Development

TABLE OF CONTENTS

The skeleton code implements a basic client that can submit jobs to the coordinator and includes the entire RPC application definition for coordinators and workers to communicate with one another. A full implementation of worker processes has been provided as well. You will be implementing the coordinator to assign tasks to workers and complete submitted jobs in a fault-tolerant manner.

> Keep in mind that the RPC server used in this assignment is not multi-threaded, so you do not need synchronization.

Usage

Running `make` creates 3 binaries in the `bin/` folder: `mr-coordinator`, `mr-worker`, and `mr-client`. The intended flow for starting your own MapReduce cluster and submitting jobs to it is explained below (commands are run from the root directory of the homework):

> **NOTE**
> This procedure will not result in any output until you begin implementing your MapReduce system.

1. Allow the Docker container to run the `rpcbind` calls by executing `sudo service rpcbind start`. You won't need to run the command more than once, however, you will have to execute it everytime you restart your Docker workspace.

2 Start the coordinator by running the `mr-coordinator` binary with no arguments. Specifically, you can run the following:

```
./bin/mr-coordinator
```

3 Start one or more workers by running the `mr-worker` binary with no arguments in several terminal windows (or multiple times in the background of the same terminal). They will then connect to the coordinator's RPC server to request tasks and will die once the coordinator is killed.

We strongly recommend using `tmux`. Commands for running a single worker (should be run in multiple terminals) and several workers (in the background of one terminal) are provided below:

```
./bin/mr-worker                                  # Multiple terminals
for i in {1..5}; do (./bin/mr-worker &); done    # Single terminal (5 workers)
```

4 Submit one or more jobs by running `mr-client submit`. The argument parsing step has already been implemented for you. Usage details are shown below.

```
Usage: mr-client submit [OPTION...] [FILES]...
Submit a job to a MapReduce cluster

  -a, --app=APP                      The name of the MapReduce application
  -n, --num-reduce=NUM_REDUCE        The number of reduce tasks [default: 5]
  -o, --output-dir=OUTPUT_DIR        The directory in which output files should be stored
  -w, --wait                         Whether or not to wait until the job is finished
  -x, --args=ARGS                    Arguments to the supplied MapReduce application
  -?, --help                         Give this help list
      --usage                        Give a short usage message
```

The available options are:

`--app`

> Chooses which app to run on the chosen input files. For example, `--app wc` runs the word count application.

`--num-reduce`

> Number of reduce tasks to split key/value pairs into.

`--output-dir`

> Directory to write MapReduce output files to. **Make sure to not submit multiple jobs with the same output directory as this will lead to race conditions when reading from intermediate files.**

`--args`

> Auxiliary arguments for the supplied MapReduce application (for the grep application, this would be the word being searched for).

For example, to submit a word count job on the entire `gutenberg` dataset and wait for it to complete, you can run the following from the root directory of the homework:

```
./bin/mr-client submit -a wc -o out -w -n 10 data/gutenberg/*
```

5 Once the job is complete, the final output is postprocessed by invoking `mr-client process`. Usage details are shown below.

```
Usage: mr-client process [OPTION...]
Post-process the output of a completed MapReduce job.

  -a, --app=APP                    The name of the MapReduce application
  -n, --num-reduce=NUM_REDUCE      The number of reduce tasks [default: 5]
  -o, --output-dir=OUTPUT_DIR      The directory in which output files should be stored
  -?, --help                       Give this help list
      --usage                      Give a short usage message
```

The options are the same as above. Note that you do not need to specify input files here, however.

To process the example job submitted in the previous step, you would run the following from the root directory of the homework:

```
./bin/mr-client process -a wc -o out -n 10
```

To check if your word count output is correct, you can compare the output of using `mr-client process` with that of the provided `words` binary, which can be run with `./words [FILES]...`. Similarly, you can

use the `grep` command to validate your grep output (e.g. `grep test data/gutenberg/*` will search for lines containing `test` in the `gutenberg` dataset).

Testing and debugging

We generally recommend inserting `printf` statements to sanity-check that your code is correct. **Make sure to end your `printf` statements with a newline so that they are immediately flushed to the console.** If you make your code modular, you can also write your own unit tests to verify that the data structures you implement operate correctly.

For more general debugging tips from TAs for MIT's 6.824, see here and here.

When debugging the fault tolerance portion of this assignment, we recommend inserting `printf` and `sleep` statements before workers begin executing map tasks. You can then manually crash a worker before it completes a task by pressing `Ctrl-C`, and verify that the coordinator reassigns the appropriate task(s) to other workers. Specifically, you will want to add a `sleep` statement right after the `worker` calls `get_task_1` on line 23 of `worker/worker.c` in the starter code so that the worker crashes after it receives a task but before it executes it.

Also, make sure to try different cluster sizes (i.e. different numbers of workers) and different parameters (e.g. `n_map` and `n_reduce`) when trying to reproduce failing tests locally. Many of the tests test your code while varying all of these parameters, and certain errors may not appear for the default settings.

Detailed testing breakdown

While the autograder will provide you with basic information on what each test checks, it is sometimes difficult to figure out where to start debugging. Here are some additional details for each test as well as some hints on what issues may cause them to fail.

We recommend that you reference this section as little as possible since you will learn much more by coming up with potential issues with your implementation on your own, but we understand that sometimes it can be frustrating to debug opaque tests that are hard to reproduce locally. Refer to the hints below before you ask a question relating to autograder tests on Ed or in OH.

- `poll-bad-id`
- `wc`
- `grep`
- `vertex-degree`
- `map-parallel`

- `map-parallel-add-worker`
- `reduce-parallel`
- `reduce-parallel-add-worker`
- `mr-concurrent`
- `coordinator-no-work`
- `mr-queue-order`
- `queue-map-multiple`
- `mr-no-duplicates`
- `map-crash`
- `reduce-crash`
- `rand-crash`
- `crash-multi-job`
- `crash-no-duplicates`
- `submit-bad-app`
- `map-error`
- `reduce-error`

`poll-bad-id`

> Checks that polling for an invalid job ID returns an error without crashing the coordinator. The test also submits valid jobs and ensures that their outputs are correct even when interspersed with invalid `POLL_JOB` requests.

> Potential bugs:
>
> - Not checking that the job exists before trying to access its fields (Error: segmentation fault and RPC issues when trying to contact the crashed coordinator).
> - Incorrectly assigning job IDs (Error: IDs that should be invalid are valid).
> - Not returning `invalid_job_id = true` in the `POLL_JOB` RPC when the provided ID is invalid (Error: IDs that are invalid do not result in an error, or timeout if `done` is set to `false`).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`wc`

> Run the word count application. The test submits jobs with different types of inputs (small or large file sizes, a few or several input files, low or high `n_reduce`, etc.), so you may need to modify these parameters if you aren't able to replicate the failure using the default parameters.

> Potential bugs:
>
> - Incorrect memory access/management (Error: segmentation fault and RPC issues when trying to contact the crashed coordinator).
> - Incorrectly implementing Job submission.
>   - Not returning `done = true` in the `POLL_JOB` RPC when a job completed (Error: timeout).
>   - Returning `done = true` in the `POLL_JOB` RPC before a job is completed (Error: incorrect output).
> - Incorrectly implementing Tasks.
>   - Incorrectly assigning tasks.
>   - Assigning the same task multiple times (Error: incorrect output).
>   - Forgetting to assign a certain task, (Error: timeout or incorrect output).
>   - Not waiting for all map tasks to finish before assigning reduce tasks (Error: error opening intermediate files, segmentation fault in workers due to writes to file while reading).
>   - Incorrectly handling finished tasks.
>   - Not marking jobs as completed when all tasks are completed (Error: timeout).
>   - Marking jobs as completed before all tasks are completed (Error: timeout).

`grep`

> Run the grep application.

> Potential bugs:
>
> - Incorrectly passing `args` down to the workers (Error: incorrect output).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`vertex-degree`

> Run the vertex degree application.

> Potential bugs:
>
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`map-parallel`

> Check that map tasks can be run in parallel. Specifically, the test doesn't allow any map tasks to complete until all are assigned to a pool of workers that is the same size as the number of available map tasks.

> Potential bugs:
>
> - Waiting for a `FINISH_TASK` RPC for an assigned task before assigning new tasks (Error: timeout).
> - Assigning map tasks multiple times (Error: timeout).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`map-parallel-add-worker`

> Check that map tasks can be run in parallel, even when workers join the cluster after a job has started. Due to the nature of the RPCs, this test is effectively the same as `map-parallel`.

> Potential bugs:
>
> - Allowing reduce tasks to run before map tasks complete (Error: job completes without having enough workers to accept map tasks).
> - See `map-parallel`.

`reduce-parallel`

> Check that reduce tasks can be run in parallel. Specifically, the test doesn't allow any reduce tasks to complete until all are assigned to a pool of workers that is the same size as the number of available reduce tasks.

> Potential bugs:
>
> - Waiting for a `FINISH_TASK` RPC for an assigned task before assigning new tasks (Error: timeout).
> - Assigning reduce tasks multiple times (Error: timeout).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`reduce-parallel-add-worker`

> Check that reduce tasks can be run in parallel, even when workers join the cluster after a job has started. Due to the nature of the RPCs, this test is effectively the same as `reduce-parallel`.

> Potential bugs:
>
> - Marking job as complete without all reduce tasks being complete (Error: job completes without having enough workers to accept reduce tasks).

> - See `reduce-parallel`.

`mr-concurrent`

> Run multiple MapReduce jobs concurrently. The test submits several jobs and ensures that they all complete and produce the correct output.

> Potential bugs:
>
> - Incorrect queuing logic where jobs submitted while another job is running are not registered (Error: timeout or invalid job ID error).
> - Incorrect task assignment logic with multiple jobs.
>   - Finishing a task from job 2 marks a task from job 1 as completed (Error: timeout or incorrect output).
>   - Jobs are removed from the queue before they are completed (Error: timeout).
>   - Jobs are not removed from the queue when they are completed and tasks for later jobs are not assigned (Error: timeout).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`coordinator-no-work`

> Check that the coordinator does not execute MapReduce tasks.

> Potential bugs:
>
> - Calling worker functions from the coordinator directly instead of assigning tasks via RPCs.
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`mr-queue-order`

> Check that tasks are assigned to workers in the correct order. Jobs should run in the order they are submitted (i.e. tasks for job 1 should run before tasks for job 2).

> Potential bugs:
>
> - Assigning tasks in a random order or not keeping track of the order in which jobs are submitted (Error: incorrect task assignment order).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`queue-map-multiple`

> Check that map tasks for multiple jobs can run in parallel. Specifically, this tests that map tasks from job 2 are assigned to idle workers even while map tasks from job 1 are still in progress.

> Potential bugs:
>
> - Waiting on a job to complete before running tasks from later jobs (Error: timeout).
> - Waiting for the map stage of the previous job to complete before running the map stage of the next job (Error: timeout).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`mr-no-duplicates`

> Check that no tasks are assigned more than once.

> Potential bugs:
>
> - Incorrect fault tolerance logic that re-assigns tasks before the necessary timeout has expired (Error: task assigned multiple times).
> - Assigning tasks solely based on whether they have completed or not instead of keeping track of which tasks have alread ybeen assigned (Error: task assigned multiple times).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`map-crash`

> Check that MapReduce jobs complete even when a worker crashes during a map task. This test runs with several different worker pool sizes and a varying number of crashes.

> Potential bugs:
>
> - Incorrect fault tolerance logic.
> - Re-assigns tasks before the necessary timeout has expired (Error: incorrect output).
> - Does not re-assign a failed task after the necessary timeout has expired (Error: timeout).
> - Marks the job as failed if a `FINISH_TASK` RPC is not received instead of re-assigning the task (Error: job failed).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`reduce-crash`

> Check that MapReduce jobs complete even when a worker crashes during a reduce task. This test runs with several different worker pool sizes and a varying number of crashes.

> Potential bugs:
>
> - Incorrect fault tolerance logic (see `map-crash`).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`rand-crash`

> Check that MapReduce jobs complete even when workers crash randomly.

> Potential bugs:
>
> - Incorrect fault tolerance logic (see `map-crash`).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`crash-multi-job`

> Check that MapReduce jobs complete even when tasks from multiple jobs fail to complete.

> Potential bugs:
>
> - Fault tolerance logic only works for a single job due to keeping track of assigned tasks based only on task ID, rather than both task ID and job ID (Error: timeout).
> - Incorrect fault tolerance logic (see `map-crash`).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`crash-no-duplicates`

> Check that no extraneous tasks are scheduled when one or more workers crash.

> Potential bugs:
>
> - Fault tolerance logic does not keep track of whether failed tasks are reassigned (Error: task assigned multiple times).
> - Incorrect fault tolerance logic (see `map-crash`).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`submit-bad-app`

> Check that submitting a job with an invalid app results in an error.

> Potential bugs:

> - Returning a valid job ID (i.e. `job_id >= 0`) in the `SUBMIT_JOB` RPC even when a job with an invalid app is submitted.
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`map-error`

> Check that the MapReduce cluster tolerates map function errors.

> Potential bugs:
>
> - Ignoring the `success` flag in the `FINISH_TASK` RPC (Error: error reading temporary files).
> - Trying to reassign tasks with `success = false` even though these are permanent errors (Error: timeout).
> - Not replying with `done = true` and `failed = true` when `POLL_JOB` is called on a failed job (Error: timeout if `done = false`, missing error if `failed = false`).
> - Incorrectly implementing Job submission or Tasks (see `wc`).

`reduce-error`

> Check that the MapReduce cluster tolerates reduce function errors.

> Potential bugs:
>
> - Ignoring the `success` flag in the `FINISH_TASK` RPC (Error: error reading output files).
> - Incorrectly handling the `success` flag (see `map_error`).
> - Incorrectly implementing Job submission or Tasks (see `wc`).
