INFORME DEL TALLER 4 - MULTIPLICACIÓN DE MATRICES EN PARALELO

Víctor Manuel Hernández Ortiz
2259520
Universidad Del Valle

Jhon Alejandro Martinez Murillo
2259565
Universidad Del Valle

I. CORRECCIÓN DE LAS FUNCIONES IMPLEMENTADAS

A. *Multiplicación de matrices secuencial*

La función *multMatriz* realiza la multiplicación de matrices de manera secuencial, obteniendo las dimensiones de ambas matrices, transponiendo la segunda matriz y luego calculando cada elemento de la matriz resultante mediante productos punto entre las filas y columnas correspondientes(que ahora también son filas gracias a la transpuesta). Funciona para matrices de dimensiones iguales, o de diferentes dimensiones. Para esto se hace uso de *Vector.tabulate* la cual es una función que crea un nuevo vector (o matriz en este caso) y llena sus elementos utilizando una función de índice. En esta función de multiplicación de matrices se utiliza para crear una matriz y calcular sus elementos mediante una función anónima

```
val m3: Matriz = Vector.tabulate(l1, l2)((i, j) =>
     prodPunto(m1(i), m2t(j)))
```

La función anónima $(i, j) => prodPunto(m1(i), m2t(j))$ toma dos índices i y j y calcula el elemento en la posición (i, j) de la matriz resultante aplicando la función $prodPunto$ a las filas correspondientes de $m1$ y $m2t$, de esta manera calculando el elemento y almacenándolo en la nueva Matriz.

B. *Multiplicación de matrices secuencial paralela*

Para realizar la multiplicación de matrices de forma paralela, inicialmente contábamos con dos enfoques para la construcción del paralelismo. El primero se centraba en calcular cada elemento de la matriz de forma paralela usando $task$, lo cual fue poco eficiente, ya que por cada elemento se generaba un hilo para dicho cálculo. El segundo, el que dejamos implementado, se enfoca en dividir la matriz m_1 en 4 partes y calcular individualmente estas 4 partes haciendo uso de $multMatriz$ para finalmente unirlas. Para ello, se hace uso de la función $parallel$ que recibe 4 tareas; estas tareas hacen referencia a una función llamada $bloquesTarea$ que recibe el "pedazo" de la matriz (calculado externamente con otra función) y devuelve el resultado. Posteriormente, estos resultados se unen, dando como resultado la matriz resultante de la multiplicación.

```
// paralelismo aplicado
val par = parallel(bloquesTarea(matriz4Partes._1._1, m2),
     bloquesTarea(matriz4Partes._1._2, m2),
    bloquesTarea(matriz4Partes._2._1, m2), bloquesTarea(
         matriz4Partes._2._2, m2))
//Lo que devuelve, la Union de los resultados
(par._1 ++ par._2 ++ par._3 ++ par._4)
```

C. *Multiplicación de matrices recursiva secuencial*

El código esta distribuido en tres funciones auxiliares que son: **auxSumaMatriz** , **modificarVector**, **auxMultMatrizRec**.

auxSumaMatriz: Esta función auxiliar realiza la suma de los elementos de un vector de vectores con la función *sumMatriz*. La recursión se realiza sobre las filas de la matriz. La acumulación se realiza a través del parámetro $acumuladorSuma$. Cuando se alcanza la última fila, devuelve la primera fila del acumulador, que representa la matriz resultado.

modificarVector: Esta función auxiliar toma un vector y lo agrega como una fila a una matriz acumuladora. La recursión se realiza sobre las posiciones del vector, y cuando se alcanza la última posición, se devuelve la matriz acumuladora.

auxMultMatrizRec: Esta es la función principal para la multiplicación de matrices. Se realiza mediante un enfoque recursivo. Para cada fila $posicionQuieta$ de la primera matriz (m1), y para cada columna posicionCambiante de la segunda matriz (m2), se calcula el producto punto con la función *prodPunto* y *subMatriz*, y va acumulando los resultados en un vector. Luego, este vector se agrega como una fila a la matriz resultado.

La recursión continúa hasta que la matriz resultado tiene la misma longitud que la primera matriz (m1). La función maneja el desplazamiento de filas y columnas en las matrices para realizar la multiplicación correctamente. Con lo mencionado anteriormente logramos cumplir con todos los requisitos dados para implementar esta funcion de manera correcta

D. *Multiplicación de matrices recursiva paralela*

Esta implementación quedo muy similar a la implementación de la secuencial, solo que tiene una diferencia en una parte del código bastante especifica que es:

```
val vectorSuma = task(Vector.tabulate(tamanio)((i) => obj.
     prodPunto(obj.subMatriz(m1,
posicionQuieta,i,m1.length)(0), obj.transpuesta(obj.
     subMatriz(m2,i,
posicionCambiante,m2.length))(0))))
```

Decidimos que era necesario implementar un $task$ para esta parte del código, ya que notamos que aquí es donde se utilizan

más funciones y se gasta más tiempo en los cálculos. Sin embargo, observamos que los tiempos empeoraron e incluso se duplicaron en comparación con la versión secuencial. Esto se debe principalmente al enfoque que utilizamos para implementar la versión recursiva de la multiplicación de matrices. En todos los llamados recursivos, siempre usamos la acumulación de un solo dato, lo cual hace que la paralelización sea innecesaria, ya que no estamos realizando ningún cálculo que pueda considerarse independiente de los demás. Esto provoca un efecto contrario al deseado, que es mejorar el tiempo.

E. Algoritmo Strassen para Multiplicación de matrices

El algoritmo de Strassen es un método eficiente para multiplicar matrices que reduce la complejidad computacional de $O(n^3)$ a $O(n^{\log_2 7})$ mediante la división de las matrices en bloques más pequeños y la recursión. Los pasos a seguir para lograr este algoritmo son:

Caso Base: Si la matriz A tiene un tamaño de 1x1, se realiza la multiplicación simple y se devuelve el resultado en una matriz de tamaño 1x1.

División de las Matrices: Las matrices A y B se dividen en cuatro bloques más pequeños, A11, A12, A21, A22, B11, B12, B21, B22.

Llamadas Recursivas: Se realizan siete llamadas recursivas utilizando estos bloques para calcular siete productos intermedios (P1 a P7) utilizando las divisiones hechas previamente.

Cálculo de Submatrices Resultantes: Se utilizan los resultados intermedios para calcular cuatro submatrices resultantes de la multiplicación (C11, C12, C21, C22).

Combinación de Resultados: Se combinan estas submatrices resultantes para formar la matriz de resultado final.

Con esto, podemos concluir que realizamos una implementación correcta, ya que seguimos la estructura del algoritmo de manera clara y utilizamos llamadas recursivas para dividir y conquistar el problema original en subproblemas más pequeños. Además, la combinación de resultados está realizada de acuerdo con la lógica del algoritmo de Strassen. Por último, también cumplimos con los requisitos solicitados, que incluyen el uso de las funciones *sumMatriz* , *subMatriz* y *restaMatriz*.

F. Algoritmo Strassen para Multiplicación de matrices paralela

La implementación de este algoritmo prácticamente quedo igual a la version secuencial, con la diferencia de que usamos la función *Task* para mejorar el tiempo de ejecución de una parte especifica del código que es la siguiente:

```
val P1 = task(strassenParallel(A11, obj.restaMatriz(B12,
     B22)))
val P2 = task(strassenParallel(obj.sumMatriz(A11, A12), B22
     ))
val P3 = task(strassenParallel(obj.sumMatriz(A21, A22), B11
     ))
val P4 = task(strassenParallel(A22, obj.restaMatriz(B21,
     B11)))
val P5 = task(strassenParallel(obj.sumMatriz(A11, A22), obj
     .sumMatriz(B11, B22)))
val P6 = task(strassenParallel(obj.restaMatriz(A12, A22),
     obj.sumMatriz(B21, B22)))
val P7 = task(strassenParallel(obj.restaMatriz(A11, A21),
     obj.sumMatriz(B11, B12)))
```

Decidimos usar la función *Task* para esta parte porque notamos que se estaba realizando una recursión bastante extensa hasta llegar al caso base que es la matriz $1x1$, por eso decidimos que seria mejor que realice cada cálculo de forma independiente a que espere que terminen los demás, con este cambio logramos apreciar una gran diferencia en el tiempo de ejecución.

II. Desempeño de las funciones secuenciales y paralelas

***Anotación*:** Es importante recalcar las configuraciones/-plataformas de hardware sobre las cuales las pruebas de desempeño y de comparación fueron ejecutadas ya que esto influye en tiempos de ejecución y eficiencia, en este caso las distintas pruebas se ejecutaron entre un **intel core i3 10110u con 2 núcleos, 4 hilos y un ryzen 5 de 4 núcleos, 8 hilos.**

A continuación se muestran las tablas de desempeño de cada algoritmo de multiplicación para las mismas matrices. Estos datos se calcularon promediando 100 resultados para cada algoritmo.

Desempeño de todos los algoritmos

Estas tres tablas representan una en realidad, pero por efectos visuales decidimos anexarla de esta manera

Tamaño de la matriz	Multiplicación secuencial	Multiplicación paralela
2	0.023874999	0.08563700
4	0.050001	0.084999
8	0.122071	0.0621710
16	0.091161	0.1137009
32	0.611	0.3503369
64	6.267858	3.0618850
128	79.0450319	39.5588989

Tamaño de la matriz	Multiplicación recursiva	Multiplicación recursiva paralela
2	0.068845999	0.19054500
4	0.050001	0.084999
8	0.3908499	2.1938079
16	2.5607809	10.3008589
32	21.95624	54.2429610
64	203.3763620	369.499269
128	2491.38580	4278.7803739

Tamaño de la matriz	Algoritmo Strassen	Algoritmo Strassen paralelo
2	0.02743	0.074569
4	0.050001	0.084999
8	0.26881499	0.1974020
16	2.0453940	1.0148579
32	14.9977109	6.1366830
64	129.294209	43.4983070
128	1417.1652550	400.7371680

De esta manera se calcularon los desempeños

```
// funcion que corre las pruebas mandando el tamano de la
     matriz
for (i <- 1 until 7) {
    println(auxDesempenoDeFunciones(math.pow(2, i).toInt))
}

def auxDesempenoDeFunciones(tamanoMatrices: Int): Vector[
     Double] = {
    println("Tamanio: " + tamanoMatrices)
    val m1 = matrizAlAzar(tamanoMatrices, 2)
    val m2 = matrizAlAzar(tamanoMatrices, 2)

    //esta parte se repite para cada algoritmo, calculando
         un array con los tiempos de ejecucion y finalmente
         se retorna un vector con los tiempos promediados
         para cada algoritmo
    val tiemposSeq = (1 to 100).map(_ => 0.0).toArray
    for (i <- 0 until 100) {
      val time = withWarmer(new Warmer.Default) measure {
        multMatriz(m1, m2)
      }
      tiemposSeq(i) = time.value
    }

}
```

A continuación se muestran los resultados de desempeño de las funciones $prodPunto$ y $prodPuntoParD$, que calculan el producto punto de forma secuencial y de forma paralela respectivamente. Las pruebas se realizan mandando distintos tamaños de vectores (potencias de 10), generando dos vectores de ese tamaño y calculando el producto punto de esos dos vectores 100 veces por cada función para al final obtener un promedio del tiempo.

Desempeño de los productos punto

Tamaño del Vector	Producto Punto secuencial	Producto punto paralelo
10^1	0.0146219	0.5192669
10^2	0.024558	0.311517
10^3	0.0690770	0.5393759
10^4	0.6806909	0.9880450
10^5	4.8775630	6.2406690
10^6	68.2956519	57.2317770
10^7	577.1415690	521.1505330

De esta manera se calcularon los desempeños para las funciones de producto punto:

```
//Llamado a la funcion principal que proporciona los
     distintos tamanos
for (i <- 1 to 10) {
  println(desempenoProdPunto(math.pow(10, i).toInt, i))
}

// funcion que calcula los desempenos y retorna promedios
     de desempeno (de la misma forma se calcula para el
     producto punto paralelo)
def desempenoProdPunto(tamanoVectores: Int, pow: Int):
     Vector[Double] = {
  println("Tamano Vectores: " + "10^" + pow)
  val v1 = vectorAlAzar(tamanoVectores, 10)
  val v2 = vectorAlAzar(tamanoVectores, 10)

  val tiemposSeq = (1 to 100).map(_ => 0.0).toArray
  for (i <- 0 until 100) {

    val time = withWarmer(new Warmer.Default) measure {
      prodPunto(v1, v2)
    }
    tiemposSeq(i) = time.value
  }
  .
  .
  .

  Vector(tiemposSeq.sum / 100, tiemposSeqPar.sum / 100)
}
```

Desempeño de algoritmos secuenciales

Tamaño de la matriz	Multiplicación secuencial	Multiplicación recursiva	Strassen
2	0.0381420016	0.093908998	0.03746996
4	0.0247579992	0.109804999	0.129887006
8	0.0386939998	0.486354006	0.215725003
16	0.0933949998	2.657260997	1.437317997
32	0.4390439993	21.21956999	9.674953
64	5.3342230015	177.01692393	67.98518801
128	43.635844	1463.531014998	506.11463989

Desempeño de algoritmos paralelos

Tamaño de la matriz	Multiplicación secuencial paralela	Multiplicación recursiva paralela	Strassen pararelo
2	0.0490440004	0.1274450003	0.08126299
4	0.0294580005	0.2960579998	0.11990699
8	0.0434640001	0.975459004	0.234464
16	0.0934720003	5.341355	1.2639369999
32	0.3401179999	31.503583	9.300731003
64	3.744225994	230.046051001	54.591446995
128	33.363751	1723.101270994	429.24941

De esta manera se calcularon los desempeños para las algoritmos secuenciales y paralelos:

```
def desempenoDeFuncionesSecuenciales(tamanoMatrices: Int):
      Vector[Double] = {
   println("Tamanio: " + tamanoMatrices)
   val m1 = obj.matrizAlAzar(tamanoMatrices, 2)
   val m2 = obj.matrizAlAzar(tamanoMatrices, 2)

   val tiemposSeq = (1 to 100).map(_ => 0.0).toArray
   for (i <- 0 until 100) {
     val time = withWarmer(new Warmer.Default) measure {
       obj.multMatriz(m1, m2)
     }
     tiemposSeq(i) = time.value
   }

   val tiempoRec = (1 to 100).map(_ => 0.0).toArray
   for (i <- 0 until 100) {
       val time = withWarmer(new Warmer.Default) measure {
           obj.multMatrizRec(m1, m2)
       }
       tiempoRec(i) = time.value
   }

   val tiempoStraseen = (1 to 100).map(_ => 0.0).toArray
   for (i <- 0 until 100) {
       val time = withWarmer(new Warmer.Default) measure {
           obj.strassen(m1, m2)
       }
       tiempoStraseen(i) = time.value
   }

   Vector(tiemposSeq.sum / 100,  tiempoRec.sum / 100,
        tiempoStraseen.sum / 100)
 }
```

Los desempeños anteriores se tomaron solo hasta el tamaño de una matriz 128 x 128 ,debido a que la terminal nos lanzaba el siguiente error **out of memory**. Lo mismo sucedio con el calculo de vectores, a partir de 10^8 nos generaba el mismo error.

En las pruebas comparativas que van a continuacion, si logramos obtener hasta el tamaño 1024 x 1024 de la matriz, ya que las pruebas comparación, a diferencia de las pruebas de desempeño, solo constan de una operación por tamaño de matriz.

Comparación de algoritmo secuencial vs paralela

Tamaño de la matriz	Multiplicación secuencial	Multiplicación paralela	Aceleracion
2	0.1941	0.3109	0.624316500482
4	0.0969	0.1253	0.773343974461
8	0.1558	0.512	0.304296874997
16	16.8657	0.4238	39.7963662104
32	1.0945	0.8826	1.24008610922
64	9.3597	2.6625	3.51538028169
128	74.3937	23.9784	3.1025297767
256	582.3829	198.7335	2.93047171
512	4811.1526	1559.8417	3.0843851
1024	41274.5973	15009.3957	2.7499173

Comparación de algoritmo recursivo vs recursivo paralelo

Tamaño de la matriz	recursiva	recursiva paralela	Aceleracion
2	0.4017	1.3084	0.3070162029
4	0.3693	1.4949	0.2470399357
8	1.4204	6.1935	0.2293372083
16	3.5153	18.4654	0.1903722638
32	128.1462	81.0312	1.5814427035
64	289.9554	532.3358	0.5446851404
128	2244.0366	2965.9211	0.756606977
256	18234.863	21556.7414	0.845900716
512	91069.015	110813.2341	0.821824358

Esta comparación la hicimos solo hasta un tamaño 512, porque despues de **17h 10m 41s** de estar haciendo la comparación, nos volvio a lanzar el error mencionado anteriormente **out of memory**

Comparación de algoritmo strassen vs strassen paralelo

Tamaño de la matriz	Strassen	Strassen paralelo	Aceleracion
2	0.7311	0.6296	1.16121346886
4	0.7454	0.3804	1.95951629863
8	1.4337	0.8821	1.62532592676
16	9.8697	2.7326	3.61183488252
32	27.3466	6.5672	4.16411865026
64	149.8199	38.782	3.86312980248
128	1187.6159	339.3482	3.4996970663
256	8264.3622	2360.687	3.5008292924
512	54366.4652	18073.1638	3.00813215
1024	399360.8654	132219.4327	3.020440

De esta manera se calcularon las comparaciones

```
  def compararAlgoritmos(Funcion1:(Matriz,Matriz) =>
       Matriz, Funcion2:(Matriz,Matriz) => Matriz)(m1:
       Matriz, m2: Matriz): (Double, Double, Double) = {
     val timeF1 = withWarmer(new Warmer.Default) measure
         {
       Funcion1(m1, m2)
     }
     val timeF2 = withWarmer(new Warmer.Default) measure
         {
       Funcion2(m1, m2)
     }

     val promedio = timeF1.value / timeF2.value
     (timeF1.value, timeF2.value, promedio)
  }
```

Comparación de producto punto secuencial vs producto punto paralelo

Tamaño de la matriz	Producto punto secuencial	Producto punto paralelo	Aceleracion
10^1	0.0725	2.6356	0.02750796782
10^2	0.0677	1.3331	0.05078388718
10^3	0.2188	1.2477	0.175362667307
10^4	2.197	4.4658	0.49196112678
10^5	8.7597	5.8216	1.50468943245
10^6	27.6552	40.2081	0.68780171159
10^7	338.2786	548.42	0.61682396703

De esta manera se calcularon las comparaciones

```
def compararProdPunto(tamanoVectores: Int): (Double,
    Double, Double) = {
      val v1 = obj.vectorAlAzar(tamanoVectores, 10)
      val v2 = obj.vectorAlAzar(tamanoVectores, 10)
      val timeSeq = withWarmer(new Warmer.Default)
           measure {
          obj.prodPunto(v1, v2)
      }
      val timePar = withWarmer(new Warmer.Default)
           measure {
          obj2.prodPuntoParD(v1, v2)
      }

      (timeSeq.value, timePar.value, timeSeq.value /
           timePar.value)
  }
```

III. Análisis comparativo de las diferentes soluciones

Analisis comparativo de multiplicacion secuencial vs multiplicacion paralelo

Se puede notar que la multiplicación paralela comienza a ser eficiente a partir de un tamaño de 16. Esto parece deberse a las condiciones iniciales de la función, ya que a partir de un tamaño de 16 es cuando se realiza una paralelización en 4 partes. En los casos de tamaños 2 y 4, apenas se lleva a cabo una paralelización en 2 partes. Limitar el número de tareas paralelas a 4 (dividiendo siempre la matriz en 4 partes, operando y finalmente uniendo los resultados) muestra una mayor eficiencia e incluso una aceleración significativa en comparación con su versión secuencial.

Por tanto, podemos concluir que limitar el número de tareas paralelas genera un beneficio considerable. No obstante, es posible que en casos de tamaños de matriz pequeños, la paralelización tenga el efecto contrario.

Analisis comparativo de multiplicacion recursiva vs multiplicacion recursiva paralelo

El algoritmo recursivo es claramente más eficiente, demostrando una aceleración significativa en comparación con su contraparte paralela. Esto se debe a lo mencionado anteriormente en cuanto a cómo se implementó el algoritmo recursivo. Intentar paralelizarlo podría tener un efecto contrario al buscado.

Analisis comparativo de strassen secuencial vs strassen paralelo

El algoritmo de Strassen paralelo supera al Strassen secuencial en eficiencia, como se evidencia en la aceleración. Aunque para matrices pequeñas la paralelización tiene cierta sobrecarga, a partir de dimensiones 8x8, el algoritmo paralelo muestra una mejora significativa en el tiempo de ejecución. Esta ventaja se incrementa a medida que crece el tamaño de la matriz, demostrando así que la paralelización era necesaria en este caso para obtener un mejor tiempo de respuesta.

Analisis comparativo de algoritmos secuenciales

En la comparación de algoritmos secuenciales, la multiplicación secuencial se destaca como la opción más eficiente, demostrando tiempos consistentemente bajos en todos los tamaños de matriz analizados. Aunque los enfoques recursivos y el algoritmo de Strassen ofrecen mejoras en eficiencia computacional en contraste con el método básico, no logran superar la eficacia del enfoque secuencial en estas pruebas.

Analisis comparativo de algoritmos paralelos

En la comparación de algoritmos paralelos, la versión de multiplicación secuencial paralela destaca como la opción más eficiente en términos de tiempo de ejecución para todos los tamaños de matriz estudiados. A medida que aumenta el tamaño de la matriz, tanto la multiplicación recursiva paralela como la multiplicación de Strassen paralela muestran tiempos significativamente más altos en comparación con la multiplicación secuencial paralela.

Analisis comparativo de producto punto secuencial vs producto punto paralelo

Al realizar la comparación de tiempos de las funciones relacionadas con el producto punto, podemos notar que, en realidad, no existe una mejora notable, excepto en el caso de vectores de tamaño 10^5, donde la versión paralela es más eficiente que la secuencial. En los demás casos, la versión paralela no presenta ninguna mejora; de hecho, genera el efecto contrario, aumentando el tiempo de la operación.

¿Las paralelizaciones sirvieron?

Sí, en todos los algoritmos pudimos notar un mejor tiempo de respuesta en los cálculos de las matrices en sus distintos tamaños, con la excepción del algoritmo recursivo paralelo debido a la forma en que implementamos su versión secuencial, lo que resultó menos eficiente. Por otro lado, la multiplicación secuencial paralela se destacó en términos de rendimiento y demostró ser particularmente beneficiosa para reducir los tiempos de cálculo en matrices de diversos tamaños. En cuanto al algoritmo de producto punto, su versión paralela empeora su rendimiento, con la excepción de un caso en el que el tamaño 10^5 es el único donde se nota una mejora con respecto a su versión normal.

¿Es realmente mas eficiente el algoritmo de Strassen?

El algoritmo, en cuanto a los tiempos de ejecución, es aceptable en general; sin embargo, no resultó ser el mejor en los cálculos de las diferentes matrices. Se podría decir que es más eficiente desde el punto de vista computacional, pero en las pruebas realizadas no observamos un mejor desempeño en comparación con los otros algoritmos.

¿No se puede concluir nada al respecto?

Podemos concluir que las paralelizaciones ofrecen algunas ventajas, pero su beneficio varía según las necesidades de cada algoritmo, el tamaño de las matrices y una adaptación cuidadosa a las características de los datos. También podemos afirmar que la multiplicación secuencial paralela es la elección más adecuada en términos de eficiencia, mientras que la implementación de la paralelización en algoritmos recursivos y de Strassen requiere consideraciones particulares para optimizar su rendimiento. En último lugar, al comparar algoritmos de producto punto, se destaca que la versión paralela no presenta mejoras notables, excepto en casos específicos de vectores de tamaños particulares.

IV. PRUEBAS DE SOFTWARE

Link del codigo Fuente

Las pruebas de software que realizamos fueron las siguientes:

A. Comprobación que los algoritmos multiplicaran las matrices de manera correcta

B. Comparación de cada algoritmo secuencial con su version paralela

C. Comparación de los productos punto secuencial y paralelo

D. Evaluación de desempeño de todos los algoritmos

E. Evaluación de desempeño de todos los algoritmos secuenciales

F. Evaluación de desempeño de todos los algoritmos paralelos

G. Evaluación de desempeño de los productos punto secuencial y paralelo