6.824 - Spring 2016

6.824 Lab 1: MapReduce

Due: Fri Feb 12, 11:59pm

Introduction

In this lab you'll build a MapReduce library as a way to learn the Go programming language and as a way to learn about fault tolerance in distributed systems. In the first part you will write a simple MapReduce program. In the second part you will write a Master that hands out tasks to workers, and handles failures of workers. The interface to the library and the approach to fault tolerance is similar to the one described in the original MapReduce paper.

Collaboration Policy

You must write all the code you hand in for 6.824, except for code that we give you as part of the assignment. You are not allowed to look at anyone else's solution, and you are not allowed to look at code from previous years. You may discuss the assignments with other students, but you may not look at or copy each others' code. Please do not publish your code or make it available to future 6.824 students -- for example, please do not make your code visible on github.

Software

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

We supply you with parts of a MapReduce implementation that supports both distributed and non-distributed operation (just the boring bits). You'll fetch the initial lab software with git (a version control system). To learn more about git, look at the git user's manual, or, if you are already familiar with other version control systems, you may find this CS-oriented overview of git useful.

The URL for the course git repository is `git://g.csail.mit.edu/6.824-golabs-2016`. To install the files in your Athena account, you need to *clone* the course repository, by running the commands below. You must use an x86 or x86_64 Athena machine; that is, `uname -a` should mention `i386 GNU/Linux` or `i686 GNU/Linux` or `x86_64 GNU/Linux`. You can log into a public i686 Athena host with `athena.dialup.mit.edu`.

```
$ add git # only needed on Athena machines
$ git clone git://g.csail.mit.edu/6.824-golabs-2016 6.824
$ cd 6.824
$ ls
Makefile src
```

Git allows you to keep track of the changes you make to the code. For example, if you want to checkpoint your progress, you can *commit* your changes by running:

```
$ git commit -am 'partial solution to lab 1'
```

The Map/Reduce implementation we give you has support for two modes of operation, *sequential* and *distributed*. In the former, the map and reduce tasks are all executed in serial: first, 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 distributed mode runs many worker threads that first execute map tasks in parallel, and then reduce tasks. This is much faster, but also harder to implement and debug.

Preamble: Getting familiar with the source

The mapreduce package provides a simple Map/Reduce library with a sequential implementation. Applications should normally call Distributed() [located in master.go] to start a job, but may instead call Sequential() [also in master.go] to get a sequential execution for debugging purposes.

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 `master_rpc.go`), and waits for workers to register (using the RPC call `Register()` [defined in `master.go`]). As tasks become available (in steps 4 and 5), `schedule()` [`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()` [`common_map.go`] at least once for each task. It does so either directly (when using `Sequential()`) or by issuing the `DoTask` RPC on a worker [`worker.go`]. 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, there will be #files x `nReduce` files after all map tasks are done:

   ```
   f0-0, ..., f0-0, f0-, ...,
   f<#files-1>-0, ... f<#files-1>-.
   ```

4. The master next makes a call to `doReduce()` [`common_reduce.go`] at least once for each reduce task. As for `doMap()`, it does so either directly or through a worker. `doReduce()` collects nReduce reduce files from each map (`f-*-`), and runs the reduce function on those files. This produces `nReduce` result files.
5. The master calls `mr.merge()` [`master_splitmerge.go`], which merges all the `nReduce` files produced by the previous step into a single output.
6. The master sends a Shutdown RPC to each of its workers, and then shuts down its own RPC server.

> **Note:** Over the course of the following exercises, you will have to write/modify `doMap`, `doReduce`, and `schedule` yourself. These are located in `common_map.go`, `common_reduce.go`, and `schedule.go` respectively. You will also have to write the map and reduce functions in `../main/wc.go`.

You should not need to 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.

Part I: 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 `common_map.go`, and the `doReduce()` function in `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, run:

```
$ cd 6.824
$ export "GOPATH=$PWD"  # go needs $GOPATH to be set to the project's working directory
$ cd "$GOPATH/src/mapreduce"
$ setup ggo_v1.5
$ go test -run Sequential mapreduce/...
ok      mapreduce       2.694s
```

> TASK
>
> You receive full credit for this part if your software passes the Sequential tests (as run by the command above) when we run your software on our machines.

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 `common.go`, and add `-v` to the test command above. You will get much more output along the lines of:

```
$ env "GOPATH=$PWD/../../" go test -v -run Sequential mapreduce/...
=== RUN   TestSequentialSingle
master: Starting Map/Reduce task test
Merge: read mrtmp.test-res-0
master: Map/Reduce task completed
--- PASS: TestSequentialSingle (1.34s)
=== 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 (1.33s)
```

```
PASS
ok      mapreduce       2.672s
```

Part II: Single-worker word count

Now that the map and reduce tasks are connected, we can start implementing some interesting Map/Reduce operations. For this lab, we will be implementing word count — a simple and classical Map/Reduce example. Specifically, your task is to modify `mapF` and `reduceF` so that `wc.go` 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 ~/6.824/src/main, downloaded from Project Gutenberg. Go ahead and try to compile the initial software we provide you and run it with the provided input files:

```
$ cd 6.824
$ export "GOPATH=$PWD"
$ 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()`) and 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"
$ time 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
14.59user 3.78system 0:14.81elapsed
```

The output will be in the file "mrtmp.wcseq". You can remove the output file and all intermediate files with:

```
$ rm mrtmp.*
```

Your implementation is correct if the following command produces 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
```

To make testing easy for you, run:

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

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

TASK

You receive full credit for this part if your Map/Reduce word count output matches the correct output for the sequential execution above when we run your software on our machines.

- **Hint:** a good read on what strings are in Go is the Go Blog on strings.
- **Hint:** you can use `strings.FieldsFunc` to split a string into components.
- **Hint:** the strconv package (http://golang.org/pkg/strconv/) is handy to convert strings to integers etc.

Part III: 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 above, 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 lab, you will complete a version of MapReduce that splits the work up over a set of worker threads, in order to exploit multiple cores. While 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 lab 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. `mapreduce.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 I, 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:

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

TASK

You receive full credit for this part if your software passes `TestBasic` from `test_test.go` (the test you run with the command above) when we run your software on our machines.

- **Hint:** 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.
- **Hint:** 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.
- **Hint:** The easiest way to track down bugs is to insert `debug()` statements, collect the output in a file with `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.

Note: The code we give you runs the workers as threads within a single UNIX 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.

Part IV: 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.)

> **Note:** 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 later labs are 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. To run these tests:

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

> TASK
>
> You receive full credit for this part if your software passes the tests with worker failures (those run by the command above) when we run your software on our machines.

Part V: Inverted index generation (optional)

> CHALLENGE
>
> 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 challenge 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:

```
$ 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,
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
```

For full credit on this challenge, you must pass `test-ii.sh`, which runs:

```
$ 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.
won: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-frankenstein.txt,pg-great_e
wonderful: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankens
words: 15 pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great_expecta
worked: 15 pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great_expec
worse: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.
wounded: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenste
yes: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-great_expectat
younger: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenste
yours: 15 pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.
```

Running all tests

You can run all the tests by running the script `src/main/test-mr.sh`. With a correct solution, your output should resemble:

```
$ sh ./test-mr.sh
==> Part I
ok      mapreduce       3.053s

==> Part II
Passed test

==> Part III
ok      mapreduce       1.851s

==> Part IV
ok      mapreduce       10.650s

==> Part V (challenge)
Passed test
```

Handin procedure

Important:

Before submitting, please run *all* the tests one final time. **You** are responsible for making sure your code works.

```
$ sh ./test-mr.sh
```

Submit your code via the class's submission website, located at https://6824.scripts.mit.edu:444/submit/handin.py/.

You may use your MIT Certificate or request an API key via email to log in for the first time. Your API key (`XXX`) is displayed once you logged in, which can be used to upload lab1 from the console as follows.

```
$ cd "$GOPATH"
$ echo XXX > api.key
$ make lab1
```

Important:

Check the submission website to make sure you submitted a working lab!

Note: You may submit multiple times. We will use the timestamp of your **last** submission for the purpose of calculating late days.

Please post questions on Piazza.