B534 Distributed Systems: Assignment 1

Rui Zhang
rz20@iu.edu

1 Introduction

This project develops a library of distributed MapReduce system in Python 3.6, which can be used for higher level of data processing. The source code can be found in Github `https://github.iu.edu/rz20/CSCI-B534-DISTRIBUTED-SYSTEMS-A1`. The implementation follows MapReducer paper [1] to guarantee features and simplify design. The limitation is primarily due to single machine, which have to handle multiprocessing and concurrent issues.

To simulate a distributed system, this library was developed based on socket communication (*socket, socketserver*) and multiprocessing programming (*multiprocessing*), as well as other utilities (*uuid, copy, json*). There are six folders, *Cluster, Examples, MapReducer, Mapper, Master*, and *Reducer* in the source-code repository. The *Cluster* folder contains coding for cluster behaviors, *Mapper* contains coding for mapper to run map function, *Reducer* for reducer to run reduce function, and *Master* for master node to coordinate works. The *MapReducer* contains coding for class inherited from mapper and reducer, since a cluster would be able to perform map and/or reduce tasks (depending on idle). The required API (`init_cluster`, `run_mapred`, and `destroy_cluster`) are implemented in `MapReducerAPI.py`. It follows objected-orientated programming to simplify the development process.

2 Designs

2.1 Data partitioning

As original MapReducer paper implementation (i.e., *3.1 Execution Overview*) [1], the program first splits the inputs files into M pieces, where M is the number of map tasks. This was also implemented in `Util.py`, which splits a given file into a number of smaller files.

Util.py

```
def split_file(file_path, num_split=M):
```

The scheduler in master then picks idle workers and assign a task from the pool, which is implemented in `Cluster/Scheduler.py`. The mutual-exclusive lock is necessary, since there may be race conditions given multiple mappers/reducers.

Scheduler.py

```
class Scheduler:
    def __init__(self):
        self.lock = threading.semaphore()
        self.pool = []
```

Once the map task is done and saved to the buffered memory, the mapper would partition into R regions and written to local disk, where R is number of reducer tasks. Several class functions were implemented in `Mapper/MapWorker.py`.

MapWorker.py

```
class Mapper:
    def store(self):
    def _partition(self):
```

The data partition is essential for mapreduce, since the data to process is usually very large. In each stages of computation, the (input or intermediate) data are split into smaller chunks and processed on each clusters.

2.2 Fault-tolerance

A simple version of worker failure tolerance was implementation in this library. Briefly, the scheduler only schedule the tasks to idle worker that is responding server's message. It might concern that any worker that failed during the computation needs to be recovered. However, this would greatly complicated the implementation, since now we need to consistently query about worker's computing status. For the study purpose of project, I left this implementation out, but it is possible to implement within the current library framework.

Scheduler.py

```
class Scheduler:
    def add_all(self, pool):
    def append(self, data):
    def get_task(self):
```

It is also worth to note that the fault occurs in master node could be fetal, although it is not likely because of single server. As in original paper [1], the only way to recover is to reboot the master and recompute the whole tasks. The current system would be fault-tolerance as long as master and at least one mapper/reducer survival.

2.3 Dynamic membership

To achieve the dynamic membership, i.e., a cluster can work as mapper or reducer depending on master assignment, The cluster object is instantiated from a `MapReducerWorker` class that inheritance of `Mapper` and `Reducer`.

MapReduceWorker.py

```
class MapReduceWorker(Mapper, Reducer):
    def __init__(self, uid)
```

To assign the clusters into mapper or reducer, a balance function was implemented and called in Master nodes. Depending how we write this balancer, we can achieve desired assignments. For the study purpose of this project, the default balancer is simply dividing available clusters in half.

MapReducerAPI.py

```
def balance_mapper_reducer(cluster)
```

3 Implementation

3.1 Assumptions

There are four main assumptions in this library, including

1. *The input data are text files.* This is quite a reasonable assumption, which greatly simplify our data processing. Per instruction, there are two required applications (i.e., word count and inverted index). They are all text files related applications.

2. *The program runs on a single machine.* Since we are not testing on distributed clusters, the program is mainly target at single machine usage. Yet, a stronger assumption may argue that a workable library on single machine is able to extend to true distributed system with little tweaks.

3. *All file are written to local disk*, although they are blind to everyone except Master. Since assumption 2, we have to write the data on local disk. Yet, to simulate distributed system, we would restrict the access. The data is only accessible after master acknowledge and communications.

4. *Simulated fault-tolerance on single machine*. Since assumption 2 (i.e., all processes run on the single machine), it is almost impossible to have faults such as disk failure (on “separated" clusters). We would simulate such fault by simply terminate the fault.

3.2 Communication

In general, a Master node is worked as Server, while Mapper/Reducer as Client. To simply the design, a state representation of Master and Mapper/Reducer (i.e., Worker) is adopted, much like processes state in the operating system. That is, Master has three states, namely `MASTER_READY`, `MASTER_WAIT`, and `MASTER_DONE`. Mapper/Reducer has four states, including `WORKER_READY`, `WORKER_WAIT`, `WORKER_DONE`, and `WORKER_CAN_CLOSE`.

The communication between Master and Mapper/Reducer, firstly exchange their current states, and then send the corresponding data accordingly. For example, when a Master S is idle (`MASTER_READY`), it receives a message from a Mapper W saying `WORKER_READY` $S \leftarrow W(worker\_ready)$. If there are data to send out, Master S then send its current states `WORKER_READY` to the worker W, $S(master\_ready) \rightarrow W$. If worker W is available, it responses master by saying it is ready to work, $S \leftarrow W(worker\_wait)$. At last, the master is sending out data to worker $S(data) \rightarrow W$.

It is quite straightforward to implement this protocol, since we only need to take care the states and use state to figure out what to do next. Along with states information, the sender's cluster id is also sent for potential debug purpose. It is also worth noting that sending some sates (such as `WORKER_WAIT`) seems redundant, it may benefit for future development of complicated tasks.

3.3 Log

The log of distributed systems is important. In this library, the important events (such as cluster creation, tasks done, etc) are simply print out directly on console.

4 Applications and Test Usages

There are two python scripts in the repository (`app_word_count.py` and `app_inverted_index.py`), which show examples of how to use he library API in applications. To run the program, simply run python with scripts and results will be shown in `result.txt`.

Command Line

```
$ python app_word_count.py
```

Command Line

```
$ python app_inverted_index.py
```

To show an example of fault-tolerance of this library, three python scripts are provided (`test_master_server.py`, `test_mapper_client.py`, and`test_reducer_client.py`). To run the test, open three seperate terminals and run the scripts in each terminal (run `test_master_server.py` first), while the results will be shown in `result.txt`.

Command Line

```
$ python test_master_serever.py
```

Command Line

```
$ python test_mapper_client.py
```

Command Line

```
$ python test_reducer_client.py
```

References

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.