
Distributing System and Parallel Computing



Abraham Foto - 557797

JANUARY 20, 2019
INHOLLAND DIEMEN

Contents

Abstract	2
List of Acronyms	3
1 Introduction.....	4
1.1 Problem statement.....	4
1.2 Objective.....	4
1.3 Report structure	4
2 Parallel infrastructure.....	5
2.1 Issues and solutions of the Hadoop setup in Raspberry Pi	7
2.2 Issues and solutions of the Spark setup in Raspberry Pi	9
2.3 Testing the cluster	9
3 Selecting an algorithm.....	10
3.1 Serial implementation	10
3.2 Parallel implementation	10
3.3 Expected performance gain	10
4 Dataset and correctness of the algorithm.....	11
4.1 Data Description	11
4.2 Generating test data.....	11
4.3 Verifying results.....	11
5 Experiments and results	12
6 Measuring performance gain	13
7 Conclusion	14
8 Discussion and Recommendation	14
Appendix.....	15
References.....	16

Abstract

In this paper, the parallel and serial implementations of an algorithm, which analyses the average number of friends broken down by age of a social network connection is investigated. The performance of the parallel implementation is compared to the serial implementation, by running the implementations with different numbers of nodes on data of 3.1 GB. The time taken to accomplish the task is used to measure the performance gain from parallelization. The results showed that the parallel implementation with 5 nodes, 1 master and 4 slaves reduced the time taken by 1.2 minutes, having the efficiency of 0.3. It can be concluded that even though there was a slight improvement in performance, the processors are not efficiently used.

List of Acronyms

UDFs	User-Defined Functions
RDD	Resilient Distributed Data
Node	Computer

1 Introduction

Big Data or large-scale data is a term used to identify data sets that cannot be easily managed due to their large size and complexity (Wei, Feng , Chieh , & Athanasios, 2015). To get business and scientific insight, the big data needs to be mined and analysed. To analyse big data in a traditional way with a single computer and serial implemented version of an algorithm is not feasible, due to time and memory complexity. The time and memory complexity refers to the amount of time taken and this stage needed by the computer to complete the task. A solution, in response to the problems of analysing large-scale data, is to use a distributed and parallel computing environment.

Parallel computing is a form of computation or execution of a task carried out simultaneously. Parallel computation helps perform a large task or computation by dividing the workload into smaller pieces between two or more computers performing the task at the same time, in parallel. There are different forms of parallelism (Khaldi, Jouvelot, Ancourt, & Irigoin, 2012): task parallelism and data parallelism. Task parallelization is a parallelization of algorithms or codes across multiple computers in a parallel computing environment (Khaldi, Jouvelot, Ancourt, & Irigoin, 2012). Different tasks operate on the same data at the same time. In contrast, data parallelism is parallelization of data across many processors or nodes. In data parallelism, the same instruction or task is performed repeatedly and simultaneously on different parts of the data (Li & Li , 2017). The focus is to distribute the data into different processors or nodes operating in parallel. The data used for this experiment is generated using an excel sheet with a size of 3.1 GB. Details on the data can be found in section 4.1: Data description.

The focus of this paper is data parallelization using Hadoop eco system and Apache Spark in Raspberry Pi (see section 2) and measure the performance gain of the parallel environment compared to the serial.

1.1 Problem statement

A social network site is a web based service that helps users connect to friends and family. The network has active users of different age groups. The focus of this experiment is to analyse the data and figure out the average number of friends broken down by age, using serial implementation in Python and parallel implementation with the Spark framework.

1.2 Objective

The goals of this experiment is set up a cluster or parallel computing environment in Raspberry PI, to write a serial implementation of an algorithm to analyse the data and to parallelize the algorithm within the Spark framework. The desired outcome of this project is to assess the performance gain from the parallel implementation.

1.3 Report structure

This paper is organized in the following way: Section 2 describes the parallel infrastructure. Within this section, the details of the computer cluster, the issues and the solution using Hadoop and Spark configuration in Raspberry Pi are described.

Section 3 describes the selected algorithm. Within this section, there are three subsections: subsections 3.1 and 3.2 describe the serial and parallel implementations of the algorithm, while section 3.3 presents the expected performance gain.

Section 4 describes the nature of the data used for this experiment. The testing data used to verify the correctness of the algorithm is also discussed.

Section 5 contains the results of the experiment.

Section 6 contains the performance gain of the parallel implementation in the context of Amdahl's law.

Section 7 contains the conclusion drawn based on the performance gain.

Section 8 contains discussion and recommendations about the task at hand.

2 Parallel infrastructure

Computer cluster is a set of multiple computers connected to each other through LAN (Yeo, et al., 2006). A computer cluster that provides fast processing speeds and large storage is called a distributed computing environment (Gabriel, Resch, Beisel, & Keller, 1998). In parallel computing environment with more than two processors (often called 'nodes'), most of these nodes would be configured identically (Ostrouchov, 1987). Some of the nodes in a cluster have physical and logical difference based on the role they play. The node that interacts with a user and manages the cluster is called the master node, whereas a group of nodes that perform the computation are called slaves. In this experiment, a small computer, called Raspberry Pi, will be used to build a distributed computing environment. Raspberry Pi is a small, affordable computer with 4 cores which is used for teaching purposes (Foundation, 2018). The configuration of the Raspberry Pi in this experiment will be based on the manual by (Mönning & Schiller, A Hadoop data lab project on Raspberry Pi, 2015). Powering the Raspberry Pi automatically installed the NOOBS installer on the SD card. The Linux based operating system Debian 7 is installed. Thus, in this setup, commands used to configure the cluster are Linux based. Using the Raspbian command line, the network interfaces are configured - to enable the Raspberry Pi to communicate between each other, the Hadoop user account is setup - this is to prepared the Raspberry Pi for Hadoop installation and separate it from other services, SSH RSA paired keys is generated and added to the authorized key- to allowed Hadoop users to access the slaves from the master node without password, by default Hadoop uses the root account to SSH. Thus, the Raspberry Pi is fully configured and is ready for Hadoop installation.

The first step in building a parallel computing environment in a Raspberry Pi is downloading and installing the Apache Hadoop software. Only one node, the master node, will be configured and the configuration will be save on the SD card, then the SD card will be cloned to all the slave nodes. The Apache Hadoop software is an open source platform that allows for the distributed processing and storage of large data sets across clusters of computers (Apache Hadoop, 2018). It is basically intended to scale up from a single server to many machines each having a local stage and computation. The main components of Hadoop are: Hadoop Distributed File System (HDFS) - the primary data storage system used by Hadoop applications, MapReduce - software framework written in Java that is used to create application that can process large amounts of data, and the master and slave nodes, which are discussed above. Next, the environmental variables of the Hadoop and Java are configured and added to the `./bashrc` file, which will provide a centralized management for the user. This is to make the path known to the user environment. In configuring the Hadoop daemon properties, the `core-site.xml`, `mapred-site.xml`, `hdfs-site.xml` are configured, one after another. Note that before finishing the Hadoop setup, it is important to create the temporary folder on hdfs, to store temporary testing data. Finally, cloning the SD Card of the configured Raspberry into the other Raspberry. Starting the `start-hdfs.sh` and `start-yarn.sh` in the `/bin` directory, starts the cluster. The nodes in the cluster can be overviewed on the UI manager using <http://localhost:8088>, figure 1.

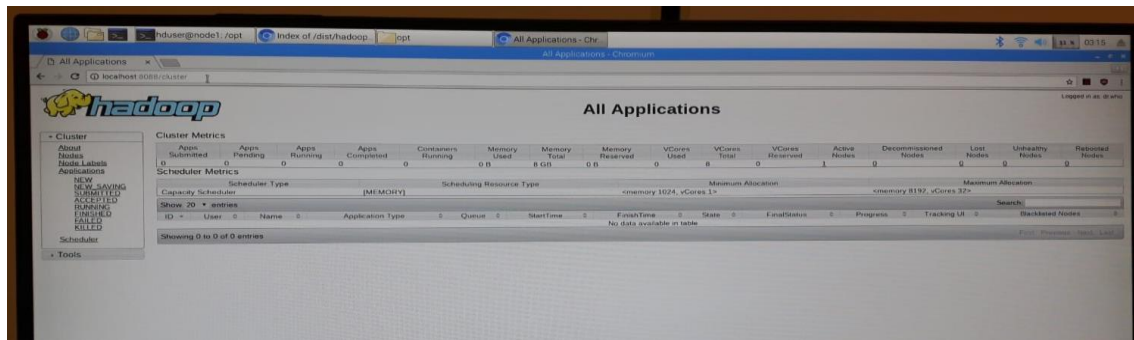


Figure 1: shows the number of nodes in the cluster

The next step is to install and configure Apache Spark on the top of the Hadoop. Apache Spark is an open source, which is known as a fast, easy-to-use and general engine for big data processing (Karau, Konwinski, Wendell, & Zaharia, 2015). Spark does not have any file system for distributed storage. In this configuration, Spark is installed on the top of Apache Hadoop YARN, which is a cluster management frameworks. This way Spark benefits from the distributed file storage. Spark environmental variables are edited in HADOOP_CONF_DIR file and added to the ./bashrc file. Finally, the Spark default template config file is replaced with Spark defaults config. The Spark running on the Raspberry can be seen on figure 2.

```
hduser@node1:/home/pi $ pyspark
Python 2.7.13 (default, Sep 26 2018, 18:42:22)
[GCC 6.3.0 20170516] on linux2
Type "help", "copyright", "credits" or "license()" for more information.
2019-01-19 11:28:33 WARN NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to

  ____      _
 / ___|  __| | | |
 \___ \  | | | | | |
  ___) | | | | | | |
 |_____|_|_|_|_|_|_|

 version 2.4.0

Using Python version 2.7.13 (default, Sep 26 2018 18:42:22)
SparkSession available as 'spark'.
>>>
```

Figure 2: shows Spark running

Sections 2.1 and 2.2 will discuss the issues encountered and the steps taken to seek the solutions on Hadoop and Spark setup in the Raspberry PI.

2.1 Issues and solutions of the Hadoop setup in Raspberry Pi

This section is concerned with the installation and configuration of Hadoop software cluster on Raspberry Pi. The focus of this setup is not step by step process of configuration, rather the problems encounter during configuration and the steps taken to resolve the problem. For more, on step by step configuration refer to (Mönning & Schiller, Nigelpond.com, 2018).

Issue - 1: The first Raspberry Pi was fully configured, cloning the configuration into the second Raspberry Pi did not work. Figure 3, shows the issue with cloning.

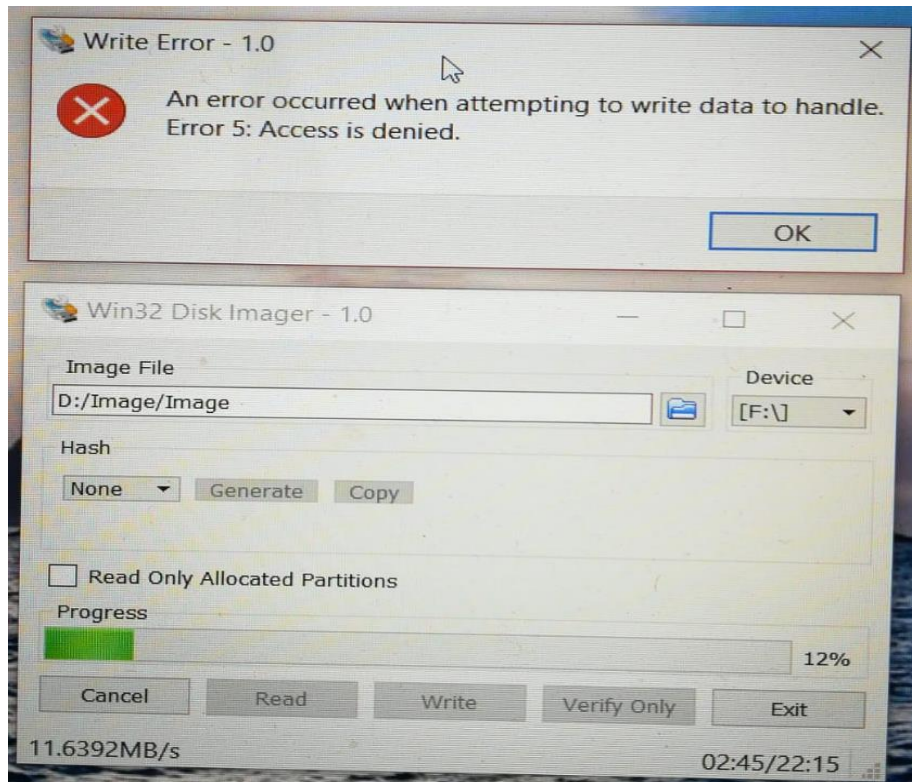
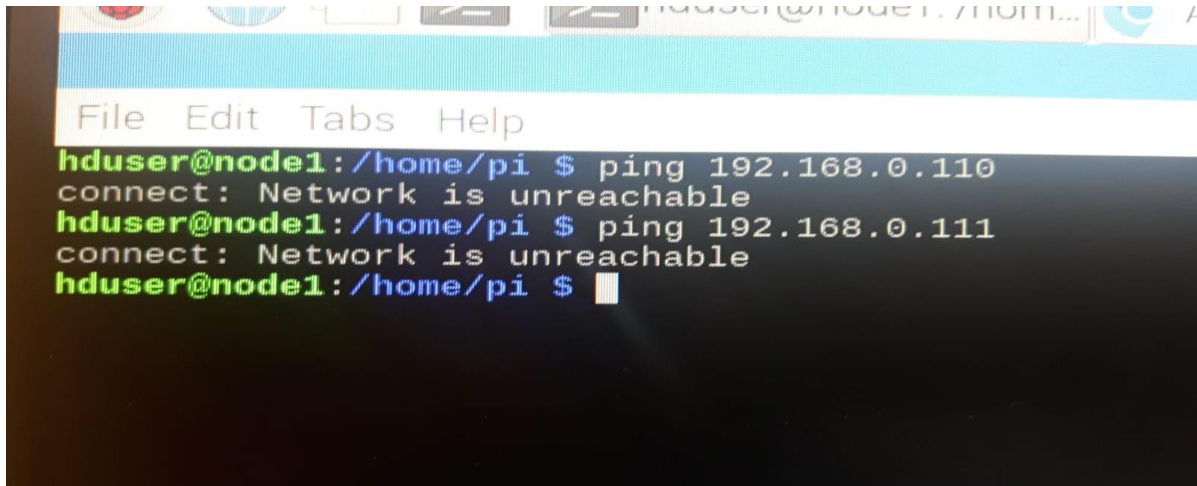


Figure 3: shows the error message while cloning the SD card

Solution - 1: The second Raspberry Pi was configured following the same procedure as the first Raspberry Pi.

Issue - 2 The two nodes were not able to communicate to each other and to themselves, figure 4.

- Pinging the first Raspberry Pi to itself and to the second Raspberry Pi, but there was no response message, unreachable to itself and to the other ones.



```
File Edit Tabs Help
hduser@node1:/home/pi $ ping 192.168.0.110
connect: Network is unreachable
hduser@node1:/home/pi $ ping 192.168.0.111
connect: Network is unreachable
hduser@node1:/home/pi $
```

Figure 4: shows that the error message Pinging from each node

- To check if the ports are correctly configured on the Raspberry, the command used is: `ifconfig`. Figure 5 below shows that only the loopback and the wlan0 interfaces are up in node2. The eth0 needs to be up.



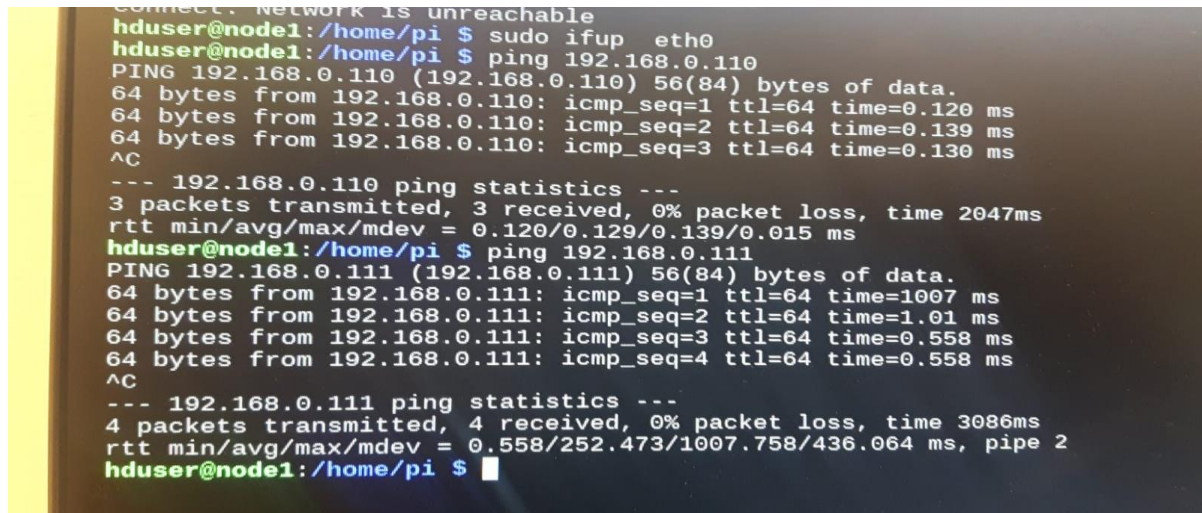
```
pi@node2:~ $ hduser
bash: hduser: command not found
pi@node2:~ $ su hduser
Password:
hduser@node2:/home/pi $ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 11 bytes 602 (602.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 11 bytes 602 (602.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 145.81.182.49 netmask 255.255.240.0 broadcast 145.81.191.255
    inet6 fe80::ba27:ebff:fef3:af8a prefixlen 64 scopeid 0x20<link>
    ether b8:27:eb:f3:af:8a txqueuelen 1000 (Ethernet)
    RX packets 7989 bytes 1553025 (1.4 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 57 bytes 8476 (8.2 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

hduser@node2:/home/pi $
```

Figure 5: shows the running ports

Solution - 2: In order to enable the eth0 interface on both Raspberry Pi's, the command used is: `sudo ifup eth0`. Then the two Pi's start to Ping to themselves and to communicate with each other. Figure 6 shows the results of the Ping between the nodes.

A terminal window screenshot showing the process of enabling the eth0 interface and performing ping tests between two Raspberry Pi nodes. The user 'hduser' is at 'node1' with a home directory of '/home/pi'. The first command is 'sudo ifup eth0', which results in a 'connect: Network is unreachable' error. The user then performs a ping to 192.168.0.110, showing successful results with 0% packet loss and a time of 2047ms. Next, the user pings 192.168.0.111, also showing successful results with 0% packet loss and a time of 3086ms. The terminal text is as follows:

```
connect: Network is unreachable
hduser@node1:/home/pi $ sudo ifup eth0
hduser@node1:/home/pi $ ping 192.168.0.110
PING 192.168.0.110 (192.168.0.110) 56(84) bytes of data.
64 bytes from 192.168.0.110: icmp_seq=1 ttl=64 time=0.120 ms
64 bytes from 192.168.0.110: icmp_seq=2 ttl=64 time=0.139 ms
64 bytes from 192.168.0.110: icmp_seq=3 ttl=64 time=0.130 ms
^C
--- 192.168.0.110 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2047ms
rtt min/avg/max/mdev = 0.120/0.129/0.139/0.015 ms
hduser@node1:/home/pi $ ping 192.168.0.111
PING 192.168.0.111 (192.168.0.111) 56(84) bytes of data.
64 bytes from 192.168.0.111: icmp_seq=1 ttl=64 time=1007 ms
64 bytes from 192.168.0.111: icmp_seq=2 ttl=64 time=1.01 ms
64 bytes from 192.168.0.111: icmp_seq=3 ttl=64 time=0.558 ms
64 bytes from 192.168.0.111: icmp_seq=4 ttl=64 time=0.558 ms
^C
--- 192.168.0.111 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3086ms
rtt min/avg/max/mdev = 0.558/252.473/1007.758/436.064 ms, pipe 2
hduser@node1:/home/pi $
```

Figure 6: shows the two nodes pinging to each other

Issue - 3: The first Pi was unable to SSH the second Pi. The reason for this was because the two nodes were configure independently, not cloned. The private and public keys of the nodes were different.

Solution - 3: The private and public keys of the name node was copied to the secondary node.

2.2 Issues and solutions of the Spark setup in Raspberry Pi

This section is concerned with the issues encountered during the installation and configuration of Apache Spark software. In this configuration Spark runs by taking advantage of dedicated cluster management frameworks Apache Hadoop YARN in the raspberry PI.

Issue - 1: after Spark was downloaded and configured on the first Raspberry PI. It was not able to communicate with the other PI, the reason could be due to the fact that the static IP Address assigned to the master node was not working. This prevented the Spark from running on the cluster. The dynamic IP address assigned by Inolland DHCP is used.

Solution - 1: due to time constrain, further steps are not taken to resolve the problem. The possible solution could be that every time the cluster starts, the IP address has to be copied and configured on the slaves and vice versa.

2.3 Testing the cluster

In verifying the correctness of the Hadoop configuration, the word counting on the apache license file is run using the map and reduce method. The result obtained can be seen in Appendix, figure 8. In case of Spark configuration, only one node is configured, the result of the word count algorithm can be seen in Appendix, figure 9.

3 Selecting an algorithm

This section presents the selected serial and parallel implementation of the algorithm. The code has enough comments to describe each line. In the serial implementation, the computation order is predefined. In other words, the computer is at one command at a time and it is an immediate execution. In contrast, in the parallel implementation, the computation order is not specified, it is more of a plan, but how will it execute or parallelize is up to the deploying environment.

Consider having a big data file of relations between users and their friends. The relation consists of a user's name, their age and their friend's name. A program must process this file and calculate the average number of friends for each age period, where age periods are: teens_youthAdult for ages [16, 20], Adult for ages [20, 40], MiddleAge for ages [40, 60] and old for ages [60, 72]. Sections 3.1 and 3.2 presents the serial and parallel implementations respectively.

3.1 Serial implementation

In the serial implementation, the file(s) was read and columns which contains user's name and age are extracted from the file and saved into a variable. Only those values are useful for the solution. Using that variable, a dictionary is built where the keys of this dictionary are pairs of (user name, user age) and the values of these keys are the number of friends for this combination which represents a user.

After finding the number of friends for each user, two dictionaries are calculated. The first dictionary contains the different people categorized in an age period (i.e. {'Adult': ('Chris', 23), ('Tom', 25) ...}). The second one contains the total number of friends for all users categorized in an age period (i.e. {'Adult': 23, 'MiddleAge': 35, ...}). Now, using these two dictionaries, it is easy to calculate the average number of user friends in each age period and that can happen by dividing the value of each key in the second dictionary by the list length – which is the value in the first dictionary – of the key in the first dictionary. The code of the serial implementation can be found in the attached file.

3.2 Parallel implementation

In the parallel implementation, reading the file(s) was done using Spark `textFile()` function. Thus, the data was saved in a RDD variable, which has its own `map`, `mapValues` and `reduceByKey` functions. As in the serial solution, a pair was made from each line of the file. This pair has as key (user name, user age) and as value the number '1'. Then, using `reduceByKey` function, an iteration over all pairs was made and the number of friends for each (username, user age) combination was computed. After this operation has completed, a mapping occurs, and every (user name, user age) combination is mapped to the correct age period (i.e. (('Chris', 25), 3) → ('Adult', 3)). Then, every value in the key-value pair is mapped to one, in order to get the average. We need to count the total number of friends for the given age and the number of times that age occurred. Finally, the total number of friends is divided by the by the total times it is encountered, which provides the average number of friends.

3.3 Expected performance gain

In general, as the dataset size becomes larger, parallel implementations produce shorter time than the serial implementation (Prodhan, Parvez, Hussain, Rumi, & Hossain, 2012). The `for` loop in the serial implementations are replaced with MapReduce in parallel implementation, where the performance gain is expected. In this problem, two mappings i.e. `map` and `mapValues` are used. These two operations do not require shuffling and thus performance gain can be expected, because the same operations were done in a serial solution. Only `reduceByKey` operation required shuffling in the parallel solution and by this a simple lack of performance is to be expected since it does not occur in the serial solution.

4 Dataset and correctness of the algorithm

4.1 Data Description

In this section the nature of the data will be discussed. The dataset in this experiment is a simulated data of a social network connection of 3.1 GB with over 72 million records (72,519,313). The data has 4 attributes: the ID of the user, the name of the user, the age and the name of friends connected to the user. The sample of the data can be seen on the figure 7 below.

	1	2	3	4	5	6	7	8	9	10
1		0	XFYI	63	AHPK					
2		1	UAIG	41	QHMS					
3		2	TSSY	47	BIPH					
4		3	TXCN	70	MUQZ					
5		4	FRPO	55	HGDM					
6		5	RLNU	40	DWO					
7		6	PJKT	49	LYXF					
8		7	EEHK	37	PXFP					
9		8	DBEV	34	BVWX					
10		9	TBUR	32	OJNE					
11		10	ZUUQ	29	YQCA					
12		11	SXWU	43	FXXM					
13		12	ZQIO	26	DHVP					
14		13	EFRL	63	IYDW					
15		14	TLPI	53	ONSW					
16		15	GNIO	61	MAMR					
17		16	LBPU	37	UVTQ					
18		17	CHVA	39	CIBE					

Figure 7: shows the sample data

4.2 Generating test data

For generating test data an excel is used and data of 1 K with 30 rows having the same attribute as the main data is created and filled with random values. The resulting generated data is displayed in Appendix, figure 10.

4.3 Verifying results

To verify the correctness of the implementation, the results can be parsed and tested. The expected result from the test data generated in the above section is calculated by hand and saved as a validating result (Appendix, figure 10). Then the result generated by the serial implementation is matched with the hand calculated validating result. If the two datasets have the matched values, the Python Boolean statement should return True and it is considered that the correctness of the algorithm is verified. The returned value was True, thus the correctness of the algorithm is verified. In the parallel implementation, the same validating procedure as the serial is followed. The returned value from the parallel implementation was float, which makes it complicated to cross compare with the validating data, thus it was cross checked manually. The results were correct and the implementation is verified. The code for verifying results can be found in the attached file.

5 Experiments and results

In general, the running time depends on the size of the data and the implementation of the algorithm. In order to evaluate the algorithm performance, the concept of time complexity is used, T . The value of T is the running time of the algorithm, i.e. the number of steps or the time required by the algorithm to perform the task (Prodhan, Parvez, Hussain, Rumi, & Hossain, 2012). The time complexity of an algorithm depends on the time complexity of each transformation step taken to perform the task.

After the correctness of the implementation is verified, to find the maximum data size that can be used on a personal computer to perform the task and investigate the performance of the serial and parallel implementations, different data sizes are used: 621 MB, 2.4 GB, 3.1 GB and 4 GB. As the data size increased from 621 MB to 2.4 GB, there was no performance gain from the parallel implementation. When the data size is increased to 3.1 GB, there was a performance gain, but when the data was further increased to 4 GB, the computer crashed due to memory issue.

In this section, the time taken for each implementations: serial implementation and the parallel implementation with different number of nodes is recorded and presented in chart 1, presented below. To calculate the time taken, the Python built-in time function is used. The time is set to start at the beginning of the algorithm and stopped when the task is over, computing the difference is the time taken for the task.

Note that in this experiment, the parallel computing environment developed on section 2 is not used. The EMR parallel computing environment from amazon is used. Amazon EMR provides a managed Hadoop and distributed framework such as Apache Spark that makes it easy and fast for computation (Amazon, 2018). The type of the computers used in amazon are the latest m4 large.

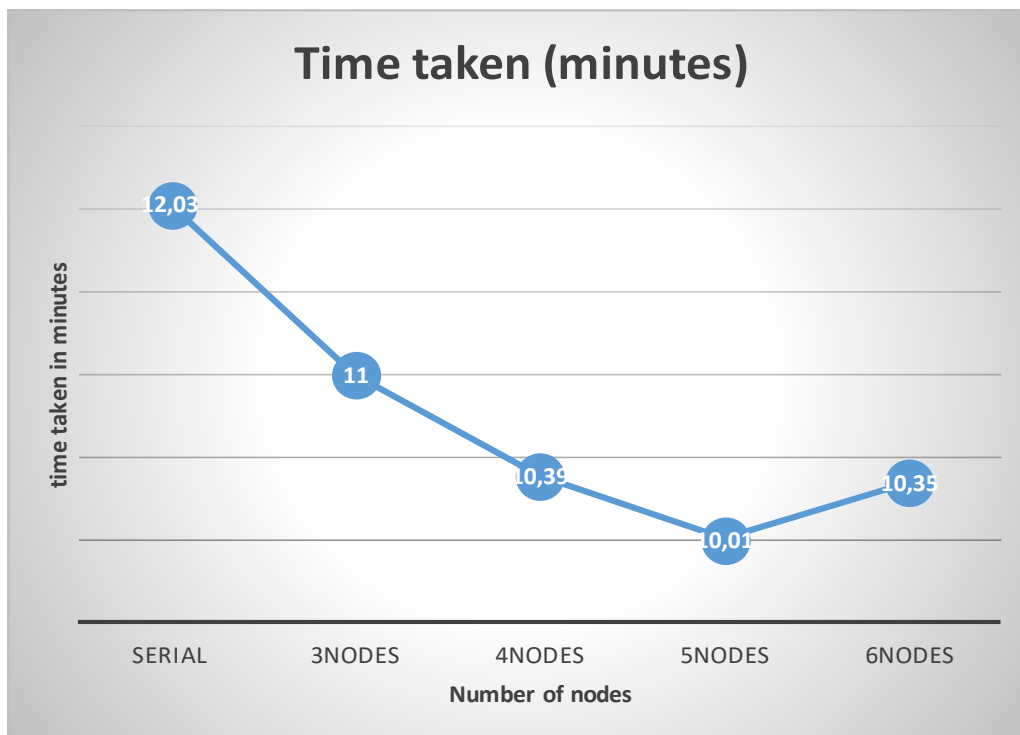


Chart 1: shows time complexity of the computation

As seen in the chart above, the time taken to accomplishing the task using parallel implementation reduces as the number of processors increased from 1 to 5. It reached its minimum value when there are five processors, 1 master and 4 slaves. Thus, the optimal number of processors required given the dataset is five and the minimum time taken is ten minutes. As the number of nodes is increased past 5, the time taken gradually starts to increase. The reason for the performance decrease could be the overhead caused by communication between workers (nodes).

6 Measuring performance gain

In order to measure the performance gain, Amdahl's law is used. Before delving into the details of Amdahl's law, it is best to define and understand the term, speedup, first. Speedup is the execution time for the algorithm in a serial implementation with a single processor divides the time of execution of the algorithm parallel with many processors (Brown, 2018). The code below shows how to compute the SpeedUp:

$$SpeedUp = \frac{T(1)}{T(i)}$$

Another important term to Efficiency. Efficiency is defined as the ratio of speedup to the number of processors. Efficiency measures the fraction of time for which a processor is usefully utilized.

$$Efficiency = \frac{SpeedUp}{number\ of\ processors\ use(N)}$$

Where T(1) – time for serial implementation with one processor and

T(i) – the time taken for parallel implementation with i number of processors.

If the SpeedUp is analysed in more detail, there are N number of threads or workers in a given task and it is natural to expect the task will be done in 1/N times faster than it takes for a single thread or worker. In the reality of parallel computing with multiprocessors, the theoretical maximum speedup of a task execution expected from a paralyzed algorithm is given by Amdahl's law. Amdahl's law is the measure of how much improvement comes from parallelism (Brown, 2018). Amdahl's law can be written mathematically as follows:

$$SpeedUp \leq \frac{(1)}{(1 - P) + \left(\frac{P}{N}\right)}$$

Where P - the fraction of the code that can be done in parallel

N – is the number of processors running

1-P – the fraction of the code that cannot be paralyzed

It is clear from the Amdahl's law formula that as the number of processors N goes to infinity, P/N drops to zero and the Speed up is completely dominated by fraction which is sequential, (1 - P), hence there is bad or no gain from the parallelism.

Back to our experiment:

$$SpeedUp = \frac{12.03(minutes)}{10.01\ (minutes)} = 1.20$$

And the efficiency is:

$$Efficiency = \frac{1.20}{4} = 0.3$$

The speedup is 1.2 and the efficiency is 0.3. Based on the formula for SpeedUp, the (1-p) fraction of the code which cannot be parallelized is not so close to zero which means not the entire code can be parallelized. Otherwise, if (1 -P) is as small as zero, the entire code could be parallel, then the speedup could actually be proportional to the number of processor used, four. Moreover, there are two shuffle statements in the parallel implementation, which makes the parallel implementation slow and computationally extensive.

7 Conclusion

To conclude, analysing the performance gain of the parallel implementation required the Amdahl's law. The parallel implementation with less than 4 nodes is ignored, since the focus is to calculate the performance gain with the optimal number of processors given the dataset. When the size of the data is small, the serial implementation is faster than the parallel. As the size increases, the parallel implementation becomes faster than the serial. The 3.1 GB is the maximum size that the personal computer used in the experiments can analyse using the serial implementation.

The performance gain was found to be 1.2, which is less than the number of nodes. Thus, it follows Amdahl's law. Based on the Amdahl's law, the maximum speedup or performance gain that can be achieved in this experiment is less than or equal to four, i.e. the number of processors used. Based on the performance gain, it can be concluded that in the parallel implementation, there is code that cannot be fully parallelized that caps how much of the performance gain we can get from parallelizing the code.

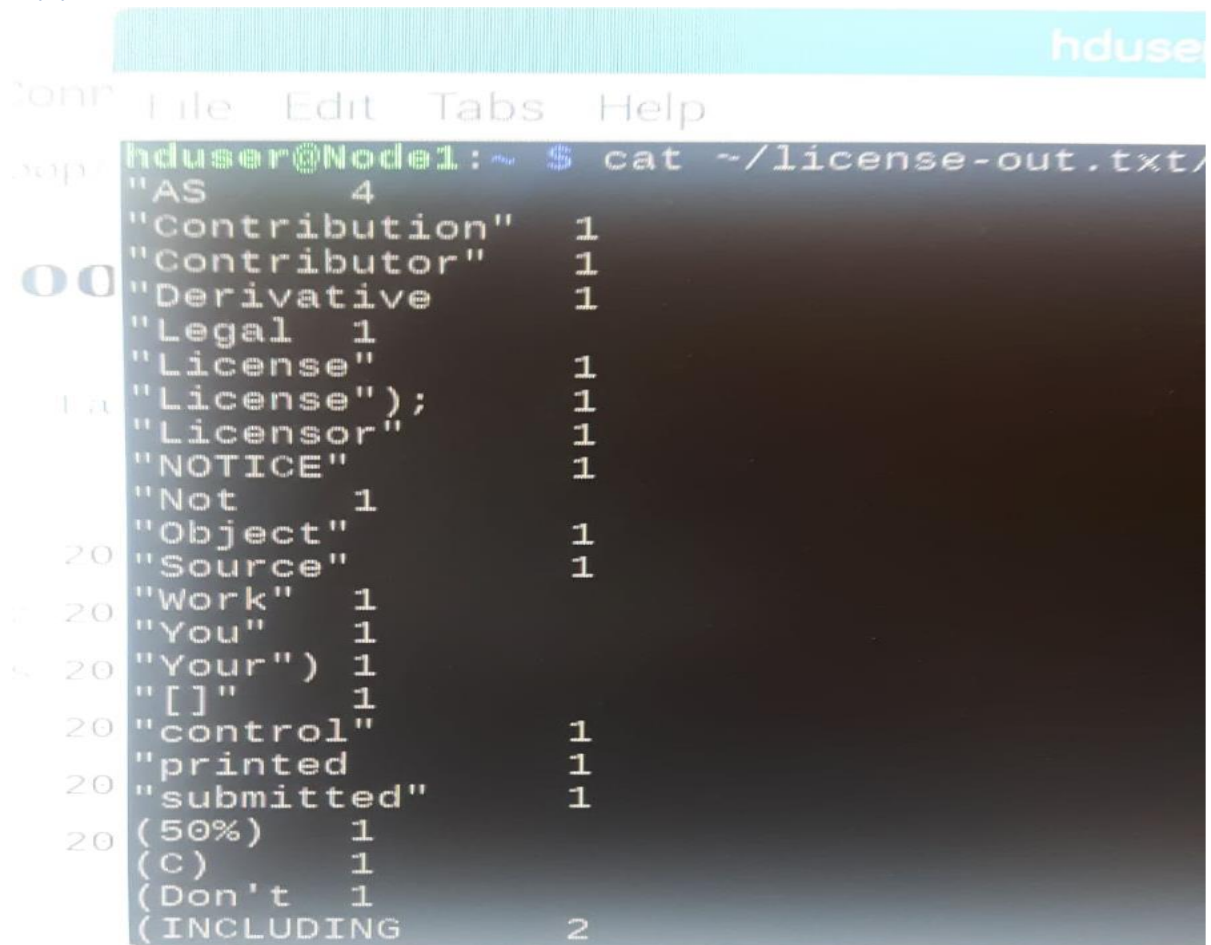
8 Discussion and Recommendation

Initially, the parallel implementation was expected to outperform the serial implementation. Based on the experiment results, the parallel implementation has outperform the serial, but the performance gain is so small that it may not significant. Therefore, there is a slight performance gain from the parallel implementation that might not be considered significant.

In the experiments and results section, it can be clearly seen that as the size of the data increases, the performance gain also increases. The maximum data size used, 3.1 GB, might not to be the right data size to get a significant gain from the parallel implementation.

As a recommendation, the data size is an important input to gain from parallel implementation. In order to get a more significant performance gain, the dataset size has to be bigger than 3.1GB. Moreover, in the parallel implementation, there are two shuffle statements which make the parallel implementation slower, they should be replaced with more efficient methods.

Appendix

A terminal window with a blue title bar labeled 'hduse'. The menu bar shows 'Conn', 'File', 'Edit', 'Tabs', and 'Help'. The prompt is 'hduser@Node1:~ \$'. The command 'cat ~/license-out.txt/' has been executed, displaying a list of words and their counts. The output is as follows:

```
"AS" 4
"Contribution" 1
"Contributor" 1
"Derivative" 1
"Legal" 1
"License" 1
"License"); 1
"Licensors" 1
"NOTICE" 1
"Not" 1
"Object" 1
"Source" 1
"Work" 1
"You" 1
"Your") 1
"[]" 1
"control" 1
"printed" 1
"submitted" 1
(50%) 1
(C) 1
(Don't 1
(INCLUDING 2
```

Figure 87: shows the testing data

(Image from Stephen)

Figure 9: shows the word counting running on Spark

R13C4				
	1	2	3	4
1	Adult	1.1		
2	teens_youthAdult	1.17		
3				
4				
5				

Figure 10: shows the validating data

References

- Amazon. (2018, jan 7). *emr*. Retrieved from aws.amazon.com: <https://aws.amazon.com/emr/>
- Apache Hadoop. (2018). Retrieved from Hadoop.apache.org: <https://hadoop.apache.org/>
- Brown, R. (2018, jan 9). *Amdahl's Law & Parallel Speedup* . Retrieved from webhome.phy.duke: <http://webhome.phy.duke.edu/~rgb/brama/Resources/als/als/node3.html>
- Foundation, R. (2018). *Raspberry Pi*. Retrieved from Raspberry Pi — Teach, Learn, and Make with Raspberry Pi: <https://www.raspberrypi.org/>
- Gabriel, E., Resch, M., Beisel, T., & Keller, R. (1998). Distributed computing in a heterogeneous computing environment. *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, n.d.
- Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). *Learning Spark: lightning-fast big data analysis*. O'Reilly Media, Inc.
- Khalidi, D., Jouvelot, P., Ancourt, C., & Irigoin, F. (2012). Task parallelism and data distribution: An overview of explicit parallel programming languages. *International Workshop on Languages and Compilers for Parallel Computing*, 174-189.
- Li, J., & Li, B. (2017). On Data Parallelism of Erasure Coding in Distributed Storage Systems . *International Conference on Distributed Computing Systems*, 45-56.
- Mönning , C., & Schiller, W. (2015). A Hadoop data lab project on Raspberry Pi .
- Mönning , C., & Schiller, W. (2018). Retrieved from Nigelpond.com: <http://www.nigelpond.com/uploads/How-to-build-a-7-node-Raspberry-Pi-Hadoop-Cluster.pdf>
- Ostrouchov, G. (1987). Parallel computing on a hypercube: an overview of the architecture and some applications. *In Computer Science and Statistics, Proceedings of the 19th Symposium on the Interface* , 27-32.
- Prodhan, U. K., Parvez, S., Hussain, I., Rumi, Y. F., & Hossain, A. (2012). PERFORMANCE ANALYSIS OF PARALLEL IMPLEMENTATION OF ADVANCED ENCRYPTION STANDARD (AES) OVER SERIAL IMPLEMENTATION. *International Journal of Information Sciences and Techniques (IJIST)*, n.d.
- Wei, C., Feng , T., Chieh , L., & Athanasios, C. (2015). Big data analytics: a survey. *Journal of Big Data*, n.d.
- Yeo, C., Buyya, R., Pourreza, H., Eskicioglu, R., Graham, P., & Sommers, F. (2006). *Cluster computing: High-performance, high-availability, and high-throughput processing on a network of computers*. Boston: Springer.