COMPSCI 532: Systems for Data Science Fall 2021

Project 1 MapReduce Documentation

Mudit Chaudhary (32607978), Ayushe Gangal (33018381), Pragyanta Dhal (33069605)

1 How to run

NOTE: The Inter-process communicator uses port 9997. Please, keep that port free to run the program.

To run the program you need the following:

- **Input files**: User need to provide input files. The location of input files need to be specified in the configuration as shown later. We have provided some input files already for testing and grading purposes in *resources/Input_files* folder.
- **Configuration**: The user needs to provide configurations for each job in *resources/configs* folder. For running the program, the configuration need to be in java properties format as in the example below:

```
inputFile=resources/Input_files/wordCount.txt
outputFileDirectory=resources/Output_files/
num_workers=3
```

  It should have the input file location, output file directory, and the number of workers.

  For testing, there need to be an extra property for specifying the Spark Output file as shown in the example below:

```
inputFile=resources/Input_files/searchWord.txt
outputFileDirectory=resources/Output_files/
num_workers=1
sparkOutputFile=resources/spark_test_outputs/searchWord_testOut/part-00000
```

  We have provided the configs files in the *resources* folder.

- **User's code**: User code which uses our mapreduce library. We have provided user code in *src/com/compsci532/usercode/Main.java* which uses our library *com.compsci532.mapreduce*.

1.1 How to run tests

The grader or the user can use the provided script *compile_run_test.sh* to run the tests. We have described the tests in a later section.

The content of the script file is as follows:

```
rm -rf runMapReduceTest
mkdir runMapReduceTest
javac -sourcepath src -d runMapReduceTest -cp \
lib/junit-platform-console-standalone-1.8.1.jar:.src/main/java/**/**/**/*.java \
src/test/java/*.java
java -jar lib/junit-platform-console-standalone-1.8.1.jar \
--class-path runMapReduceTest --scan-class-path
rm -rf runMapReduceTest
```

The output from running the tasks of the tests will be saved in *resources/Output_files* directory.

1.2 How to run an example program

The grader or the user can use the provided script *compile_run_program.sh* to run an example program with 4 tasks. The content of the script file is as follows:

```
rm -rf runMapReduce
rm -rf resources/Intermediate_files/
mkdir runMapReduce
javac -sourcepath src -d runMapReduce src/main/java/**/**/**/*.java
java -cp runMapReduce com.compsci532.usercode.Main
```

The outputs from this example program will be saved in *resources/Output_files* directory.

2 How to define and use User-defined functions

First the user needs to import our mapreduce library – *com.compsci532.mapreduce*.

2.1 Mapper

The user needs to define their Mapper function by implementing the Mapper Interface from our library as follows:

```
public static class <Mapper Name> implements Mapper {

        public void map(String key, String value, MapResultWriter writer) throws IOException {
        // key is null
        // value is a single line from input
        // writer is a MapResultWriter object from our mapreduce library
        // Implement mapper function

            writer.writeResult(<key>, <value>);
        }
    }
```

2.2 Reducer

The user needs to define their Reducer function by implementing the Reducer Interface from our library as follows:

```
 public static class <Reducer Name> implements Reducer {

        public void reduce(String key, ArrayList<String> values, ReduceResultWriter writer) thi
            // key is the key from Mapper
            // values is a list of grouped values
            // writer is a ReduceResultWriter object from our mapreduce library

            writer.writeResult(<key>, <reduced result>);
        }
    }
```

2.3 Run job

To run the job the user needs to setup job configuration, setup master, and then run the job. An example has been shown below:

```
String wordCountConfig = Paths.get("resources", "configs", "config.properties")
.toString(); // Load configuration file path

JobConf myJobConfig = new JobConf( jobName, config); // Load job config
Master masterClient = new Master(); // Start Master

myJobConfig.setMapper(MapperCls.class);   // Set UDF Mapper function in job config
myJobConfig.setReducer(ReducerCls.class);  // Set UDF Reducer function in job config

masterClient.setJobConfig(myJobConfig);  // Set Job config for the master
masterClient.runJob();  // Master runs job based on job config
```

3 MapReduce Library Structure

Our MapReduce library implements the following:

- **JobConf**: Job configuration class that maintains the configuration for each job.
- **Master**: Master class that invokes worker process, handles failures, and performs worker state management.
- **Worker**: Worker class that executes user-defined Mapper and Reducer function.
- **MapResultWriter**: Class to write mapper's result to the appropriate partition according to the user key.

- **ReduceResultWriter**: Class to write reducer's result to output files.
- **HeartbeatServer**: Class for Java RMI for inter-process communication.
- **Mapper**: Interface for Mapper.
- **Reducer**: Interface for Reducer.
- **HeartbeatRMIInterface**: Interface for RMI for inter-process communication.

4 Program Flow

In this section, we explain the MapReduce flow of our library and the fault tolerance flow.

4.1 MapReduce Flow

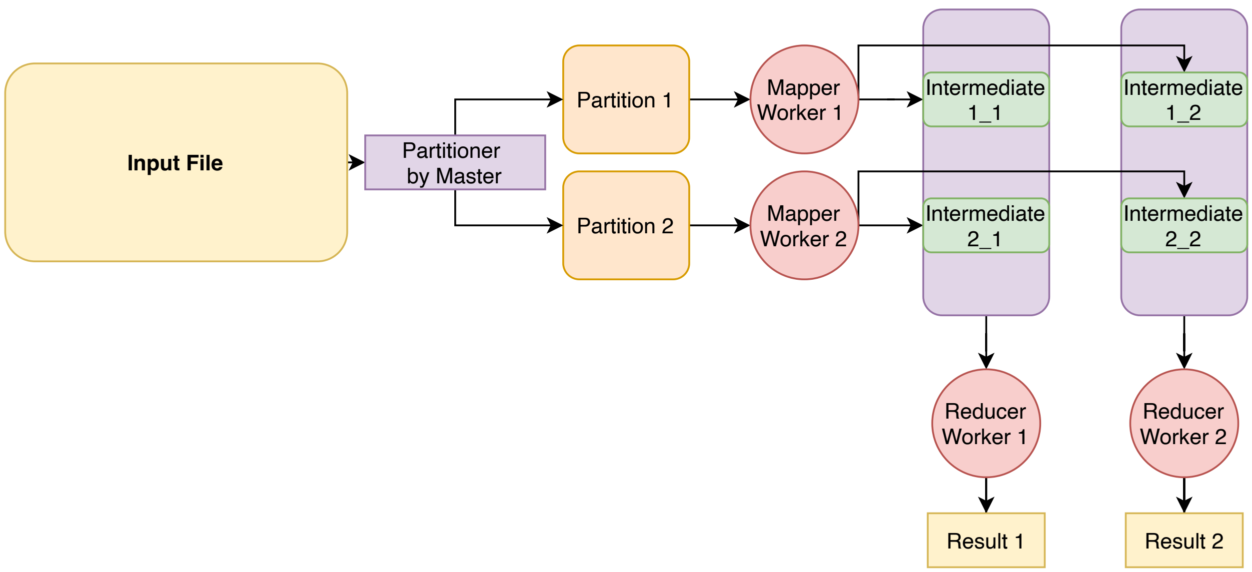

Figure 1: Map reduce program flow

In Fig. 1, we see the flow of our map reduce program. On receiving the input file, the master partitions it according to number of workers. Each mapper takes in one partition and produces intermediate files equal to number of workers. Each key-value pair is written in the intermediate file according to key's value using $hashCode()\%num_workers$. It is done so that there is no overlap of keys across intermediate file partitions. Each reducer takes one partition from each mapper, groups the values per key and then performs the user-defined reducer function to give the final output.

4.2 Fault tolerance Flow

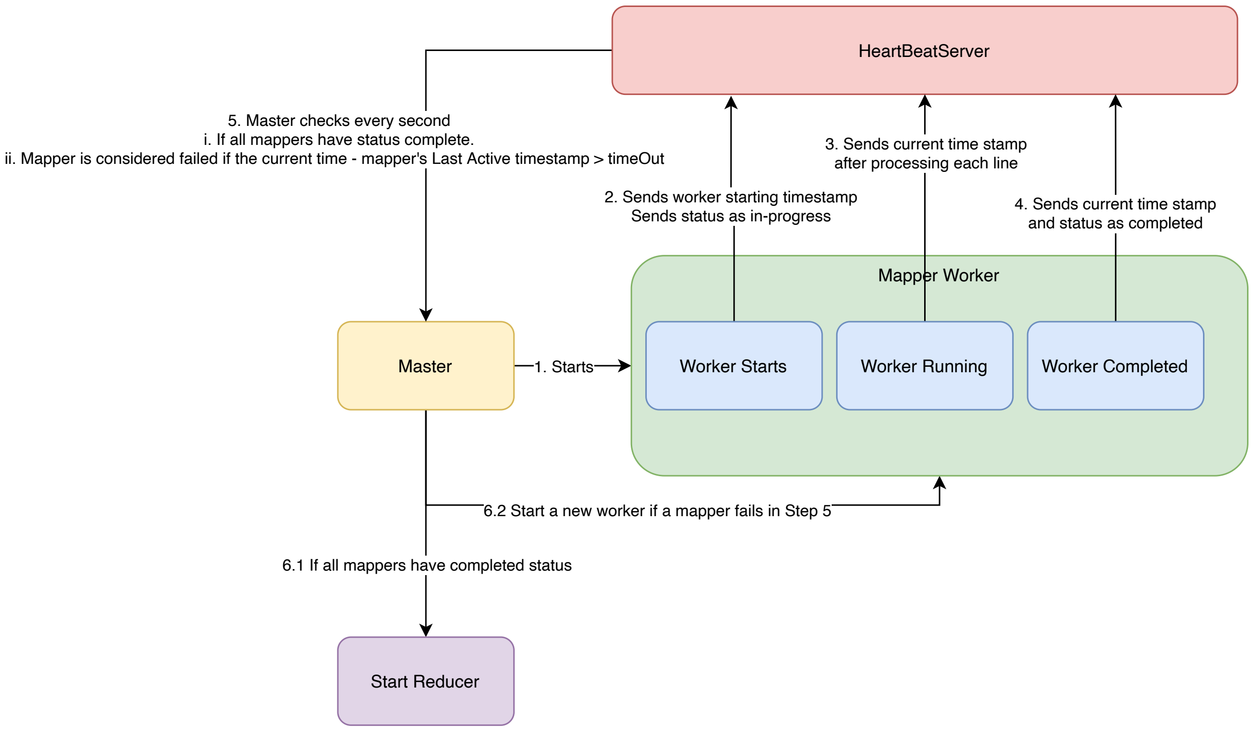

Figure 2: Fault Tolerance flow

We present our fault tolerance flow using example of 1 mapper in Fig. 2. When master creates a worker, the worker sends a heartbeat message to the server. After the initial message, it sends a message to the server after processing every line (for mapper) or key (for reducer) with the status and the timestamp. If the worker completes its job, it sends a final message with status as 'completed'. Every second, the master checks the heartbeat message. If all the mappers have completed status, it starts the reduce phase. A worker is considered failed if the $current\_time - lastActiveWorkerTimestamp > timeOut$. The $lastActiveWorkerTimestamp$ is obtained from the worker's latest timestamp in the heartbeat message. When a failed worker is detected, the master stores this info and spawns a new worker process with the same input partitions as the failed worker.

5 Design considerations and trade-offs

We made the following trade-offs for offering simplicity:

- **Mapper works only line-by-line**: For simplicity, mapper does not take *key*. Rather, it only gives each line of the input file as *value* on which the user can perform a map function.
- **Heartbeat message at every line or key**: Heartbeat message are sent everytime a mapper processes a line or a reducer processes a key. It leads to sending many heartbeat messages but it offers simplicity in terms of implementation.

- **Partitioned input is stored on-disk**: The input file is partitioned according to the number of workers. These partitions are stored on-disk, which are then processed by the mapper. This might be an issue, if the available disk space is less than twice the size of the input file.

6 Tests

We provide the tests on the following tasks:

- **Word Count:** Given a text file, count instances of each word.
- **Average Stock Price:** Given a file with stocks and their prices on different dates, find the average for each stock.
- **Search Word:** Given a text file, check if the given word appears.

For each of the above tasks, we perform the following tests:

- **Single process**: A single mapper process and a single reducer process run the tasks.
- **Multiple process**: Multiple mapper processes and multiple reducer processes run the tasks.
- **Multiple process and one fault**: Multiple mapper processes and multiple reducer processes run the tasks but one mapper fails.