Ubaid Ur Rehman

BSCS-5B

Reg. number: 00000123495

uurehman.bscs15seecs@seecs.edu.pk

Advanced Programming

Lab 4 Document (Strassen’s Algorithm in Scala)

# Introduction

# This lab is about to implement Strassen’s algorithm in Scala

# Approach

Simple Functional programming approach.

Build .scala file and run it.

# Output:

Enter the power factor so that I populate a matrix of order 2^n

(n is power factor): 2

Printing Matrix A

3 18 11 3

43 1 23 44

48 24 40 25

11 17 16 27

Printing Matrix B

39 29 10 4

34 47 19 45

17 35 8 42

46 40 34 45

Printing Product Matrix

1054 1438 562 1419

4126 3859 2129 3163

4518 4920 2106 4077

2521 2758 1479 2696

# Code:

/\*

Subject: Implementation of Strassen's Algorithm for matrices multiplication

Coded by: Ubaid ur Rehman

Email: uurehman.bscs15seecs@seecs.edu.pk

Date: October 12, 2017

\*/

import util.Random.nextInt

import io.StdIn.\_

object Strassen {

def main(args: Array[String]): Unit = {

print("Enter the power factor so that I populate a matrix of order 2^n\n(n is power factor): ")

val powerFactor: Int = readInt()

val matrixDimension: Int = Math.pow(2, powerFactor).intValue

val matrixA = Array.fill(matrixDimension, matrixDimension)(scala.util.Random.nextInt(50))

val matrixB = Array.fill(matrixDimension, matrixDimension)(scala.util.Random.nextInt(50))

println("Printing Matrix A")

printArray(matrixA)

println("\nPrinting Matrix B")

printArray(matrixB)

val productMatrix = sMultiply(matrixA, matrixB)

println("\nPrinting Product Matrix")

printArray(productMatrix)

productMatrix

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*\*

\* print a given matrix

\*/

def printArray[Int](givenArray: Array[Array[Int]]) = {

for (row <- 0 until givenArray.length){

for (column <- 0 until givenArray.length){

print(givenArray(row)(column) + "\t")

}

print("\n")

}

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*\*

\* Strassen's Multiplication Algorithm Implementation

\* product of two matrices

\*/

def sMultiply(mA: Array[Array[Int]], mB: Array[Array[Int]]): Array[Array[Int]] = {

val matrixDimension = if (mA.length == mB.length) mA.length else 0

var productMatrix = Array.ofDim[Int](matrixDimension, matrixDimension)

if (matrixDimension == 1) {

// matrix is of order 1x1

productMatrix(0)(0) = mA(0)(0) \* mB(0)(0)

return productMatrix

}

else if (math.pow(2, math.log(matrixDimension) / math.log(2)) == matrixDimension) {

/\* matrix is of order 2^n x 2^n Divide array into 4 arrays \*/

val halfDimension = matrixDimension / 2

// matrix mA

val mAa = splitMatrix(mA, 1, halfDimension)

val mAb = splitMatrix(mA, 2, halfDimension)

val mAc = splitMatrix(mA, 3, halfDimension)

val mAd = splitMatrix(mA, 4, halfDimension)

// matrix mB

val mBa = splitMatrix(mB, 1, halfDimension)

val mBb = splitMatrix(mB, 2, halfDimension)

val mBc = splitMatrix(mB, 3, halfDimension)

val mBd = splitMatrix(mB, 4, halfDimension)

// calculating values (recursive calls)

val p1 = sMultiply(mAa, subtractMatrices(mBb, mBd))

val p2 = sMultiply(addMatrices(mAa, mAb), mBd)

val p3 = sMultiply(addMatrices(mAc, mAd), mBa)

val p4 = sMultiply(mAd, subtractMatrices(mBc, mBa))

val p5 = sMultiply(addMatrices(mAa, mAd), addMatrices(mBa, mBd))

val p6 = sMultiply(subtractMatrices(mAb, mAd), addMatrices(mBc, mBd))

val p7 = sMultiply(subtractMatrices(mAa, mAc), addMatrices(mBa, mBb))

// calculating product matrix quarters

val mPa = addMatrices(subtractMatrices(addMatrices(p5, p4), p2), p6)

val mPb = addMatrices(p1, p2)

val mPc = addMatrices(p3, p4)

val mPd = subtractMatrices(subtractMatrices(addMatrices(p1, p5), p3), p7)

// product matrix quarters merged

productMatrix = mergeMatrices(mPa, mPb, mPc, mPd)

return productMatrix

}

else {

println("Matrix is not of order 2^n")

return productMatrix

}

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*\*

\* Subtract two matrices

\*/

def subtractMatrices(vA: Array[Array[Int]], vB: Array[Array[Int]]): Array[Array[Int]] = {

val matrixDimension = if (vA.length == vB.length) vA.length else 0

val result = Array.ofDim[Int](matrixDimension, matrixDimension)

for (i <- 0 until vA.length)

for (j <- 0 until vB.length)

result(i)(j) = vA(i)(j) - vB(i)(j)

result //return result matrix

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*\*

\* Add two matrices

\*/

def addMatrices(vA: Array[Array[Int]], vB: Array[Array[Int]]): Array[Array[Int]] = {

val matrixDimension = if (vA.length == vB.length) vA.length else 0

val result = Array.ofDim[Int](matrixDimension, matrixDimension)

for (i <- 0 until vA.length)

for (j <- 0 until vB.length)

result(i)(j) = vA(i)(j) + vB(i)(j)

result //return result matrix

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*\*

[upperLeft |upperRight ]

[quarter (1) |quarter (2) ]

[\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_]

[ | ]

[lowerLeft |lowerRight ]

[quarter (3) |quarter (4) ]

\*

\* split a matrix in above format

\*/

def splitMatrix(matrixGiven: Array[Array[Int]], matrixNumber: Int, matrixDimension: Int) = {

val splittedMatrix = Array.ofDim[Int](matrixDimension, matrixDimension)

var x1: Int = 0; var y1: Int = 0; var x2: Int = matrixDimension; var y2: Int = matrixDimension

if (matrixNumber == 1) {

// do nothing

} else if (matrixNumber == 2) {

y1 = matrixDimension; y2 = matrixGiven.length

} else if (matrixNumber == 3) {

x1 = matrixDimension; x2 = matrixGiven.length

} else if (matrixNumber == 4) {

y1 = matrixDimension; y2 = matrixGiven.length

x1 = matrixDimension; x2 = matrixGiven.length

}

var x = 0; var y = 0

for (row <- x1 until x2) {

y = 0

for (column <- y1 until y2) {

splittedMatrix(x)(y) = matrixGiven(row)(column)

y += 1

}

x += 1

}

splittedMatrix

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*\*

\* merge 4 matrices in one matrix

\*/

def mergeMatrices(mA: Array[Array[Int]], mB: Array[Array[Int]], mC: Array[Array[Int]], mD: Array[Array[Int]]) = {

val matrixDimension = mA.length \* 2

val halfDimension = mA.length

val mergedMatrix = Array.ofDim[Int](matrixDimension, matrixDimension)

for (row <- 0 until matrixDimension) {

for (column <- 0 until matrixDimension) {

if (row >= halfDimension && column >= halfDimension) {

// loop rowNo is more than upperHalf and loop columnNo is more than leftHalf (4)

mergedMatrix(row)(column) = mD(row - halfDimension)(column - halfDimension)

} else if (row >= halfDimension) {

// loop rowNo is more than upperHalf (3)

mergedMatrix(row)(column) = mC(row - halfDimension)(column)

} else if (column >= halfDimension) {

// loop columnNo is more than leftHalf (2)

mergedMatrix(row)(column) = mB(row)(column - halfDimension)

} else {

// enter values in upperLeft quarter (1)

mergedMatrix(row)(column) = mA(row)(column)

}

}

}

mergedMatrix

}

}

Github link: https://github.com/uurehman/strassens-algoritm-for-matrix-multiplication