Program Design Process

Milestone 1

In this stage, we implemented only one mapper and one reducer to complete the MapReduce task. When the program starts, the master simply calls mapper and reducer to do the job. A mapper calls a user-defined map function to convert all words to key-value pairs and store sorted pairs in an intermediate container. Then a reducer merges all values by keys and calls a user-defined reduce function to handle all pairs and write them to an output file.

The issues are that(as pointed out by the grader):

1. The master invoked workers as functions instead of processes.
2. Intermediate data are passed directly from mapper to the reducer instead of passed through intermediate files. We actually planned and tried to use intermediate files but we encountered some golang package problems.
3. The UDFs are passed to workers as functions but we should read them from the other file.

Milestone 2

In order to fulfill multi-process implementation and fix problems in Milestone 1, we drastically modify the frame of our design. Rather than putting all things in one file, we separate them into `master.go`, `worker.go` and `message.go`. We use RPC for communication between master and worker, and the message information for communication is in `message.go`.

In `master.go`, the master repeatedly handles task requests and task reports from multiple workers. The master will tell workers some important information including: the location of the input text file; the current stage of the task i.e. map stage/reduce stage; the task index and etc. After all map tasks are finished, the master will then start to distribute the reduce tasks, and after all reduce tasks completed, the master will inform the workers and stop working.

In worker.go, a worker repeatedly asks for tasks and reports task status(mission complete/fail) to master.

In map stage, a worker acts as a mapper to:

1) firstly find the corresponding lines in the input file based on offset produced by task index;
2) encode and map key-value pair to multiple intermediate files according to the hash function;
3) report mission status.

While in reduce stage, a worker acts as a reducer to:

1) decodes and reads from intermediate files by task index;
2) sort by keys;
3) deal with values by key;
4) write to an output file.

To ensure the safety of communication between one master and multiple worker processes, we use `chan` for queue and `mutex` to lock variables in case of modifying the same data at the same time.

Issues:

1. The output file is incorrect for some number of N. There is a bug when mappers try to find their corresponding lines.
2. Need more apps to test the program.

Milestone 3

We solved the bug of incorrectly reading blocks and added more apps and tests to the test script. In addition, we put more python spark programs and correct outputs produced by spark in the `testfiles`.

Running Test or Program

- To run the test script, please type the following commands:

  ```
  $ cd src/main

  $ sh test.sh
  ```

  The test script includes:

  1. Running single process

2. Running multiple processes with different apps
3. Running multiple processes and with a possible crash app.

- To run programs directly, please type the following commands:

```
$ cd src/main

$ go run master_main.go inputFile N

$ go run worker_main.go app.so
```

where `inputFile` is the name of the input file, `N` is the number of mappers/reducers; `app.so` is the name of the UDF plugin file with `.so` as filename extension.

mapReduce UDF explain:

Wc.go: this will count the number of each word

occureByLength.go: this will output the number of words with length of 1, 2, 3…

wordLength.go: this will output the length of each word

Tests

For basic correctness, it is shown by the 3 mapreduce applications with the result from pySpark.

For multiprocesses, we run multiple workers at the same time, and each worker gets a split of its part and only reads the part. We could get the correct result and it means that we could do multiple processes.

For fault tolerance, we know that sometimes servers could fail or use too much time. For the fault tolerance test, we imitate the behavior of a crashing worker by setting the exit code os.Exit(1). This code will make the worker crash, then we can see what will happen after the code crashes. For our code, if the mapper or reducer crashed, the job will not be done at all and could be redistributed by the master. In our test, we will run new worker processes to accept those failed jobs of crashed workers, and at last we could see that the jobs could be done perfectly, and it means that our mapReduce could tolerate failed workers.