COMS 4113 Homeworks

Mingen Pan

Agenda

- Overview of homeworks (deadlines, grading, related topics, difficulty)
- Introduction to Go
- Introduction to MapReduce

Homeworks

8 Homeworks in total (excluded HW0).

Homework	Submission Deadline	Weights
HW0	W 9/16 (<1 week)	0 (but required)
HW1	W 9/23 (1 week)	12.5%
HW2a	W 10/7 (2 weeks)	12.5%
HW2b	W 10/14 (1 week)	12.5%
HW3a	W 10/28 (2 weeks)	12.5%
HW3b	Th 11/5 (1 week + 1 day)	12.5%
HW4a	W 11/18 (2 weeks - 1 day)	12.5%
HW4b	W 12/2 (2 weeks)	12.5%
HW5	W 12/16 (2 weeks)	12.5%

Grading

- Unit tests are used to grade your assignments.
- Unit tests in the same homework have the same scores (if 10 unit tests, then each contribute to 1.25%).
- Each unit tests will be run 50 times. Every time a unit test fails, the score of this unit test will be multiplied by 0.9.
- Grading machines are run in Linux with Go version of 1.13.

#fails	0	1	2	3	4	5	6	7	8	25
score	100%	90%	81%	73%	66%	59%	53%	48%	43%	7%

Related Topics

Homework	Project	Related Topics
HW1	MapReduce	MapReduce, RPC
HW2	Primary/Backup Server	Fault Tolerant
HW3	Paxos and KV Database	Consensus, Paxos, Availability
HW4	Sharded KV Database	Scalability, Paxos, Atomic commitment.
HW5	Coming soon	

Difficulty

- HW1 << HW2, HW3, HW4
- Part a < Part b

Tips

- Read papers and understand the protocol before coding.
- Frequently print your results when debugging a distributed system.
- Start your part b before the deadline of part a.
- If you are struggling on HW1, you need to schedule more time on the remaining assignments.
- Reference: 5 - 15 hours a week (Instagram time excluded).

Go

- Go is a statically typed, compiled programming language designed at Google. Go is syntactically similar to C, but with memory safety, garbage collection, structural typing, and CSP-style concurrency.

- More and more popular: Docker, Kubernetes, InfluxDB, ...

- Take the tour before you code: https://golang.org/doc/

Go

Some useful features that may help you in the assignments:

- go - command to run a function asynchronously.
- chan - communicate between different threads.
- sync.Mutex - structure to provide lock service
- defer - defer close right after open, defer unlock right after lock.
- Be careful about pointers and values.
- Refer to the godoc when you use built-in packages.

Go

Coding time

https://gist.github.com/mingen-pan/170a79709065df8ebd7a84149e1c9016

MapReduce

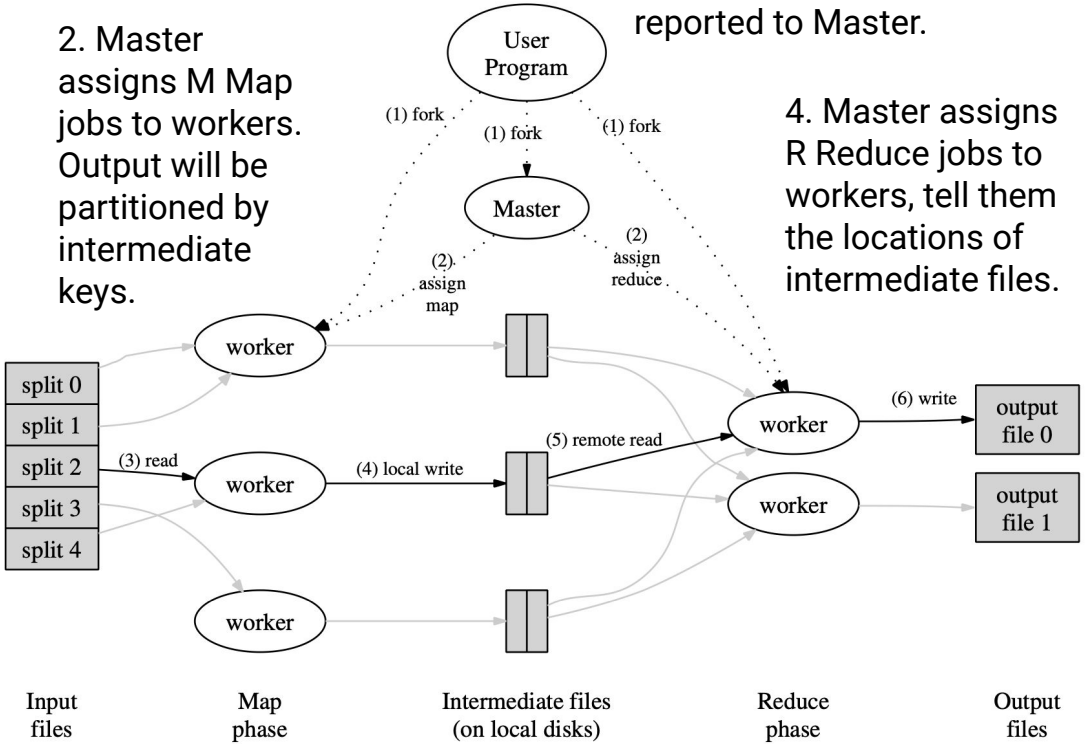
1. Split the file in M chunks.

Define Map and Reduce Functions.

2. Master assigns M Map jobs to workers. Output will be partitioned by intermediate keys.

3. There are MxR intermediate results from Map phases. Their location are reported to Master.

4. Master assigns R Reduce jobs to workers, tell them the locations of intermediate files.



5. Reduce jobs output R final outputs.

From the MapReduce paper (link)

MapReduce Example: word count

Split file: input-1.txt, input-2.txt, ..., input-m.txt

Map Phase: "..., a dog, a cat ..." => {"a": ["1", "1"], "dog": ["1"], "cat": ["1"], ...}

Intermediate files: file-1-1.txt {"a": ["1", "1"], "cat": ["1"], ...}, file-1-2.txt {"dog": ["1"]}

Reduce Phase: {"a": ["1", "1"], "cat": ["1"], ...}, {"a": ["1", "1", "1"], "apple": ["1", "1"]} => {"a": "5"}, {"apple": "2"}, {"cat": "1"}, ...

Output: output-1.txt, output-2.txt, ..., output-r.txt

How to start?

1. Start with common.go.
Get familiar with the RPC signature.

```go
type DoJobArgs struct {
  File string
  Operation JobType
  JobNumber int       // this job's number
  NumOtherPhase int   // total number of jobs in other phase (map or reduce)
}

type DoJobReply struct {
  OK bool
}

type ShutdownArgs struct {
}

type ShutdownReply struct {
  Njobs int
  OK bool
}
```

RPC exposed by Worker

```go
type RegisterArgs struct {
  Worker string
}

type RegisterReply struct {
  OK bool
}
```

RPC exposed by Master

2. Understand the code in worker.go and logic in mapreduce.go.

```go
type Worker struct {
  name string
  Reduce func(string, *list.List) string
  Map func(string) *list.List
  nRPC int
  nJobs int
  l net.Listener
}

// The master sent us a job
func (wk *Worker) DoJob(arg *DoJobArgs, res *DoJobReply) error {
  fmt.Printf("Dojob %s job %d file %s operation %v N %d\n",
             wk.name, arg.JobNumber, arg.File, arg.Operation,
             arg.NumOtherPhase)
  switch arg.Operation {
  case Map:
    DoMap(arg.JobNumber, arg.File, arg.NumOtherPhase, wk.Map)
  case Reduce:
    DoReduce(arg.JobNumber, arg.File, arg.NumOtherPhase, wk.Reduce)
  }
  res.OK = true
  return nil
}

// The master is telling us to shutdown. Report the number of Jobs we
// have processed.
func (wk *Worker) Shutdown(args *ShutdownArgs, res *ShutdownReply) error {
  DPrintf("Shutdown %s\n", wk.name)
  res.Njobs = wk.nJobs
  res.OK = true
  wk.nRPC = 1   // OK, because the same thread reads nRPC
  wk.nJobs--   // Don't count the shutdown RPC
  return nil;
}
```

How to start?

3. Implement the Master following the protocol from the paper. You need to write in both master.go and mapreduce.go

4. Finally, run each test cases 50 times to make sure your code is correct.

```go
func (mr *MapReduce) RunMaster() *list.List {
  // Your code here
  return mr.KillWorkers()
}
```

```go
type MapReduce struct {
  nMap int // Number of Map jobs
  nReduce int  // Number of Reduce jobs
  file string  // Name of input file
  MasterAddress string
  registerChannel chan string
  DoneChannel chan bool
  alive bool
  l net.Listener
  stats *list.List

  // Map of registered workers that you need to keep up to date
  Workers map[string]*WorkerInfo

  // add any additional state here
}
```

Questions?

Thank you for listening

Feel free to ask me any questions