A project Report On

Simulation of Distributed MapReduce Algorithm In Big Data Analytics

Submitted in partial fulfilment for the award of the degree of

B.tech Computer Science and Engineering

By

Sanjit Kumar
18BCE0715

Under the guidance of
Dr. Narayanamoorthi M

CSE4001
Parallel and Distributed Computing
J Component

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering,
June 2021

Contents

Abstract

The MapReduce algorithm first introduced by google was a ground breaking advancement in the field of big data processing. It allowed for the efficient use of multiple workstations or servers in a cluster, for the huge volumes of data. In this project, I aim to simulate the working of MapReduce algorithm in a distributed system. The idea, is to use docker containers for multiple instantiations of the Map and Reduce nodes and therefore create a virtual environment that can simulate the a distributed system.

Introduction

A domain that has shown tremendous growth in recent times is the field of Big Data. Data is just information of any kind that is stored or moved from one computer to another. Big data is just data that's enormous in size but still grows exponentially in real time. The data from a road traffic monitoring system for example gets huge volumes of data every second. To process such data in a single machine is impossible. Thats why one or several clusters are used. Almost always in cases such as these the data is growing exponentially. The New York stock exchange's millions of stock value changes, social media, air transport monitoring and payment systems are just a few of the most widely known big data systems.

Parallel processing of data has existed for decades now but the capacity and efficiency of systems have always exponentially increased over the years. But the volume of data that had to be processed is also tremendous. So Google came up with a smart way to distribute and recollect the computational task into sub-tasks (as MapReduce jobs) in a cluster with high computational power. This was released as a white paper by google's research and development team. It soon became popular and was adopted to be a widespread framework in the form of Apache Hadoop.

Such frameworks, like Apache Hadoop in Java, provide a layer of abstraction over the distribution and management of multiple devices or threads. The complexity is masked to the programmer. So in this project, I aim to build from scratch the MapReduce algorithm by implementing it in Python and simulating the distributed environment with the help of Docker. Docker is used for the creation of multiple containers (or nodes) that act like individual nodes in a network (since they are completely isolated and independent from the host) - they are layered on top of a lightweight linux distribution called Alpine.

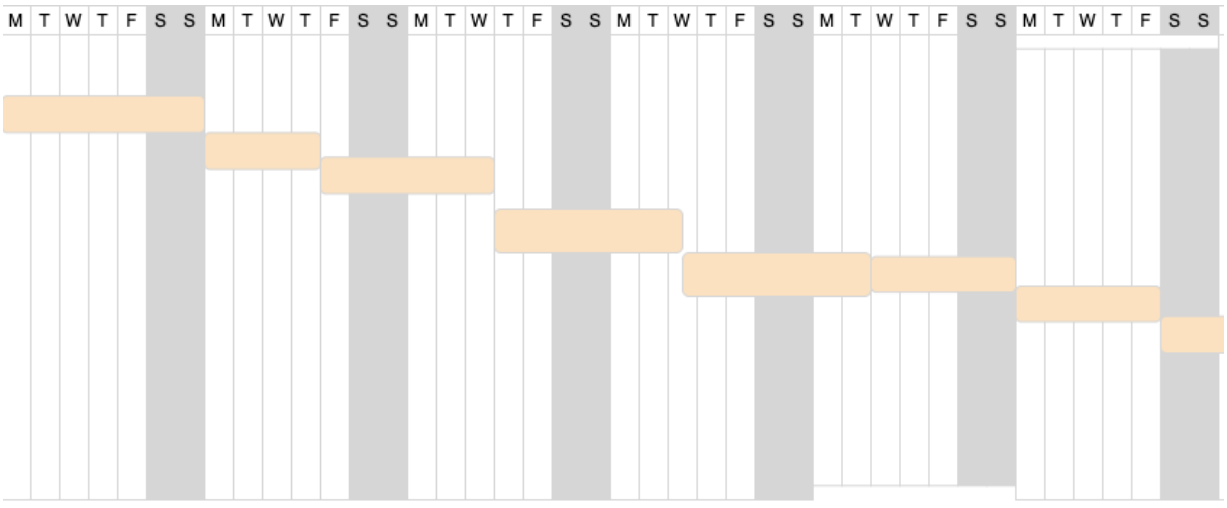
This is a very widely adopted strategy and in this project I simply try to simulate the whole process in a single local machine. There has been renewed interest in this area as there are variations of this algorithm coming out as new research articles. The project could be improved by incorporating such new algorithms that improve efficiency. The key highlight of the project would be that the whole algorithm had been build from scratch with no frameworks but just parallel instantiation of docker containers.

Problem Statement

The process of querying data from a distributed file system. A range of queries may be done based on the wide spectrum of MapReduce algorithms that are available for making data selections. Using large scale graph analysis in conjunction with the above needs an efficient solution to do mapping and reducing Jobs in parallel. This is solved by using Map Reduce and highly powerful distributed systems.

Gantt Chart

#	Task Name	Duration	Start	ETA	Dependence
Complete project execution					
1	Problem Statement and Overall Solution	7 days	1-04-21	7-04-21	None
2	Docker Commands and Context	4 days	08-04-21	13-04-21	None
4	Map Phase Modules	5 days	13-04-21	18-04-21	1
5	Combine Phase Modules	8 days	18-04-21	26-04-21	1
6	Reduce Phase Moduels	5 days	27-04-21	03-05-21	3
7	Bash Unix Shell Scripting for Orchastration	5 days	03-05-21	08-05-21	4
8	Docker Container Communication	7 days	15-05-21	21-05-21	5,6
9	Testing and Debugging	4 days	24-05-21	28-05-21	6
10	Version Control and Deployment	3	28-05-21	01-06-21	2



Literature Review

Big data analytics is a huge domain and MapReduce since its inception in 2004 has been accepted as the standard programming model for processing large amount of data in a distributed cluster. It helps distributed and redistribute work against all the nodes in the system. Venkata Swamy et al. in [1] talk about the need for improved algorithm that addresses load balancing issues. The paper proposes a new algorithm called h-mapreduce to do the same. In addition to the exponential performance gains, our investigations also found a negative effect of deploying h-MapReduce due to an inappropriate definition of heavy tasks, which provides us a guideline for an effective application of h-MapReduce.

The MapReduce framework is being increasingly used in the scientific computing and image/video processing fields. Relevant research has tailored it for the field's specificities but there are still overwhelming limitations when it comes to temporal locality-sensitive computations. The performance of this class of computations is closely tied to an efficient use of the memory hierarchy, concern that is not yet taken into consideration by the existing distributed MapReduce runtimes. Consequently, implementing temporal locality-sensitive computations, such as stencil algorithms, on top of MapReduce is a complex chore not rewarded with proportional dividends.

The paper by Daniel Magro et al. [2] tackles both the complexity and the performance issues by integrating tiling techniques and memory hierarchy information into MapReduce's split stage. We prototyped our proposal atop the Apache Hadoop framework, and applied it to the context of stencil computations. Our experimental results reveal that, for a typical stencil computation, our prototype clearly outperforms Hadoop MapReduce, specially as the computation scales.

Big data processing is one of the hot scientific issues in the current social development. MapReduce is an important foundation for big data processing. In this paper, we propose a semantic++ MapReduce. The paper deals with (1) Semantic++ extraction and management for big data, (2) SMRPL (Semantic++ MapReduce Programming Language), (3) Semantic++ MapReduce compilation methods,4) Semantic++ MapReduce computing technology. It includes three parts.

Renting a set of virtual private servers (VPSs for short) from a VPS provider to establish a virtual MapReduce cluster is cost-efficient for a company/organization. To shorten job turnaround time and keep data locality as high as possible in this type of environment, the paper by Jia-Chun Lin et al. in [4] proposes a Best-Fit Task Scheduling scheme (BFTS for short) from a tenant's perspective.

Hadoop cluster is widely used for executing and analyzing a large data like big data. It has MapReduce engine for distributing data to each node in cluster. Compression is a benefit way of Hadoop cluster because it not only can increase space of storage but also improve performance to compute job. Recently, there are some popular Hadoop's compression codecs for example; deflate, gzip, bzip2 and snappy. The paper by Kritwara et al. [5] talks about a detailed study of the same using a word count example.

Proposed Model

Methodology Proposed

In this project I attempt to incorporate parallel and distributed computing in simulating the big data landscape for a large amount of textual data. The approach towards this is via the famous MapReduce Algorithm that will be programmed from scratch in the Haskell programming language.

The approach will entail as modules as described in the block diagram. The input module will obtain the data (here it is a large textual data) and the cluster size of the worker nodes (will be discussed later). The data is preprocessed and segmented in this input module.

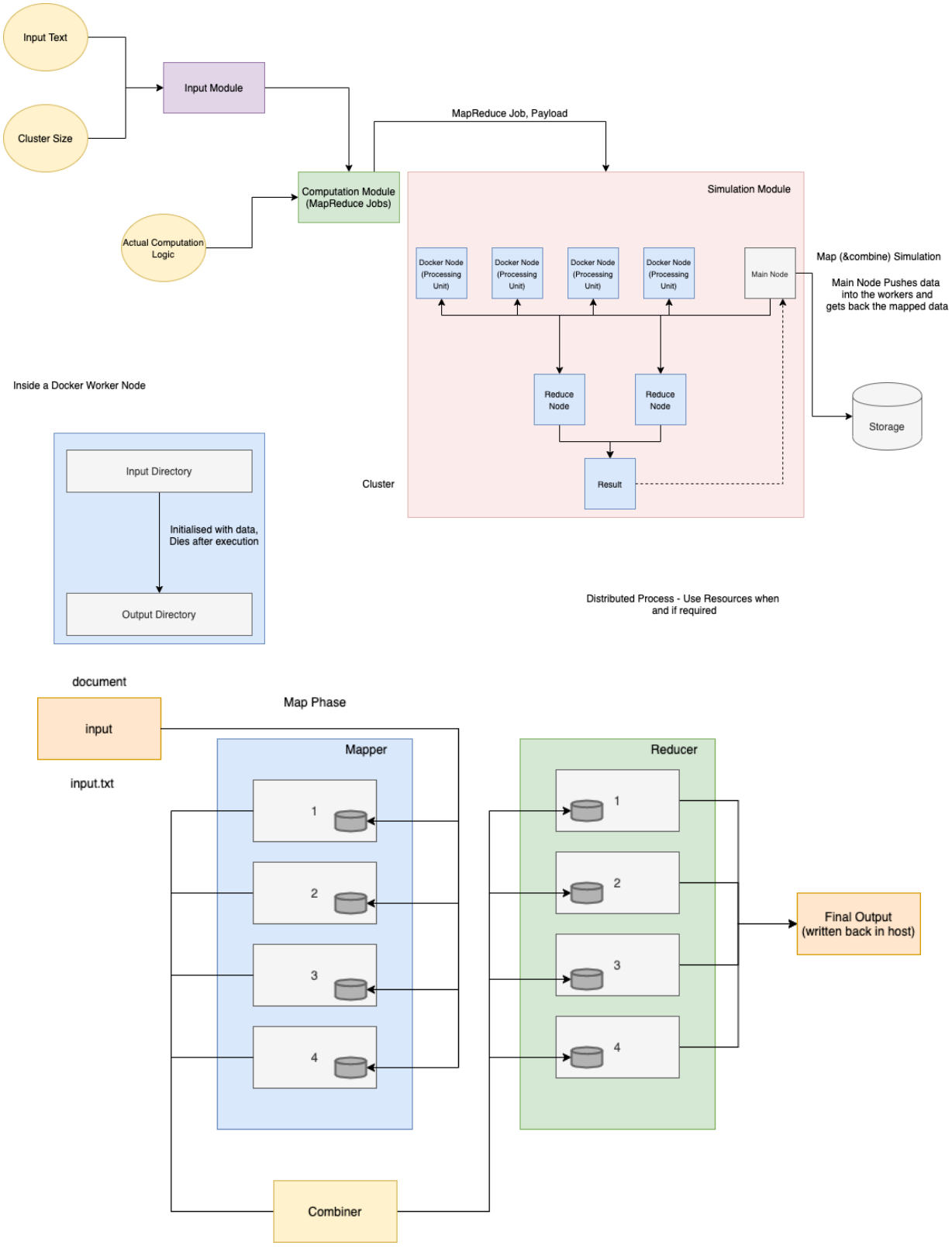
The preprocessed data will be sent to the computation module. This is where the computational logic for the Map and Reduce Jobs are written. The map job in our example is finding word frequency, and its logic is written into the map and reduce functions here. Here it is worth mentioning that these are very similar to how it is done in Java with the Apache Hadoop Framework.

The jobs are then passed to the simulation module along with the text data. Inside here a distributed system is simulated in a single local machine using docker nodes. Each of these docker instances are a worker node that are synonymous to an individual system. Here there is a main node that receives the data and maps them to the other slave nodes. These perform the computation as in the map jobs and then combine and shuffle them out to 2 other systems (also docker instances). The mapped words are them computed for number of occurrences in the reduce phase and then sent back to the main node.

This data is then stored in a permanent storage such as a database like MongoDB or ElasticSearch.

So the overall objective of this project is to create a simulated distributed system which computes the number of occurrences of each word in a given large-sized textual data.

Architecture Block Diagram



Algorithm

The MapReduce Algorithm is a computational model for parallel and distributed computing used in big data and cloud computing on a cluster. The idea is to use multiple machines in the distributed system that all contain portions of the data that has to go though a certain computation. For this purpose the MapReduce algorithm has 2 major phases - The Map Phase and The Reduce Phase.

Splitting and Map Phase

Map Function is the first step in MapReduce Algorithm. Map phase will work on key & value pairs input. It takes input tasks and divides them into smaller sub-tasks and then perform required computation on each sub-task in parallel.
In the map phase, key & value is in the form of byte offset values. A list of data elements is provided to mapper function called map(). Map() transforms input data to an intermediate output data element.
Mapper output will be displayed in the form of (K,V) pairs. Map phase performs the following two sub-steps -

- **Splitting** - Takes input dataset from Source and divide into smaller sub-datasets.
- **Mapping** - Takes the smaller sub-datasets as an input and perform required action or computation on each sub-dataset.

The output of the Map Function is a set of key and value pairs as <Key, Value>.

Shuffle & Sort Phase

This is the second step in MapReduce Algorithm. Shuffle Function is also known as “Combine Function”. Mapper output will be taken as input to sort & shuffle.
The shuffling is the grouping of the data from various nodes based on the key. This is a logical phase. Sort is used to list the shuffled inputs in sorted order.
It performs the following two sub-steps -

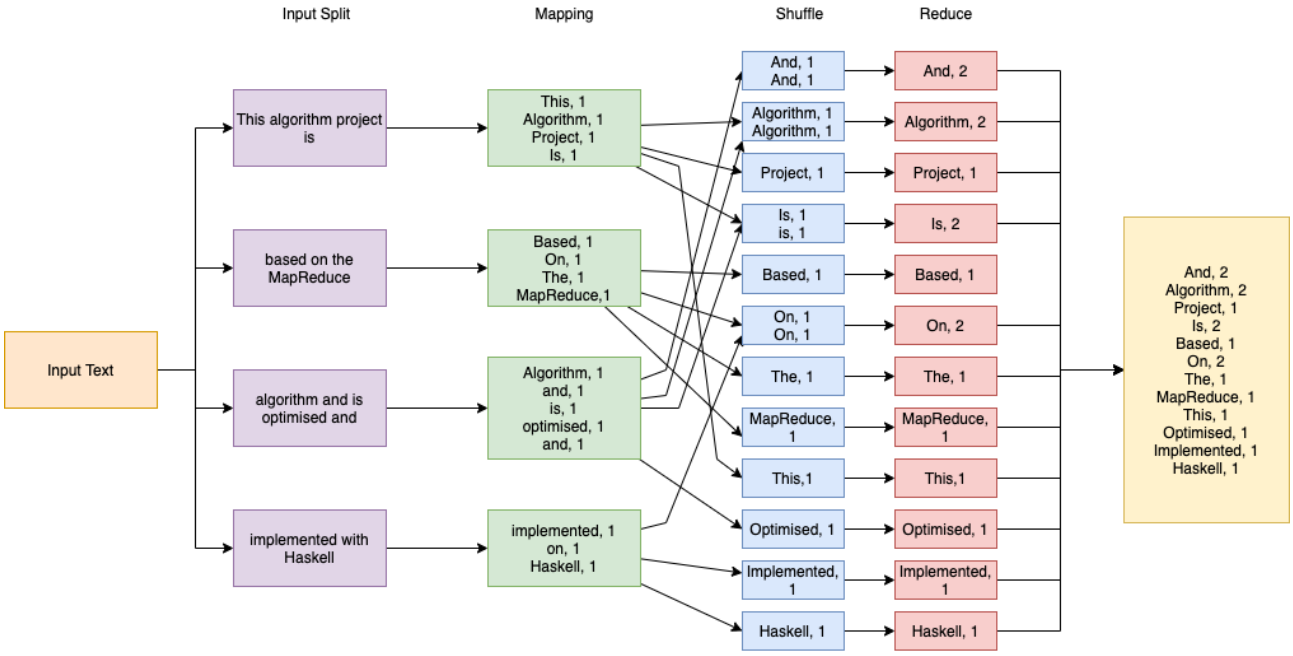
- **Merging** - combines all key-value pairs which have same keys and returns <Key, List<Value>>.
- **Sorting** - takes output from Merging step and sort all key-value pairs by using Keys. This step also returns <Key, List<Value>> output but with sorted key-value pairs.

It takes a list of outputs coming from “Map Function” and perform these two sub-steps on each and every key-value pair. Finally, Shuffle Function returns a list of <Key, List<Value>> sorted pairs to Reducer phase.

Reduce Phase

Reduce phase is the final step in MapReduce Algorithm. Reduce is inherently sequential unless processing multiple tasks. It takes list of <Key, List<Value>> sorted pairs from Shuffle Function and perform reduce operation.
Reduce function receives an iterator values from an output list for the specific key. Reducer combines all these values together and provide single output value for the specific key. This phase performs only one step - Reduce step.
Reduce step <Key, Value> pairs are different from map step <Key, Value> pairs. Reduce step <Key, Value> pairs are computed and sorted pairs. After completion of the Reduce Phase, the cluster collects the data to form an appropriate result and sends it back to the Hadoop server.

Flow Chart

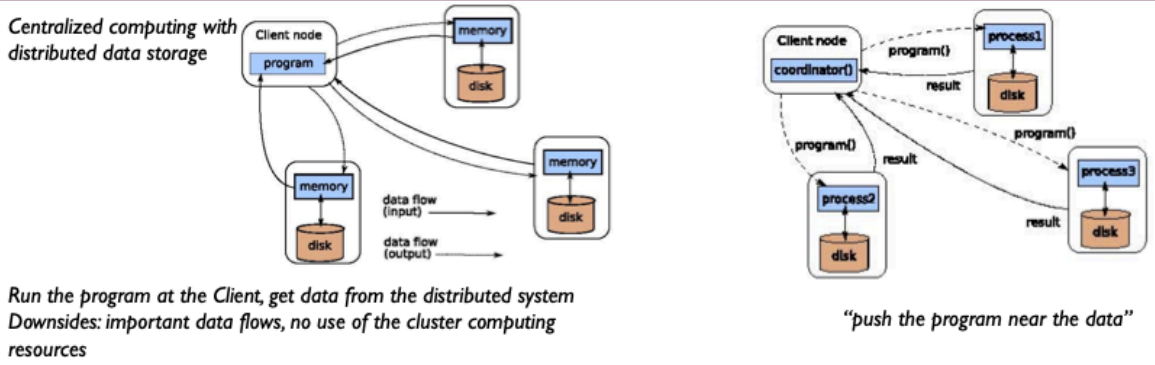


The above example shows the flow of text data from the input module through the different phases of the MapReduce algorithm. Towards the end it reaches the final calculation section that's reduce section, after which the result is given back to the host.

Working Principles and Model

Instead of moving large amounts of data around, it is far more efficient, if possible, to move the code to the data

The complex task of managing storage in such a processing environment is typically handled by a distributed 6 file system that sits underneath MapReduce

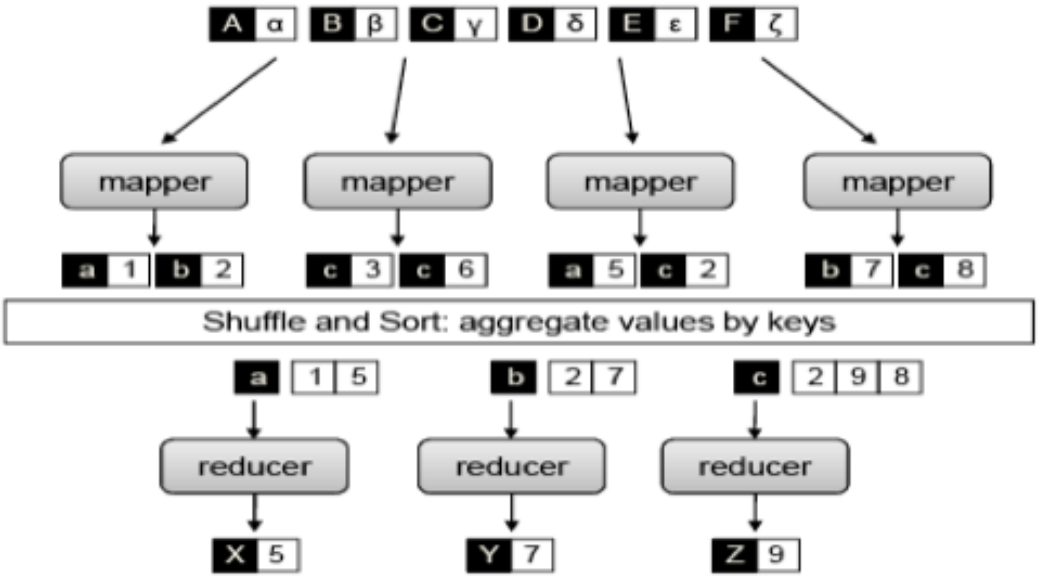


Stage 1: Apply a user-specified computation over all input records in a dataset.

- These operations occur in parallel and yield intermediate output (key-value pairs)

Stage 2: Aggregate intermediate output by another user-specified computation

- Recursively applies a function on every pair of the list



Requirements Specification

Dataset

The text data is derived from the novel called ***The Tale Of Two Cities.*** The text data is extracted and written into a .txt plain text file.

Tech Stack

1. Python 3.7.7
2. Distributed Libraries
3. Unix Shell Scripting (MacOS Termnial)
4. Docker 20.10.5 (build 55c4c88)

Docker - Linux (Alpine/Ubuntu), Python Instances

The program was written and tested on MacOS BigSur 11.2.3

Hardware

The hardware will only need the host machine, which has the following specifications:
MacProcessor - 1.8 GHz Dual-Core Intel Core i5 - Quadcore CPU

The simulation can be done with any set up since the use of docker containers brings cross compatibility.

Code Snippets

The Full Implementation of the Project can be found in the Github repository: https://github.com/sanjitk7/MapReducePython

---

The Map Phase

```
#!/usr/bin/env python3
import sys
import datetime
begin_time = datetime.datetime.now()
output_no = sys.argv[1]
total_chunks = sys.argv[2]
print(output_no)
data_location = "./mapper_data/"
input_file_path = data_location+"split_"+str(output_no)
+"_"+str(total_chunks)+".txt"
# output_file_location = "./mapped_data/mapped_data_"
mapped_dict = {}
try:
    print("input file path in map.py of node
",output_no,input_file_path)
    with open(input_file_path, encoding = 'utf-8') as f:
        d = f.read()
        l = d.split(" ")
        for x in l:
                if x not in mapped_dict:
                    mapped_dict[x] = 1
                else:
                    mapped_dict[x] = mapped_dict[x] + 1
        with open(data_location+"mapped_"+str(output_no)
+"_"+str(total_chunks)+ ".txt","w", encoding = "utf-8") as
mapped_data_file:
            tup_view = mapped_dict.items()
            tup_list = list(tup_view)
            for y in tup_list:
                mapped_data_file.write(str(y)+"\n")
    end_time = datetime.datetime.now()
```

```
    print("COMPUTATION TIME: MAP NODE "+str(output_no)+" =
"+str(end_time-begin_time))

except IOError as e:
    print(IOError,e)
```

Combine Phase

```
#!/usr/bin/env python3
import sys
import os.path
import time
from ast import literal_eval as make_tuple

total_chunks = int(sys.argv[1])

mapped_files_path = "./mapper_data"

def file_len(fname):
    with open(fname) as f:
        for i, l in enumerate(f):
            pass
    return i + 1

def count_map_output_files():
    mapped_counter = 0
    for root, dirs, files in os.walk(mapped_files_path):
        for file in files:
            if file.startswith('mapped'):
                mapped_counter += 1
    return mapped_counter

def list_chunks(l, n):
    n = max(1, n)
    return [l[i:i+n] for i in range(0, len(l), n)]

def combine():
    combined_dict = {}
    for i in range(total_chunks):
```

```
        input_file_path = "./mapper_data/
mapped_"+str(i+1)+"_"+str(total_chunks) +".txt"
        with open(input_file_path, encoding = 'utf-8') as f:
            d = f.read()
            # read each line - tuple strings
            tuples_list = d.split("\n")
            count = 0
            for j in tuples_list:
                count+=1
                # parse the tuple
                try:
                    j_tuple = make_tuple(j)
                except:
                    pass
                if j_tuple[0] not in combined_dict:
                    # creates single item list in value
                    combined_dict[j_tuple[0]] = [j_tuple[1]]
                else:
                    combined_dict[j_tuple[0]].append(j_tuple[1])

    # creating a list of tuples to write to reduce nodes
    tup_view = combined_dict.items()
    tup_list = list(tup_view)
    tup_list_n = len(tup_list)
    tup_list_chunks = list_chunks(tup_list,tup_list_n//
total_chunks)
    for i in range(total_chunks):
        combined_output_path = "./mapper_data/combined_split_" +
str(i+1)+"_"+ str(total_chunks) + ".txt"
        tup_list_chunk = tup_list_chunks[i]
        with open(combined_output_path,"w",encoding="utf-8") as
ff:
            for y in tup_list_chunk:
                ff.write(str(y)+"\n")


flag = False
while not flag:
    print("*Current Status: Number of Mapped Nodes =
",str(count_map_output_files())+"*")
```

```
    if (count_map_output_files()==total_chunks):
        combine()
        break
    time.sleep(3)
```

Reduce Phase

```
#!/usr/bin/env python3
import sys
import datetime
from ast import literal_eval as make_tuple
begin_time = datetime.datetime.now()


output_no = sys.argv[1]
total_chunks = sys.argv[2]
print(output_no)
data_location = "./mapper_data/"
input_file_path = data_location+"combined_split_"+str(output_no)
+"_"+str(total_chunks)+".txt"
# output_file_location = "./mapped_data/mapped_data_"
reduced_dict = {}
try:
    print("input file path in reduce.py of node
",output_no,input_file_path)
    with open(input_file_path, encoding = 'utf-8') as f:
        d = f.read()
        # convert to tuple
        tuples_list = d.split("\n")
        count = 0
        for j in tuples_list:
            count+=1
            # parse the tuple
            try:
                j_tuple = make_tuple(j)
            except:
                pass
            # print(j_tuple,type(j_tuple))
```

```
            # reduce logic
            tuple_reduce_sum = sum(j_tuple[1]) # total the values
of the list
            reduced_dict[j_tuple[0]] = tuple_reduce_sum
        # print("reduced_dict: ",reduced_dict)
        with open(data_location+"reduced_"+str(output_no)
+"_"+str(total_chunks)+ ".txt","w", encoding = "utf-8") as
reduced_data_file:
            tup_view = reduced_dict.items()
            tup_list = list(tup_view)
            for y in tup_list:
                reduced_data_file.write(str(y)+"\n")
    end_time = datetime.datetime.now()
    print("COMPUTATION TIME: REDUCE NODE "+str(output_no)+" =
"+str(end_time-begin_time))

except IOError as e:
    print(IOError,e)
```

Docker Composing the Images (examples)

```
FROM python:3
WORKDIR /usr/src/app
RUN mkdir data
COPY map.py .
CMD [map.py]
ENTRYPOINT ["python3"]
```

```
FROM python:3
WORKDIR /usr/src/app
RUN mkdir data
COPY combine.py .
CMD [combine.py]
ENTRYPOINT ["python3"]
```

```
START_TIME=$(ruby -e 'puts (Time.now.to_f * 1000).to_i')

#!/bin/sh

echo "---------------------- INITIALISING MAPREDUCE SIMULATION
--------------------\n\n"

echo "ENTER THE NUMBER OF AVAILABLE NODES: "
read chunks

# RUN CHUNKING MODULE
echo "------------ INITIALISING CHUNKING SCRIPT OF INPUT TEXT
FILE ------------\n"

cd orchestration
python main.py $chunks
cd ..

echo "------------ TERMINATING CHUNKING SCRIPT OF INPUT TEXT FILE
------------\n"

# CREATE VOLUME FOR DATA USAGE
echo "------------ CREATE VOLUME FOR DATA COMMUNICATION
------------\n"

docker volume create mapper_data
echo "----- DETAILS OF CREATED VOLUME: -----"
# docker volume ls
docker volume inspect mapper_data


# MAP PHASE

echo "------------ INITIALISING MAP PHASE ------------\n"

# Build Map Script Image
cd map
# sudo docker build -t map .
```

```
cd ..

# Create and Run Containes in Parallel
for (( i=1; i<=$chunks; i++ ))
do
    temp_name="map_node_$i"
    split_str="split_${i}_${chunks}.txt"
    echo "Process Creation:\nProcess Name: $temp_name\nSplitting
and Mapping $split_str in Parallel\n"
    sh ./create_map_nodes.sh $split_str $temp_name $i $chunks &
done


# COMBINE PHASE

# Build Combine Script Image
cd combine
# sudo docker build -t combine .
cd ..


echo "------------ INTERMEDIATE COMBINE PHASE INITIATION IN
PARALLEL------------\n"
docker container run --entrypoint /bin/sh -itd --mount
source=mapper_data,destination=/usr/src/app/mapper_data --name
combine_node combine:latest
docker exec -it combine_node python3 combine.py $chunks

echo "------------ MAP AND COMBINE PHASE TERMINATION
------------\n"


# REDUCE PHASE
cd reduce
# sudo docker build -t reduce .
cd ..

echo "------------ REDUCE PHASE INITIATION ------------\n"
for (( i=1; i<=$chunks; i++ ))
do
```

```
    temp_name="reduce_node_$i"
    split_str="combined_split_${i}_${chunks}.txt"
    echo "Process Creation:\nProcess Name: $temp_name\nReducing
and Computing: $split_str in Parallel\n"
    sh ./create_reduce_nodes.sh $split_str $temp_name $i $chunks &
done

wait
echo "------------- REDUCE PHASE TERMINATION -------------\n"


# COPY RESULT FILES BACK TO HOST

echo "------------- GETTING RESULT BACK TO MAIN NODE -------------
\n"


SECONDS=0
REDUCED_FILES="$(docker exec -it reduce_node_1 find . -name
"reduced_*" -printf '.' | wc -m)"
while [ $REDUCED_FILES != $chunks ]
do
echo "$SECONDS"
sleep .5
REDUCED_FILES="$(docker exec -it reduce_node_1 find . -name
"reduced_*" -printf '.' | wc -m)"
echo "Result Files: $REDUCED_FILES and $chunks"
done


for (( i=1; i<=$chunks; i++ ))
do
reduced_str="reduced_${i}_${chunks}.txt"
echo $reduced_str
docker cp reduce_node_$i:/usr/src/app/mapper_data/$reduced_str ./
data/reduced
done
```

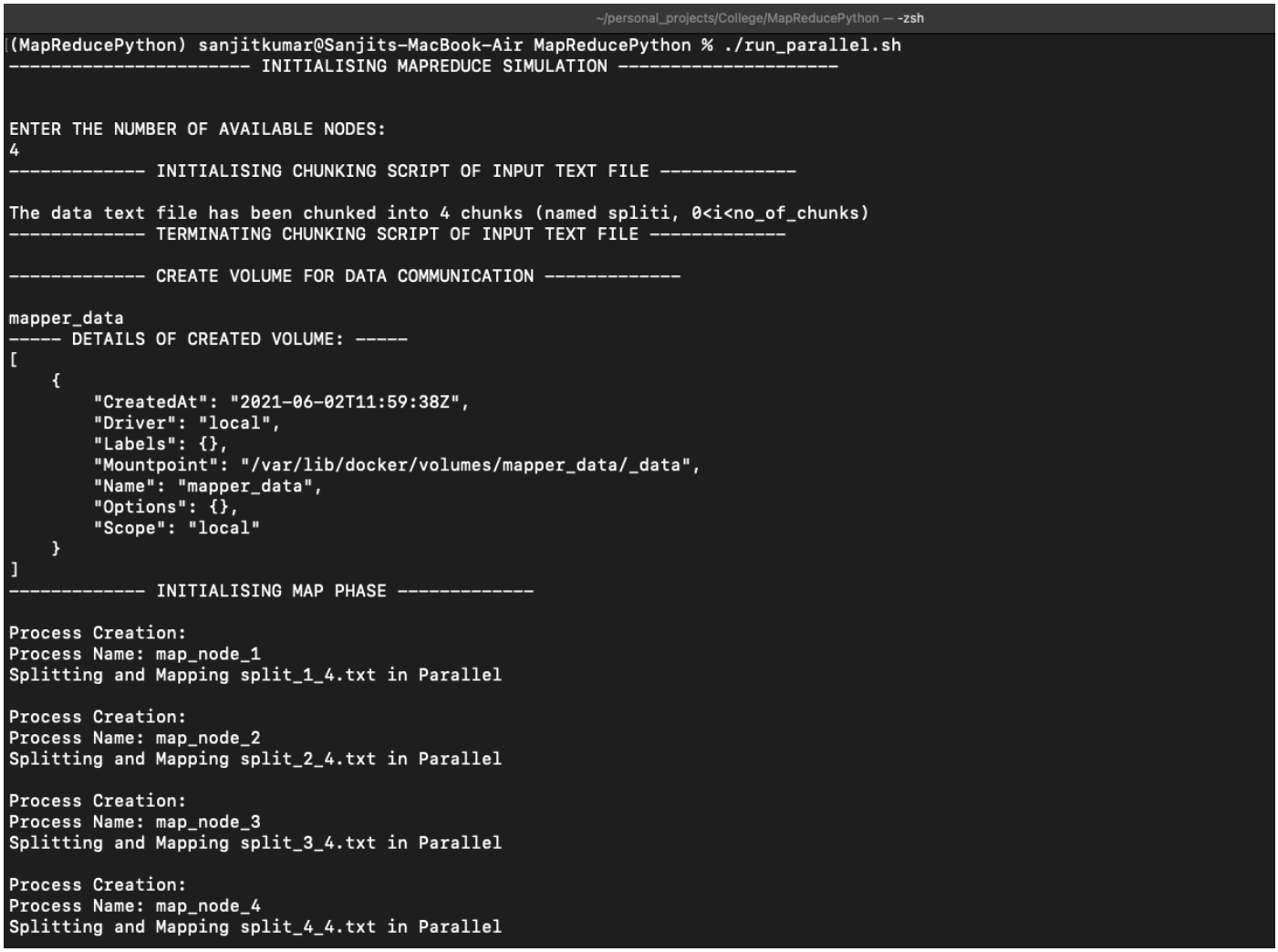
```
python orchestration/combine_files.py

echo "------------ RESULT OBTAINED AND WRITTEN INTO HOST DATA
FOLDER ------------\n"

echo "---------------------- TERMINATING MAPREDUCE SIMULATION
--------------------\n\n"

# COMPUTATION TIME CALC
END_TIME=$(ruby -e 'puts (Time.now.to_f * 1000).to_i')
COMP_TIME=$((END_TIME - START_TIME))
echo "****TOTAL COMPUTATION TIME OF THE ALGORITHM: $COMP_TIME ms
**** "

wait
```

Result Observation

The text data file exists as ***a-tale-of-two-cities.txt*** in the data directory inside the project directory. On running ./run_parallel.sh we get the following simulation output.

All the events are logged in the console as and when they occur.

Enter the number of nodes you want to simulate with and the simulation will start.



```
~/personal_projects/College/MapReducePython — -zsh

Process Creation:
Process Name: map_node_4
Splitting and Mapping split_4_4.txt in Parallel

------------ INTERMEDIATE COMBINE PHASE INITIATION IN PARALLEL------------

Map Process map_node_1 Start Time:
Map Process map_node_2 Start Time:
Map Process map_node_3 Start Time:
Map Process map_node_4 Start Time:
17:29:38
17:29:38
17:29:38
17:29:38
6f724c784dc7d42df6de82b955ca3500b813e5b79e2e116d003c5c7c430918da
21502c4e10b2d878433a0e78806ecd0735276e6d8b6f28262f8015cb770c3418
c67b506e002ffff7205b102a6717cfa6782543a9db786b032556885dd18d2bf2
59e5e5078ec399cd3633ddd0ee8ea3fc8b26db657b664de2310b370802623177
802a10d018f0593897b0e28a180a6b37ed0f5611e87712345d75233b5013930b
*Current Status: Number of Mapped Nodes =  0*
3
 input file path in map.py of node  3 ./mapper_data/split_3_4.txt
                                                      COMPUTATION TIME: MAP NODE 3 = 0:00:00.157933
                                                                                                  2
                                                                                                   input file path in map.py of n
ode  2 ./mapper_data/split_2_4.txt
                                COMPUTATION TIME: MAP NODE 2 = 0:00:00.212817
                                                                              4
                                                                               input file path in map.py of node  4 ./mapper_data/split_4_4.
txt
   COMPUTATION TIME: MAP NODE 4 = 0:00:00.065448
                                           1
                                            input file path in map.py of node  1 ./mapper_data/split_1_4.txt
                                                                                                    COMPUTATION TIME: MAP NODE 1
 = 0:00:00.092579
              *Current Status: Number of Mapped Nodes =  4*
------------ MAP AND COMBINE PHASE TERMINATION ------------

------------ REDUCE PHASE INITIATION ------------

Process Creation:
Process Name: reduce_node_1
```

```
~/personal_projects/College/MapReducePython — -zsh

------------ REDUCE PHASE INITIATION ------------

Process Creation:
Process Name: reduce_node_1
Reducing and Computing: combined_split_1_4.txt in Parallel

Process Creation:
Process Name: reduce_node_2
Reducing and Computing: combined_split_2_4.txt in Parallel

Process Creation:
Process Name: reduce_node_3
Reducing and Computing: combined_split_3_4.txt in Parallel

Process Creation:
Process Name: reduce_node_4
Reducing and Computing: combined_split_4_4.txt in Parallel

Reduce Process reduce_node_1 Start Time:
Reduce Process reduce_node_4 Start Time:
Reduce Process reduce_node_2 Start Time:
Reduce Process reduce_node_3 Start Time:
17:29:50
17:29:50
17:29:50
17:29:50
2beca50d2721e00525c03252a7622c9a57b6404d792466119a72053fbd7d591e
5cc8cc5389e10166df04e2adc9f761f28e2cc085056c63a8127559cbf11772d6
6c6ec311a3b7890eb9a1f1a60dcd0eb563d10f774a018ca748c296e547f2f25b
de54c447cbd3e2685d9592383f1490cbaa1fd40d81a23d8ee2bb6d1961b866af
3
input file path in reduce.py of node  3 ./mapper_data/combined_split_3_4.txt
COMPUTATION TIME: REDUCE NODE 3 = 0:00:00.490525
4
input file path in reduce.py of node  4 ./mapper_data/combined_split_4_4.txt
COMPUTATION TIME: REDUCE NODE 4 = 0:00:00.628095
2
input file path in reduce.py of node  2 ./mapper_data/combined_split_2_4.txt
COMPUTATION TIME: REDUCE NODE 2 = 0:00:00.487405
1
input file path in reduce.py of node  1 ./mapper_data/combined_split_1_4.txt
COMPUTATION TIME: REDUCE NODE 1 = 0:00:00.579403
------------ REDUCE PHASE TERMINATION ------------
```

The computation time taken is also calculated

```
------------ REDUCE PHASE TERMINATION ------------

------------ GETTING RESULT BACK TO MAIN NODE ------------

reduced_1_4.txt
reduced_2_4.txt
reduced_3_4.txt
reduced_4_4.txt
------------ RESULT OBTAINED AND WRITTEN INTO HOST DATA FOLDER ------------

---------------------- TERMINATING MAPREDUCE SIMULATION --------------------


****TOTAL COMPUTATION TIME OF THE ALGORITHM: 15076 ms ****
(MapReducePython) sanjitkumar@Sanjits-MacBook-Air MapReducePython %
```

It is important to clear the processes before running the simulation again. By doing ./ clean.sh you clear out the existing containers and volumes.

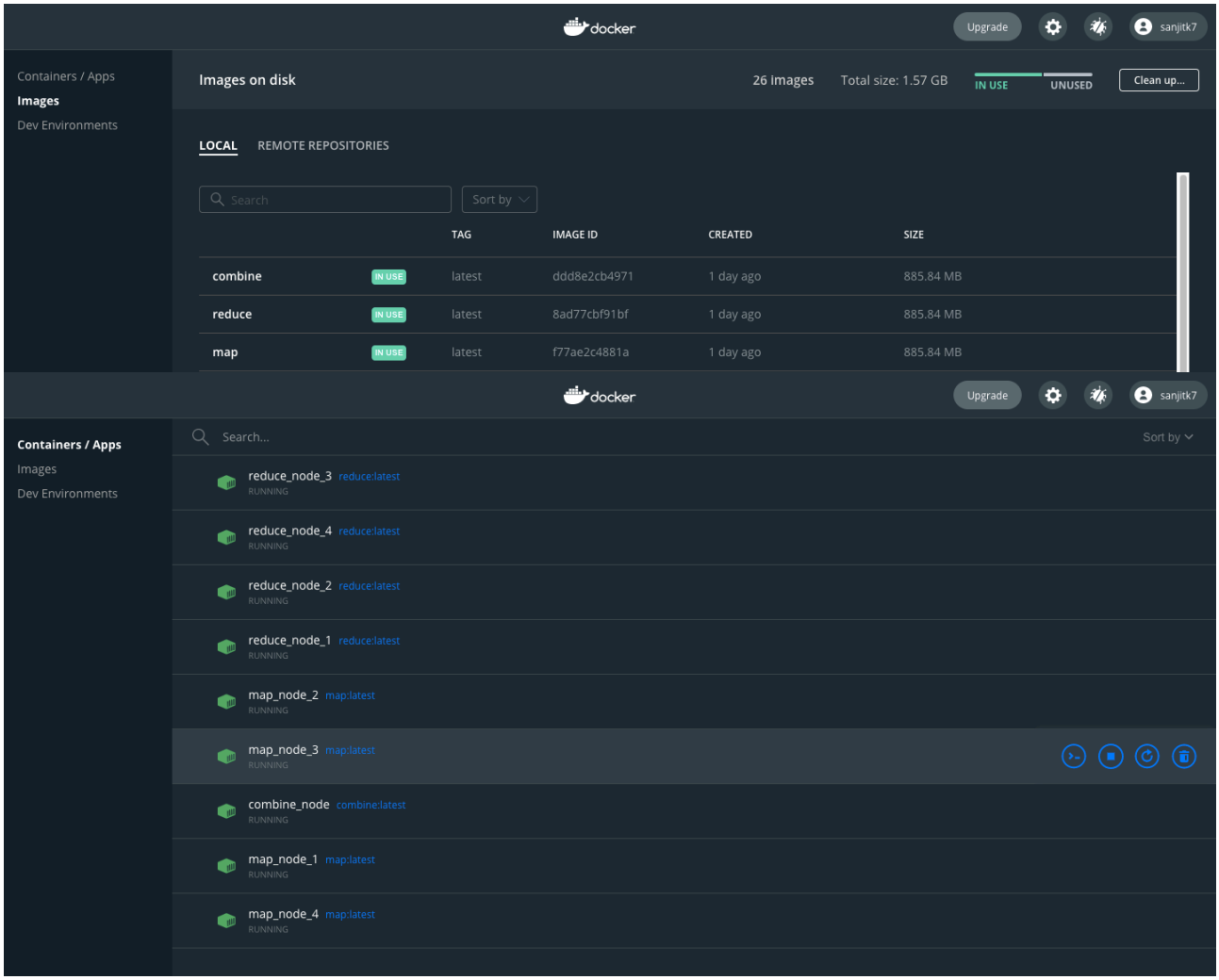
```
(MapReducePython) sanjitkumar@Sanjits-MacBook-Air MapReducePython % ./clean.sh
de54c447cbd3
5cc8cc5389e1
6c6ec311a3b7
2beca50d2721
802a10d018f0
59e5e5078ec3
21502c4e10b2
c67b506e002f
6f724c784dc7
Deleted Containers:
de54c447cbd3e2685d9592383f1490cbaa1fd40d81a23d8ee2bb6d1961b866af
5cc8cc5389e10166df04e2adc9f761f28e2cc085056c63a8127559cbf11772d6
6c6ec311a3b7890eb9a1f1a60dcd0eb563d10f774a018ca748c296e547f2f25b
2beca50d2721e00525c03252a7622c9a57b6404d792466119a72053fbd7d591e
802a10d018f0593897b0e28a180a6b37ed0f5611e87712345d75233b5013930b
59e5e5078ec399cd3633ddd0ee8ea3fc8b26db657b664de2310b370802623177
21502c4e10b2d878433a0e78806ecd0735276e6d8b6f28262f8015cb770c3418
c67b506e002ffff7205b102a6717cfa6782543a9db786b032556885dd18d2bf2
6f724c784dc7d42df6de82b955ca3500b813e5b79e2e116d003c5c7c430918da

Total reclaimed space: 2.99MB
mapper_data
WARNING! This will remove all local volumes not used by at least one container.
Are you sure you want to continue? [y/N] y
Total reclaimed space: 0B
(MapReducePython) sanjitkumar@Sanjits-MacBook-Air MapReducePython %
```

Running the same algorithm in serial we get the following computation time.

```
------------ REDUCE PHASE TERMINATION ------------

------------ GETTING RESULT BACK TO MAIN NODE ------------

reduced_1_4.txt
reduced_2_4.txt
reduced_3_4.txt
reduced_4_4.txt
------------ RESULT OBTAINED AND WRITTEN INTO HOST DATA FOLDER ------------

---------------------- TERMINATING MAPREDUCE SIMULATION --------------------


****TOTAL COMPUTATION TIME OF THE ALGORITHM: 19696 ms ****
```

Therefore we can say that the MapReduce algorithm improves the computation time of any computation in the distributed environment.

Docker Nodes that are running can be monitored via the Docker Desktop GUI



Conclusion and Future Scope

In this project I have created a visual simulation environment for the functioning of MapReduce in a distributed ecosystem. We saw that need for such an algorithm, its history and context of usage. It was further proved by creating a simulation environment in the scale of a single host machine running multiple docker containers. We later saw the performance improvement that comes with the MapReduce usage.

As the domain of Big Data continues to grow in importance and popularity, there is an increasing need to research techniques that effectively process such massive amounts of data. The most powerful aspect of MapReduce is its simplicity and scalability. By horizontally scaling the number of devices it's possible to improve computation time. But if there is ultimately a more direct algorithmic improvement that could improve the computation time, then the research and literature should focus on that.

References

[1] V. S. Martha, W. Zhao and X. Xu, "h-MapReduce: A Framework for Workload Balancing in MapReduce," 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), 2013, pp. 637-644, doi: 10.1109/AINA.2013.48.

[2] D. Magro and H. Paulino, "In-cache MapReduce: Leverage Tiling to Boost Temporal Locality-Sensitive MapReduce Computations," 2016 IEEE International Conference on Cluster Computing (CLUSTER), 2016, pp. 374-383, doi: 10.1109/CLUSTER.2016.33.

[3] G. Zhang, J. Wang, W. Huang, C. Li, Y. Zhang and C. Xing, "A Semantic++ MapReduce: A Preliminary Report," 2014 IEEE International Conference on Semantic Computing, 2014, pp. 330-336, doi: 10.1109/ICSC.2014.63.

[4] J. Lin, M. Lee and R. Yahyapour, "Scheduling MapReduce tasks on virtual MapReduce clusters from a tenant's perspective," 2014 IEEE International Conference on Big Data (Big Data), 2014, pp. 141-146, doi: 10.1109/BigData.2014.7004223.

[5] K. Rattanaopas and S. Kaewkeeree, "Improving Hadoop MapReduce performance with data compression: A study using wordcount job," 2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2017, pp. 564-567, doi: 10.1109/ECTICon.2017.8096300.