CS 434: Parallel and Distributed Computing Lab 4

Kweku Yamoah: 71712022

April 21, 2021

Introduction

This report was generated as a requirement of Lab 4 for Parallel and Distributing computing. The report contains two sections; Part 1 and Part 2. Part 1 is the Lab where I experiment with the MaoReduce framework and my understanding in class. Part 2 is the project which focuses on implementing matrix multiplication using MapReduce algorithms.

Part 1

MapReduce Framework & Why

My selected MapReduce framework was MrJob. I chose MrJob because of the following reasons:

- it provides an easy route for writing python programs that run on Apache Hadoop.
- it provides an easy switch for input and output formats with a single line of code.
- All codes are written in a single python class which enforces the understanding of OPP concepts.
- it has an extensive documentation with examples as reference, which, makes learning of the framework easy.

Installing MrJob was easy because it require running a command with the python pip package. However, understanding the documentation and how to submit my first job took some time to understand.

Word-count Algorithm

Firstly, the word-count algorithm prints out the frequency of each word in a text file.

The word count algorithm inherited the **MrJob** class to implement a job that executes the task. This job class required the implementation of the methods **mapper_init, configure_args, mapper and reducer**.

- The *configure_args* method creates command-line options. I used tis to pass the stop words file as a command line argument.
- The *mapper_init* method initializes or setups the resources that the mapper method needs. I read the stop words and converted the words to a python set in this method.
- *mapper* method splits every sentence from the text file into words. All the stop words are removed before further processing. The method finally returns a key-value pair consisting of the word as the key and a one as the value for every word. E.g., word, 1.
- *reducer* method in this program acts as a combiner and a reducer at the same time. This method finds the occurrences of each word in the text by summing up the results from the mapper method passed to it. The output of the reducer method is a key-value pair containing a word as the key and the value as the number of times it appears in the text.

Running the Program

The program needs some requirements to be satisfied first before running.

- The program requires python 3 or later to run.
- The program requires the correct command-line arguments to be passed before it can run.
- The program must be run on a Linux subsystem; Ubuntu is preferred.

To run the program type this in your terminal whilst ensuring that you are in the directory which contains the program files; **python3 mrjob_wordcount.py <input.txt - -stop-words=stop_words.txt**

Part 2

In this section I implement matrix multiplication using the MapReduce framework MrJob with block-block partitioning. I assume that my matrices are of dimensions $N * N$ and P which is the number of processors is an even square number.

Overview of Algorithm

The first step of the algorithm is to partition the matrices into blocks. Matrix A is split into $\sqrt{P}$ blocks along the rows and matrix B is also split into $\sqrt{P}$ blocks along the columns. After, $P = \sqrt{P} * \sqrt{P}$ tasks are spawned for the mapping and reduce phases.

Pseudocode of Map and Reduce Phases

Algorithm 1: mapper_raw(self,input_file,input_uri)

```
Brief:  Maps a given N x N matrix with block partitioning
Input:  matrix file, size(where size = N)
Output:  key value pairs for row block and column block
```

for *i in generate_partitions.rowMajor(matrix,bands,offset)* **do**
  for *j in generate_partitions.columnMajor(matrix,bands,offset)* **do**
    yield str(count), i yield str(count), j
  end
end

Algorithm 2: reducer(self,key,values

```
Brief:  Reduces a given N x N matrix with block partitioning
Input:  key value pairs)
Output:  matrix multiplication for row block and column block
```

for *i in values* **do**
  put i in localList
end
foreach *row in localList* **do**
  foreach *column in localList* **do**
    answer.append(row * column)
  end
end
yield key, answer

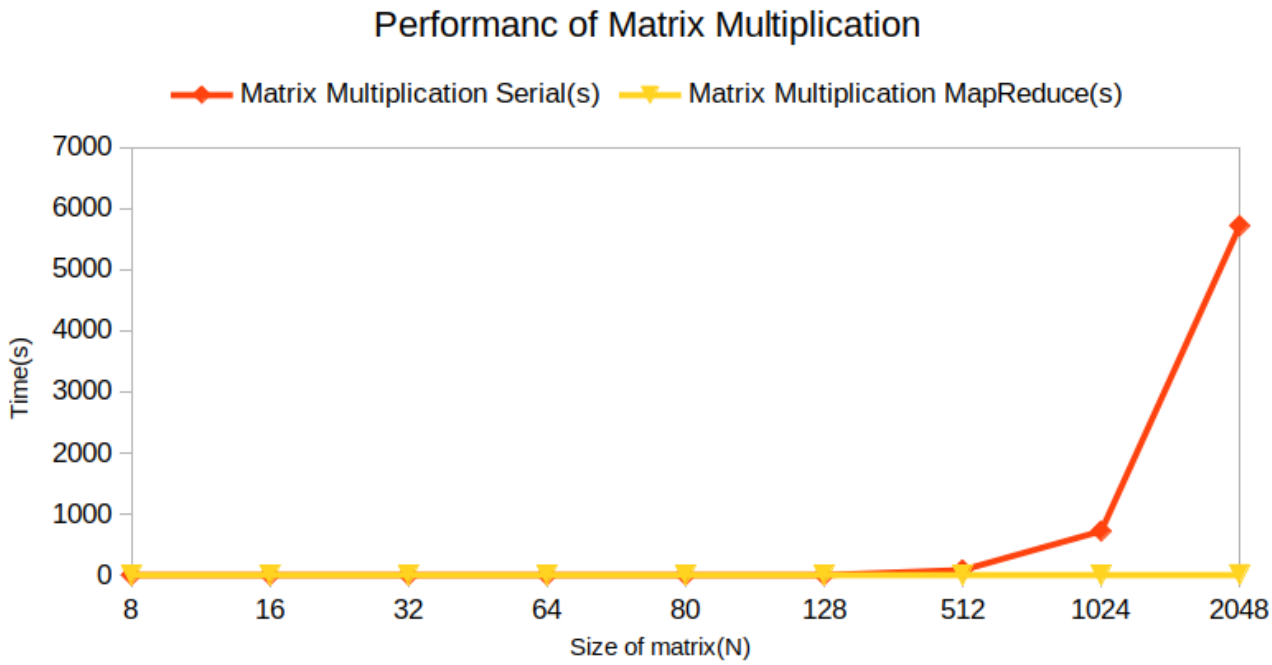
Performance Results

The table below shows the performance of the two versions as implemented in the python programming language. The largest matrix that was computed was a 2,048 element matrix. No larger matrices were attempted due to hardware limitations.

N_0=N_1	P	Matrix Multiplication Serial(s)	Matrix Multiplication MapReduce(s)
8	24	0.0025	0.105
16	64	0.0144	0.115
32	128	0.031	0.104
64	256	0.1513	0.128
80	256	0.277	0.136
128	512	1.1108	0.15342
512	1024	84.7971	1.5077
1024	4096	721.4393	2.786
2048	8192	5718.2588	10.92

After generating the comparative table, I observed that for small sizes of N the serial Implementation of the matrix multiplication algorithm executed faster. However, as N grew exponentially, the parallel implementation in MapReduce executed faster than the serial implementation. This goes to show the importance of parallel computing and message passing communications in Computing.

Here is a graph visualising the difference between the serial implementation and the MapReduce implementation as the matrix size increases.



|

Running the Program

The program needs some requirements to be satisfied first before running.

- The program requires python 3 or later to run.
- The program requires the correct command-line arguments to be passed before it can run.
- The program must be run on a Linux subsystem; Ubuntu is preferred.

First one needs to run the serial version of the code before the parallel version. To run the serial version run the command; **python3 matrix_mults.py N**; where *N(matrix size)*= 16,64,80,128 etc.

To run the program type this in your terminal whilst ensuring that you are in the directory which contains the program files; **python3 mrjob_matrixmult.py matrix.txt - -P=n**; where *n(number of processors)*= 64,256,512 etc