# Distributed Hadoop Mapreduce Cluster on Android Devices

Dustin McAfee and Alan Grant

**Abstract**—New Android mobile devices are becoming increasingly powerful with larger amounts of memory, with most equipped with quad or even octa-core processors. Hadoop Mapreduce is installed and optimized on top of a distributed cluster of Android devices. Utilizing a distributed system of Android mobile devices for processing of small data sets with Hadoop Mapreduce is shown as feasible. Example benchmark algorithms such as TestDFSIO and Terasort are used for measuring network IO and sorting run-times on a three node distributed cluster of two Android devices as slaves and a laptop device running Linux as the master node. These benchmarks are performed and compared for a few different configurations of the Hadoop cluster.

## 1 Introduction

Hadoop Mapreduce is one of the most popular algorithms for big data processing. Big data is termed here as a large number of data to be processed in a short period of time. Hadoop is a scalable, open source, fault-tolerant framework that allows for distributed processing of large data sets across clusters of computers using simple programming models. Mapreduce is the programming model used to process large data sets stored in Hadoop [1]. Porting Hadoop Mapreduce to the mobile Android platform could have hardware advantages, since the cost of Android hardware is generally low. Android applications use the filesystem directly, making it possible to manage files similar to that of a Unix system. An Android Hadoop cluster could also enable a mobile workforce instead of a desktop workforce.

In order to port Hadoop to Android, a Linux image is installed and run inside of the Android operating system; this requires root privileges. Another requirement is Busybox, an open source package installed with root privileges which provide replacements for many GNU utilities, including chroot [2]. An application called Linux Deploy [3] from the Google Play Store is used to automatically install and configure the Linux image; it allows for custom configurations of Linux boot images and uses chroot to change root directory and boot Linux over Android by making a subprocess tree for a specific system. In this case chroot is used with an Ubuntu Xenial image to boot Ubuntu on top of the Android operating system.

Open JDK 8 and Hadoop 2.7.5 are installed on a rooted Samsung Galaxy S7 running Android Nougat, a Hikey960 development board running AOSP 4.9, and a laptop running Ubuntu Xenial. Sample Mapreduce benchmarks are performed on the cluster and runtimes compared to different slave configurations. Different Hadoop cluster configurations are benchmarked and compared for efficiency in runtime for a data sort and efficiency in network IO throughput.

### 1.1 Related Work

Previous work most related with this project involves deploying an image of Linux over the Android operating system [4] [5]. While it is also possible to set up the appropriate environment manually in Android using BusyBox and chroot, a minimal installation of a Linux environment can be well optimized for running within an Android operating system [3]. The Linux image chosen for the Android nodes is Ubuntu Xenial 16.04, while the laptop master-node also runs Ubuntu Xenial 16.04. Unlike previous work, the Ubuntu image running within Android runs with an ssh server, but without a visual server, since the visual server is not necessary for the slave nodes.

In Android, java classes can not be installed and executed directly from the command line. Instead, the application must always exist in an *.apk* file. Running a Linux image inside of Android helps alleviate this issue. However, earlier work done with Hadoop version 0.19.0 extracted the java libraries, edited source code, and compiled them into *.apk* files [6]. This is not practical for our implementation, but certain configuration aspects were adopted from this, such as DFS block size reduction.

### 1.2 Installation

Hadoop 2.7.5, JDK8, and openSSH is installed on each machine. Hadoop is in the directory */usr/local/hadoop* and JDK8 is in the directory */usr/lib/jvm/default-java*. The slave hostnames are node1 for the Hikey960 and node2 for the Samsung Galaxy S7, and the master node hostname is node-master. The file */etc/hostname* for the master node is changed from localhost to node-master, and the slave nodes are kept localhost. The file */etc/hosts* is updated for each machine to have the ip address of node1, node2, and node-master. For each machine, an rsa key pair is generated and authorized for every other machine using the following command:

```
$ ssh-copy-id remote_username@remote_hostname
```

The environment must be set up as well, and so the following is added to the */.bashrc* file:

```
export JAVA_HOME=/usr/lib/jvm/default-java
export HADOOP_CLASSPATH=$JAVA_HOME/lib/tools.jar
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_INSTALL=/usr/local/hadoop
export PATH=$PATH:$JAVA_HOME/bin
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
```

```
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib
    /native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME
    /lib"
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

The $JAVA_HOME variable is reset when starting Hadoop, so the file *$HADOOP_INSTALL/etc/hadoop/hadoop-env.sh* must be updated like so:

```
export JAVA_HOME=/usr/lib/jvm/default-java
```

The XML configuration files for the initial configuration in the directory */usr/local/hadoop/etc/hadoop/* are shown in the appendices. The *yarn-site.xml* file (appendix D) has a property *yarn.resourcemanager.hostname* (in red font), which value is different for each node: It is not existent in the master node, and for the slave nodes it contains the hostname of the slave (e.g. node1 or node2). The properties in the appendices that are in red font indicate different values specific to the node.

## 2 METHODS

The Hadoop configuration is installed on a Sprint Samsung Galaxy S7 Android cell phone and Hikey 960 Android Open Source Project development board. The Galaxy S7 has four cores each running at 2.1GHz, and 4GB of RAM. The Hikey 960 has only 3GB of RAM, but is equipped with eight cores: four running at 2.3GHz and four running at 1.8GHz. Both devices have a limited amount of virtual memory: about 10GB. Block size of 64MB are chosen (see appendix B), which is half the default block size of 128MB. It is important to note here that the block size of the cluster should be chosen based on the job at hand, and this block size will remain constant in order to measure network IO and runtime performance of other cluster configurations.

The number of V-cores set up (see appendix D) are exactly one less than the number of cores at each node. The initial amount of memory allocated to the resource manager (see *yarn.nodemanager.resource.memory-mb* and *yarn.scheduler.maximum-allocation-mb* in appendix D) is 1.5GB for both slave nodes, and the minimum allocation for each container is set to be 512MB. Also, the virtual memory check is manually turned off. For each node, the mapper memory allocation is initially set to 1GB, and the reducer memory allocation is initially set to 1.5GB (see appendix C).

This configuration is first benchmarked as a reference for later optimizations. After measuring the network IO throughput using TestDFSIO (see section 2.1) and the runtime of sorting random data using Terasort (see section 2.2), Mapreduce intermediate compression is turned on (see section 3.1) and compared to the initial configuration, and then the sort buffer size is increased (see section 3.2) and compared to the initial configuration. Section 3.3 discusses issues found in the cluster and solutions if there are any, and section 4 discusses the findings.

### 2.1 TestDFSIO

TestDFSIO is a distributed IO benchmark tool that is used to test the network configuration. There are two calls: First to test the write throughput, and second to test the read throughput. The output measures total throughput and average throughput per task. The dataset chosen for this benchmark is 1GB in various file sizes and number of files. Sample write and read benchmark calls for two 512MB files are listed, respectfully, below:

```
$ hadoop jar /usr/local/hadoop/share/hadoop/
    mapreduce/hadoop-mapreduce-client-jobclient
    -2.7.5-tests.jar TestDFSIO -write -nrFiles 2 -
    fileSize 512MB
$ hadoop jar /usr/local/hadoop/share/hadoop/
    mapreduce/hadoop-mapreduce-client-jobclient
    -2.7.5-tests.jar TestDFSIO -read -nrFiles 2 -
    fileSize 512MB
```

The results of the TestDFSIO benchmark writing and reading 1GB of data are shown below in Figures 1 and 2, respectfully.

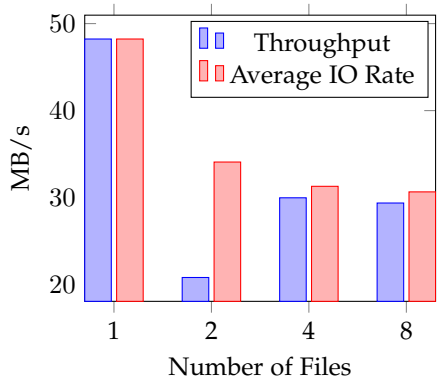


Fig. 1. TestDFSIO Write on 1GB of Data

The throughput and IO rate results are based on individual map tasks, and not the overall throughput of the cluster. The number of files determines the number of map tasks that exist. As one might expect, the throughput for one file is much higher than when split into multiple files. Many small files tend to slow the cluster down, and thus the lowest throughput is splitting the data into 128MB files, and is not very practical in a real-world scenario. The standard deviation for the write tests ranges from 6.07MB/s in 8 files to 10.56MB/s in 2 files (the test on a single file shows a standard deviation of zero).

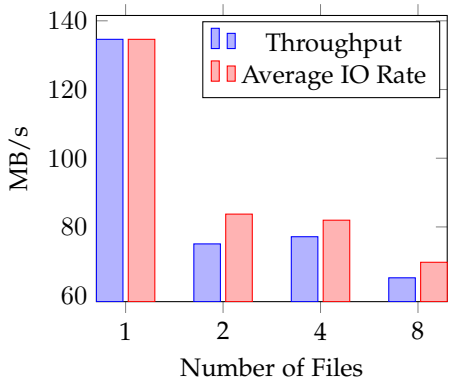


Fig. 2. TestDFSIO Read on 1GB of Data

The IO throughput in the this local network shown here seems to be efficient for this project. However, the standard

deviation of the read tests ranges from 17.59MB/s in 8 files to 26.95MB/s in 2 files (and a standard deviation of zero for one file), which seems to be a little high. This indicates that there might be room for a performance increase (likely one of the cluster nodes is exhibiting performance related issues).

### 2.2 Terasort

Terasort is a popular benchmark that measures the amount of time to sort randomly distributed data using MapReduce. It is commonly used to measure MapReduce performance on an Apache Hadoop cluster. In this study with two mobile slave nodes, 128MB to 1GB of randomly distributed data is generated, and the calls to Terasort are timed and plotted.

First, the random data must be generated with a call to Teragen. A sample call to Teragen that generates 1GB of data is shown below.

```
$ hadoop jar hadoop-mapreduce-examples-2.7.5.jar
    teragen 10000000 <output-dir>
```

The target focus here is not the generation of the dataset, however, Teragen generates a specified amount of 100-byte rows of data. Thus, this one-time invocation of Teragen command generates 1GB of data to be used with multiple invocations of Terasort for the same dataset. The Terasort benchmark is performed with a simple time command:

```
$ time hadoop jar hadoop-mapreduce-examples-2.7.5.
    jar terasort <input-dir> <output-dir>
```

The run-times of sorting 128MB to 1GB of data is shown below in Figure 3.

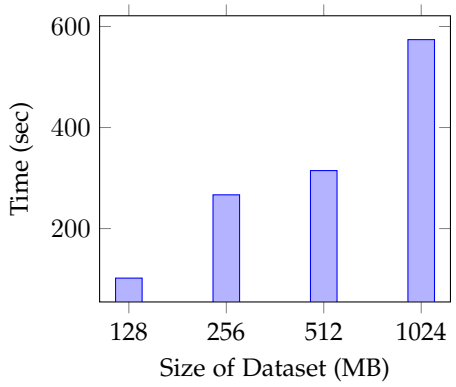


Fig. 3. Run Times of Terasort

These results show that this cluster is feasible for running a Hadoop Mapreduce job on a small set of data, but quickly reaches an unacceptable performance level on larger-scaled jobs (about ten minutes for sorting one gigabyte of data). Adding more nodes would most likely improve the accuracy of measurements and performance of this cluster, and future work should definitely take this into account.

In the next section, these benchmarks are re-executed with slight changes in configuration, and compared to this initial configuration. The cluster properties that are changed include intermediate value compression and increased sort buffer memory. There are many configuration values that should be observed to find an optimal configuration, and not all values are checked in this paper. The properties added for performance comparisons are not reflected in the appendices.

## 3 Results

The initial configuration benchmarks show that this Hadoop Cluster will work on small jobs, but also does not scale to large datasets with the limited number of nodes. The goal here is to tune the configuration to yield better results for this three node distributed cluster. The first attempt is a compression of intermediate Mapreduce values in a hope that the network performance per map task from TestDFSIO increases.

### 3.1 Compress Intermediate Values

The first configuration optimization attempt is to compress the intermediate map data via setting the *mapreduce.output.fileoutputformat.compress* to true in the *mapred-site.xml* file. The compression codec is left to the default codec.

The TestDFSIO is run first on this configuration. The results are presented in Figures 4 and 5, respectfully, below.

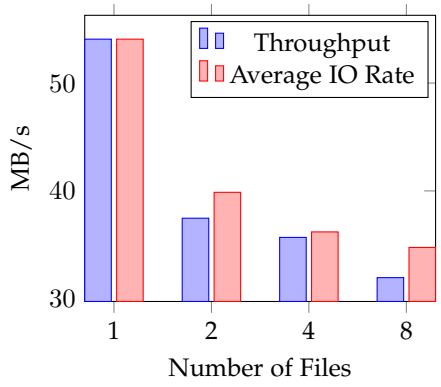


Fig. 4. TestDFSIO Write on 1GB of Data with Intermediate Compression

The standard deviation for the write tests ranges from 4.34MB/s in 4 files to 9.79MB/s in 2 files (with a standard deviation of 0.01MB/s for 1 file). The write measurements here are about 5MB/s faster than before, with an exception to the throughput for 2 files, which is about 17MB/s faster. This seems as though it might have improved the clusters' writing performance.

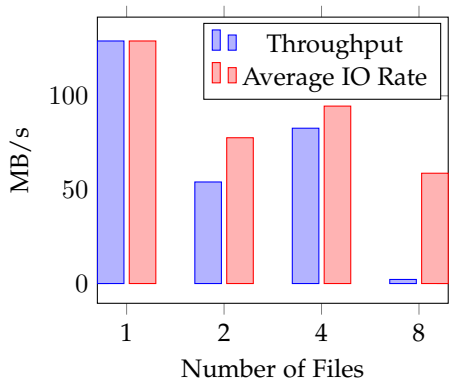


Fig. 5. TestDFSIO Read on 1GB of Data with Intermediate Compression

The reading test was even more inconclusive this time. The standard deviation for the read tests ranges from 33.31MB/s in 4 files to 57.22MB/s in 8 files (with a 0.03MB/s standard deviation for 1 file). The throughput is dramatically low for eight 128MB files (2.18MB/s), most likely because of a bottleneck in the network. The read throughput and IO rate visually seems to have increased, but these results are inconclusive. There may be instability within this configuration on one of the nodes causing performance issues here.

Terasort was ran with this configuration to test for significant performance increases, as well. These results are shown in Figure 6 below.

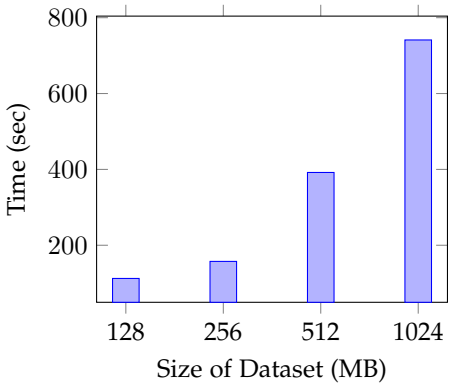


Fig. 6. Run Times of Terasort with Compressed Intermediate Values

The same initial datasets are used for this configuration for comparison. There seemed to be a significant performance increase in the data size of 256MB (about 109 seconds faster), but also a significant performance decrease in the data size of 1GB (about 167 seconds slower). Larger jobs may call for compression, but for this job size, it seems that the compression is not necessary. It also seems here that these results would likely be more conclusive if tested on a larger cluster (with more mobile slave nodes) and on a larger data set.

In the next section, the spill buffer memory is increased and the Terasort benchmark performed.

### 3.2 Increased Buffer Space

Mapreduce uses a circular memory buffer to hold the intermediate map results in memory. When this buffer reaches a certain threshold, it will spill to disk. Too many files spilled to disk can hinder performance. The total amount of buffer memory (*mapreduce.task.io.sort.mb* in *mapred-site.xml*) to use while sorting files is 100MB by default. This value is changed to 500MB for this benchmark, and the *mapreduce.map.sort.spill.percent* configuration variable (in *mapred-site.xml*) is set to 1.0 (default is 0.8), which will make use of the complete buffer space (instead of only eighty percent before spilling to disk). The JVM heap sizes for map and reduce tasks are adjusted to 80 percent of the maximum size of the map and reduce tasks. The intermediate compression is also removed for this benchmark. The expected result is that the run-time of Terasort should decrease, with less data being spilled to disk. The TestDFSIO benchmark is not shown, since there were not enough spilled records to justify.

With the initial configuration, a call to Terasort for 1GB of data spills 23355441 records (Terasort on 512MB of data spills 10000000 records). After an increase in the buffer size, the number of spilled records for 1GB of data decreases to 20000000. Contradictory to the original hypothesis, however, it was observed that this number did not change for the 512MB and below sets of data. This indicates that the performance will likely increase for larger datasets, but not for the smaller ones. Terasort was ran with this configuration to test for significant performance increases in run-time. These results are shown in Figure 7 below.

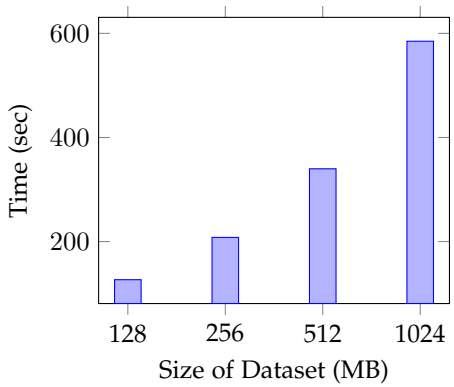


Fig. 7. Run Times of Terasort with Increased Buffer Space

There does not seem to be any significant decrease in run-time for Terasort on this cluster with an increase of buffer space. In fact, it even seems comparable to the initial configuration benchmark results. This is probably due to the size of the cluster, which seems to be too small to test for significant differences in these configurations.

### 3.3 Known Issues

A major issue in the first implementations of the cluster comes from a known bug in the procps package in the Xenial release of Ubuntu. This bug was unclear to the authors at first, and caused the slave nodes to crash randomly while the cluster is running. Upon investigating, it was found that, when the nodemanager called kill with a negative argument, there was incorrect parsing that lead to call sys_kill(-1), which killed all processes in the system and caused a system restart. This bug is reported in [7]. The

fix to this issue involves compiling an older version of the procps package and installing the older version kill system call. This bug is particularly annoying, because the Linux over Android software re-installs the faulty kill command each time it is initialized.

Another known issue is that Linux on Android is sometimes unstable when installed in the free space partition of the device. It is recommended to partition an external level 10 micro-sdcard and install the Linux distribution on the partitioned storage device. However, fast external flash memory was not available at the time, so the Linux image was installed into a folder in the free memory of the device. No known issues came from this installation, aside from an occasional crash when attempting to stop and unmount the Linux container from the Android device. This has been reported in [3].

Finally, the tests were ran in a matter of three weeks, which varied on time of day and internet traffic. The entire cluster is on a single local area network, and the network performance seemed to be a great obstacle when benchmarking the cluster. There was great effort expended to minimize the amount of traffic on the network for the best performance measurements when executing the benchmarks, but sometimes it seemed as thought the network was still a bottleneck in some cases (See the read throughput rating in Section 3.1 on Figure 5)

## 4 Conclusion

With the increasing availability of Android operating systems and mobile devices, and with the increasing memory and speed of these devices, forming a distributed cluster for big data processing seems to be becoming a feasible project. A Linux over Android Hadoop Mapreduce cluster installation is shown in a reproducible manner, with known issues about the cluster discussed. The cluster is configured, successfully executed, and benchmarked multiple times with minor configuration differences in an attempt to optimize the performance of the cluster. Unfortunately, there were a limited number of nodes in the cluster due to limited resources, and therefore most of the tests were inconclusive. It is shown, however, that it is likely that intermediate compression of Mapreduce results increases the average writing throughput and IO rate per map task (see section 3.1). Also, there is an observed decrease in spilled records for the Terasort benchmark on the dataset of 1GB from the map tasks with the buffer space increased (see section 3.2), which indicates an increase in performance for the cluster on larger jobs. There were no instances of failed tasks in any of the benchmarks that were executed.

## References


[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[2] D. V. Rob Landley and B. Reutner-Fisher, "Busybox," 2018.

[3] A. Skshidlevsky, "The software to automate processes of installation and launch of gnu/linux applications on android devices," *Information and Control Systems/Informazionno-Upravlyaushie Sistemy*, vol. 66, no. 5, pp. 56–60, 2013.

[4] K. S. S. K. D. N. Bothe B. Namrata, Mate N. Anagha, "Migration of hadoop to android platform using chroot," Apr. 2016.

[5] B. B. R. M. S. J. N. R. J. Prachil S. Tambe, Prof. S. Y. Raut, "Implementation of hadoop pseudo-distributed cluster on android using chroot," *International Journal Of Engineering And Computer Science*, vol. 4, Apr. 2015.

[6] E. Marinelli, "Hyrax: Cloud computing on mobile devices using mapreduce," Sep. 2009.

[7] D. Frazier, "Kill incorrectly parses negative pids," Oct. 2016.


## Appendix

### A

The */usr/local/hadoop/etc/hadoop/core-site.xml* file is updated like so on each node:

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://node-master:9000</value>
  </property>
</configuration>
```

### B

The */usr/local/hadoop/etc/hadoop/hdfs-site.xml* file is updated like so on each machine:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/usr/local/hadoop/dfs/nameNode</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>/usr/local/hadoop/dfs/dataNode</value>
  </property>
  <property>
    <name>dfs.blocksize</name>
    <value>67108864</value>
  </property>
</configuration>
```

### C

The */usr/local/hadoop/etc/hadoop/mapred-site.xml* file is updated like so on each machine:

```
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
  <property>
    <name>yarn.app.mapreduce.am.resource.mb</name>
    <value>512</value>
  </property>
  <property>
    <name>mapreduce.map.memory.mb</name>
    <value>1024</value>
  </property>
  <property>
    <name>mapreduce.reduce.memory.mb</name>
    <value>1536</value>
  </property>
  <property>
    <name>mapred.job.tracker</name>
    <value>node-master:9001</value>
  </property>
</configuration>
```

## D

The */usr/local/hadoop/etc/hadoop/yarn-site.xml* file is updated like so on each machine (the red text is specific to the node; *yarn.nodemanager.hostname* property is nonexistent on the master node configuration):

```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.acl.enable</name>
    <value>0</value>
  </property>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>node-master</value>
  </property>
  <property>
    <name>yarn.nodemanager.hostname</name>
    <value> slave-node-hostname </value>
  </property>
  <property>
    <name>yarn.nodemanager.resource.cpu-vcores</name>
    <value> 3 </value>
  </property>
  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>1536</value>
  </property>
  <property>
    <name>yarn.scheduler.maximum-allocation-mb</name>
    <value>1536</value>
  </property>
  <property>
    <name>yarn.scheduler.minimum-allocation-mb</name>
    <value>512</value>
  </property>
  <property>
    <name>yarn.nodemanager.disk-health-checker.max-
        disk-utilization-per-disk-percentage</name>
    <value>98.5</value>
  </property>
  <property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
  </property>
</configuration>
```