# Assignment 2 (Map Reduce)
# Due: Tuesday, March 5th, 11:59 PM

**Overview.** In this assignment, you will complete the implementation of:

- A sequential map-reduce library that can support any arbitrary map and reduce functions. (Task 1)
- A word-count map-reduce application. (Task 2)
- A distributed implementation of map-reduce that is tolerant to worker failures. (Task 3)

The starting code can be found in this directory. The code is structured as follows:

- mapreduce package folder.
  - This is where you will develop Tasks 1 and 3.
  - Task 1 will be developed in map_reduce.go
  - Task 3 will be developed in schedule.go
- main folder
  - This is where you will develop Task 2 (as a programmer who is using the library to write a map-reduce application without worrying about the underlying details).
  - The code of Task 2 should be added to word_count.go

**Acknowledgement.** This assignment is an *adapted version* of assignments used in Princenton's COS418 and MIT 6.824 distributed systems courses.

## Detailed Description

The given mapreduce package provides a simple mapreduce library with both a sequential and distributed implementation. Normally, the applications should call Distributed() [located in master.go] to run a map-reduce job. However, you will start with the Sequential() variant [also in master.go] to make things simpler.

The following gives a high-level overview of the flow, and how the tasks you will complete are connected together. The steps here do not reflect the order of the tasks you will work on. It reflects how to use the library and what is happening in the background in order.

> **Step 1:** The application programmer will provide a list of input files and define a custom map and reduce functions as in the paper in addition to the number of reduce tasks. The number of map tasks will be equal to the number of input files. (You will simulate the role of the application programmer later in Task 2 by writing a word count application after completing the base library in Task 1.)
>
> **Step 2:** The inputs in step 1 are passed to a master. More specifically, a path to the directory that has the input files, the defined map and reduce functions, and the number of reduce tasks will be passed. See the main() function in main/word_count.go.
> The sequential variant of the master will run the map and reduce jobs sequentially.
> The distributed variant of the master spins up an RPC server (see master_rpc.go), and waits for workers to register (using the RPC call Register() [defined in master.go]). The

schedule() method in [schedule.go] decides how to assign the map and reduce tasks to workers, and how to handle worker failures. Working on this will be deferred to Task 3.

**Step 3:** The master makes a call to runMapTask() [map_reduce.go] at least once for each task. It does so either directly (when using Sequential()) or by issuing the RunTask RPC on a worker [worker.go]. Each call to runMapTask() reads the appropriate file, calls the map function on that file's contents, and produces nReduce files for each map file. The total number of files that will be generated after all map tasks will be (nReduce * the number of input files). The name of each output file will have the index of the map task that generated it, and the index of the reducer.

**Step 4:** The master next makes a call to runReduceTask() [map_reduce.go] at least once for each reduce task. As for runMapTask(), it does so either directly or through a worker. runReduceTask() collects nReduce reduce files from each map, and runs the reduce function on those files. This will produce nReduce result files.

**Step 5:** The master calls mr.merge() [master_splitmerge.go], which merges all the nReduce files produced by the previous step into a single output.

**Step 6:** The master sends a Shutdown RPC to each of its workers, and then shuts down its own RPC server.

## Getting started with the code

In order to run the test commands of task 1 and the code in the next tasks, you will need to properly setup module and workspace files. For example, this can be done by `go mod init mapreduce` (from the mapreduce directory) and `go work init ./mapreduce`. (from the asg2 directory). See https://go.dev/doc/tutorial/workspaces if you need more information.

## Task 1

In the sequential implementation, all the map and reduce tasks are run by the master process. Before writing the word-count map reduce application in Task 2, the sequential implementation will need to be completed. More specifically, the functions runMapTask and runReduceTask in map_reduce.go will need to be written.

**Task 1.1:** write runMapTask()
This function runs all the steps of a map task:

1. It reads one of the input files (the file given by inputFile),
2. calls the user-defined map function (mapFn) on that file's contents,
3. and partitions the output into nReduce intermediate files stored locally on disk.

```
func runMapTask(
    jobName string,   // The name of the whole mapreduce job
    mapTaskIndex int, // The index of the map task
    inputFile string, // The path to the input file assigned to this
task
    nReduce int,      // The number of reduce tasks that will be run
    mapFn func(file string, contents string) []KeyValue,
                      // The user-defined map function
) {}
```

**Guidelines for runMapTask():**

- Steps 1 and 2 should be straightforward.
- Note that you do not have to worry about what the user-defined mapFn does internally. It will just return a key-value map that you can use independently.
- To call mapFn, the first argument is the name of the file that is being processed, and the second argument is the file's contents.
- There are several issues that need to be taken care of in step 3.
    - The output of the mapFn will need to be stored in multiple files. The number of files equals the number of reduce tasks.
    - To find the name of the file corresponding to reduce task i, call getIntermediateName(jobName, mapTaskNumber, i)
    - To know which file you should write a (key, value) pair to, use the hash32 function define in the code, but note that it returns a 32-bit integer
    - Finally, the format to use while writing the (key, value) pairs to the intermediate files should be JSON. It is straightforward to do this in Go

        Example (You will need to import "encoding/json")

        file, err := os.Create(filename)

        encoder := json.NewEncoder(file)

        err := encoder.Encode(&kv)  where kv is a key value pair

        The above code shows how to have an encoder for one file, but recall that you will write to multiple files.
    - Remember to close the files at the end and to handle any possible errors.

**Task 1.2:** write runReduceTask()

This function runs the steps of a reduce task:

1. It reads the intermediate files produced by the mappers for reduce task # reduceTaskIndex,
2. groups all the received intermediate key-value pairs by key,
3. calls the user-defined reduce function (reduceFn) for each key,
4. and writes the output (key, value) pairs to disk. The output should be sorted by key.

```
func runReduceTask(
    jobName string,      // The name of the whole MapReduce job
    reduceTaskIndex int, // The index of the reduce task
    nMap int,            // The number of map tasks that were run
    reduceFn func(key string, values []string) string,
) {   }
```

**Guidelines for runReduceTask():**

- Step 1:
    - Assume the files are on disk. You can get the names of the files that should be read using the same method that you used to generate them during the map phase, i.e., via getIntermediateName
    - As you used Json to encode the intermediate results during the map phase, you will need to decode them in this phase:
      Example:
      decoder := json.NewDecoder(file)
      var kv KeyValue
      err := decoder.Decode(&kv)
      You can repeat the above till decode returns an error, which will indicate that there are no more records to read.
- Step 2: You can do the grouping while reading from the files (step 1).
- Step 4: All the output key value pairs should appear in a single file. The name of the file can be obtained by getReduceOutName(jobName, reduceTaskIndex). The output key value pairs should be encoded as Json files as done earlier in the map phase. You can use native sorting to sort the keys.

**Testing of Task 1**

The provided test cases test the sequential implementation through automatic generation of input files, running the sequential implementation and checking the outputs.

In the asg2 folder, you can test your implementation via these commands:

```
go test -run TestSequentialSingle  mapreduce/...

go test -run TestSequentialMany  mapreduce/...
```

You can add -v to view printed messages.
To disable caching of test results, you can add -count=1.
To see debugging messages, see debugEnabled in common_mr.go.

## Task 2

In this task, you will complete the implementation of a word count application based on the map-reduce paradigm. You will write the mapFn and reduceFn in main/word_count.go.

**Guidelines:**

- `func mapFn(docName string, value string) (res []mapreduce.KeyValue)`
  - This function is expected to do the same as the paper's wordcount map example, i.e., include (w, 1) in the output, but for words that have 8 letters or more
  - Combining is optional
  - The input to this function is the document name and its content. Note: you won't need the documentName while implementing the function.
  - The output of this function should be a slice of key/value pairs, each of type mapreduce.KeyValue.
- `func reduceFn(key string, values []string) string {`
  - This function is called once for each key generated during the map phase.
  - The second parameter is a list of the string values that correspond to that key after collecting them from the mappers.
  - The return value should be a single output value for that key, representing the frequency of the word.

**Running the code**

Note: You won't be able to test the code without completing task 1.

From the main folder, run

```
go run word_count.go master sequential papers
```

Input dataset: The text of the most cited 10 papers in security conferences. These are included in the papers folder. To check your implementation, we are providing the top 8 words with their frequencies in the given mr-wcnt-expected.txt file.

In order to check the most common words in the output, you will need to sort the output file (mrtmp.wcnt_seq) by value. This does not have to happen using the code. You can do it externally or using command line on Linux for example, e.g., sort -n -k2 mrtmp.wcnt_seq | tail -8

## Task 3

To run the map-reduce job in a distributed manner, the Distributed() variant of the master will need to be completed. More specifically, the implementation of the schedule() function in schedule.go.

```
func (mr *Master) schedule(phase jobPhase) {
    var ntasks int
    var numOtherPhase int
    switch phase {
    case mapPhase:
        ntasks = len(mr.files)     // number of map tasks
        numOtherPhase = mr.nReduce // number of reducers
    case reducePhase:
        ntasks = mr.nReduce           // number of reduce tasks
        numOtherPhase = len(mr.files) // number of map tasks
    }

      // TODO
}
```

This function runs all the tasks for a given phase (either map or reduce) in a distributed manner. All the ntasks tasks have to be run by worker threads in parallel. The function should not return unless all of the tasks have been completed successfully.
Note that workers might fail, and that any given worker may help with multiple tasks.

Hints:

- The available workers can be found through the registerChannel of the master.
- The registerChannel provides the addresses of the available workers as strings.
- To ask a worker to run a job, use the call function (see definition in common_rpc.go) to issue an RPC call to the worker and pass the worker address obtained from the channel, worker function (RunTask), and a pointer to a RunTaskArgs object that has the task parameters. See the definition of RunTaskArgs in common_rpc.go
- Note that if the above call returned false, this will indicate worker failure. To make sure that the assigned task has been completed, you will need to reassign it to another available worker.
- If a worker completed a job successfully, it could re-registered again to indicate its availability.
- No need to worry about having two workers writing to the same file. This case will not appear in testing. Also, there is no need to consider the failure of the master.

**Testing Task 3**

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

go test -run TestOneFailure mapreduce/...

go test -run TestManyFailures mapreduce/...
```

The first will test the distributed implementation assuming no failures. The other two tests will test the distributed implementation in case of one worker failure and many worker failures.

You can also run the word count application in a distributed manner by using the commands in word_count.go and make sure that the result is similar to the sequential case, but you will have to run at least one worker in addition to the master. Note: for the given command to run background processes on Windows, you will need to use powershell (version 6 or higher).

**Submission**

- The submission form will be posted to the MS team of the course.
- Only the three files: map_reduce.go, word_count.go and schedule.go will be delivered. Don't edit any other code (unless you want to do add logging statements for debugging purposes)