Design Document
PROJECT 1 - MAP REDUCE

Team Members: Jurgen Yu, Muskan Jain, Venkata Bramara Parthiv Dupakuntla

System Design:
The given project implements the MapReduce functionalities. This implementation allows users to write their custom Map and Reduce functions, compile their source code to obtain the executables and, finally, run those executables to get the job done. Even though the system works on a single device, we make use of multiple cores to mimic multiprocessing. We achieve communication across processes using the Remote Method Invocation mechanism. Using reflection we ensure that the library code invokes the User Defined Functions (UDFs). The system is also robust to fault tolerance of upto one mapper or reducer during a job.

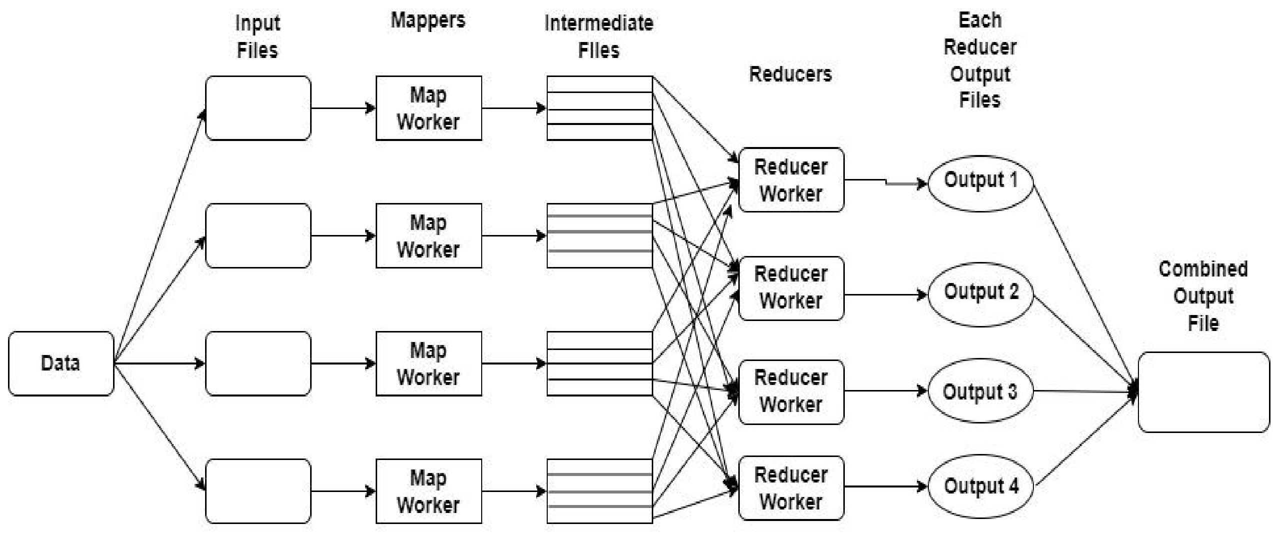
The user provides the input and output location, an arbitrary number of mappers and reducers (N).
Along with the input and output location, the user also provides the number of mappers/reducers (N). Fault Tolerance Logic is implemented to handle any number of Mappers or Reducers becoming unavailable at random times.

Test Cases:
We have chosen "Word Count", "Word Length", and "Movie Profile" as our test cases.

1. Word Count: Given a large text file, find the frequency of each word.
2. Word Length: Given a large text file, return all the words with their respective lengths.
3. Movie Profile: Given a large data set which consists of the names of people and the movies they have watched, find the number of movies each person watched.

How it works:
The user defined map and reduce functions and data are stored in the respective 'tasks' and 'data' folders. These are used as inputs to the master MapReduce implementation which first creates the partition of the input data and then invokes the map and reduce functions sequentially. The Reflection API makes sure that every Mapper worker and every Reducer worker call their respective mapper and reducer functions implemented by the user. The mapper workers read their partitioned block of texts, process them as key-value pairs and create intermediate files. Once all the mappers are done executing, the reducers will launch. The inputs to these reducers are the intermediate files generated. After every reducer execution, output files are generated, which are at the end coalesced to give the final results. If at any time fault occurs either within mapper or reducer, the process will be relaunched and executed again, this can be observed while executing the batch file.

Control flow:



Parallelism:
This is achieved by using Java's ProcessBuilder API. We can't use multithreading because threads usually communicate through a common memory stack which isn't the case in MapReduce. The number of processes used is defined in the MapReduceSpecs. This number controls the degree of parallelism our code will be using. The degree of parallelism is specified by the user given configuration file which has an input "N". "N" different workers are created for the map and reduce tasks for each test case. N mappers are launched simultaneously and reducers are only launched when all the mappers are complete. If there is a fault in any of the workers, the corresponding worker gets relaunched and after all the reducers finish their job, the system concludes its execution.

Communication between processes:
Initially, an RMI registry will be started to facilitate the communication between the master and the worker nodes. The RMI registry will be used by the workers to access the user defined map and reduce objects. The various methods used from RMI to facilitate this inter process communication is given in the 'utils/RMITools.java' file.

Inputs and Outputs:
The following are the inputs from the user:

1. Number of processes (N)
2. Input File Location
3. Output File Location
4. Timeout for each worker
5. Object of the user defined mapper class which is to be added to RMI registry for Master and mapper worker to access.

6. Object of the user defined reducer class which is to be added to RMI registry for Master and reducer worker to access.

The following are the outputs and their storage specifications:

1. finalOutput: contains the final output for all the test cases which is obtained after the system execution is concluded. Example of an output file for the Movie Profile: “finalOutput/com.mapreduce.tasks.movieprofile.MovieProfileMapperoutput.txt”
2. sparkOutput: contains the final output for all the test cases which is obtained from the same jobs run in Spark to perform output comparison. This is performed by the “outputComparison.py” file.
3. Datamapperoutput: This folder contains the output files of all the mappers. If there are 5 processes, there will be 5 mapper outputs in the form of a .txt file. The size of this folder depends on “N”
4. testCasesOutput: This folder contains 3 sub folders corresponding to each of the test case and each sub folder contains the respective outputs from each mapper.

Fault Tolerance:

We take care of two types of Faults in this system:

1. Worker Failure
2. Stragglers

Worker Failure fault is associated with a process being killed. To check this, we randomly select one of the mapper worker processes and one of the reducer processes, and cause them to fail. To deal with these dead workers, the master creates a new process with the same process ID and invokes its implementation.

To deal with the Stragglers, we have specified a “timeout” variable. If any of the processes take more time than this to execute, it is killed and restarted. The timeout value set for our project is 5s.

Design Tradeoffs and Considerations:

The design considers that multiple files will be provided as input in the input directory to facilitate multiple test cases. Hence, each file can be treated as a separate partition. Without this consideration, the entire file will have to be read by a process for it to be split into partitions, which should be avoided. Hence, in our implementation the master does not read the input files, it directly splits them among the workers.
We also consider that at any given time only one mapper or reducer will fail and this is achieved by simulating a faulty worker, where we drop any of the workers randomly.

Code Execution:

System Requirements:
To execute the whole code, a Java version greater than 8 is required and also a Python version >=3.0 is required to execute the output comparison file.

Windows:
Run $runTestCases.bat

MacOS:
Run sh runTestCases.sh *or*
Run ./runTestCases.sh