COS418 Assignment 1 (Part 2): Sequential Map/Reduce

Introduction

In parts 2 and 3 of the first assignment, you will build a Map/Reduce library as a way to learn the Go programming language and as a way to learn about fault tolerance in distributed systems. For part 2, you will work with a sequential Map/Reduce implementation and write a sample program that uses it.

The interface to the library is similar to the one described in the original MapReduce paper.

Software

You'll implement this assignment (and all the assignments) in Go. The Go web site contains lots of tutorial information which you may want to look at.

For the next two parts of this assignment, we will provide you with a significant amount of scaffolding code to get started. The relevant code is under this directory. We will ensure that all the code we supply works on the CS servers (cycles/courselab).

In this assignment, we supply you with parts of a flexible MapReduce implementation. It has support for two modes of operation, *sequential* and *distributed*. Part 2 deals with the former. The map and reduce tasks are all executed in serial: the first map task is executed to completion, then the second, then the third, etc. When all the map tasks have finished, the first reduce task is run, then the second, etc. This mode, while not very fast, can be very useful for debugging, since it removes much of the noise seen in a parallel execution. The sequential mode also simplifies or eliminates various corner cases of a distributed system.

Getting familiar with the source

The mapreduce package (located at `src/mapreduce`) provides a simple Map/Reduce library with a sequential implementation. Applications would normally call `Distributed()` — located in `mapreduce/master.go` — to start a job, but may instead call `Sequential()` — also in `mapreduce/master.go` — to get a sequential execution, which will be our approach in this assignment.

The flow of the mapreduce implementation is as follows:

1. The application provides a number of input files, a map function, a reduce function, and the number of reduce tasks (nReduce).
2. A master is created with this knowledge. It spins up an RPC server (see `mapreduce/master_rpc.go`), and waits for workers to register (using the RPC call `Register()` defined in `mapreduce/master.go`). As tasks become available, `schedule()` — located in `mapreduce/schedule.go` — decides how to assign those tasks to workers, and how to handle worker failures.

3. The master considers each input file one map task, and makes a call to `doMap()` in `mapreduce/common_map.go` at least once for each task. It does so either directly (when using `Sequential()`) or by issuing the `DoTask` RPC — located in `mapreduce/worker.go` — on a worker. Each call to `doMap()` reads the appropriate file, calls the map function on that file's contents, and produces `nReduce` files for each map file. Thus, after all map tasks are done, the total number of files will be the product of the number of files given to map (`nIn`) and `nReduce`.

```
f0-0, ...,  f0-[nReduce-1],
...
f[nIn-1]-0, ..., f[nIn-1]-[nReduce-1].
```

- The master next makes a call to `doReduce()` in `mapreduce/common_reduce.go` at least once for each reduce task. As with `doMap()`, it does so either directly or through a worker. `doReduce()` collects corresponding files from each map result (e.g. `f0-i, f1-i, ... f[nIn-1]-i`), and runs the reduce function on each collection. This process produces `nReduce` result files.
- The master calls `mr.merge()` in `mapreduce/master_splitmerge.go`, which merges all the `nReduce` files produced by the previous step into a single output.
- The master sends a Shutdown RPC to each of its workers, and then shuts down its own RPC server.

You should look through the files in the MapReduce implementation, as reading them might be useful to understand how the other methods fit into the overall architecture of the system hierarchy. However, for this assignment, you will write/modify **only** `doMap` in `mapreduce/common_map.go`, `doReduce` in `mapreduce/common_reduce.go`, and `mapF` and `reduceF` in `main/wc.go`. You will not be able to submit other files or modules. In other words, any helper functions must reside within these listed files.

Part A: Map/Reduce input and output

The Map/Reduce implementation you are given is missing some pieces. Before you can write your first Map/Reduce function pair, you will need to fix the sequential implementation. In particular, the code we give you is missing two crucial pieces: the function that divides up the output of a map task, and the function that gathers all the inputs for a reduce task. These tasks are carried out by the `doMap()` function in `mapreduce/common_map.go`, and the `doReduce()` function in `mapreduce/common_reduce.go` respectively. The comments in those files should point you in the right direction.

To help you determine if you have correctly implemented `doMap()` and `doReduce()`, we have provided you with a Go test suite that checks the correctness of your implementation. These tests are implemented in the file `test_test.go`. To run the tests for the sequential implementation that you have now fixed, follow this (or a non-bash equivalent) sequence of shell commands, starting from the `418/assignment1-2` directory:

```
$ cd 418/assignment1-2
$ ls
```

```
README.md src
# Go needs $GOPATH to be set to the directory containing "src"
$ export GOPATH="$PWD"
$ cd src
$ go test -run Sequential mapreduce/...
ok  mapreduce 4.515s
```

If the output did not show *ok* next to the tests, your implementation has a bug in it. To give more verbose output, set `debugEnabled = true` in `mapreduce/common.go`, and add `-v` to the test command above. You will get much more output along the lines of:

```
$ go test -v -run Sequential
=== RUN   TestSequentialSingle
master: Starting Map/Reduce task test
Merge: read mrtmp.test-res-0
master: Map/Reduce task completed
--- PASS: TestSequentialSingle (2.30s)
=== RUN   TestSequentialMany
master: Starting Map/Reduce task test
Merge: read mrtmp.test-res-0
Merge: read mrtmp.test-res-1
Merge: read mrtmp.test-res-2
master: Map/Reduce task completed
--- PASS: TestSequentialMany (2.32s)
PASS
ok  mapreduce4.635s
```

Part B: Single-worker word count

Now that the map and reduce tasks are connected, we can start implementing some interesting Map/Reduce operations. For this assignment, we will be implementing word count — a simple and classic Map/Reduce example. Specifically, your task is to modify `mapF` and `reduceF` within `main/wc.go` so that the application reports the number of occurrences of each word. A word is any contiguous sequence of letters, as determined by `unicode.IsLetter`.

There are some input files with pathnames of the form `pg-*.txt` in the `main` directory, downloaded from Project Gutenberg. Remember to set `GOPATH` before continuing. This is the result when you initially try to compile the code we provide you and run it:

```
$ cd "$GOPATH/src/main"
$ go run wc.go master sequential pg-*.txt
# command-line-arguments
./wc.go:14: missing return at end of function
./wc.go:21: missing return at end of function
```

The compilation fails because we haven't written a complete map function (`mapF()`) nor a complete reduce function (`reduceF()`) in `wc.go` yet. Before you start coding read 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 it into words, and return a Go slice of key/value pairs, of type `mapreduce.KeyValue`. Your `reduceF()` will be called once for each key, with a slice of all the values generated by `mapF()` for that key; it should return a single output value.

You can test your solution using:

```
$ cd "$GOPATH/src/main"
$ go run wc.go master sequential pg-*.txt
master: Starting Map/Reduce task wcseq
Merge: read mrtmp.wcseq-res-0
Merge: read mrtmp.wcseq-res-1
Merge: read mrtmp.wcseq-res-2
master: Map/Reduce task completed
```

The output will be in the file `mrtmp.wcseq`. We will test your implementation's correctness with the following command, which should produce the following top 10 words:

```
$ sort -n -k2 mrtmp.wcseq | tail -10
he: 34077
was: 37044
that: 37495
I: 44502
in: 46092
a: 60558
to: 74357
of: 79727
and: 93990
the: 154024
```

(this sample result is also found in main/mr-testout.txt)

You can remove the output file and all intermediate files with:

```
$ rm mrtmp.*
```

To make testing easy for you, from the `$GOPATH/src/main` directory, run:

```
$ sh ./test-wc.sh
```

and it will report if your solution is correct or not.

Resources and Advice

- a good read on what strings are in Go is the Go Blog on strings.
- you can use `strings.FieldsFunc` to split a string into components.
- the strconv package is handy to convert strings to integers etc.

Point Distribution

Test	Points
SequentialSingle	10
SequentialMany	10
test-wc.sh	10

Submitting Assignment

You hand in your assignment exactly as you've been letting us know your progress:

```
$ git commit -am "[you fill me in]"
$ git tag -a -m "i finished assignment 1-2" a12-handin
$ git push origin master a12-handin
```

You should verify that you are able to see your final commit and your a12-handin tag on the Github page in your repository for this assignment.

Recall, in order to overwrite a tag use the force flag as follows.

```
$ git tag -fam "i finished assignment 1-2" a12-handin
$ git push -f --tags
```

We will use the timestamp of your **last** tag for the purpose of calculating late days, and we will only grade that version of the code. (We'll also know if you backdate the tag, don't do that.)

Our test script is not a substitute for doing your own testing on the full go test cases detailed above. Before submitting, please run the full tests given above for both parts one final time. **You** are responsible for making sure your code works.

You will receive full credit for Part A if your software passes the Sequential tests in `test_test.go` and `test-wc.sh` (as run by the `go test` commands above) on the CS servers. You will receive full credit for Part B if your Map/Reduce word count output matches the correct output for the sequential execution above when run on the CS servers.

Acknowledgements

This assignment is adapted from MIT's 6.824 course. Thanks to Frans Kaashoek, Robert Morris, and Nickolai Zeldovich for their support.