For your first project you will create a MapReduce library as a gentle way to get familiar with the Go programming language and with the challenges of building fault-tolerant systems.

We organized this project in two parts, each in turns divided in two steps; for the first part, you are asked to write a simple MapReduce program and a sample program that use it. In the second part, you will write a distributed version of your MapReduce with a Master that assigns tasks to workers and handles failures. The interface to the library is similar to that described in the original MapReduce paper we discussed in class.

Project Setup

The lab is to be implemented in Go. For reference on the Go language, the course web site has a pointer to a good tutorial. There are other good references that you may want to check out, including:

- Effective Go
- Go by example
- And if you want a book, The Go Programming Language is worth the money.

To get you started with these projects, we provide a basic framework for both the sequential and distributed versions. Please check the link below.

lab1_mapreduce.tar.gz

Project Overview

The mapreduce package reveals a simple MapReduce library typically started by calling Distributed() (in master.go) or Sequential() (also master.go). The sequential mode executes map and reduce jobs one at a time which, while slower, is useful for understanding the project. The distributed version runs jobs in parallel, first the map tasks and then reduce ones. It is faster but also harder to debug.

A job is executed in the following order:

1. A number of input files are provided to the application as well as two functions (map and reduce) and a number of reduce tasks (nReduce).
2. The application uses this knowledge to create a master. This, in turn, starts an RPC server (in master_rpc.go) and waits for workers to register (RPC call Register() from master.go). Upon availability, tasks are assigned to workers with schedule() (schedule.go).
3. Each input file is considered by the master to be one map task, calling doMap() (common_map.go) at least once for each map task. It does so either directly (when proceeding sequentially) or by issuing the DoTask RPC to a worker (worker.go). Each doMap() call reads the appropriate file, executes the map function on the contents, and writes the key/value pairs to nReduce intermediate files. The keys are hashed in order to pick the intermediate file and thus the reduce task that will process the key. In the end, there will be nMap x nReduce files. Each file name consists of a prefix, map task number,

and reduce task number. Workers must be able to read files written by any other worker in addition to the input files. In the real world, a distributed storage system such as GFS is used, however in this lab, all the workers are on the same machine.

4. The master then calls doReduce() (common_reduce.go) at least once for each reduce task. As with doMap(), it does so either directly or through a worker. For reduce task r, doReduce() collects the r'th intermediate file for each map task and calls the reduce function for each key that appears in those files. The reduce tasks produce nReduce result files.
5. The master calls mr.merge() (master_splitmerge.go) to merge the nReduce result files into a single output.
6. Finally, the master sends a Shutdown RPC to each of its workers before it shuts down its own RPC server.

Your work: Over the course of the following exercises, you will have to write/modify following files yourself (Do not modify other files).

Part 1.A

- doMap (common_map.go)

- doReduce (common_reduce.go)

Part 1.B

- mapF and reduceF (../main/wc.go)

Part 2

- *schedule (schedule.go)*

Part 1.A. Sequential MapReduce

The code you are given contains holes for you to fill in. Before you write your first MapReduce program, it is necessary to implement the sequential mode. In particular, there are two functions you must write: the function to divide up the output of a map task and the function that gathers all the inputs for a reduce task. These tasks are done by the **doMap() function in common_map.go and the doReduce() function in common_reduce.go** respectively. The files contain comments that should help you.

To determine if you have correctly implemented the functions, we have provided you with a Go test suite that checks the correctness of your implementation. These tests are contained in the file test_test.go. To do that run:

```
cd YOUR_MAP_REDUCE_DIR
export "GOPATH=$PWD"
cd src/mapreduce
go test -run Sequential
```

If the code is correct the output will show **ok** next to the tests, otherwise your implementation has a bug in it. For more verbose output, set debugEnabled = true in common.go and -v to the test command above.

Part 1.B. Test MapReduce with WordCounter example

Now you have 1.A, you need to implement a word counter - a simple MapReduce example. In main/wc.go you'll find empty mapF() and reduceF() functions. Your job is to insert code so that wc.go reports the number of occurrences of each word in the input. For this project, a word is any contiguous sequence of letters, as determined by unicode.IsLetter.

There are input files with pathnames of the form pg-*.txt in "src/main" taken from Project Gutenberg. Here's how to run wc with the input files:

```
cd YOUR_MAP_REDUCE_DIR
export "GOPATH=$PWD"
cd src/main
go run wc.go master sequential pg-huckleberry_finn.txt
```

The compilation would fail here because **mapF() and reduceF()** are not complete. When a test is succeeded, try to test all with the following command.

```
sh test-wc.sh
```

Review Section 2 of the MapReduce paper. Your mapF() and reduceF() functions will differ a bit from those in the paper's Section 2.1. Your mapF() will be passed the name of a file, as well as that file's contents; it should split the contents into words, and return a Go slice of mapreduce.KeyValue. While you can choose what to put in the keys and values for the mapF() output, for word count it only makes sense to use words as the keys. Your reduceF() will be called once for each key, with a slice of all the values generated by mapF() for that key. It must return a string containing the total number of occurrences of the key.

Part 2.A. Distributed MapReduce

The second part continues the work you started - building a MapReduce library. Now that you have a mapreduce library and a simple application, your next task is make it distributed and able to tolerate workers' failures. The popularity of MapReduce comes from the fact that it can automatically parallelize ordinary sequential code without any work from the developer. Here, you will complete the distributed MapReduce mode to split work over a set of worker threads that run in parallel on multiple workers. While not distributed across multiple machines as in real MapReduce deployments, your implementation will use RPC to simulate distributed computation. The interface to the library and the approach to fault tolerance is similar to the one described in the original MapReduce paper (链接到外部网站。) we discussed in class.

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 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.

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.
- To check if your code has **race conditions**, run Go's race detector with your test:

  go test -race -run TestParallel > out.

Part 2.B. Handling worker failures

For your last *mission,* 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 2 should only involve modifications to **schedule.go**.

Submission instruction

- File type: gz or zip

- File contents: ALL FILES of the project

- File name: project1_YOUR_NAME (e.g. project1_Jin_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.

- **DO NOT** leave any codes for printing/debugging. Additional text output in the test result may significantly distract grading, so it will deduct minor points from your grade.

- Do not include extra files unrelated to the project.