# ELEN4020 Data Intesive Computing for Data Science Lab 3 Report

Leantha Naicker (788753), Fiona Rose Oloo (790305),
Boitumelo Mantji (823869), Justine Wright (869211)

*The University of Witwatersrand*

April 10, 2018

## 1 Introduction

## 2 Python3 MapReduce Framework: MrJob

Map Reduce is programming tool to solving big data problems by utilising a multitude of computers in parallel. The logic is, as the name suggests, divided into two parts; the Map part and the Reduce part. The algorithms A and B were written in Python 3 and use the Map Reduce framework MrJob.

## 3 Matrix Multiplication: Algorithm A

In Map, the code generates the data in a key-value format; where the key indicates the co-ordinates of a value in a 2D-Array or matrix in reference to (i,j).
The data provided for each matrix is in the format below.

| row | column | value |
|---|---|---|

The algorithm assumes multiplication of two matrices M[i,j,value] and N[j,k,value] which are found in files 'matrixA.txt' and 'matrixB.txt' respectively. The matrices are generated using a a function found in MatrixGen.py, which also checks matrix dimension compatibility.
The method used to multiply these two matrices are to map, reduce and then map and reduce again. The map function will produce key value pairs whilst the reduce function uses the output of the function to perform the row-column calculations, after sorting the values into lists.

Psuedo Code for the map function:
**for** (each element of M) **do**
    produce(key, value) pairs as (i,k), (M,j,m of ij) for k =1,...
**for** (each element of N) **do**
    produce (key, value) pairs as (i,k), (N, j, n of jk) for i =1, 2,...
**return** set of key, value pairs

Psuedo Code for the reduce function:
**for** each key **do**
    sort values into lists for values_A and values_B
    multiply (m of ij) and (n of jk) for each value in the list
    sum up (m of ij)*(n of jk)
**return** (i,k), sidesetj=1sum (m of ij)*(n of jk)

## 4 Matrix Multiplication: Algorithm B

Algorithm B, uses the Strassen algorithm to multiply two matrices. Additionally, a divide and conquer method was also used alongside the Strassen algorithm, which could in theory be threaded to have multiple Strassen algorithm's running in parallel on the chunks of memory. It also has two map-reduce steps. It is worth noting that the Strassen algorithm only works on square matrices with dimensions $2^n$. So, matrices which do not meet these requirements are padded with zeros

Pseudo-code for Strassen algorithm:

**RecursiveStrassen**(A,B)
**in:** Matrix A and B
**out:** Matrix C

If n = 1 call matrix_product(A,B)
Else
  Sub-divide both matrices A and B into four
  $P_1 \leftarrow$ RecursiveStrassen$(A_{11}, B_{12} - B_{22})$
  $P_2 \leftarrow$ RecursiveStrassen$(A_{11} + A_{12}, B_{22})$
  $P_3 \leftarrow$ RecursiveStrassen$(A_{21} + A_{22}, B_{11})$
  $P_4 \leftarrow$ RecursiveStrassen$(A_{22}, B_{21} - B_{11})$
  $P_5 \leftarrow$ RecursiveStrassen$(A_{11} + A_{22}, B_{11} + B_{22})$
  $P_6 \leftarrow$ RecursiveStrassen$(A_{12} - A_{22}, B_{21} + B_{22})$
  $P_7 \leftarrow$ RecursiveStrassen$(A_{11} - A_{21}, B_{11} + B_{12})$
  $C_{11} \leftarrow P_5 + P_4 - P_2 + P_6$
  $C_{12} \leftarrow P_1 + P_2$
  $C_{21} \leftarrow P_3 + P_4$
  $C_{22} \leftarrow P_1 + P_5 - P_3 - P_7$
  Combine $C_{11}...C_{22}$ to make C
  Return C
End If

Psuedo-code for Divide-and-Conquer algorithm:

**multiply**(A,B)
**in:** Matrix A and B
**out:** Matrix C

Sub-divide A into $A_{11}, A_{12}, A_{21}, A_{22}$
Sub-divide B into $B_{11}, B_{12}, B_{21}, B_{22}$
$C_{11} \leftarrow A_{11} \times B11$
$C_{12} \leftarrow A_{11} \times B12$
$C_{21} \leftarrow A_{21} \times B11$
$C_{22} \leftarrow A_{21} \times B12$
$T_{11} \leftarrow A_{12} \times B21$
$T_{12} \leftarrow A_{12} \times B22$
$T_{21} \leftarrow A_{22} \times B21$
$T_{22} \leftarrow A_{22} \times B22$
$C_1 1 \leftarrow C_{11} + T_{11}$
$C_1 2 \leftarrow C_{11} + T_{11}$
$C_2 1 \leftarrow C_{11} + T_{11}$
$C_2 2 \leftarrow C_{11} + T_{11}$
Output C

Pseudo-code for mapperOne:

**mapperOne**(_,line)
**in**: lines from the file
**out**: (key, value)

Sub-divide A into $A_{11}, A_{12}, A_{21}, A_{22}$
Sub-divide B into $B_{11}, B_{12}, B_{21}, B_{22}$
key ← the matrix multiplication needed for divide-and-conquer as a string
value ← matrix, i, j, A[i][j] or B[i][j]
yield (key, value)

Pseudo-code for reducerOne:

**reducerOne**(key, values)
**in**: lines from the file
**out**: (key, value)

$l \leftarrow max(m, n, p)$
$l \leftarrow PowerOf2(l)/2$
$C \leftarrow strassen(A, B, l)$

depending on the key
$identity \leftarrow C_{11}, C_{12} ... or T_{22}$
For a to length of C
    For b to length of C[a]
        $yield(identity[0], [(identity, C[a][b], a, b)])$
    End For
End For

Pseudo-code for second mapperTwo:

**mapperOne**(key,values)
**in**: key, value
**out**: key, value

$l \leftarrow max(m, n, p)$
$l \leftarrow PowerOf2(l)/2$
For v in values
    if v[0] = "C11" or "T11"
        $i \leftarrow v[2]$
        $k \leftarrow v[3]$
    Else If v[0] = "C12" or "T12"
        $i \leftarrow v[2]$
        $k \leftarrow v[3] + l$
    Else If v[0] = "C21" or "T21"
        $i \leftarrow v[2] + l$
        $k \leftarrow v[3]$
    Else If v[0] = "C22" or "T22"
        $i \leftarrow v[2] + l$
        $k \leftarrow v[3] + l$
End For
yield (i,k), (v[1])

Pseudo-code for reducerTwo:

**reducerTwo**(key,values)
**in**: key values
**out**: key, value

$C[i][k] \leftarrow$ sum of the values
yield (i, k), C[i][k]

# 5 Performance

Between algorithm A and algorithm B, which were run using outA1.list and outB1.list as the matrices for multiplication, the run time for algorithm A was 27.63s and algorithm B was 18.94s. Clearly the the recursive Strassen algorithm performed better.Strassen's algorithm is specifically designed to compute matrix multiplication for larger matrices.