Data Intensive Computing - ELEN 4020
Lab 3

William Becerra Gonzalez 789146
Sailen Nair 1078491
Kyle Govender 571133

April 11, 2018

Introduction

This report documents the procedure taken, as well as the reasons behind the proposed solution within the laboratory. The purpose of this laboratory is to become familiar with the principles of the Mapreduce framework, through the design and implementation of two different matrix multiplication algorithms. The implementation was done using MrJob as the library of choice to interface with Hadoop.The main problem posed by this laboratory was dividing the matrix multiplication algorithm into tasks that can be computed by task-manager nodes in a Hadoop cluster.

1 Proposed Solution

Matrix Multiplication is a basic linear algebra operation used in variety of fields, with one of them being the state space analysis of linear, time-invariant systems. Matrix multiplication is a computationally intensive operation and most implementations use the Strassen algorithm with $O(n^{2.807355})$ time complexity. The Strassen algorithm was pubished in 1969 by Volker Strassen and since then, no improved algorithms with practical use have been proposed. There has been a few theoretical algorithms proposed, and currently the fastest of these is by François Le Gall in 2014 with a time complexity of $O(n^{2.3728639})$.

The time performance of the matrix multiplication algorithm can be improved by making use of Hadoop's distributed computing architechture. Hadoop facilitates the monitoring and scheduling of task-based problems on a network, where tasks can be run in parallel on different nodes within a cluster. Hadoop make use of the MapReduce programming model to perform distributed data storage and operations. MrJob was the MapReduce framework of choice, due to it's readily accessible documentation. MrJob allows for the algorithms to be tested locally i.e. without requiring access to a Hadoop cluster.

2 MapReduce

MapReduce is a method of writing software for large amounts of data and is the heart of Hadoop. MapReduce is designed to run on multiple machines all within a single cluster. Hadoop does all the configuration and task allocation in the background thus, the programmer need not worry about how or on which machine the code will be executed.

MapReduce refers to two different tasks. The first being the 'Map' task and the second being the 'Reduce' task.

The Map aspect of MapReduce processes the input data to output key-value pairs. This means that once the data has been mapped, each data entry will then contain a key-value pair. The key-value pairs are defined by the programmer. To illustrate how mapping works, an example is shown below. This example will take in a set of data and compute how many people are born in each month of the data set.

Table 1: Example Data Set

Name	Age	Birth Month
John	20	January
Steve	24	December
Mary	21	January
Dean	12	January
Alice	15	March

The table above illustrates five peoples names, ages and months in which they were born. This is the raw data that will be fed into the Map section of the MapReduce program. The data will be mapped to a key-value pair, for this example the key will be defined as the month of birth and the value will be a 1. This indicates that there is a person born in that particular month. The mapped data is shown below.

Table 2: Mapped Data

Key	Value
January	1
December	1
January	1
January	1
March	1

The next step is to reduce the data. The Reduce task takes in the key-value pair that was generated in the Map section of the algorithm. The Reduce algorithm will then combine the keys that are the same and perform tasks on the values. Below shows the reduced data after summing the values of matching keys, which in this example are months.

Table 3: Reduced Data

Key	Value
January	3
March	1
December	1

The example presented above shows a basic example of MapReduce. This lab required the computation of matrix multiplication to be programmed using the MapReduce framework.

2.1 MrJob

MrJob was chosen due to its ease of use, allowing one to run MapReduce programs locally. All of the code of a MapReduce program can also be encapsulated within a single class using MrJob. This makes executing the code much easier than a scenario where the Mapper and Reducer were in separate files.

2.2 Matrix Multiplication

The mathematics behind matrix muliplication is illustarted in this section. To multiply two matrices together, they need to adhere to rules of matrix multiplication. With the fundamental rule being, the number of columns in the first matrix must equal the number of rows in the second matrix. If this constraint is met, the two matrices can be multiplied together. The resultant matrix has the form: rows equal to the number of rows in the first matrix, columns equal to the number of columns in the second matrix.

Example: Matrix A has dimensions [n][m] Matrix B has dimensions [m][p]

As can be seen the two matrices adhere to the rule mentioned above, thus the two matrices can be multiplied together. AxB = C

C will have dimensions [n][p].

The formula for matrix multiplication is presented in equation 1 below:

$$c_{ij} = a_{i1}b_{1j} + ... + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj} \tag{1}$$

for $i = 1..n$, and $j = 1..p$

The above formula is seen to be the dot product of the ith row of matrix A with the jth column of matrix B. This above calculation is shown visually in the three matrices below. Where the first row of matrix A is multiplied with the first column of matrix B to yield the first element in matrix C.

Table 4: Matrix A

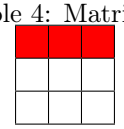


Table 5: Matrix B

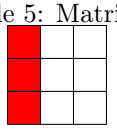
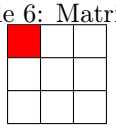


Table 6: Matrix C



2.3 Divide And Conquer

The above algorithm for matrix multiplication shows a naive approach. Another algorithm that can be used is known as 'Divide and Conquer;. Due to time constraints, this algorithm was not implemented in a MapReduce framework. The Divide and Conquer algorithm breaks up a given matrix into smaller matrices that is then used to compute the multiplication of the overall matrix. The figure below shows how the Divide and Conquer algorithm would work. As can be seen, the matrix is broken into four smaller matrices. The final resultant matrix is then computed by a series of multiplications and additions of the smaller matrices. This algorithm much like the naive has a time complexity of O^3. However due to the reduced size of the matrices, the smaller matrices can be stored in cache memory, rather than RAM memory. Data in cache memory is accessed quicker than data in RAM due to the smaller distance between the CPU and cache.

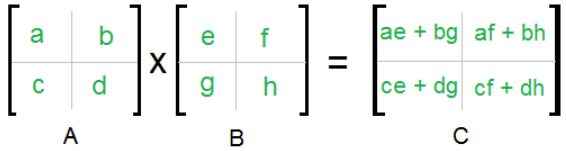


Figure 1: Figure showing the Divide and Conquer approach, where a,b,c...h represent smaller matrices [3]

2.4 pseudo-code

Two different MapReduce algorithms are presented within this laboratory. The two algorithms perform the same the mathematics of matrix multiplication as discussed above, however the actual

algorithms for MapReduce are different. The first algorithm makes use of a 2-step MapReduce solution, whereas the second method makes use of a single step MapReduce solution. The pseudo-code for each algorithm is detailed below. Both algorithms take in two matrices as input. The matrices are defined as:

```
Matrix M with element mij has i rows and j columns ~ M [i][j]
Matrix N with element njk has j rows and k columns ~ N [j][k]
Matrix C is the resultant MxN with element cik has i rows and k columns ~ C [i][k]
```

It is assumed the two matrices, M and N, meet the requirements for matrix multiplication.

2.4.1 Algorithm 1

```
First Map Function:
    for each element mij of M do
        yield (key,value) pair (j, (M,i,mij)
    for each element njk of N do
        yield (key,value) pair (j, (N,k,njk)

First Reduce Function:
    for each key j
        yield (key,value) pair (j, (i,k,mij*njk))

Second Map Function:
    for each (key,value) pair (j, (i,k,mij*njk))
        yield (key,value) pair ((i,k), (mij*njk))

Second Reduce Function:
    for each (key, value) pair ((i,k), (mij*njk))
        sum(mij*njk)
        yield (key, value) pair ((i,k),sum(mij*njk))
```

The above algorithm makes use of two MapReduce steps to complete the matrix multiplication. This algorithm has uses a row*column matrix multiplication approach. Using a two-step algorithm is advantageous as less memory is used in each step. Unlike the one-step algorithm discussed below where more memory is used in the mapper and reducer stage. It will be shown in the results that the two-step MapReduce performs faster than the single step MapReduce.

2.4.2 Algorithm 2

```
Map Function:
    for each element mij of M do
        produce (key,value) pair ((i,k), (M,j,mij))
        (k = 1...no of cols in matrix N)
    for each element nij of N do
        produce (key, value) pair ((i,k), (N,j,nij))
        (i = 1...no of rows in matrix M)
    yield (key, value) pairs ((i,k),(M,j,mij)/(N,j,nij)) for
    all values of j

Reduce Function:
    for each key (i,k) do
        put M values in listM
        put N values in ListN
        mij*njk for element j of each list
        put result in listAns
    yield (key,value) pairs ((i,k) sum(listAns))
```

The above algorithm uses a single MapReduce step to compute the matrix multiplication, unlike the two steps in algorithm 1. The mathematics behind both algorithms are very similar, whereby the dot product between row i in matrix M and column k in matrix N yields element C_{ik}.

This algorithm in MapReduce is computed by getting (key,value) pairs and performing operations on all keys that are the same. Algorithm 2 does this by obtaining all values whereby the key is (i,k). This encompasses all values in row i of matrix M and all values of row k in matrix N. Each value in row i of matrix M is then multiplied with the corresponding value in column k of matrix N. The summation of results from the multiplication then yields the entry C_{ik} in the resultant matrix. This shows that in order to compute the matrix multiplication of two matrices, a large amount of calculations need to carried out. Hence, running the process in parallel will speed up computation time for very large matrices.

3 Results

Table 8 shows the computation times of both algorithms for a varying matrix sizes. The largest matrix that was computed was a 100 000 element matrix. No larger matrices were attempted due to hardware limitations. The hardware used to compute the times was an i5-4210u processor with 16 GiB DDR3 memory.

Due to the hardware limitations only two out of the three matrix multiplication files were computed. The two include the (100*100)x(100*100) matrix, as well as the (1000*500)x(500*1). The computation times for the two algorithms are shown below.

Table 7: Table showing computation times for the two provided input files

Matrix Dimensions	Algorithm 1 Time (s)	Algorithm 2 Time (s)
(100*100)(100*100)	7.917	11.517
(1000*500)(500*1)	8.479	10.233

Table 8: Table showing computation times for large range of matrix size.

No. of Elements per matrix	Algorithm 1 Time (s)	Algorithm 2 Time (s)	Matrix Dimensions
625	0,4635	0,543	(25*25)(25*25)
2500	1,837	1,717	(50*50)(50*50)
10000	7,917	11,517	(100*100)(100*100)
15000	24,11	27,48	(125*125)(125*125)
15625	26,231	28,556	(100*150)(150*100)
20000	46,53	51,79	(200*100)(100*200)
40000	94,53	105,78	(200*200)(200*200)
100000	3327	5056	(1000*100)(100*1000)

4 Evaluation of Solutions

According to the two graphs shown in Figure 2 and Figure 3, it is evident that algorithm 1 (two-step MapReduce) performs better than the algorithm 2 (single-step MapReduce). It can be seen that the larger the matrices were, the larger the compute-time difference between the two algorithms were.

Although both algorithms perform similar mathematical operations, algorithm 1 executes faster due to the smaller amount of memory required at each step. This two-step MapReduce performs a smaller amount of tasks per step, hence breaking the overall computation into smaller steps. By doing this, less memory is needed at each step, thus most of the data can be stored in cache memory.

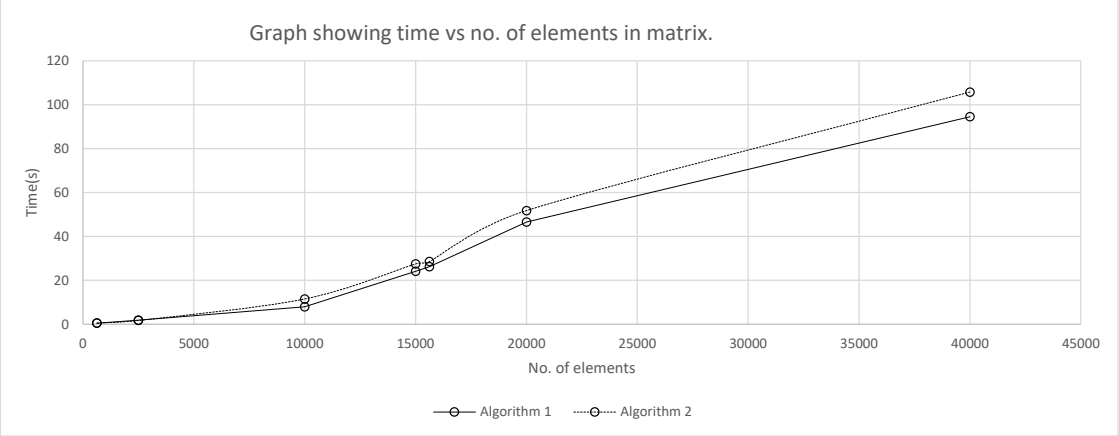


Figure 2: Figure showing the time vs number of elements in matrix. (For a maximum of 40 000 elements)

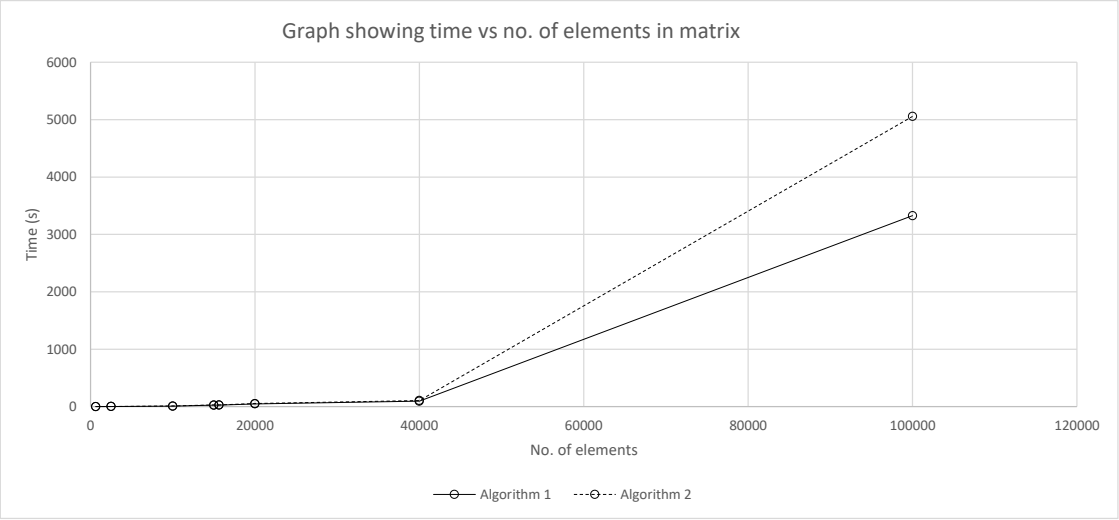


Figure 3: Figure showing the time vs number of elements in matrix. (For a maximum of 100 000 elements)

5 Unweighted directed graph

This component required finding the pairs of nodes that are connected by paths of length 3, within a unweighted directed graph $G = (v, e)$, where $|v|$ is the number of nodes and $|e|$ is the number of edges.

A graph can be represented in matrix format. Hence, by using matrix multiplication the path lengths can be found. Where the square of the original matrix will yield all nodes connected together by path length of 2, the cube of the original matrix will yield all nodes connected together by path a length of 3, and so on.

A simple example is illustrated below.

Figure 4 shows a graph consisting of four nodes. This graph as well as its connections can be represented in matrix format. This is shown in eq. (2).

Figure 4: Figure showing a unweighted directed graph.

$$\alpha = \begin{array}{c} \begin{matrix} A & B & C & D \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} A \\ B \\ C \\ D \end{matrix} \tag{2}$$

The matrix above shows the graph in matrix format where a '1' indicates a connection.

$$\alpha^2 = \alpha * \alpha \tag{3}$$

To obtain all connections which have a path length of 2, the square of the matrix is computed through the above formula. The resultant matrix is shown below in eq. (4). This matrix shows the nodes that can be joined by a path length of two (omitting the values in the diagonal).

$$\alpha^{\mathbf{2}} = \begin{array}{c} \begin{matrix} A & B & C & D \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} A \\ B \\ C \\ D \end{matrix} \tag{4}$$

$$\alpha^3 = \alpha^2 * \alpha \tag{5}$$

To obtain all connections which have a path length of 3, the formula above is used. The resultant matrix is shown below in eq. (6). This matrix shows the all nodes that can be joined by a path length of 3.

$$\alpha^{\mathbf{3}} = \begin{array}{c} \begin{matrix} A & B & C & D \end{matrix} \\ \begin{pmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 3 & 0 \\ 0 & 3 & 0 & 2 \\ 1 & 0 & 2 & 0 \end{pmatrix} \end{array} \begin{matrix} A \\ B \\ C \\ D \end{matrix} \tag{6}$$

This procedure was carried out for question 6 of the laboratory. The input file of a (500*500) matrix was used. The resultant cubed matrix can be found on the github submission page. The total number of paths that are of length 3 were found to be: 39 784 709. This number includes non-unique paths whereby a node can be traversed more than once in a single path.

To perform the matrix multiplication, the two step MapReduce was used.

6 Conclusion

The implementation of two MapReduce matrix multiplication algorithms is presented. Namely, a single-step MapReduce and a two-step MapReduce. It was shown that the two-step MapReduce outperforms the single-step, due to the different amount of memory required. A method for finding the number of paths with the same length using matrix multiplication is also presented.

References

[1] Strassen, Volker, Gaussian Elimination is not Optimal, Numer. Math. 13, p. 354-356, 1969

[2] Andrey Vladimirov, Colfax Research, August 12, 2013 $https : //colfaxresearch.com/multithreaded - transposition - of - square - matrices - with - common - code - for - intel - xeon - processors - and - intel - xeon - phi - coprocessors/$ Last accessed 07 March 2018

[3] GeeksforGeeks, Divide and Conquer | Set 5 (Strassen's Matrix Multiplication) (https://www.geeksforgeeks.org/strassens-matrix-multiplication/) Last Accessed 7 April 2018