Project 2: A MapReduce Library (Part 2)

Due May 3 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip and gz
Available Apr 20 at 12am - May 5 at 11:59pm 16 days

This assignment was locked May 5 at 11:59pm.

The second project continues the work you started - building a MapReduce library. Here you will tackle a distributed version of the library, writing code for a master that hands out tasks to multiple workers and handles failures in workers. The interface to the library and the approach to fault tolerance is similar to the one described in the original **MapReduce paper (https://www.usenix.org/legacy/event/osdi04/tech/full_papers/dean/dean.pdf)** we discussed in class. As with the previous project, you will also complete a sample Map/Reduce application.

Currently, your MapReduce implementation runs tasks one at a time. The popularity of MapReduce comes from the fact that it can automatically parallelize ordinary sequential code without any work from the developer. In this part of the lab, you will complete the distributed MapReduce mode to split work over a set of worker threads that run in parallel on multiple cores. While not distributed across multiple machines as in real MapReduce deployments, your implementation will use RPC to simulate distributed computation.

The code in mapreduce/master.go handles the majority of managing a MapReduce job. Additionally, we give you the complete code for a worker thread, found in mapreduce/worker.go, and some of the code to deal with RPCs, found in mapreduce/common_rpc.go.

Your work: Over the course of the following exercises, you will have to write/modify following files yourself.

- schedule (schedule.go)

There are two parts for Project 2.

Part A - Distributing MapReduce

In the first part, your job is to implement **schedule()** within mapreduce/schedule.go. This function is called twice by the master for each MapReduce job, once for the Map phase and once for the Reduce phase. schedule()'s job is to distribute tasks to available workers. Usually, there will be more tasks than worker threads so schedule() must give each worker a sequence of tasks. The function should wait until all tasks have completed before returning.

To learn about the set of workers, schedule() reads off its registerChan argument. The channel yields a string for each worker, containing the RPC address of the worker. While some workers may exist before schedule() is called and some may start while schedule() is running, all will appear on registerChan. **schedule() should use all the workers.**

schedule() tells a worker to execute a task by sending a RPC to the worker in the format of Worker.DoTask. This RPC's arguments are defined by DoTaskArgs in mapreduce/common_rpc.go. The File element is only used by Map tasks as the name of the file to read. schedule() can find these file names in mapFiles.

To send an RPC to a worker use the call() function in mapreduce/common_rpc.go. The first argument of the call is the worker's address, received from registerChan. The second argument should be "Worker.DoTask". Finally, the third argument should be the DoTaskArgs structure and the last argument should be nil.

This part only requires modifications to schedule.go. If you modify other files while working on this part, please remove your changes before submitting.
To test your solution:

```
go test -run TestParallel
```

This will execute two tests, TestParallelBasic and TestParallelCheck, the latter of which will verify that your scheduler successfully orchestrates the execution of tasks in parallel by the workers.

Hints:

- RPC package documents the Go RPC package.
- schedule() should send RPCs to the workers in parallel so that the workers can work on tasks concurrently. You will find the go statement useful for this purpose; see Concurrency in Go.
- schedule() must wait for a worker to finish before it can give it another task. You may find Go's channels useful.
- You may find sync.WaitGroup useful.
- The easiest way to track down bugs is to insert print statements (perhaps calling debug() in common.go), collect the output in a file with *go test -run TestParallel > out*, and then think about whether the output matches your understanding of how your code should behave. The last step is the most important.
- To check if your code has **race conditions**, run Go's race detector with your test:
  go test -race -run TestParallel > out.

Part B - Handling worker failures

In this part, you will write the code for the master to handle failed workers. This is relatively easy due to the design of MapReduce: workers don't have persistent state. If a worker fails while it is handling an RPC from the master, the call() function call will timeout eventually and return false. In this case, the master should assign the task to another worker.

Just because an RPC fails it doesn't necessarily mean that the worker didn't execute the task, but rather the worker may have successfully executed it but the reply was lost or that the worker may still be executing but the master's RPC timed out. Therefore, it is possible that two workers receive the same task, compute it, and generate output. Because of this, it is necessary for two invocations of a map or reduce function to generate the same output for a given input (i.e. map and reduce functions are "functional") in order for there to be no inconsistencies if subsequent processing sometimes reads one output and sometimes the other. In

addition, the MapReduce framework ensures that map and reduce function output appears atomically: the output file will either not exist, or will contain the entire output of a single execution of the map or reduce function (the lab code doesn't actually implement this, but instead only fails workers at the end of a task, so there aren't concurrent executions of a task).

Your implementation must pass the two remaining test cases in test_test.go. The first case tests the failure of one worker, while the second test case tests the handling of many failures of workers. Periodically, the test cases start new workers that the master can use to make forward progress, but these workers fail after handling a few tasks.
To run these tests:

```
go test -run Failure
```

Your solution to Part B should only involve modifications to **schedule.go**. If you modify other files as part of debugging, please restore their original contents and then test before submitting.

Submission instruction:

File type: gz or zip

Filename: project2_YOUR_NAME (e.g. project2_Byungjin_Jun.tar.gz)

- **Only one person in a group** should submit your file. The filename should be the name of one of the group member.

File contents: ALL FILES of the project (please don't modify any file other than schedule.go)