12 = subMatriz(m2, 0, m, m)
      val b21 = subMatriz(m2, m, 0, m)
      val b22 = subMatriz(m2, m, m, m)

      val s1 = resMatriz(b12, b22)
      val s2 = sumMatriz(a11, a12)
      val s3 = sumMatriz(a21, a22)
      val s4 = resMatriz(b21, b11)
      val s5 = sumMatriz(a11, a22)
      val s6 = sumMatriz(b11, b22)
      val s7 = resMatriz(a12, a22)
      val s8 = sumMatriz(b21, b22)
```

```
      val s9 = resMatriz(a11, a21)
      val s10 = sumMatriz(b11, b12)

      val p1 = task(multStrassenPar(a11, s1))
      val p2 = task(multStrassenPar(s2, b22))
      val p3 = task(multStrassenPar(s3, b11))
      val p4 = task(multStrassenPar(a22, s4))
      val p5 = task(multStrassenPar(s5, s6))
      val p6 = task(multStrassenPar(s7, s8))
      val p7 = task(multStrassenPar(s9, s10))

      val c11 = task(resMatriz(sumMatriz(p6.join(),
sumMatriz(p5.join(), p4.join())), p2.join()))
      val c12 = task(sumMatriz(p1.join(), p2.join()))
      val c21 = task(sumMatriz(p3.join(), p4.join()))
      val c22 = task(resMatriz(sumMatriz(p5.join(), p1.join()),
sumMatriz(p3.join(), p7.join())))

      c11.join().zip(c12.join()).map { case (filaC11, filaC12) =>
filaC11 ++ filaC12 } ++ c21.join().zip(c22.join()).map { case (filaC21,
filaC22) => filaC21 ++ filaC22 }
    }
  }
}
```

Es una versión paralela de multStrassen. Utiliza la misma lógica de división y conquista, pero aprovecha la capacidad de procesamiento paralelo de múltiples núcleos. Los cálculos de los productos intermedios se distribuyen en diferentes tareas que se ejecutan de forma concurrente. Esto permite acelerar significativamente el proceso, especialmente para matrices grandes, ya que se pueden aprovechar los recursos de múltiples procesadores.

INFORME DE PARALELIZACIÓN:

Estrategia de paralelización:

En nuestro enfoque para paralelizar el algoritmo decidimos usar el algoritmo de Strassen para la multiplicación de matrices. La ventaja principal de este algoritmo es que reduce la cantidad de multiplicaciones pasando de 8 a 7 lo que hace que sea más eficiente. Lo que se hizo fue paralelizar las multiplicaciones de las submatrices y para esto dividimos el problema en

partes más pequeñas que se pueden ejecutar en paralelo usando varios hilos y de esta manera las operaciones que antes se hacían de forma secuencial ahora se reparten entre varios hilos lo que acelera todo el proceso de cálculo.

Ley de Amdahl:

Con Ley de Amdahl, sabemos que el rendimiento de un algoritmo paralelizado depende de cuánto de ese algoritmo se puede paralelizar y cuánto sigue siendo secuencial. En nuestro caso las multiplicaciones de las submatrices se paralelizan pero las sumas y restas de las submatrices siguen siendo secuenciales, esto quiere decir que aunque conseguimos mejorar el rendimiento paralelizando una parte importante del cálculo, las ganancias de tiempo siempre van a estar limitadas por la parte del algoritmo que no se puede paralelizar.

Pruebas y resultados:

Hicimos varias pruebas con matrices de diferentes tamaños: 2x2, 4x4, 8x8, 16x16 y 32x32. Los resultados mostraron que para matrices más grandes (como las de 16x16 y 32x32) la paralelización tiene un efecto más grande en la mejora de los tiempos de ejecución, porque la parte paralelizable del algoritmo es más significativa en estos tamaños, pero para matrices más pequeñas (como 2x2 y 4x4) no se vió gran diferencia en el tiempo de ejecución y esto es por la sobrecarga de manejar los hilos y al tamaño reducido de las matrices que hace que la mejora no sea tan notoria.

CASOS DE PRUEBA

```
def sumMatriz(m1: Matriz, m2: Matriz): Matriz =
{
  Vector.tabulate(m1.length, m1.length)((a, b) => m1(a)(b) + m2(a)(b))
}
```

```
11    test("sumMatriz suma correctamente dos matrices cuadradas") {
12      val m1 = Vector(Vector(1, 2), Vector(3, 4))
13      val m2 = Vector(Vector(5, 6), Vector(7, 8))
14      val esperado = Vector(Vector(6, 8), Vector(10, 12))
15      assert(taller.sumMatriz(m1, m2) == esperado)
16    }
17
18    test("sumMatriz con matrices de diferentes valores") {
19      val m1 = Vector(Vector(0, -1), Vector(-3, 4))
20      val m2 = Vector(Vector(1, 1), Vector(3, -4))
21      val esperado = Vector(Vector(1, 0), Vector(0, 0))
22      assert(taller.sumMatriz(m1, m2) == esperado)
23    }
```

```
25    test("sumMatriz con matrices cero") {
26      val m1 = Vector(Vector(0, 0), Vector(0, 0))
27      val m2 = Vector(Vector(0, 0), Vector(0, 0))
28      val esperado = Vector(Vector(0, 0), Vector(0, 0))
29      assert(taller.sumMatriz(m1, m2) == esperado)
30    }
31
32    test("sumMatriz con matrices de 1x1") {
33      val m1 = Vector(Vector(3))
34      val m2 = Vector(Vector(-3))
35      val esperado = Vector(Vector(0))
36      assert(taller.sumMatriz(m1, m2) == esperado)
37    }
```

```
def multMatrizRec(m1: Matriz, m2: Matriz): Matriz
```

```
40  test("multMatrizRec multiplica correctamente dos matrices cuadradas") {
41    val m1 = Vector(Vector(1, 2), Vector(3, 4))
42    val m2 = Vector(Vector(5, 6), Vector(7, 8))
43    val esperado = Vector(Vector(19, 22), Vector(43, 50))
44    assert(taller.multMatrizRec(m1, m2) == esperado)
45  }
46
47  test("multMatrizRec con matrices de 1x1") {
48    val m1 = Vector(Vector(2))
49    val m2 = Vector(Vector(3))
50    val esperado = Vector(Vector(6))
51    assert(taller.multMatrizRec(m1, m2) == esperado)
52  }
```

```
54  test("multMatrizRec con matriz identidad") {
55    val m1 = Vector(Vector(1, 0), Vector(0, 1))
56    val m2 = Vector(Vector(5, 6), Vector(7, 8))
57    val esperado = m2
58    assert(taller.multMatrizRec(m1, m2) == esperado)
59  }
60
61  test("multMatrizRec con matrices cero") {
62    val m1 = Vector(Vector(0, 0), Vector(0, 0))
63    val m2 = Vector(Vector(0, 0), Vector(0, 0))
64    val esperado = Vector(Vector(0, 0), Vector(0, 0))
65    assert(taller.multMatrizRec(m1, m2) == esperado)
66  }
```

```
70  test("multMatrizRecPar multiplica correctamente matrices pequeñas") {
71    val m1 = Vector(Vector(1, 2), Vector(3, 4))
72    val m2 = Vector(Vector(5, 6), Vector(7, 8))
73    val esperado = Vector(Vector(19, 22), Vector(43, 50))
74    assert(taller.multMatrizRecPar(m1, m2) == esperado)
75  }
```

```scala
def resMatriz(m1: Matriz, m2: Matriz): Matriz = {
  Vector.tabulate(m1.length, m1.length)((a, b) => m1(a)(b) - m2(a)(b))
}
```

```scala
77    test("restaMatriz resta correctamente dos matrices cuadradas") {
78      val m1 = Vector(Vector(5, 6), Vector(7, 8))
79      val m2 = Vector(Vector(1, 2), Vector(3, 4))
80      val esperado = Vector(Vector(4, 4), Vector(4, 4))
81      assert(taller.resMatriz(m1, m2) == esperado)
82    }
83
84    test("restaMatriz con matrices de valores negativos") {
85      val m1 = Vector(Vector(-5, -6), Vector(-7, -8))
86      val m2 = Vector(Vector(-1, -2), Vector(-3, -4))
87      val esperado = Vector(Vector(-4, -4), Vector(-4, -4))
88      assert(taller.resMatriz(m1, m2) == esperado)
89    }
90
91    test("restaMatriz con matrices de ceros") {
92      val m1 = Vector(Vector(0, 0), Vector(0, 0))
93      val m2 = Vector(Vector(0, 0), Vector(0, 0))
94      val esperado = Vector(Vector(0, 0), Vector(0, 0))
95      assert(taller.resMatriz(m1, m2) == esperado)
96    }
```

```scala
99    test("restaMatriz con matrices de 1x1") {
100     val m1 = Vector(Vector(10))
101     val m2 = Vector(Vector(3))
102     val esperado = Vector(Vector(7))
103     assert(taller.resMatriz(m1, m2) == esperado)
104   }
```

```
def multStrassen(m1: Matriz, m2: Matriz): Matriz = {
  val n = m1.length
```

```
106   test("multStrassen multiplica correctamente matrices cuadradas") {
107     val m1 = Vector(Vector(1, 2), Vector(3, 4))
108     val m2 = Vector(Vector(5, 6), Vector(7, 8))
109     val esperado = Vector(Vector(19, 22), Vector(43, 50))
110     assert(taller.multStrassen(m1, m2) == esperado)
111   }
112
113   test("multStrassen con matrices identidad") {
114     val m1 = Vector(Vector(1, 0), Vector(0, 1))
115     val m2 = Vector(Vector(5, 6), Vector(7, 8))
116     val esperado = m2
117     assert(taller.multStrassen(m1, m2) == esperado)
118   }
119
120   test("multStrassen con matrices de 1x1") {
121     val m1 = Vector(Vector(2))
122     val m2 = Vector(Vector(3))
123     val esperado = Vector(Vector(6))
124     assert(taller.multStrassen(m1, m2) == esperado)
125   }
```

```
127   test("multStrassen con matrices de ceros") {
128     val m1 = Vector(Vector(0, 0), Vector(0, 0))
129     val m2 = Vector(Vector(0, 0), Vector(0, 0))
130     val esperado = Vector(Vector(0, 0), Vector(0, 0))
131     assert(taller.multStrassen(m1, m2) == esperado)
132   }
```

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

```scala
135  test("prodPunto calcula correctamente el producto punto") {
136    val v1 = Vector(1, 2, 3)
137    val v2 = Vector(4, 5, 6)
138    val esperado = 32 // 1*4 + 2*5 + 3*6
139    assert(taller.prodEscalar(v1, v2) == esperado)
140  }
141
142  test("prodPunto con vectores de ceros") {
143    val v1 = Vector(0, 0, 0)
144    val v2 = Vector(0, 0, 0)
145    val esperado = 0
146    assert(taller.prodEscalar(v1, v2) == esperado)
147  }
148
149  test("prodPunto con vectores negativos") {
150    val v1 = Vector(-1, -2, -3)
151    val v2 = Vector(4, 5, 6)
152    val esperado = -32 // -1*4 + -2*5 + -3*6
153    assert(taller.prodEscalar(v1, v2) == esperado)
154  }
```

```scala
156   test("prodPunto con un vector vacío") {
157     val v1 = Vector[Int]()
158     val v2 = Vector[Int]()
159     val esperado = 0
160     assert(taller.prodEscalar(v1, v2) == esperado)
161   }
```

```scala
163  test("prodPunto con vectores de diferentes tamaños (debería fallar)") {
164    val v1 = Vector(1, 2)
165    val v2 = Vector(3, 4, 5)
166    assertThrows[IllegalArgumentException] {
167      taller.prodEscalar(v1, v2)
168    }
169  }
```

```scala
def multStrassenPar(m1: Matriz, m2: Matriz): Matriz = {
  val n = m1.length
```

```scala
  test("multStrassenPar multiplica correctamente matrices cuadradas") {
    val m1 = Vector(Vector(1, 2), Vector(3, 4))
    val m2 = Vector(Vector(5, 6), Vector(7, 8))
    val esperado = Vector(Vector(19, 22), Vector(43, 50))
    assert(taller.multStrassenPar(m1, m2) == esperado)
  }

  test("multStrassenPar con matrices identidad") {
    val m1 = Vector(Vector(1, 0), Vector(0, 1))
    val m2 = Vector(Vector(5, 6), Vector(7, 8))
    val esperado = m2
    assert(taller.multStrassenPar(m1, m2) == esperado)
  }

  test("multStrassenPar con matrices de ceros") {
    val m1 = Vector(Vector(0, 0), Vector(0, 0))
    val m2 = Vector(Vector(0, 0), Vector(0, 0))
    val esperado = Vector(Vector(0, 0), Vector(0, 0))
    assert(taller.multStrassenPar(m1, m2) == esperado)
  }
```

```scala
  test("multStrassenPar con matrices de 1x1") {
    val m1 = Vector(Vector(2))
    val m2 = Vector(Vector(3))
    val esperado = Vector(Vector(6))
    assert(taller.multStrassenPar(m1, m2) == esperado)
  }

  test("multStrassenPar con matrices grandes cuadradas") {
    val size = 4
    val m1 = Vector.tabulate(size, size)((i, j) => i + j + 1)
    val m2 = Vector.tabulate(size, size)((i, j) => i - j + 1)
    val esperado = taller.multMatriz(m1, m2) // Validamos contra el resultado de la versión no paralela
    assert(taller.multStrassenPar(m1, m2) == esperado)
  }
```

CONCLUSIONES

En este documento, se ha presentado la implementación y evaluación de algoritmos para la multiplicación de matrices: métodos convencionales, recursivos y técnicas avanzadas como el algoritmo Strassen y su versión paralela. A través de técnicas de programación funcional y concurrencia se ha demostrado como la “divide & conquer” puede aplicarse para descomponer problemas complejos en subproblemas más menores manejables, lo que resulta en una cantidad menor de operaciones necesarias, implica una optimización del tiempo de ejecución.

El algoritmo Strassen en particular experimenta una gran mejora en términos de complejidad computacional al disminuir el número de multiplicaciones requerido; es especialmente óptimo para matrices de grán tamaño. La versión paralela del algoritmo se distingue por ocupar los recursos adicionales proporcionados por múltiples núcleos de procesamiento y acelerar significativamente en comparación con su versión secuencial. Referencia y pruebas hechas con esta técnica junto con herramientas, como la función task para realizar tarea concurrente, evidencia el valor de realizar paralelización en situaciones de alta demanda computacional.

Además, el análisis hecho a través de la Ley de Amdahl desagrava cómo se mejora en grande el rendimiento al apoyar paralelismos en algunas secciones críticas del algoritmo pero, también, política las capacidades limitadas de las partes que se resguardan secuencialmente. Las pruebas hechas con diferentes tamaños de entrada han corroborado el adecuado rendimiento de estas optimizaciones y abogan por un diseño algorítmico que equilibre entre el costo de computo de las tareas.

En conclusión, este trabajo arroja luz sobre cómo la programación funcional y concurrente no solo mejora el rendimiento de algoritmos existentes, sino que también permite innovar en la gestión de grandes cantidades de datos, lo que equipara las bases para el uso en otras abordajes de aprendizaje automático, simulación científica y de alto rendimiento.