B534 : MapReduce

Siddharth Pathak

sidpath@iu.edu

February 11, 2019

Design:

Initializing:

The master initially spawns reducers and mappers as new processes using Python's multiprocessing module. Each mapper and reducer also starts a RPC server in a different thread and are ready to listen to incoming method calls. The master keeps track of the mappers and reducers started in a Python list. Master starts these new processes and servers on IP and port as per specified in the config file given by the user.

The user interacts with master using RPC calls. To initialize a cluster, master server supports `init_cluster` method over RPC which takes the path of config file as a parameter. Master also supports `destroy_cluster` method over RPC which destroys all the mappers and reducers and deletes their local files too.

Master to Mapper communication:

Once the cluster has been initialized, user can start a job on it via a RPC call to method `start_job`. The method takes config file path as a parameter.

Master splits the input file data evenly into chunks/sections depending on the number of mappers available in the cluster. The data which cannot be split evenly is assigned to the last mapper. The master tells each mapper about the lines in the files it has to work on, the files list, the mapper function name, master IP, and master port. Each mapper works on the same section (e.g. line 1-200) on each input file. The master also tells the number of reducers in the cluster to each mapper. It is required for partitioning the data evenly into files same as the reducers.

The master copies the input file to a location to which only the mappers have access to. Once copied, master calls the RPC method $start_working$ for each mapper. Once the mapper receives a request, it immediately responds to the master and starts a new thread which does works on the mapping job.

Mapper calculates the map of the data, combines it, and hashes each key and divides it into R(number of reducer) buckets. Once the keys are divided into buckets, each bucket is stored in a different file for easy retrieval by each reducer. The file is stored locally on mapper's disk using Python's pickle module. Also, each file contains the same key i.e the file 1 will contain the same keys as each mapper locally.

While the mapper jobs are running, master keeps checking every 4 seconds if the mappers are alive. It checks by sending a request to each mappers $check_alive$ method. If the request is not successful, then the whole cluster is destroyed and the job is restarted.

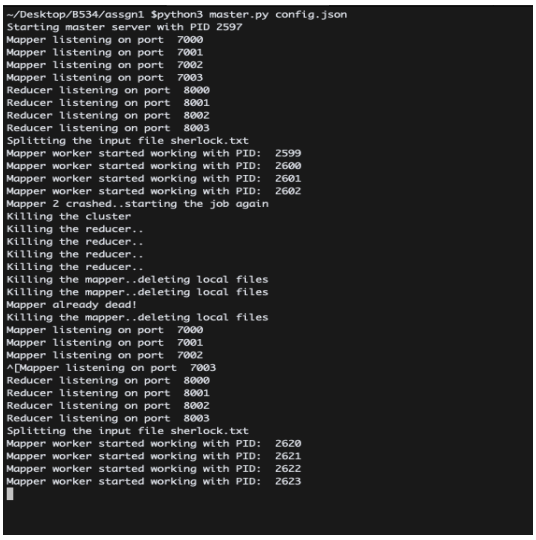
Mapper to Master communication: As each mapper completes its job, it responds back to master by calling master's method `send_keys` over RPC. It sends along the list of file it has generated as the map output.

Master to Reducer communication:
Master keeps a track of the number of mappers completed. Once the number of mappers completed as equal to the total mappers, it makes the reducers start their job. It assigns each file to each reducer, since the number of files are at most equal to the number of reducer. Because the mappers divided them into R buckets. If the number of files formed is less than the number of reducers, the master starts job only on that many number of reducers. Master calls the `start_working` method of reducer, and sends them the list of mappers along with their IP and port, and the file it has assigned to that particular reducer. It also sends the master's IP and port so the reducers can respond back later.
Once the reducer receives master's request, it similarly like mapper immediately responds back to the master, and starts the reducer job in a new thread.

Reducer to Mappers communication:
Since, master sent the list of mappers to each reducer, each reducer broadcasts a request to each mapper asking the data it has been assigned to. It calls the `get_keys` function along the file name as the parameter of each mapper, and calls the reducer function on the data. The mapper's `get_keys` function reads the data from the file, and returns it to the reducer. The data is essentially a list of key/value pairs.

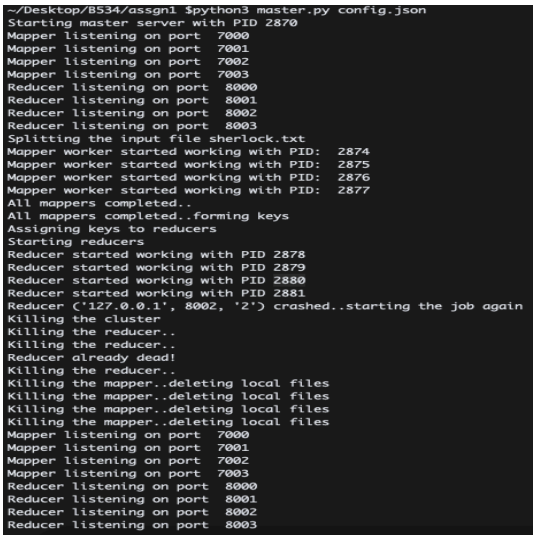
Reducer to Mapper communication:
Once reducer completes the job, it sends the list of key value pairs to master using the master's `send_reducer_keys`. The master receives the result of all the reducers, and responds back to the user with the output.

Fault Tolerance:
While the mappers or reducers are working, master sends a request to the `check_alive` method of each of them over RPC. If master gets a response, then it considers that they are working. If it doesn't get a response, it destroys all the mappers and reducers, and starts a new job.



Sample Fault Tolerance output (mapper)

For testing fault tolerance, I manually killed one of the mappers. Master was able to detect the crash and it restarted the whole job again.



Sample Fault Tolerance output (reducer)

Similarly, I killed one of the reducers and it detected it and restarted the whole job.

Config File:

Config file is a JSON file, which contains the master's IP, port, the list of mappers and reducers and their IP, port. It also contains the name of mapper function like `word_count` or `inverted_index`. Furthermore, it also contains the list of input files.

Testing:

For testing, I have provided a file `tester.py` which takes config file as input. It assumes that master server is running, so before running the tester file, the master server should be running.

Run the master server: `python3 master.py config.json`
Run the tester: `Python tester.py config.json`
The tester file compares the output of the map reducer job, to the same job if its run sequentially. If the output matches it prints out `Output Matched!`. The tester also initializes the cluster, based on the config file, and once the job is finished it destroys the cluster too.

The default config file options are 4 mappers, 4 reducers, 2 input files, and inverted index as the mapper function. The mapper function can be changed to `word_count` for performing word count.

Assumptions:

The MapReduce paper stores the output in R files in a global storage. While my framework returns the output to the master. In the paper, the size of key value pair is so huge, that the master can't store it in main memory.

The mappers and reducers access file in GFS using GFS client, but in the current implementation they access the file locally. But each mapper can access only its own file stored on a local directory, and can't access files stored in other directory.