Here, you will complete the distributed MapReduce mode to split work over a set of worker threads that run in parallel on multiple workers. While not distributed across multiple machines as in real MapReduce deployments, your implementation will use RPC to simulate distributed computation. The interface to the library and the approach to fault tolerance is similar to the one described in the original MapReduce paper (Links to an external site.) we discussed in class.

Your job is to implement **schedule()** within mapreduce/schedule.go. This function is called twice by the master for each MapReduce job, once for the Map phase and once for the Reduce phase. schedule()’s job is to distribute tasks to available workers. Usually, there will be more tasks than worker threads so schedule() must give each worker a sequence of tasks. The function should wait until all tasks have completed before returning.

To learn about the set of workers, schedule() reads off its registerChan argument.

The channel yields a string for each worker, containing the RPC address of the worker. While some workers may exist before schedule() is called and some may start while schedule() is running, all will appear on registerChan. **schedule() should use all the workers.**

schedule() tells a worker to execute a task by sending a RPC to the worker in the format of Worker.DoTask. This RPC’s arguments are defined by DoTaskArgs in mapreduce/common\_rpc.go. The File element is only used by Map tasks as the name of the file to read. schedule() can find these file names in mapFiles.

To send an RPC to a worker use the call() function in mapreduce/common\_rpc.go. The first argument of the call is the worker’s address, received from registerChan. The second argument should be “Worker.DoTask”. Finally, the third argument should be the DoTaskArgs structure and the last argument should be nil.

schedule() 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; see Concurrency in Go.

schedule() must wait for a worker to finish before it can give it another task. You may find Go's channels useful.

You may find sync.WaitGroup useful.