Project 1: A MapReduce Library (Part 1)

Due Apr 22 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip and gz
Available Apr 8 at 12am - Apr 24 at 11:59pm 17 days

This assignment was locked Apr 24 at 11:59pm.

In the first two projects, you will create a MapReduce library as a gentle way to learn the Go programming language and how to build fault-tolerant systems.

For the first project, 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 gives workers tasks and handles failures. The interface to the library is similar to that described in the original **MapReduce paper (https://www.usenix.org/legacy/event/osdi04/tech/full_papers/dean/dean.pdf)** we discussed in class.

Project Setup

The lab is to be implemented in **Go (https://golang.org/)**. 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 (https://golang.org/doc/effective_go.html)**
- **Go by Example (https://gobyexample.com/)**
- And if you want a book, **The Go Programming Language (https://www.amazon.com/gp/product/0134190440)** is worth the money.

To get you started with these projects, we will provide a basic framework for both the distributed and sequential versions. 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's faster but also harder to debug.

Starting Code: **mapreduce_lab1.tar.gz**

Project - Basics

The mapreduce package reveals a simple MapReduce library typically started by calling Distributed() (in master.go) or Sequential() (also master.go). 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.

- doMap (mapcommon_map.go)
- doReduce (common_reduce.go)
- mapF and reduceF (../main/wc.go)
- schedule (schedule.go) -> Project 2

There are two parts to project 1.

Project - Part A

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_MAPREDUCE_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.

Project - Part B

In this part, you will 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_MAPREDUCE_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.

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.

Submission instruction:

File type: gz or zip

File name: project1_YOUR_NAME (e.g. project1_Byungjin_Jun.tar.gz)

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

File contents: ALL FILES of the project