Lab3: Matrix Multiplication using MapReducing techniques

Gift Langa (1043882) *, Pierre Nayituriki (1045397) †,
Zanele Malinga (0706885N) ‡, Mokhulwane Nchabeleng (1117206) §
April 12, 2018
ELEN4020: Data Intensive Computing
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050, South Africa

I. Introduction

This paper presents the implementation and the testing of two algorithms for multiplying matrices with different values using Map Reducing techniques. The first algorithm uses one step map-reduce procedure, the second algorithm uses a two step map-reduce procedure. The code implementation was done in Python 3.6 using the MrJob module to run the map-reduce algorithms.

II. First Algorithm Implementation

The first algorithm uses one map reduce step, the pseudo code for the first algorithm is shown in appendix A 1. The first matrix is assigned the letter M and each element is denoted as m_{ij}, and the second matrix is assigned the letter N and each element in N is denoted n_{jk}. The algorithm's map functions generates a key-value pair for each element in M and in N as $((i,k),(Matrix1,j,m_{ij}))$, for $k = 1,2,3,...$ up to the number of columns of N, and $((i,k),(Matrix2,j,n_{ij}))$ respectively, for $k = 1,2,3,...$ up to the number of rows of M. The reducer on the hand, connects all the values on the list generated by the map function with the same value of j, for each j in the list, element m_{ij} is multiplied by n_{ij} and stored in another list. the reducer functions output key is (i,k) and the value is the sum of all the elements with the same key.

III. Second Algorithm Implementation

The second algorithm uses two map-reduce steps, the pseudo code for algorithm two is shown in appendix A 2. The naming nomenclature used to name the matrices is the same as that of algorithm 1. The code implementation was done with one map and two reduce functions. The map function generated key-value pairs as $(j,(M,i,m_{ij}))$ for each element in matrix M and generated the key-value pairs $(j,(N,i,m_{ij}))$ for each element in matrix N. The map functions sorts the elements by rows of M and columns of N .The first reducer generates key-value pairs as $((i,k),m_{ij}n_{jk})$ from the list generated by the map functions. The first reduce function executes the row-column multiplication depending on the value of j. The second reduce function sums all the values in the list produced by the first reducer with the same (i,k) key.

IV. Results and Analysis

Table I shows the time taken for the two algorithms to run the 3 matrices that were used. One can see that the second set of matrices which had a dimension of 1000x500 for outA2.list and 500x1000 for outB2.list did not produce results. The reason for this was that the cluster which was used crashed when the algorithms ran these matrices. This could be as a result of 16GB ram of the cluster which was

TABLE I
TIMES TAKEN FOR ALGORITHMS TO RUN THE SET OF MATRICES IN SECONDS

	Sets of matrices		
matrice dimensions	$100x100$ * $100x100$	$1000x500$ * $500x1000$	$1000x500$ * $500x1$
algorithm 1	36.22	-	10.52
Algorithm 2	13.044	-	8.60

suppose to hold $1,000,000$ elements but was unable to.

One can also see that algorithm one performed poorly as compared to algorithm two. This could be as a result of algorithm one using one mapper and one reducer function and algorithm two using one mapper function and two reducer function making it more efficient. In algorithm one there is only one reducer which is responsible for multiplying and adding the elements .In algorithm two when reducer one is done with multiplying the first set of elements, reducer two will start adding those elements with the same key.Therefore algorithm two is faster because it uses two reducers that are run in parallel.

V. CONCLUSION

Two map-reduce matrix multiplication algorithms were tested and the results showed that algorithm two was faster than algorithm one. Algorithm two was faster because it consisted of 1 mapper function and 2 reducer functions that were run in parallel. The multiplication of larger matrices caused the computer to crash because the RAM memory was not sufficient to store all the values.

APPENDIX A
ALGORITHMS

Algorithm 1 One map one Reduce function

procedure MAPPER($self, , value$)
    for each element m_{ij} of M **do**
        $key, value = (i, k), (Matrix1, j, m_{ij})\ for\ k\ =\ 1\ up\ to\ number\ of\ columnns\ of\ N$
    end for
    for each element n_{ij} of N **do**
        $key, value = (i, k), (Matrix2, j, n_ij)\ for\ i\ =\ 1\ up\ to\ number\ of\ rows\ of\ M$
    end for
    Return $key\ value\ pairs$
end procedure
procedure REDUCER($self, key, values$)
    initialize listA , listB , MatrixC to no elements
    for index in values **do**
        if $index[0] == Matrix2$ **then**
            listB.add(values and x-coordinate that originated from matrix N)
        elselistA.add(values and y-coordinate that originated from matrix M)
        end if
    end for
    for elem1 in listA **do**
        for elem2 in listb **do**
            if $elem1[1] == elem2[1]$ **then** add elem1[0]*elem2[0] to MatrixC
            end if
        end for
    end for
     Return key , sum(MatrixC)
end procedure

Algorithm 2 One map function, two Reduce functions

procedure MAPPER($self, , value$)
    for each element m_{ij} of M **do**
        yield$(j, ('M', j, m_{ij}))$
    end for
    for each element n_{kj} of N **do**
        yield$(j, ('N', k, n_i j)$
    end for
end procedure
procedure REDUCER1($self, key, values$)
    initialize listA , listB to no elements
    for index in values **do**
        if $index[0] ==' N'$ **then**
            listB.append(n_{jk}, k)
        elselistA.add(m_{ij}, i)
        end if
    end for
    for elem1 in listA **do**
        for elem2 in listb **do**
            if $elem1[1] == elem2[1]$ **then yield** (elem1[1],elem2[1]) to MatrixC
            end if
        end for
    end for
end procedure
procedure REDUCER2($self, key, values$) **yield** $[int(key[0]), int(key[1])], sum(values)$
end procedure