Coded MapReduce for Terasort Application using Dropbox

Indu Vadhani Suresh Jing Li Shraddha Thakkar

***Abstract*- MapReduce is a commonly used framework for executing data-intensive tasks on distributed server clusters[1]. Coded MapReduce is a new framework that enables and exploits a particular form of coding to significantly reduce the inter-server communication load of MapReduce. In particular, Coded MapReduce exploits the repetitive mapping of data blocks at different servers to create coded multicasting opportunities in the shuffling phase, cutting down the total communication load by a multiplicative factor that grows linearly with the number of servers in the cluster. we are extending the work to cover terasort application. TeraSort is a popular benchmark that measures the amount of time to sort one terabyte of randomly distributed data on a given computer system[4].**

I. Introduction

In the present day of communication scenario, there is only a curve that increases exponentially connecting almost the entire world and transferring humungous amount of data between them. Load on the systems that support such connectivity is highly undesirable. We present a comparison between two benchmarking techniques such as Mapreduce and coded Mapreduce and implement the techniques on the terasort application. The main goal of terasort is to sort 1 Tb or any other amount of data as fast as possible[5]. This ideally requires minimal load conditions . These requirements are analysed further with the help of MapReduce and Coded MapReduce.

A. Original MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster [2]. This is the primitive benchmarking technique without any coding . A general MapReduce job consists of an input file that contains N subfiles and output keys Q[3].Each key needs to be evaluated using N intermediate values one from each subfile. K available servers are connected via multicast LAN network. The load is decided by the paramters N,K,Q. Each server evaluates Q/K output keys and processes N/K subfiles. They need another Q/K(N-N/K)intermediate values. Communication load is given as QN(1-1/K). In order to reduce the communication load which is a bottleneck , we resort to coded Mapreduce.

B. Coded MapReduce

The main goal of coded MapReduce is to slash the communication load by coding[3]. When doubling the processing load (r), Coded mapReduce can cut the communication load by more than 50%. Without

coding ,the reduction would have been only a factor of (K-1)/(K-2) which vanishes as K inrceases.The key coding technique used is the bit-wise XORs of the required intermediate values.The key challenge here is the careful assignment of map tasks to the server such that multicast coding opportunity of size r arises in the shuffling phase.The problem resembles prefetching design in cache networks.

II. Implementation

A. Original MapReduce

Step1: Map Task Assignment

In our implementation, we take six chapters as an example and assign map tasks as follows: Server1 = {1, 2}, Server2 = {3, 4}, Server3 = {5, 6}. We take server2 as an example to explain the code:

```
#load data from chapter 3 and chapter 4
raw3 = loadtxt("chap3.txt", comments="#", delimiter=",", unpack=False)
raw4 = loadtxt("chap4.txt", comments="#", delimiter=",", unpack=False)
```

Step2: Map Task Execution

After the Map tasks assignment, server k (k = 1, 2, 3) starts to execute each chapter assigned to it. We need to get the three partitions for each chapter. For example, if we deal with numbers in range [1 102], and we assign key range [1 34] to server 1, key range [35 68] to server 2 and key range [69 102] to server 3, i.e. In the end, server 1 will have sorted numbers in range 1 to 34, server 2 will have sorted numbers in range 35 to 68, server 3 will have sorted numbers in range 69 to 102:

```
chap3range1 = list()
chap3range2 = list()
chap3range3 = list()

for i in raw3:
   if i < FIRST_RANGE_UPPER:
      chap3range1.append(i)
      #print chap1range1
   if  FIRST_RANGE_UPPER<=i<SECOND_RANGE_UPPER:
      chap3range2.append(i)
   if i >= SECOND_RANGE_UPPER:
      chap3range3.append(i)

chap4range1 = list()
chap4range2 = list()
chap4range3 = list()

for i in raw4:
   if i < FIRST_RANGE_UPPER:
      chap4range1.append(i)
   if  FIRST_RANGE_UPPER<=i<SECOND_RANGE_UPPER:
      chap4range2.append(i)
```

```
    if i >= SECOND_RANGE_UPPER:
        chap4range3.append(i)
```

We need to get the numbers within our own key range and keep them to the server itself so that we will not exchange those numbers with the other two servers. Take server 2 as an example, server 2 takes care of range [35 68], so numbers within 35 to 68 will be kept to server 2 itself:

```
localList = list(itertools.chain(chap3range2, chap4range2))

#fors11:numbers for server 1 from chap3
#fors12:numbers for server 1 from chap4
fors11 = chap3range1
fors12 = chap4range1
fors31 = chap3range3
fors32 = chap4range3
```

Then we create four files, each of which stores the data in each key range for each server from each chapter. We upload those four files to dropbox:

```
#fors1froms2c1.txt: stores the data within range1 for server 1 from chapter 3,
etc.
f = open("fors1froms2c1.txt", "w")
f.write("\n".join(map(lambda x: str(x), fors11)))
f.close()

f = open("fors1froms2c2.txt", "w")
f.write("\n".join(map(lambda x: str(x), fors12)))
f.close()

f = open("fors3froms2c1.txt", "w")
f.write("\n".join(map(lambda x: str(x), fors31)))
f.close()

f = open("fors3froms2c2.txt", "w")
f.write("\n".join(map(lambda x: str(x), fors32)))
f.close()

#upload the four files to dropbox
subprocess.call(['./uploading21.sh'])
subprocess.call(['./uploading22.sh'])
subprocess.call(['./uploading212.sh'])
subprocess.call(['./uploading222.sh'])
```

The following is the decoding part. In our implementation we use a waiting mechanism, which will wait for the other four files from the other two servers. As long as the server get its files, it will access those files through folder paths and convert those strings to float type through map function:

```
#we need to acccess the shared folder through folder path
file_path="/Users/jingli/Dropbox/MapReduce/fors2froms1c1.txt"
```

```
file_path1="/Users/jingli/Dropbox/MapReduce/fors2froms1c2.txt"
file_path2="/Users/jingli/Dropbox/MapReduce/fors2froms3c1.txt"
file_path3="/Users/jingli/Dropbox/MapReduce/fors2froms3c2.txt"

#check if we have received all the needed files, if not, we will wait
while not os.path.exists(file_path) or not os.path.exists(file_path1) or not
os.path.exists(file_path2) or not os.path.exists(file_path3):
    time.sleep(1)
if os.path.isfile(file_path) and os.path.isfile(file_path1) and
os.path.isfile(file_path2) and os.path.isfile(file_path3):
   file1 = [line.strip() for line in
open("/Users/jingli/Dropbox/MapReduce/fors2froms1c1.txt", 'r')]
        file12 = [line.strip() for line in
open("/Users/jingli/Dropbox/MapReduce/fors2froms1c2.txt", 'r')]
   file3 = [line.strip() for line in
open("/Users/jingli/Dropbox/MapReduce/fors2froms3c1.txt", 'r')]
   file32 = [line.strip() for line in
open("/Users/jingli/Dropbox/MapReduce/fors2froms3c2.txt", 'r')]

    #convert numbers from string type to float type for calculating
   file1 = map(float, file1)
   file12 = map(float, file12)
   file3 = map(float, file3)
   file32 = map(float, file32)

   #concatenate all numbers in range2 together
   totalsort =
quickSort(list(itertools.chain(localList,file1,file12,file3,file32)))

else:
   print "error"
```

In the end, we check if the numbers are sorted successfully and if they are within their corresbonding ranges:

```
if all(90001<=totalsort[i]<=180000 for i in xrange(len(totalsort)-1)) == True:
   sys.stderr.write("CORRECT: Numbers are in range [90001 180000]\n")
else:
   sys.stderr.write("ERROR: Unexpected numbers occur!!\n")

if all(totalsort[i] <= totalsort[i+1] for i in xrange(len(totalsort)-1)) ==
True:
   sys.stderr.write("SUCCESS: numbers have been sorted successfully\n")
else:
   sys.stderr.write("ERROR: Sorting failed!!\n")
```

Step3: Reduce tasks

The reduce process is simple. We just need to concatenate all the numbers from different servers together, which we do not represent in our code.

B. Coded MapReduce

Step1: Map Task Assignment

In Code MapReduce, We take six chapters as an example and assign map tasks as follows: Server1 = {1, 2, 3, 4}, Server2 = {1, 2, 5, 6}, Server3 = {3, 4, 5, 6}. In this way we follow the rule of Coded MapReduce that each chapter is assigned to exactly two servers and every server share exactly two chapters. We take server2 as an example to explain the code:

```
raw1 = loadtxt("chap1.txt", comments="#", delimiter=",", unpack=False)
raw2 = loadtxt("chap2.txt", comments="#", delimiter=",", unpack=False)
raw5 = loadtxt("chap5.txt", comments="#", delimiter=",", unpack=False)
raw6 = loadtxt("chap6.txt", comments="#", delimiter=",", unpack=False)
```

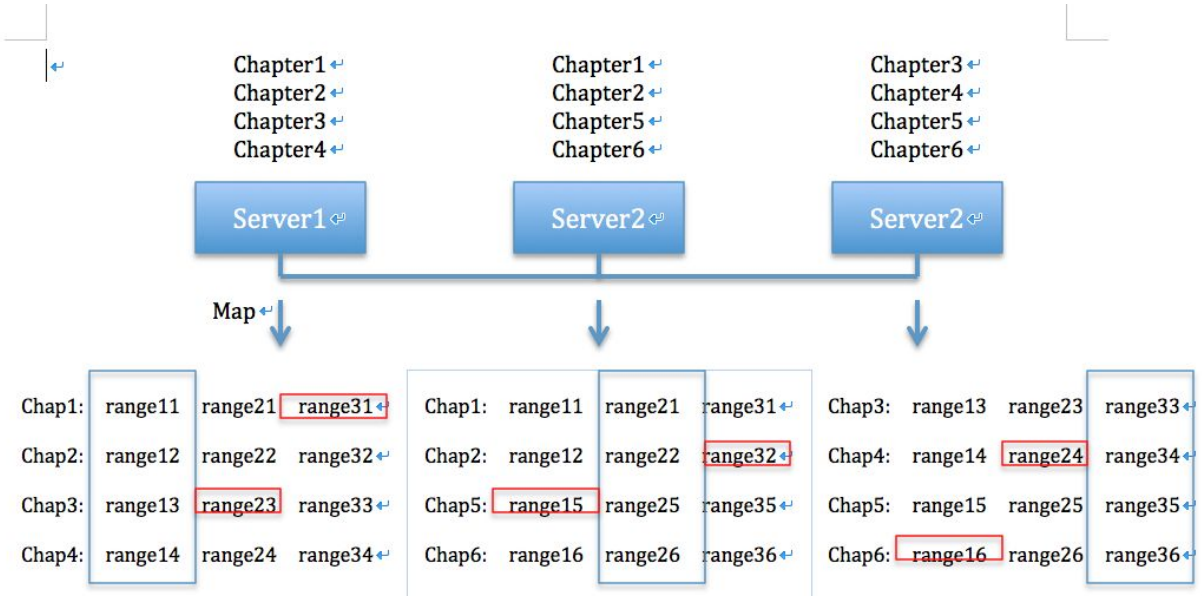
Step2: Map Task Execution

The following part is the same as original MapReduce. We input the data in their respective partitions and obtain the results.

```
chap1range1 = list()
chap1range2 = list()
chap1range3 = list()

for i in raw1:
    if i < FIRST_RANGE_UPPER:
        chap1range1.append(i)
        #print chap1range1
    if  FIRST_RANGE_UPPER<=i<SECOND_RANGE_UPPER:
        chap1range2.append(i)
    if i >= SECOND_RANGE_UPPER:
        chap1range3.append(i)

chap2range1 = list()
chap2range2 = list()
chap2range3 = list()

for i in raw2:
    if i < FIRST_RANGE_UPPER:
        chap2range1.append(i)
    if  FIRST_RANGE_UPPER<=i<SECOND_RANGE_UPPER:
        chap2range2.append(i)
    if i >= SECOND_RANGE_UPPER:
        chap2range3.append(i)

chap5range1 = list()
```

```
chap5range2 = list()
chap5range3 = list()
for i in raw5:
    if i < FIRST_RANGE_UPPER:
        chap5range1.append(i)
    if  FIRST_RANGE_UPPER<=i<SECOND_RANGE_UPPER:
        chap5range2.append(i)
    if i >= SECOND_RANGE_UPPER:
        chap5range3.append(i)
#print chap5range3
chap6range1 = list()
chap6range2 = list()
chap6range3 = list()

for i in raw6:
    if i < FIRST_RANGE_UPPER:
        chap6range1.append(i)
    if  FIRST_RANGE_UPPER<=i<SECOND_RANGE_UPPER:
        chap6range2.append(i)
    if i >= SECOND_RANGE_UPPER:
        chap6range3.append(i)

#concatemate all the numbers that will not be exchanged
localList = list(itertools.chain(chap1range2, chap2range2, chap5range2,
chap6range2))
```

Then we generate coded pairs. We present this process in the following figure:



The numbers in blue boxes are those need to be kept to servers themselves. The numbers in red boxes are numbers need to be exchanged with other servers, i.e. numbers need to do XOR to generate coded pairs. For example, if range31 = (68, 69, 70, 71), range23 = (42, 43, 44, 45), we need to XOR numbers

correspondingly so we get four coded pairs in total: 68 XOR 42, 69 XOR 43, 70 XOR 44, 71 XOR 45. In our case, we generate 4 * 3 = 12 coded pairs.

```
#chap2range3: numbers from chap2 in range3
chap2range3 = chap2range3
chap5range1 = chap5range1

#Using map function to add numbers corresbondingly. Since we are using dropbox
app not #an router, we do not use XOR here.
pair = map(sum, zip(chap2range3, chap5range1))
```

The follwing part is the same as original MapReduce code, we store the data in one file, upload the file and wait for other coded pairs from server 2 and server 3 through a waiting mechanism.

```
f = open("s2pair.txt", "w")
f.write("\n".join(map(lambda x: str(x), pair)))
f.close()

subprocess.call(['./uploading2.sh'])

file_path="/Users/jingli/Dropbox/MapReduce/s1pair.txt"
file_path1="/Users/jingli/Dropbox/MapReduce/s3pair.txt"

while not os.path.exists(file_path) or not os.path.exists(file_path1):
    time.sleep(1)
```

Upon received all the coded pairs from the other two servers, we start the decoding process:

if os.path.isfile(file_path):

#decode

```
  file1 = [line.strip() for line in
open("/Users/jingli/Dropbox/MapReduce/s1pair.txt", 'r')]
    file3 = [line.strip() for line in
open("/Users/jingli/Dropbox/MapReduce/s3pair.txt", 'r')]
    file1 = map(float, file1)
    file3 = map(float, file3)

   #chap1range3: the corresbonding numbers servers themselves have for
decoding.
    chap1range3 = map(lambda x: -1*x, chap1range3)
    chap6range1 = map(lambda x: -1*x, chap6range1)

   #froms1: numbers server 2 need from server 1
    froms1 = map(sum, zip(file1, chap1range3))
    froms3 = map(sum, zip(file3, chap6range1))

    totalsort = quickSort(list(itertools.chain(localList, froms1, froms3)))
else:
    raise ValueError("%s isn't a file!" % file_path)
```

Then we check the numbers if they are in order and within the desinated range. For server 2 it is range2. Then we calculate the total time we use in the whole process.

```
if all(FIRST_RANGE_UPPER<=totalsort[i]<=SECOND_RANGE_UPPER for i in
xrange(len(totalsort)-1)) == True:
   sys.stderr.write("CORRECT: Numbers are in range [90001 180000]\n")
else:
   sys.stderr.write("ERROR: Unexpected numbers occur!!\n")

if all(totalsort[i] <= totalsort[i+1] for i in xrange(len(totalsort)-1)) ==
True:
   sys.stderr.write("SUCCESS: numbers have been sorted successfully\n")
else:
   sys.stderr.write("ERROR: Sorting failed!!\n")

stop = time.asctime(time.localtime(time.time()))
stop1 = time.time()
print stop
print "End CodedMapReduce"
x = stop1 - start1
print "Total Computation Time using Coded MapReduce"
print x
```

Step3:Reduce tasks

We just need to concatenate all the ranges together.

III. Results & Performance Analysis

We test three different data sizes, 12MB, 67.8MB, and 271.2MB, respectly. The following figures show the results we get:

Table is interpreted as,

Trial No- corresponds to the experiment number

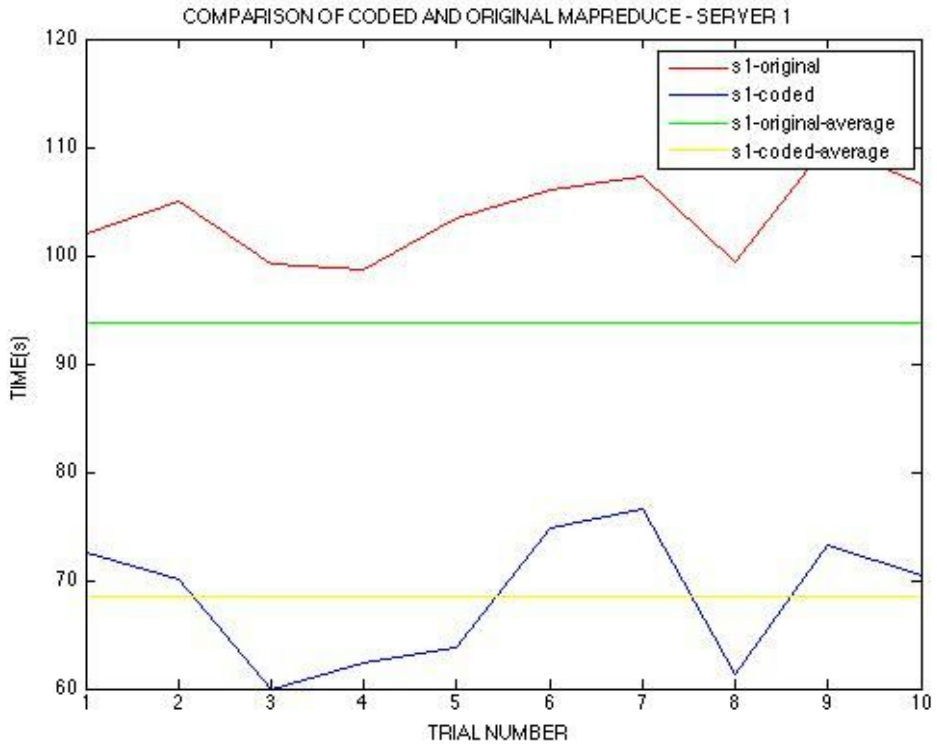
S1,S2,S3-Server 1, Server 2 and Server 3 respectively

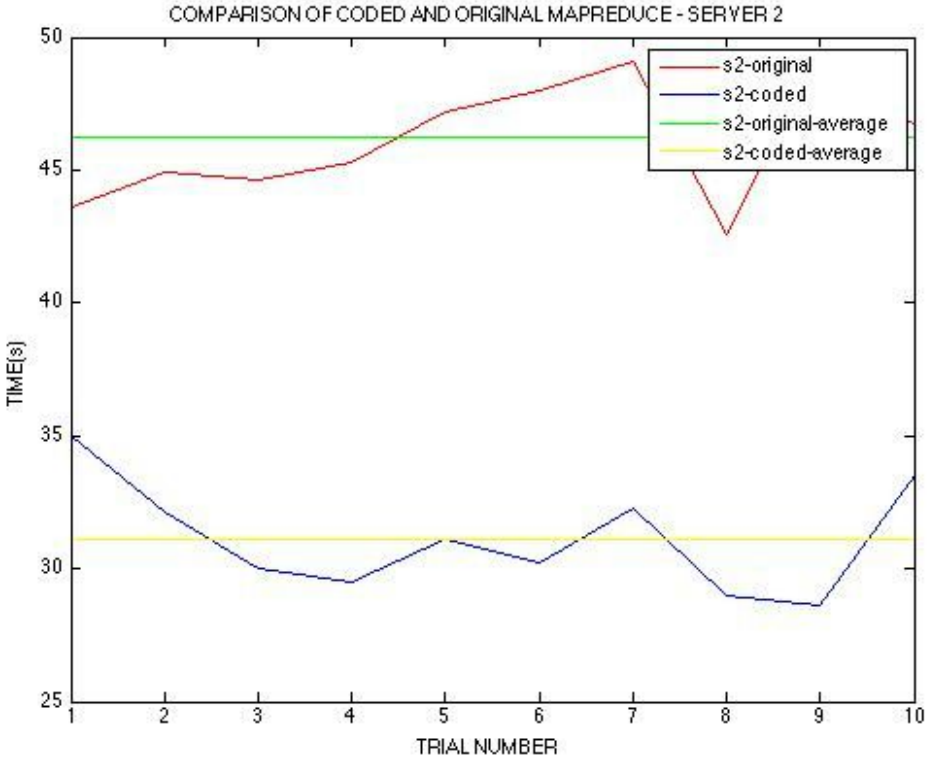
ORIGINAL- Original MapReduce

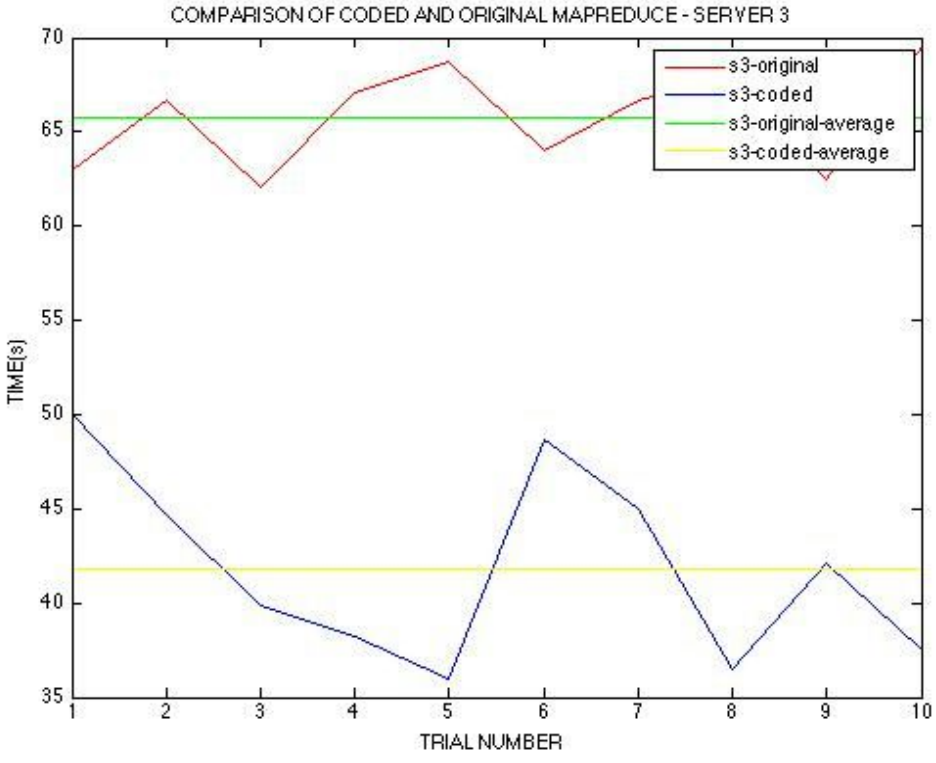
CODED- Coded MapReduce

DATA SIZE: 12 MB

TRIAL NO.	S1-ORIGINAL	S1-CODED	S2-ORIGINAL	S2-CODED	S3-ORIGINAL	S3-CODED
1	102.0	72.5	43.6	35.0	63.0	50.0
2	105.0	70.1	44.9	32.1	66.7	44.7
3	99.2	60.0	44.6	30.0	62.1	39.9
4	98.6	62.4	45.3	29.5	67.1	38.2
5	103.4	63.7	47.2	31.1	68.7	36
6	106	74.8	48	30.2	64	48.7
7	107.2	76.5	49.1	32.3	66.7	45
8	99.4	61.4	42.6	29	68	36.5
9	110.2	73.2	49.5	28.6	62.5	42.1
10	106.5	70.4	46.7	33.5	69.5	37.5
AVERAGE	93.8	68.5	46.2	31.1	65.8	41.8



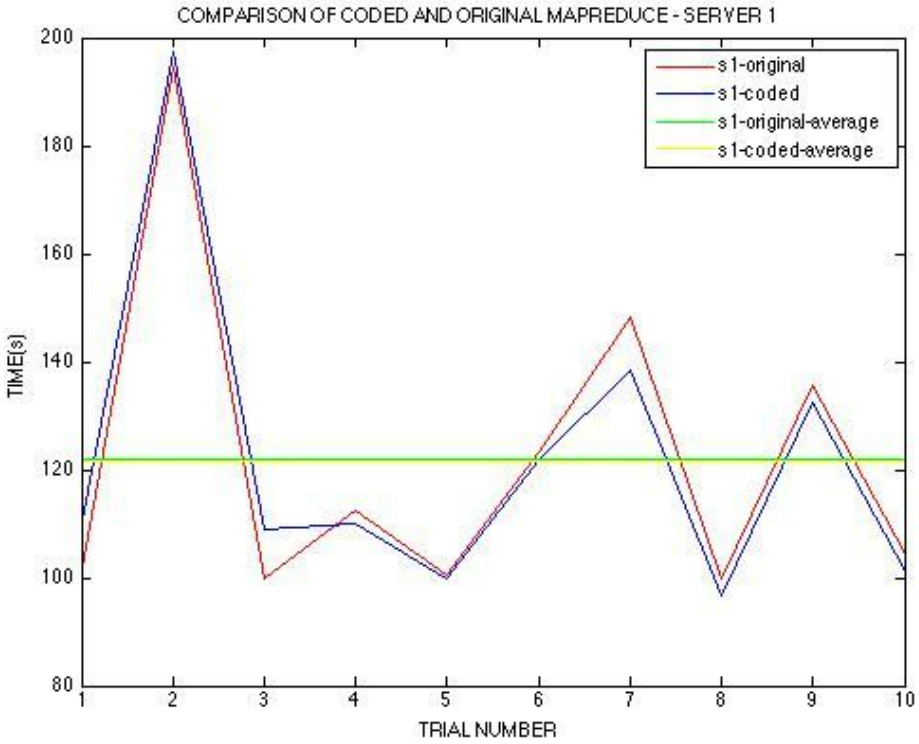

COMPARISON OF CODED AND ORIGINAL MAPREDUCE - SERVER 2
s2-original
s2-coded
s2-original-average
s2-coded-average
TIME(s)
TRIAL NUMBER
50
45
40
35
30
25
1
2
3
4
5
6
7
8
9
10

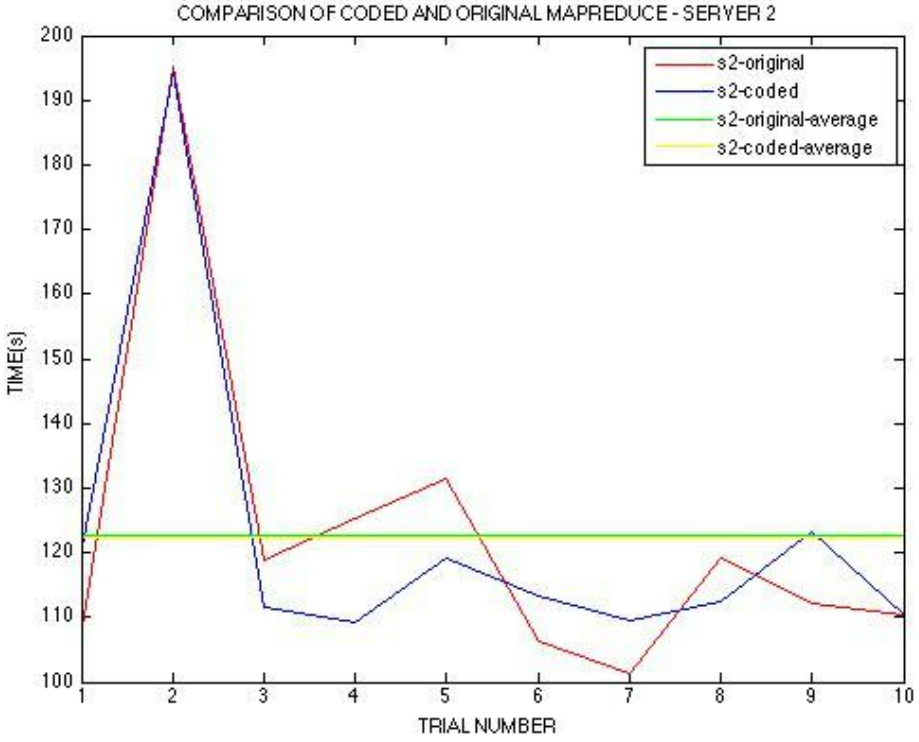

COMPARISON OF CODED AND ORIGINAL MAPREDUCE - SERVER 3
s3-original
s3-coded
s3-original-average
s3-coded-average
TIME(s)
TRIAL NUMBER
70
65
60
55
50
45
40
35
1
2
3
4
5
6
7
8
9
10

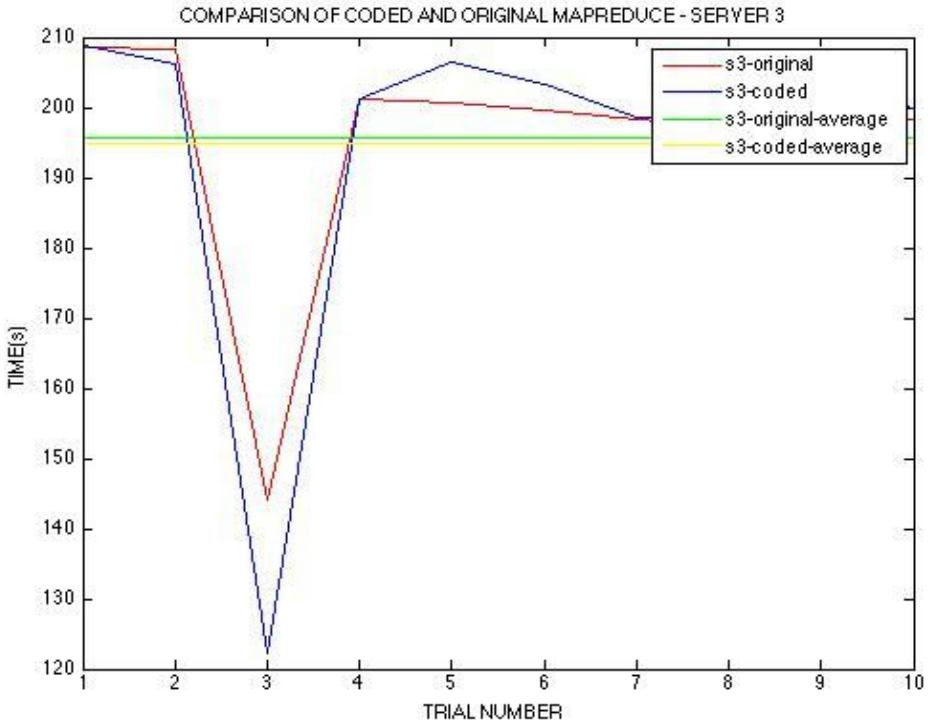
DATA SIZE: 67.8 MB

Time: in seconds

TRIAL NO.	S1-ORIGINAL	S1-CODED	S2-ORIGINAL	S2-CODED	S3-ORIGINAL	S3-CODED
1	101.0	109.8	107.5	120.0	208.6	208.9
2	194.4	197.4	195.1	194.4	208.3	206.1
3	99.8	109.0	118.9	111.6	144.2	122.3
4	112.4	110.1	125.4	109.2	201.2	201.2
5	100.4	99.8	131.5	119.2	200.7	206.5
6	123.4	122.0	106.3	113.2	199.5	203.2
7	148.3	138.5	101.2	109.6	198.2	198.6
8	99.9	96.5	119.2	112.4	197.3	199.4
9	135.6	132.3	112.2	123.3	200.1	202.3
10	104.4	101.2	110.5	110.4	198.3	200.0
AVERAGE	122.0	121.5	122.8	122.3	195.6	195.0



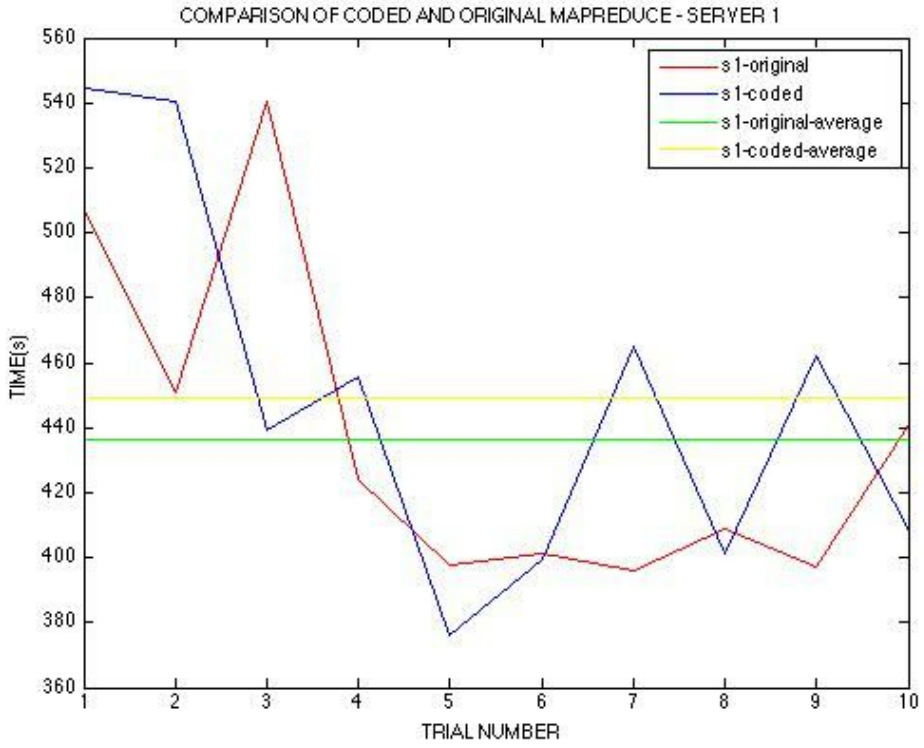

COMPARISON OF CODED AND ORIGINAL MAPREDUCE - SERVER 2
s2-original
s2-coded
s2-original-average
s2-coded-average
TIME(s)
TRIAL NUMBER
200
190
180
170
160
150
140
130
120
110
100
1
2
3
4
5
6
7
8
9
10

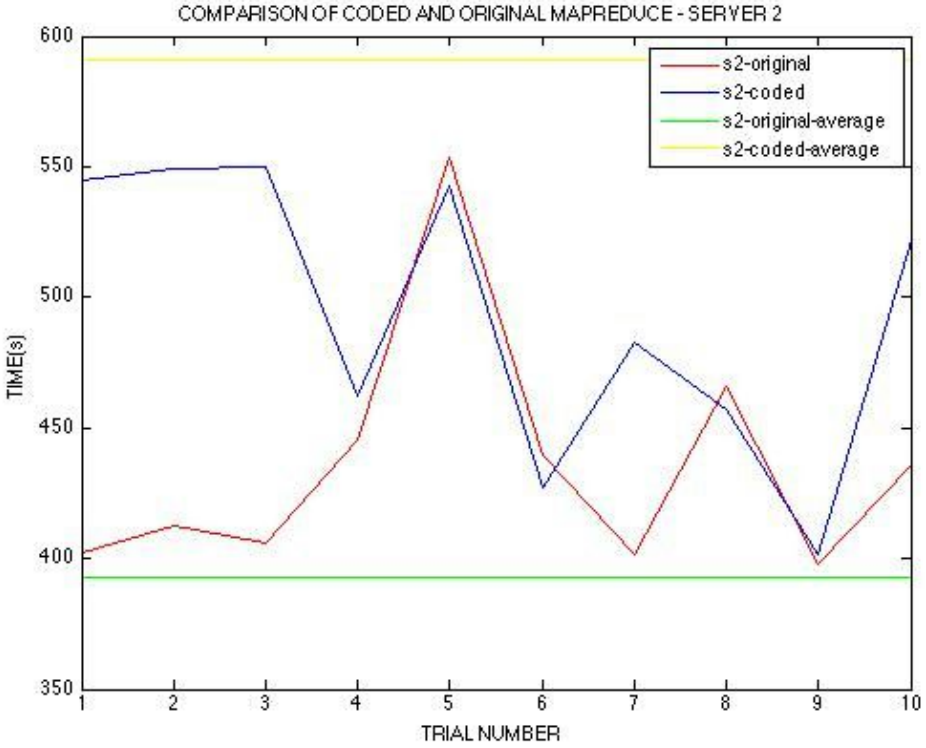

COMPARISON OF CODED AND ORIGINAL MAPREDUCE - SERVER 3
s3-original
s3-coded
s3-original-average
s3-coded-average
TIME(s)
TRIAL NUMBER
210
200
190
180
170
160
150
140
130
120
1
2
3
4
5
6
7
8
9
10

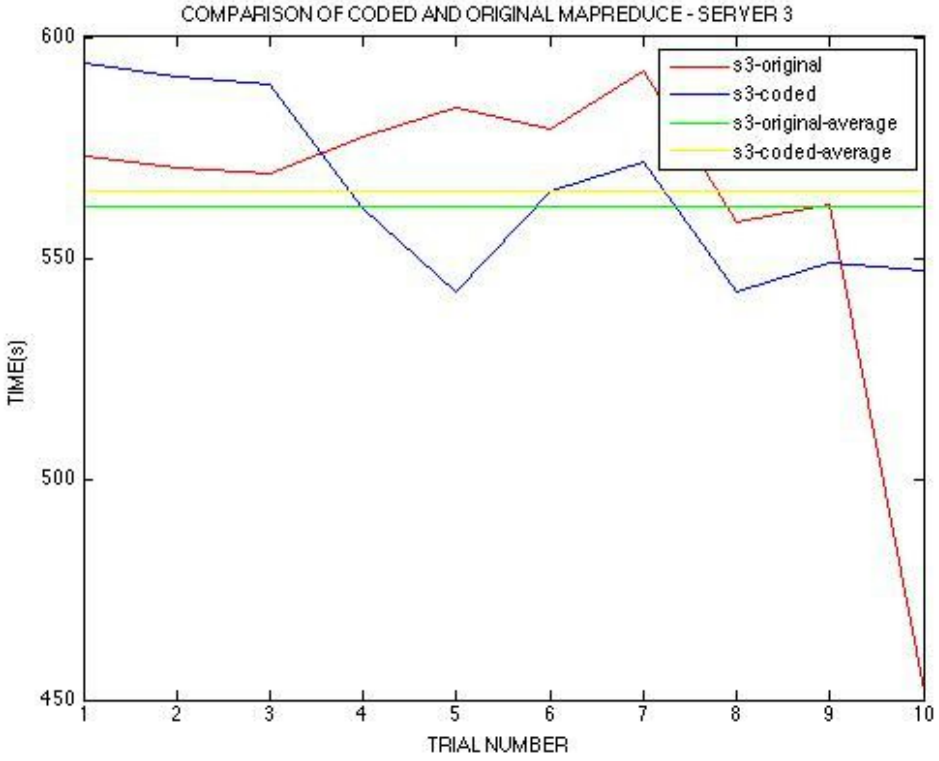
DATA SIZE: 271.2MB

Time: in seconds

TRIAL NO.	S1-ORIGINAL	S1-CODED	S2-ORIGINAL	S2-CODED	S3-ORIGINAL	S3-CODED
1	507.8	544.3	544.8	402.3	573.0	594.0
2	450.7	540.2	549.3	412.2	570.3	591.1
3	540.3	439.5	550.2	406.1	569.2	589.3
4	424.3	455.7	462.1	445.1	577.3	561.2
5	397.5	376.1	542.7	553.6	584.1	542.2
6	401.5	399.7	426.8	439.5	579.3	565.3
7	395.7	465.2	482.7	401.6	592.5	571.7
8	409.1	401.5	457.4	466.1	558.0	542.3
9	397.2	462.1	401.8	398.2	562.1	549.0
10	441.2	408.4	522.4	436.0	452.1	547.0
AVERAGE	436.5	449.2	591	392.5	561.8	565.3




COMPARISON OF CODED AND ORIGINAL MAPREDUCE - SERVER 2
s2-original
s2-coded
s2-original-average
s2-coded-average
TIME(s)
TRIAL NUMBER
600
550
500
450
400
350
1
2
3
4
5
6
7
8
9
10


COMPARISON OF CODED AND ORIGINAL MAPREDUCE - SERVER 3
s3-original
s3-coded
s3-original-average
s3-coded-average
TIME(s)
TRIAL NUMBER
600
550
500
450
1
2
3
4
5
6
7
8
9
10

From the above graphs and results, we can observe that the designed application gives excellent results for lower data size which is reflected in the total time taken for sorting of data. The results show considerable improvement in the computation as well as in the communication load between the servers. The number of files to be communicated between the 3 servers by each server is just 2 for coded mapreduce as compared to 4 files to be transmitted for the conventional mapreduce. This gives a 50% reduction in the communication load between the servers using the coded mapreduce method described above. Considering the total time taken for communication and computation of the data sets is twice in the conventional mapreduce when compared to the coded mapreduce.

But, as seen from the results above, as the data size increases the computation load does not show considerable .difference but the communication load is still reduced to 50%.

REFERENCES

[1] http://ee.caltech.edu
[2] https://en.wikipedia.org/wiki/MapReduce
[3] http://www-bcf.usc.edu/~avestime/papers/CMR_slides.pdf
[4] https://www.mapr.com/sites/default/files/terasort-comparison-yarn.pdf
[5] https://hadoop.apache.org
[6] Songze Li, Mohammad Ali Maddah-Ali, A. Salman Avestimehr,"Coded Mapreduce",Distributed, Parallel,and cluster computing.