Design Document - MapReduce

Nikita Agarwal | Varun Prasad | Ajinkya Ghadge

Overall Program Design

The following is a description of the different classes that are used in the project

- MapReduce.java contains the mapreduce interface that is to be implemented by all the test cases
- WordCount.java, DistributedGrep.java and URLfrequency.java are the test cases which implement this interface
- Master.java contains the implementation of the master. In this, the workers are created, and the global synchronization barrier is also implemented. We create separate processes for mapper phase and reducer phase. This is done through a mechanism called heartbeats, in which all the workers send a message to the master at regular intervals (2 sec), and the master assumes a worker has crashed if no message is received in that interval.
- WorkerHeartBeat.java extends the thread class, and is used to implement the heartbeat mechanism, so that workers can send messages to the master
- Worker.java contains the implementation of the workers. Here, all the file reading/writing operations are dealt with, and the heartbeat thread is started
- Finally, RunTests.java is used to run all the test cases, and implement fault tolerance (using class *TestFaultTolerance*)

The current implementation supports multiple workers for MapReduce. The test cases which it supports are:

Word Count: Count the words in the given text file
Distributed Grep: Find a specific pattern in the text file
URL Frequency count: Count the URL access frequency

System Specifications

- Windows - 64 bit system
- JAVA (JDK 14) (Set the path of JDK in gradle.properties file)
- Gradle 6.7 - the environment path variable also needs to be set (Download: link)
- JAVA_HOME: path variable set in the environment variables.

How to run the tests

Run the **createjar.bat** file. This will build the project using gradle, run it and then start all the tests (defined in the "test" folder). If the tests are passed/failed, a message will be displayed in the CLI that the respective test case is passed/failed.

The input, output and intermediate file locations are specified in the config files, which are wordcount.json, distributedgrep.json and urlfrequency.json respectively. The reason for having separate config files is because the outputs are dumped into different output folders, since we are running multiple tests at the same time and the output of all tests need to be displayed.

If you want to give an input file other than the one provided in the project directory, then please change the *inputfile* path in the respective config file. We faced some issues with different versions of Java. We have used jdk 14.0.2. If there are any issues faced with building gradle, the jdk file can be downloaded using this link. Also, in this case change the gradles.properties file to indicate the path to the jdk 14.0.2, for example: org.gradle.java.home=C:\\Program Files\\Java\\jdk14.0.2

How the tests work

The tests are started by running startTests.py present in the test folder. This file then runs wordCountTest.py, distributedGrepTest.py and URLFrequencyTest.py. All of these test files take the output of our MapReduce program as "actual output" and the actual results that are used for verification as "expected output ". These expected results are computed by the respective test python files. In this way, we are checking whether our distributed implementation of the tasks returns the same results as a non-distributed, single process implementation of the task.

Note: The tests need to be run from the root directory of the project, and not from the test folder, since the config files are present in the root directory.

How the project works

We have implemented a MapReduce interface which has the following function declarations:

- The *mapper* function of each worker takes in a row from the partition of the data that is assigned to that worker and gives out a list of key value pairs, which are then written to intermediate files
- The *reducer* function takes in the values as a list for a particular key and gives out the final result in string format. This values list is obtained after after applying the combiner function on the intermediate file data
- The *getResourceSeparator* function is used to obtain the separator that is used to separate rows in the input for each test case

The execution is started with the creation of a Master process. The master refers to the config file that is provided, and takes in the desired locations for the input file, output file and intermediate file, along with the value of N which defines the number of workers. The master then takes an object of the test case, which implements the MapReduce interface that our library defines, serializes that object and places it in a file. The worker processes are then created with unique IDs. A socket is initialized to facilitate communication between the master and all the workers, and the master constantly listens for messages from the workers. The file containing the serialized object, the communication port, and the file paths are passed to all of the workers. The workers are all initialised to perform the map phase first, and a global barrier is implemented at the master side. In this phase, the input files are taken as input and the intermediate files are considered as output files. The workers access the serialized object in read-only mode, deserialize it, start the worker heartbeat thread and then access the user-defined map function using that object. This is followed by the execution of the task specified in the UDF. Upon completion, each worker sends a message to the master on the designated port stating that it has completed its task. The master acknowledges, and maintains a tally

to ensure global synchronicity. The first phase ends once all the workers have sent a message to the master stating their tasks are completed.

The second phase is then started, and workers are created and initialised for the reduce phase. The same process described above is followed again, with the workers acting as reducers now. The intermediate files are considered as the input files for this phase, and the output files are the final output files.

Note: We are aware that in pure distributed fashion, the serialized object file will not be available to all the workers, but we can always send the serialized object to the worker nodes. Therefore, the code can generalize with few modifications.

Design Choices & Tradeoffs

In order to implement fault tolerance, we required the heartbeat mechanism between our master and workers. There were two ways to implement it: Either the client could send heartbeats at regular intervals or the master could ask the status at regular intervals. We chose to go with the option of the worker sending the heartbeat as this required usage of a lesser number of ports. The worker can simply connect to the port the master is listening to in the global barrier stage and write the message there. Also, this was a more feasible option to go with after implementing milestone 2. While implementing the heartbeat mechanism, we had to come up with the threshold after which the master concludes that the worker process has died and respawns it. We consider that we will not have any network delay in our system, and have chosen the threshold as 2000 ms which is equal to the interval within which the workers send heartbeats. This value could be altered depending on the network delay in a distributed system.

We had the option of dumping the combiner output to files, or we could keep it in the memory. For this project, we assumed that the combiner output would fit in memory and chose the in-memory combiner output approach. This also helped decide when to call the combiner function. It can be called after the mapping phase or just before the reducer phase. As we were using an in-memory approach, calling before the reduce phase was the best option.

While implementing different test cases, we were posed with the question of how to generalize the row separation for different types of resources. For example, for word count a row can be considered as anything separated on whitespace, for URL Frequency, it is usually a new line character. Therefore, we added a new method(getResourceSeperator()) in the MapReduce.java interface which returns the separator for each test case.

In numerous parts of the project, we have chosen approaches that favour ease of implementation. Ideally, these can be made more efficient. Thus, we have made small tradeoffs of simplicity for efficiency