COS418 Assignment 1 (Part 3): Distributed Map/Reduce

Introduction

This part of the assignment continues from assignment 1 part 2 — building 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. You will tackle a distributed version of the Map/Reduce 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. As with the previous part of this assignment, you will also complete a sample Map/Reduce application.

Software

You will use the same mapreduce package as in assignment 1-2, focusing this time on the distributed mode.

Over the course of this assignment, you will have to modify `schedule` in `schedule.go`, as well as `mapF` and `reduceF` in `main/ii.go`.

As with the previous part of this assignment, you should not modify any other files, but reading them might be useful in order to understand how the other methods fit into the overall architecture of the system.

To get started, copy all source files from `assignment1-2/src` to `assignment1-3/src` as follows.

```
# start from your 418 GitHub repo
$ cd 418
$ ls
README.md     assignment1-1 assignment1-2 assignment1-3 assignment2
assignment3   assignment4   assignment5   setup.md
$ cp -r assignment1-2/src/* assignment1-3/src/
$ ls assignment1-3/src
main      mapreduce
```

Part C: Distributing MapReduce Tasks

One of Map/Reduce's biggest selling points is that the developer should not need to be aware that their code is running in parallel on many machines. In theory, we should be able to take the word count code you wrote in Part B of assignment 1-2, and automatically parallelize it!

Our current implementation runs all the map and reduce tasks one after another on the master. While this is conceptually simple, it is not great for performance. In this part of the assignment, you will complete a version of MapReduce that splits the work up over a set of worker threads in order to exploit multiple cores. Computing the map tasks in parallel and then the reduce tasks can result in much faster completion, but is also harder to implement and debug. Note that for this part of the assignment, the work is not distributed across multiple machines as in “real” Map/Reduce deployments, your implementation will be using RPC and channels to simulate a truly distributed computation.

To coordinate the parallel execution of tasks, we will use a special master thread, which hands out work to the workers and waits for them to finish. To make the assignment more realistic, the master should only communicate with the workers via RPC. We give you the worker code (`mapreduce/worker.go`), the code that starts the workers, and code to deal with RPC messages (`mapreduce/common_rpc.go`).

Your job is to complete `schedule.go` in the `mapreduce` package. In particular, you should modify `schedule()` in `schedule.go` to hand out the map and reduce tasks to workers, and return only when all the tasks have finished.

Look at `run()` in `master.go`. It calls your `schedule()` to run the map and reduce tasks, then calls `merge()` to assemble the per-reduce-task outputs into a single output file. `schedule` only needs to tell the workers the name of the original input file (`mr.files[task]`) and the task `task`; each worker knows from which files to read its input and to which files to write its output. The master tells the worker about a new task by sending it the RPC call `Worker.DoTask`, giving a `DoTaskArgs` object as the RPC argument.

When a worker starts, it sends a Register RPC to the master. `master.go` already implements the master's `Master.Register` RPC handler for you, and passes the new worker's information to `mr.registerChannel`. Your `schedule` should process new worker registrations by reading from this channel.

Information about the currently running job is in the `Master` struct, defined in `master.go`. Note that the master does not need to know which Map or Reduce functions are being used for the job; the workers will take care of executing the right code for Map or Reduce (the correct functions are given to them when they are started by `main/wc.go`).

To test your solution, you should use the same Go test suite as you did in Part A of assignment 1-2, except swapping out `-run Sequential` with `-run TestBasic`. This will execute the distributed test case without worker failures instead of the sequential ones we were running before. Remember to set your `GOPATH` first.

```
$ go test -run TestBasic mapreduce/...
```

As before, you can get more verbose output for debugging if you 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 TestBasic mapreduce/...
=== RUN   TestBasic
/var/tmp/824-32311/mr8665-master: Starting Map/Reduce task test
Schedule: 100 Map tasks (50 I/Os)
/var/tmp/824-32311/mr8665-worker0: given Map task #0 on file 824-mrinput-
0.txt (nios: 50)
/var/tmp/824-32311/mr8665-worker1: given Map task #11 on file 824-mrinput-
11.txt (nios: 50)
/var/tmp/824-32311/mr8665-worker0: Map task #0 done
/var/tmp/824-32311/mr8665-worker0: given Map task #1 on file 824-mrinput-
1.txt (nios: 50)
/var/tmp/824-32311/mr8665-worker1: Map task #11 done
```

```
/var/tmp/824-32311/mr8665-worker1: given Map task #2 on file 824-mrinput-
2.txt (nios: 50)
/var/tmp/824-32311/mr8665-worker0: Map task #1 done
/var/tmp/824-32311/mr8665-worker0: given Map task #3 on file 824-mrinput-
3.txt (nios: 50)
/var/tmp/824-32311/mr8665-worker1: Map task #2 done
...
Schedule: Map phase done
Schedule: 50 Reduce tasks (100 I/Os)
/var/tmp/824-32311/mr8665-worker1: given Reduce task #49 on file 824-
mrinput-49.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker0: given Reduce task #4 on file 824-
mrinput-4.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker1: Reduce task #49 done
/var/tmp/824-32311/mr8665-worker1: given Reduce task #1 on file 824-
mrinput-1.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker0: Reduce task #4 done
/var/tmp/824-32311/mr8665-worker0: given Reduce task #0 on file 824-
mrinput-0.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker1: Reduce task #1 done
/var/tmp/824-32311/mr8665-worker1: given Reduce task #26 on file 824-
mrinput-26.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker0: Reduce task #0 done
...
Schedule: Reduce phase done
Merge: read mrtmp.test-res-0
Merge: read mrtmp.test-res-1
...
Merge: read mrtmp.test-res-49
/var/tmp/824-32311/mr8665-master: Map/Reduce task completed
--- PASS: TestBasic (25.60s)
PASS
ok  mapreduce25.613s
```

Part D: Handling worker failures

In this part you will make the master handle failed workers. MapReduce makes this relatively easy because workers don't have persistent state. If a worker fails, any RPCs that the master issued to that worker will fail (e.g., due to a timeout). Thus, if the master's RPC to the worker fails, the master should re-assign the task given to the failed worker to another worker.

An RPC failure doesn't necessarily mean that the worker failed; the worker may just be unreachable but still computing. Thus, it may happen that two workers receive the same task and compute it. However, because tasks are idempotent, it doesn't matter if the same task is computed twice — both times it will generate the same output. So, you don't have to do anything special for this case. (Our tests never fail workers in the middle of task, so you don't even have to worry about several workers writing to the same output file.)

You don't have to handle failures of the master; we will assume it won't fail. Making the master fault-tolerant is more difficult because it keeps persistent state that would have to be recovered in order to resume operations after a master failure. Much of the rest of this course is devoted to this challenge.

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 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. Run these tests as follows. Remember to set your `GOPATH` first.

```
$ go test -run Failure mapreduce/...
```

Part E: Inverted index generation

Word count is a classical example of a Map/Reduce application, but it is not an application that many large consumers of Map/Reduce use. It is simply not very often you need to count the words in a really large dataset. For this application exercise, we will instead have you build Map and Reduce functions for generating an ***inverted index***.

Inverted indices are widely used in computer science, and are particularly useful in document searching. Broadly speaking, an inverted index is a map from interesting facts about the underlying data, to the original location of that data. For example, in the context of search, it might be a map from keywords to documents that contain those words.

We have created a second binary in `main/ii.go` that is very similar to the `wc.go` you built earlier. You should modify `mapF` and `reduceF` in `main/ii.go` so that they together produce an inverted index. Running `ii.go` should output a list of tuples, one per line, in the following format. Remember to set your `GOPATH` first.

```
$ go run ii.go master sequential pg-*.txt
$ head -n5 mrtmp.iiseq
A: 16 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great_expectations.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-les_miserables.txt,pg-metamorphosis.txt,pg-moby_dick.txt,pg-sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-tom_sawyer.txt,pg-ulysses.txt,pg-war_and_peace.txt
ABC: 2 pg-les_miserables.txt,pg-war_and_peace.txt
ABOUT: 2 pg-moby_dick.txt,pg-tom_sawyer.txt
ABRAHAM: 1 pg-dracula.txt
ABSOLUTE: 1 pg-les_miserables.txt
```

If it is not clear from the listing above, the format is:

```
word: #documents documents,sorted,and,separated,by,commas
```

We will test your implementation's correctness with the following command, which should produce these resulting last 10 items in the index:

```
$ sort -k1,1 mrtmp.iiseq | sort -snk2,2 mrtmp.iiseq | grep -v '16' | tail -10
```

```
women: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-
emma.txt,pg-frankenstein.txt,pg-great_expectations.txt,pg-
huckleberry_finn.txt,pg-les_miserables.txt,pg-metamorphosis.txt,pg-
moby_dick.txt,pg-sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-
tom_sawyer.txt,pg-ulysses.txt,pg-war_and_peace.txt
won: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-
frankenstein.txt,pg-great_expectations.txt,pg-grimm.txt,pg-
huckleberry_finn.txt,pg-les_miserables.txt,pg-metamorphosis.txt,pg-
moby_dick.txt,pg-sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-
tom_sawyer.txt,pg-ulysses.txt,pg-war_and_peace.txt
wonderful: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-
emma.txt,pg-frankenstein.txt,pg-great_expectations.txt,pg-grimm.txt,pg-
huckleberry_finn.txt,pg-les_miserables.txt,pg-moby_dick.txt,pg-
sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-tom_sawyer.txt,pg-
ulysses.txt,pg-war_and_peace.txt
words: 15 pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-
frankenstein.txt,pg-great_expectations.txt,pg-grimm.txt,pg-
huckleberry_finn.txt,pg-les_miserables.txt,pg-metamorphosis.txt,pg-
moby_dick.txt,pg-sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-
tom_sawyer.txt,pg-ulysses.txt,pg-war_and_peace.txt
worked: 15 pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-
frankenstein.txt,pg-great_expectations.txt,pg-grimm.txt,pg-
huckleberry_finn.txt,pg-les_miserables.txt,pg-metamorphosis.txt,pg-
moby_dick.txt,pg-sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-
tom_sawyer.txt,pg-ulysses.txt,pg-war_and_peace.txt
worse: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-
emma.txt,pg-frankenstein.txt,pg-great_expectations.txt,pg-grimm.txt,pg-
huckleberry_finn.txt,pg-les_miserables.txt,pg-moby_dick.txt,pg-
sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-tom_sawyer.txt,pg-
ulysses.txt,pg-war_and_peace.txt
wounded: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-
emma.txt,pg-frankenstein.txt,pg-great_expectations.txt,pg-grimm.txt,pg-
huckleberry_finn.txt,pg-les_miserables.txt,pg-moby_dick.txt,pg-
sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-tom_sawyer.txt,pg-
ulysses.txt,pg-war_and_peace.txt
yes: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-
emma.txt,pg-great_expectations.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
les_miserables.txt,pg-metamorphosis.txt,pg-moby_dick.txt,pg-
sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-tom_sawyer.txt,pg-
ulysses.txt,pg-war_and_peace.txt
younger: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-
emma.txt,pg-frankenstein.txt,pg-great_expectations.txt,pg-grimm.txt,pg-
huckleberry_finn.txt,pg-les_miserables.txt,pg-moby_dick.txt,pg-
sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-tom_sawyer.txt,pg-
ulysses.txt,pg-war_and_peace.txt
yours: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-
emma.txt,pg-frankenstein.txt,pg-great_expectations.txt,pg-grimm.txt,pg-
huckleberry_finn.txt,pg-les_miserables.txt,pg-moby_dick.txt,pg-
sherlock_holmes.txt,pg-tale_of_two_cities.txt,pg-tom_sawyer.txt,pg-
ulysses.txt,pg-war_and_peace.txt
```

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

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

```
$ sh ./test-ii.sh
```

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

Resources and Advice

- The master 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 and the Go RPC documentation.

- The master may have to wait for a worker to finish before it can hand out more tasks. You may find channels useful to synchronize threads that are waiting for reply with the master once the reply arrives. Channels are explained in the document on Concurrency in Go.
- The code we give you runs the workers as threads within a single process, and can exploit multiple cores on a single machine. Some modifications would be needed in order to run the workers on multiple machines communicating over a network. The RPCs would have to use TCP rather than UNIX-domain sockets; there would need to be a way to start worker processes on all the machines; and all the machines would have to share storage through some kind of network file system.
- The easiest way to track down bugs is to insert `debug()` statements, set `debugEngabled = true` in `mapreduce/common.go`, collect the output in a file with, e.g., `go test -run TestBasic mapreduce/... > out`, and then think about whether the output matches your understanding of how your code should behave. The last step (thinking) is the most important.
- When you run your code, you may receive many errors like `method has wrong number of ins`. You can ignore all of these as long as your tests pass.

Point Distribution

Test	Points
TestBasic	15
OneFailure	10
ManyFailure	10
test-ii.sh	15

Submission

You hand in your assignment as before.

```
$ git commit -am "[you fill me in]"
$ git tag -a -m "i finished assignment 1-3" a13-handin
$ git push origin master a13-handin
```

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

```
$ git tag -fam "i finished assignment 1-3" a13-handin
$ git push -f --tags
```

You should verify that you are able to see your final commit and tags on the Github page of your repository for this assignment.

You will receive full credit for Part C if your software passes `TestBasic` in `test_test.go` on the CS servers. You will receive full credit for Part D if your software passes the tests with worker failures (`TestOneFailure` and `TestManyFailures` in `test_test.go`) on the CS servers. You will receive full credit for Part E if your index output matches the correct output when run on the CS servers (as in `test-ii.go`).

Acknowledgements

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