UNIVERSITY OF WITWATERSRAND

ELEN4020

DATA INTENSIVE COMPUTING IN DATA SCIENCE

Laboratory Exercise No 3

Authors:
KISHAN NAROTAM - 717 931
JESAL CHANA - 603 177

Authors:
SYED HUSSAIN - 600 524
YUSUF ALLY - 604 973

5th April 2018

I. PROBLEM DESCRIPTION

Two matrices, *A* and *B* of different dimensions and sizes must be multiplied together to produce the product matrix, *C*, i.e. $C = A \times B$. In order for *A* and *B* to be multiplied, the rules of matrix multiplication must be met. These matrices are read in from text files, where the respective matrices are stored in the *Matrix-Market* format. For example, if matrix *A* is a 3×4 matrix, represented as:

$$A[3][4] = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix}$$

This matrix will be read in from the file as follows:

$$\begin{array}{ccc} 3 & 4 & \\ 0 & 0 & a \\ 0 & 1 & b \\ 2 & 3 & l \\ 0 & 3 & d \\ 1 & 0 & e \\ 2 & 0 & i \\ 1 & 1 & f \\ 1 & 2 & g \\ 2 & 1 & j \\ 0 & 2 & c \\ 2 & 2 & k \\ 1 & 3 & h \end{array}$$

The first line of the text file is the dimensions of the matrix, and following that, the first two elements of each line is the position, A_{ij} of the third element in the line which is the value at that respective position within the matrix. The matrix could be an integer which is 4 bytes or a short floating point which too is 4 bytes. The elements of the matrix must be generated using the formula presented in equation 1:

$$A\langle i, j\rangle = i * N_j + j \tag{1}$$

II. BACKGROUND

A. MapReduce

MapReduce is said to be the heart of the *Apache Hadoop* software framework. This framework is written and built in order to process large amounts of data in parallel on large computer clusters, i.e. multiple nodes, with each node of the cluster having its own storage. There are two essential tasks that the program performs. The first of which is the `map` function which is said to take a set of data and convert it into another set. This *mapper* filters and divides the work to the various nodes on the cluster (map). The individual elements are reduced to tuples, i.e. (*key, value pairs*) [1], [2], [3].

The second task is the `reduce` function, which is responsible for taking the output from the `map`, which in turn becomes the input, and combines the data tuples into smaller set of tuples. The *reducer* organizes and reduces the results from each node into a cohesive answer to a query. As the name suggests, the `reduce` function will always happen after the `map` function [1], [2], [3].

An example of how the *MapReduce* can be used could be as follows. Consider 5 files, containing two columns each. In terms of the framework, each file has a key and a value. Within each file, the first column is the name of a city and the second column would be the temperature on a random day. From these files, the goal is to find the maximum temperature for each city recorded. Using the *MapReduce* framework, five mappers will work on its own file. The mapper will return the maximum temperature for each city. The reducer will then take the results from each mapper, and produce one output of each city and their maximum temperature recorded [2].

III. ALGORITHMS

Using the *MrJob* framework, the algorithm was written in $Python$ programming language.

A. Algorithm A: Using MapReduce Repeated Times

Given two matrices, *M* and *N*, with the respective elements m_{ij} and n_{jk}, the product, *P*, will be $P = MN$, with the elements p_{ik}, where

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

The matrix will three key attributes, the first two would be regarded as the key, i.e. row number and column number, with the third being the value at that position. Matrix M has the relation $M(I, J, V)$ with the corresponding tuples (i, j, m_{ij}). Matrix N would have a similar relation as $N(J, K, W)$ with the tuples (j, k, n_{jk}). Since *MapReduce* was created in order to handle larget sets of data, many large matrices are sparse, i.e. most elements are 0, and thus these tuples can be omitted. i, j and k could be implicit in the position of an element rather than written explicitly with the element itself. In a situation like this, the `map` function must be designed to construct I, J, and K from the tuples from the position of the data [4].

The product output of the two matrices is a natural join, after the grouping, i.e. mapping. This means that between the two matrices, the J attribute will be in common, resulting in the tuples (i, j, k, v, w), where (i, j, v) in M and $j(j, k, w)$ in N. The natural join tuples, i.e. the five component tuple, represents a pair of matrix elements, (m_{ij}, n_{jk}). The desired tuple should have four components, $(i, j, k, v \times w)$ as this would represent the desired product result. Now that this relation has been established, one `map-reduce` operation can be done. The grouping can be done with I and K being the attributes for the grouping, the sum of $V \times W$ will be the aggregation [4].

Ultimately, the matrix multiplication can be cascaded of two `MapReduce` operations:

1) **`Map`**:
   - sends matrix element to key value pair:
     $m_{ij} \rightarrow (j, (M, i, m_{ij})$
     $n_{jk} \rightarrow (j, (N, k, n_{jk})$
2) **`Reduce`**:
   - for each key, j, examine list of associated values and produce the tuple $(i, j, m_{ij} n_{jk})$

- Output of this function is key j with the list of all of the tuples of the from from j

3) **`Map`**:
   - this function is applied to the pairs which are the outputs from the prior `reduce` function
   - The pairs are of the form $\rightarrow$
     $(j, [(i_1, k_1, v_1), (i_2, k_2, v_2), ..., (i_q, k_q.v_q)])$
     $v_q = m_{i_q j} \times n_{j k_q}$
   - From this, p key-value pairs are produced:
     $((i_1, k_1), v_1), ((i_2, k_2), v_2), ..., ((i_p, k_p), v_p)$
4) **`Reduce`**:
   - for each key, (i, k), produce sum of list of values associated with the respective key
   - The result $\rightarrow ((i, k), v)$ where v is the value of the element in row i and column k in the product matrix P [4].

B. Algorithm B: Using One MapReduce *Step*

A more efficient method of performing the basic matrix multiplication method would be to improve the two functions and ultimately use `MapReduce` only once to solve for $P = MN$. This is done by using the `map` function to create sets of elements that are needed to compute each element for the matrix P. One input element, will be turned into many key-value pairs, (i, k) where i is a row of M and k is a column of N [4].

1) **`Map`**:
   - for each element of M, m_{ij}, produce key-value pair:
     $((i, k), (M, j, m_{ij}))$ for $k = 1, 2, ...$ number of columns of N
   - for each element of N, n_{jk}, produce key-value pair:
     $((i, k), (N, j, n_{jk}))$ for $i = 1, 2, ...$ number of rows of M
2) **`Reduce`**:
   - each key, (i, k) will have a associated list with the values (M, j, m_{ij}) and (N, j, n_{jk}) for all possible values of j
   - function must connect the two values on the list that have the same value of j for each j

C. Algorithm C: Strassen Algorithm

The *Strassen* Algorithm, or the *Divide and Conquer* method for solving matrix multiplication can only be applied to square matrices, $n \times n$. However, the size of n can only be in powers of 2. The multiplying matrices are divided into 4 smaller, sub matrices of size $\frac{n}{2}$ [5], [6]. An example of the *Strassen* algorithm would be as follows.

Given that $P = M \times N$, $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$, $N = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$ and $P = \begin{bmatrix} I & J \\ K & L \end{bmatrix}$

Using *Strassen*'s algorithm, the following equations must be calculated:

$$M_1 := (A + C) \times (E + F)$$
$$M_2 := (B + D) \times (G + H)$$
$$M_3 := (A - D) \times (E + H)$$
$$M_4 := A \times (F - H)$$
$$M_5 := (C + D) \times (E)$$
$$M_6 := (A + B) \times (H)$$
$$M_7 := D \times (G - E)$$

Then,

$$I := M_2 + M_3 - M_6 - M_7$$
$$J := M_4 + M_6$$
$$K := M_5 + M_7$$
$$L := M_1 - M_3 - M_4 - M_5$$

With regards to using the `MapReduce` functions, it could be implemented many ways, one of which would be using *MapReduce* in a straight-forward fashion:

1) Compute $fv(A)$ and $fv(B)$ which takes $\log_n$ passes
2) Each value in $fv(A)$ is multiplied to the corresponding value from $fv(B)$

   These two steps take one `MapReduce` *pass*

3) Using various equations, the output from step 2 is transformed into the values in the product matrix.
   This last step takes $\log_n$ `MapReduce` *passes* [7].

IV. FUNCTION DESCRIPTION

A. `MatrixGenerator`

Takes in an input (stored as N) from the user which defines the matrix dimensions (square matrix). A function is defined as create with parameter `matrix_file`. `matrix_file` is a text file with a unique name which will store a generated matrix. A `for` loop is started, from $x = 0$ to $x = N$, which contains another `for` loop. This has an overall nested `for` loop structure and allows the program to create rows and coloumns of the same size. The second `for` loop, from $y = 0$ to $y = N$, generates a random integer from 1 to 10 using the `randint` function. This random integer is then written to the file, with a single space inbetween, using the file name provided via the parameter provided to the `create` function

The code then calls the function twice using the parameter file names *File1ForLab3.txt* and *File2ForLab3.txt*

B. `MatrixMultiply1`

The function aims to multiply two matrices together by first using a *MapReduce* system to improve the speed and efficiency of the calculations when dealing with large data sets. The function reads in two text files which contain a matrix each. These matrices are in *Matrix-Market* format and the first line defines their 2D parameters. The algorithm

makes use of a single step *MapReduce* before multiplication.

The `Map` function reads in the position (the i and j positions) and the corresponding value at that position. The matrix is then mapped, which means each value in the matrix is converted to a key-value format. The key being an identifier and the value is the corresponding element for that key. This is done by defining each element in terms of its co-ordinates and for Matrix 1 given each element a matrix identifier. The terms, in Matrix 1, are duplicated equal to the number of columns in Matrix 2 and are stored as the mapped results for Matrix 1. The format of these mapped views are show in equation 2.

$$Z_{ij} = ((i,k),(M,j,m_{ij}) \tag{2}$$

where:
Z_{ij} = Final key-value pair for that particular value
I = Co-ordinate of element in Matrix
K = Iterator which goes from $1, 2, ., N$ where n = number of columns in matrix 2
M= Matrix identifier
J = Second co-ordinate of the element within the matrix
$M_{i,j}$ = Value of element at the give $i-j$ position.

The result of the mapping is then passed to the `Reduce` function which combines the relevant elements into a list. The elements that are combined need to have the same matrix identifier terms as well as the same value of j. These elements are then added to the key-value pair for that key and are joined to form a list of values. The same process is done for all possible matrix identifiers and j values. The result from the Reduce Function is then multiplied and summed to give you a set of final key-value pairs which represents the product of matrix 1 and 2.

C. `MatrixMultiply2`

This method is similar to the standard *MapReduce* method shown in Section IV-B, however, it makes use of slightly different keys and two separate *MapReduce* systems to improve on the speed of the first algorithm.

In the first round of Mapping the first matrix uses its j values as the key rather than iterating to k for each $i-j$ value as in Section IV-B. For matrix 2 the I value is used as the key instead. Example of this key value system is shown in equation 3.

$$Z_{ij} = (j,(M,i,m_{ij}) \tag{3}$$

where:
Z_{ij} = key-value pair
J = J position of element and key in this element (I for matrix 2)
M = Matrix identifier
I = I position of element
M_{ij} = Element at position $i-j$

Once this mapping is done it is sent to the `Reduce` function which follows a similar pattern as in ection IV-B, it appends the values of the same keys to form a list for that key and stores them. Once the initial *MapReduce* is complete the second *MapReduce* system a second *MapReduce* is done where the matrix identifier and the the co-ordinate not used in the first mapping are used as the key and the list is appended with the values corresponding to those keys. The second `Reduce` function sums the values of those elements with the same key values.

V. PSEUDO CODE

Algorithm 1 The `Map` function [8]

for each element m_{ij} of M **do**
  produce $(key, value)$ pairs $((i,k),(M,j,m_{ij}))$ for $k = 1, 2, 3...$ number of columns of N
end for
for each element n_{jk} of N **do**
  produce $(key, value)$ pairs $((i,k),(N,j,n_{jk}))$ for $i = 1, 2, 3...$ number of rows of M
end for
return set of $(key, value)$ pairs that each key, (i,k), has a list with values (M,j,m_{ij}) and (N,j,n_{jk}) for all possible values of j

Algorithm 2 The `Reduce` function [8]

for each key (i,k) **do**
  sort values begin with M by j in $list_M$
  sort values begin with N y j in $list_N$
  multiply M_{ij} and n_{jk} for j^{th} value of each list
  sum up $m_{ij} * n_{jk}$
end for
return (i,k), $\sum_{j=1} m_{ij} * n_{jk}$

VI. FINAL CODE AND OUTPUT

The final code was developed in *Python* using the *MrJob* framework. The testing that was done calculated the time taken to multiply randomly generated matrices, using the `MatrixGenerator` function, using both matrix multiply functions created. These generated matrices are not sparse, i.e. have a density of 100%. An ASUS N550JV high performance laptop was used for testing. The machine has an *Intel(R) Core(TM) i7-4700HQ CPU* with $12GB$ of RAM installed with a 64-bit operating system installed. The operating system installed is *Windows 10 Pro* however for testing, a virtual machine was installed with the *Ubuntu 16.04.3* 64-bit installed. $5GB$ of the total RAM is dedicated to the virtual machine. A custom created *Makefile* was created in order to create the matrix using the `MatrixGenerator.py` file and to run the first and second algorithms, `MatrixMultiply1.py` and `MatrixMultiply2.py` respectively. The times taken

to run each algorithm is written to respective text files.

Due to the limited resources, the maximum matrix size that was tested was 128×128 before any memory dumps occurred. Table I shows the times recorded for each algorithm, and figure 1 is the plotted results.

TABLE I
TABLE SHOWING THE TIME TAKEN, IN SECONDS, TO RUN EACH ALGORITHM WITH MATRICES OF VARIOUS SIZES

Matrix Size	`MatrixMultiply1`	`MatrixMultiply2`
2×2	1.29130792618166	0.316252946853637
4×4	1.32429695129149	0.300330638885498
8×8	1.39224815369512	0.395497798919677
16×16	1.57556509972988	0.543861150741577
32×32	4.33468699455935	1.550762176513670
64×64	14.37436580669240	9.622877836227410
128×128	89.40903306018110	67.783106565475400

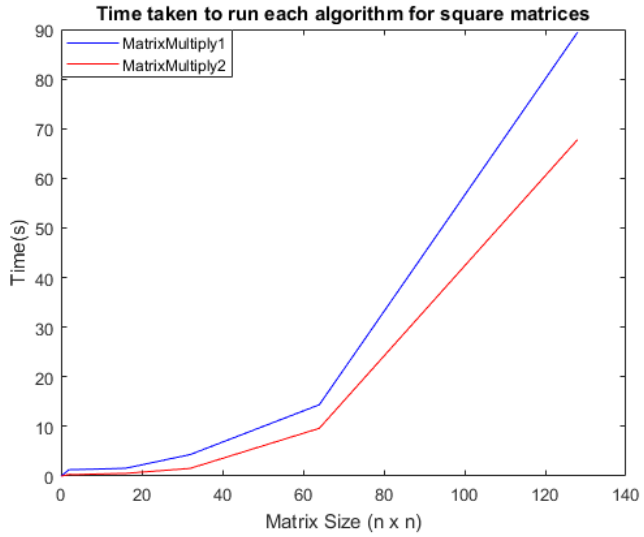

Fig. 1. Graph showing time taken to run algorithm

Although `MatrixMultiply1` should be more efficient, it seems that using the algorithm which utilizes the `MapReduce` function twice is more efficient. This could be due to the way the algorithm was executed, as `MatrixMultiply2` was executed using *Python3* and `MatrixMultiply1` was executed using *Python2*.

VII. CONCLUSION

Matrix multiplication using *MapReduce* framework was presented. A variety of algorithms and methods were presented using the *MapReduce* framework in order to calculate matrix multiplication. It was shown that the algorithm that utilizes `MapReduce` twice was more efficient than the algorithm that uses it once.

REFERENCES

[1] Bigelow, SJ; Chu-Carroll, MC; *What is MapReduce? - Definition from WhatIs.com*; https://searchcloudcomputing.techtarget.com/definition/MapReduce; Last Accessed: 04/04/2018

[2] IBM Analytics; *What is MapReduce? — IBM Analytics*; https://www.ibm.com/analytics/hadoop/MapReduce; Last Accessed: 04/04/2018

[3] Hadoop; *MapReduce Tutorial*; https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html; Last Accessed: 04/04/2018

[4] Leskovec, J; Rajaraman, A; Ullman, JD; *Mining of Massive Datasets - Chapter 2*; Stanford University; Palo Alto, CA; March 2014

[5] TutorialsPoint; *Design and Analysis of Algorithms Strassens Matrix Multiplication*; https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_strassens_matrix_multiplication.htm; Last Accessed: 04/04/2018

[6] GeeksforGeeks; *Divide and Conquer — Set 5 (Strassen's Matrix Multiplication) - GeeksforGeeks*; https://www.geeksforgeeks.org/strassens-matrix-multiplication/; Last Accessed: 04/04/2018

[7] Deng, M; Ramanan, P; *MapReduce Implementation of Strassen's Algorithm for Matrix Multiplication*; Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems fro MapReduce and Beyond Article No. 7; Chicago, IL, USA; 14-19 May 2017

[8] Aytekin, M; *Matrix Multiplication with MapReduce — lendapp*; https://lendap.wordpress.com/2015/02/16/matrix-multiplication-with-mapreduce/; Last Accessed: 04/04/2018