CSE 511 Programming Assignment 1: A MapReduce Framework

Out: February 20, 2020 Due: March 20, 2020

1 Introduction

This project is designed to give you experience in writing multi-threaded distributed systems by implementing a simplified "MapReduce" framework. The end product will be a base on which a variety of different parallel computations can be run, including parallel versions of the word-counting and grep applications.

Many commonly occurring data processing tasks can be expressed as "feed-forward sequences" of stages wherein: (i) the first stage s_1 reads inputs from one or more files, (ii) stage s_k $(1 \leq k \leq n)$ applies some transformation upon its input to create an intermediate output which serves as the input for s_{k+1}, and (iii) the last stage s_n writes the final output to one or more files. Furthermore, very often individual stages are highly parallelizable (offering the user an effective knob for using available machines/resources for performance improvements). Given this staged structure, many researchers/engineers have found it beneficial to implement a common software infrastructure ("framework") for the task-independent portions of this processing (e.g., setting up threads). This allows programmers to focus on coding their task-specific stages and simply re-use the task-independent functionality offered by the framework. MapReduce, developed by Google, is among the most popular examples of such a framework. MapReduce deploys two main stages, *Map* and *Reduce*. *Map* stage takes as input a set of tokens and converts them into a list of key-value pairs. These pairs are called intermediate key-value pairs because they are consumed by the *Reduce* to generate a final set of key-value entities.

Consider how the following two popular tasks can be programmed to fit this basic structure:

- *wordcount:* The *Map* stage reads individual words and produces key-value pairs of the form `(word, 1)`. The *Reduce* stage consumes these, groups them by key, and sums up the counts in each group to produce the final output. Notice how the first stage can be parallelized.

- *grep:* The first stage reads individual lines from a file or set of files and matches each line against a given string, producing key-value pairs of the form `(filename, lineNumber:line)`. The *Reduce* stage, as in wordcount, consumes these key-values and and outputs a sorted list containing matched lines for each key, ie., filename.

This project brings together many different ideas covered in class and will introduce you to several challenges and best practices in systems programming. Most notable among these are as follows.

- You will learn to write multi-threaded code that correctly deals with race conditions.

- You will familiarize yourself with the use of remote procedure calls (RPCs) and a distributed file system (HDFS).

- You will appreciate the role of layering and interfaces in the design of systems software.

- You will carry out simple performance evaluation experiments to examine the performance impact of (i) the degree of parallelism in one of the stages and (ii) the size of the shared buffers using which the two stages of your application communicate.

2 Description

You will implement a simplified version of the MapReduce framework (illustrated in Figure 1). Note that you are not writing the *application* (e.g., wordcount); you are writing the *system*, or framework, on which these applications will run. In other words, you will still be using the the C library and UNIX system call APIs to implement the given functions, but you will also be *providing* a precisely specified API to allow other software to utilize your system. You are required to ensure that your framework correctly implements this precise interface expected by the application. We will call the individual threads executing *Map* and *Reduce* operations as `Mappers` and `Reducers`, respectively. We will use the term Job to refer to the individual units of work meaningful to the user and parallel entities within a job will be described as Tasks. The application will specify the job, by providing *Map* and *Reduce* functions as well as a few configuration parameters to the framework. It's the responsibility of the framework to orchestrate their concurrent execution and result accumulation.

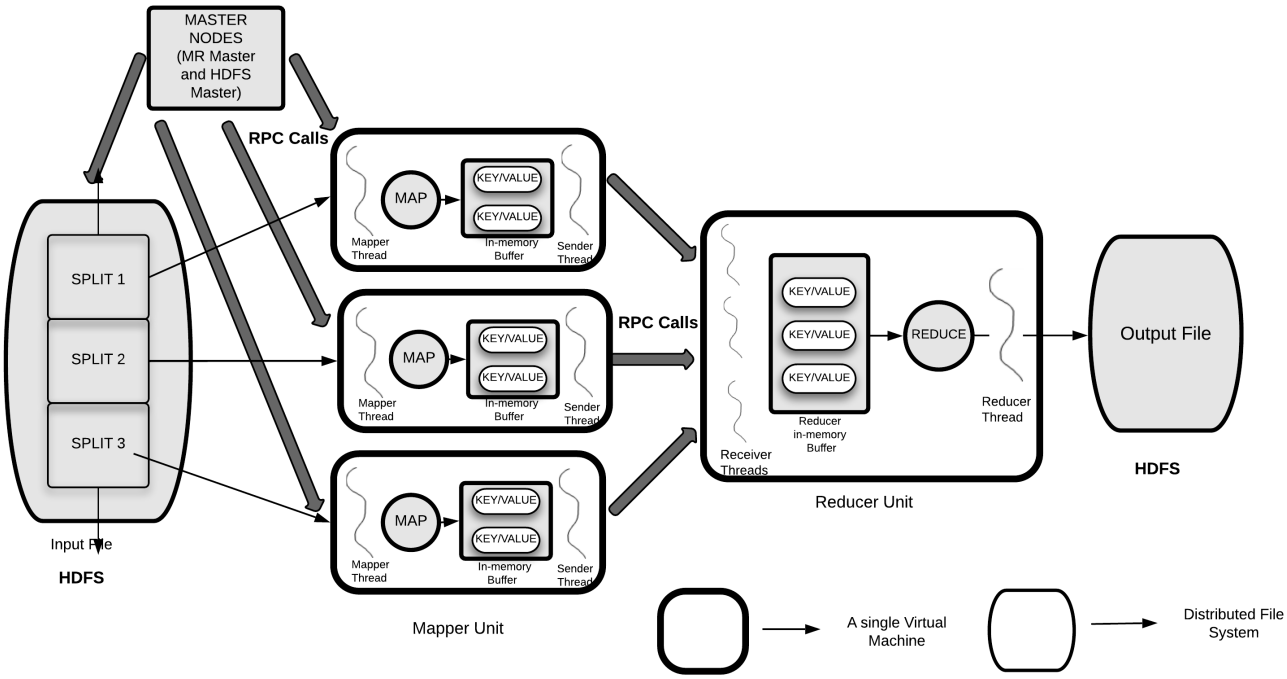

Figure 1: Overview of our simplified MapReduce framework.

2.1 The User's Perspective

Although your task in this project is the implementation of the framework, it is important to understand the expectations from the framework well. That will help you in formulating a good design for your implementation. This section aims to throw some light on the overall architecture expected in your implementation. It will also elaborate on the expected characteristics of individual components and the subsequent design decisions involved. To accomplish the aforementioned goals, let us begin by summarizing the sequence of steps involved in executing a typical MapReduce job on the framework.

- **Job specification** : User defines a job that needs to be run by specifying the input and output files as well as some config parameters like, number of threads or barrier enable. User is also responsible for instantiating the framework by calling the MapReduce API used for initialisation and registering the *Map* and *Reduce* callback operations with the framework.
- **Framework Initialization** : When the user calls `mr_init` API, framework registers the callback functions provided by the user and also prepares the input on which framework should operate. Given the distributed nature of your framework, the input file needs to be fetched from the user specified path and pushed onto the distributed file system.
- **Job Start** : Application starts the job by calling `mr_start` API. As a result, the framework needs to start the threads responsible for executing the *Map* and *Reduce* functions, based on the input parameters specified by the user, such as the degree of parallelism.
- **Job Execution** : The framework ensures that the mapper and reducer threads coordinate to produce the correct output.
- **Output creation and Job clean-up** : The output generated by the reducer thread is populated to the specified path on the distributed file system, with the help of APIs provided by the framework. After the job is finished, framework outputs some performance measurements (more details on this later) and also performs some clean-up, to free up the resources which were in use.

2.1.1 Job Specification

The design choices you make for your MapReduce framework should be dictated by the goal of a "general" computational framework. The application is going to use the APIs exposed by the framework and provide both *Map* and *Reduce* callback functions. The application will also provide the input file for the job. To provide more control over the execution to the user/application, the framework will accept a few other parameters - the number of parallel threads, Enable/Disable barrier and the delimiters to be used for tokenization (More details in 2.1.3). It's job of the application code to call APIs for framework init and job start (More details about the API in 3). The application binary that will utilize the framework API and provide all these parameters will be provided to you. Your job is to implement the API functionality expected by the application, while adhering strictly to the interface defined.

In order to run a job using the framework, you first need to setup a set of Virtual Machines (VM) that will enable you to have multiple threads running concurrently. You also need to create a config file, that your framework will use, containing a list of configuration parameters such as the connection details (IP address) of all VMs part of your cluster as well as the sizes of in-memory buffers used by the mappers and reducers.

2.1.2 Framework Initialisation

Setting up the framework for a job requires the initialization of two main entities, namely **MapReduce(MR) Master** and **HDFS Master** (also called Namenode). The application code will be started on the VM which is designated as the MapReduce Master. MR Master is responsible for orchestrating the framework execution, which includes input file setup, management of mappers and reducers as well as the clean-up on finish. Other nodes in the cluster are designated as worker nodes, which will be running mapper and reducer threads. MR Master will use RPC calls to communicate to each of the worker nodes. So, you have to ensure that before the application is run, the workers should have RPC servers running to service any incoming requests from the MR Master. When the

application uses `mr_init` API call to initialize your framework, MR Master will consume the parameters specified by the user and initialize the state of the MR framework. All the functionality expected from MR Master has to be implemented by you, as a part of the framework API implementation.

The second entity that constitutes an important part of framework initialisation is the HDFS Master/Namenode. The framework should utilize a Distributed File System to store all the input and output files. You are required to use HDFS for this project. In the context of distributed file systems(DFS), Namenode is responsible for maintaining the metadata about all the files in the DFS, such as their location, replication factor, etc. It's also responsible for managing the cluster state in case of node failures. You will be using the Namenode daemon provided by HDFS and don't need to implement any aspect of the DFS yourself. You can initialize the Namenode daemon on the same VM as the MR Master or use a separate VM for performance reasons. You will use the start-up scripts available in HDFS to spawn both the Namenode daemon as well as DataNode (other nodes in the cluster which store the actual file data) daemons on the other VMs. All you need to provide is the connection details of the nodes in cluster, which you already have in the config file you used for MR Master. You can either do it before the start of the application or make it the responsibility of your `mr_init` function. Once, HDFS is initialized, MR Master can load the input file provided by the user, from the specified input path, onto the HDFS. This can be accomplished using C API provided by HDFS, `libhdfs` or using the HDFS shell commands.

2.1.3 Starting the Job

After initialization, the job will be started when `mr_start` is invoked by the application. MR Master node will use gRPC to contact each of the Worker nodes and spawn a number of mapper threads, based on the number `threads` given by the application. It will also create a single reducer thread, on one of the VMs. Each Mapper thread also provisions a local in-memory buffer, that is used to store intermediate key-value pairs, generated by the *Map* function. Similarly, an in-memory buffer is used by the reducer to receive and aggregate all the key-value pairs generated by the mappers.

An important responsibility of the MR Master here is to divide the input file into well-defined equal chunks and assign it to each mapper thread. You need to come up with a particular protocol for this splitting, that divides the work equally while respecting object boundaries, i.e., lines in this case. Each mapper thread, tokenizes the file input based on `delimiters` provided by the application. For each token generated, a callback to the application's *Map* function will be initiated and the subsequent key-value pair generated will be stored in the local buffer. The *Map* will use `mr_produce` function, provided by the framework, to store the key-value pair in the buffer. The mapper threads will terminate asynchronously, after they are done processing their input. When you design the RPC services provided by the reducer, make sure you incorporate a way to signal to the reducer about a mapper thread's completion. This is done, so that the reducer knows when to stop waiting for new key-value pairs.

2.1.4 Job Execution

After the previous stage, all the mapper and reducer threads will be in execution. You are required to ensure that all the entities stored to the memory buffers are serialized first. Protobuf allows a highly efficient way to serialize the objects and should be used for the implementation of your framework. `mr_produce` and `mr_consume` will be responsible for serialization to and from the buffer, respectively.

The buffer available to each mapper is of a limited size and needs to be periodically flushed to accommodate new entries. To achieve this, paired with each mapper thread, a separate thread called *sender* thread should be active for each memory buffer. It should be woken up when the total memory utilization breaches the threshold value given by `MR_BUFFER_THRESHOLD`, defined in `mapreduce.h`. Use

pthreads library for the implementation of your threads. The responsibility of the *sender* thread is to flush the contents of the buffer and send the data to the memory buffer maintained at the reducer node. You will utilize Google RPC to send the serialized data across to the reducer. You will have to ensure synchronisation between the thread producing key-value pairs and the harvester thread, to avoid data races as well as buffer overflows.

At the reducer's memory buffer, data production (writes by the mapper threads) and consumption (by the reducer) need to be synchronized. As the buffer gets full, writes by the mappers will have to block. The reducer thread will use the function `mr_consume` to read serialized values from the buffer and convert into a form acceptable by the *Reduce* function. The values read on each call to `mr_consume` will be flushed from the buffer and further key-value pairs can now be received from the mappers. How the buffer flushing and the thread synchronization between writers (mapper threads) and readers (reducer thread) is handled, will be dictated by your design choices. For the sake of optimization, `mr_consume` reads a bunch of key-value pairs at once and converts them into an array of `kvpair` structs, that can be provided to the *Reduce* function. How the reducer consumes the incoming data, will be determined by the `enableBarrier` flag, supplied to the `mr_consume` function. If the flag is enabled, then the framework should ensure that the callback to application's *Reduce* function is made, only after all the mapper threads have finished. Otherwise, the callback can be made without any waiting.

Since, you will be reading from a serialized data stream and converting it into a new format by allocating new buffers, be very careful of the scope of the pointers being passed to the *Reduce* function. Here the framework design assumes that the *Reduce* function will retain state between consecutive calls to the function. You need to ensure that reducer thread terminates when all the mappers have finished producing data. The output of the reducer should again be stored onto a file in the HDFS, using the function `mr_output` provided by the framework.

2.1.5 Output creation and Job clean-up

Reducer keeps aggregating all the key-value pairs provided by `mr_consume`, until it returns 0; signalling all the mapper threads have finished. *Reduce* function then prepares the output in the format required by the application and stores it in a data buffer. Application then uses the `mr_output` API to create the output file on the HDFS and supplies the data buffer to it. The MR Master Node should be informed, in some way, of the completion of the job. When the MR Master is informed, `mr_finish` can return and subsequent clean up will be done by `mr_destroy`.

3 Detailed API Description

3.1 MapReduce framework API (your code)

Your code will provide an API of seven functions which applications can use to perform parallel MapReduce computations. The application will provide its own Map and Reduce code (using callback function pointers), and your framework will use these to run the MapReduce operation. It is up to you to determine how best to implement these functions within the constraints laid out in the assignment. (See the file `mapreduce.h` for more detail on these functions and the related types and structures.)

The first four functions have to do with the overall setup of the framework:

- `mr_init(mapfn, reducefn, threads, inpath, outpath, delimiters)`: Allocates, initializes, and returns a new instance of your MapReduce framework, which will eventually execute the given Map and Reduce callbacks using the specified number of mapper threads. Also, initializes the distributed file system and loads the input file on the HDFS cluster. Returns `NULL` on error.

- `mr_destroy(mr)`: Destroys and cleans up an existing instance of your MapReduce framework. Any resources which were acquired or created in `mr_init` should be released or destroyed here.

- `mr_start(mr, barrierEnable)`: Begins a multi-threaded MapReduce operation using the initialized framework. When `barrierEnable` flag is set, a barrier should be implemented at the reducer, i.e., the reducer should start processing the key/value pairs only when all the mappers are done producing values. Returns 0 if successful and nonzero on error.

- `mr_finish(mr)`: Blocks until the entire MapReduce operation is complete and all threads are finished. Returns 0 if every Map and Reduce callback returned 0 (success), or nonzero if any of them returned nonzero.

The Map and Reduce functions will communicate using a general key-value pair structure called `kvpair`. This structure has pointers called `key` and `value` which point to the data for the key and value, respectively, and integers `keysz` and `valuesz` which give their sizes in bytes. These pointers are opaque to your framework, so you will not need to interpret the data, only move it from one place to another. The functions which you will implement to provide this communication are:

- `mr_produce(mr, kv)`: Serializes the key-value pair `kvpair` and adds it to the buffer for the given mapper. If there is not enough room in the shared buffer, this function should block until there is. How you handle the invocation and synchronisation with the harvester thread is a design choice you are free to make. Returns 1 if successful and -1 on error (this convention mirrors that of the standard "write" function).

- `mr_consume(mr, kvset, num, barrierEnable)`: Retrieves the next set of key-value pairs from the reducer buffer and converts them into the `kvpair` format, understood by the *Reduce* function. `num` defines the number of pairs read. The timing and the amount of data fetched from the buffer will be dictated by your design choices. If no pair is available, this function should block until one is produced or all the mapper threads return. Returns 1 if successful, 0 if the mappers have finished producing pairs, and -1 on error (this convention mirrors that of the standard "read" function).

- `mr_output(mr, writeBuffer, bufferLength)`: This function will be used by the application to write the final output to a file on the HDFS. The format of the output will be dictated by the application code and will be provided to this function as a data buffer. If the output file does not exist, it should be created; if it does, it should be truncated and overwritten.

3.2 Map and Reduce functions (not your code)

Every application in our MapReduce framework defines its computation in terms of a different Map and Reduce function. The application will provide these two "callback" functions when it initializes the framework, and your code will "call them back" to do the processing work for each stage. They are defined as follows:

- `map(mr, kv)`: Implements the application's Map function, which will be run in parallel using the number of mapper threads specified in `mr_init`. You will pass this function a pointer to your MapReduce structure and the token generated by the framework for processing by the Map function.

- `reduce(mr, kvset, num)`: Implements the application's Reduce function. You will pass this function a pointer to your MapReduce structure, pointer to a buffer containing a collection of `kvpair` structs and the number of pairs available in the buffer passed. The function uses static

entities to maintain state between subsequent calls to the function and accumulate all the results. Output format is prepared by the function and `mr_output` is used by the application to dump the output to a file on the Distributed File System.

Each of these functions returns 0 to indicate success and nonzero to indicate failure. Note that your framework and implementation remain exactly the same from application to application, but by changing these two functions it can be applied to a wide variety of parallel tasks.

4 Performance Evaluation

Being Written. You will be given further details on this soon.

5 Experimental Setup

You can work in teams of atmost 2 people for this project. Each team will be given access to a private repository on GitHub Classroom, just like it was done for the last project. You can create your teams using the Github Classroom invitation link, which will be provided to you. You are required to deploy and test your code on a public cloud platform, preferably AWS. There should be mutiple VMs instantiated to run the different components of the framework, such as the Master, mappers and reducers, to avoid interference and performance penalties. As in real-world systems programming, you do not have the source code for the applications that will be using your system.

You will be provided with two application binaries, with which your framework should correctly work.

- `mr-wordc`: MapReduce version of a word count application.
- `mr-grep` : MapReduce version of a grep application.

Additionally, you will be provided with a Header file `mapreduce.h`, defining the API you must provide as well as a `map_reduce` struct type for storing your framework's state. We will be using GitHub classroom to provide the above files and any additional files that will be given later. You are expected to push your code periodically to the GitHub repository, so that the progress can be tracked.

6 Submission

- Like in the previous project, GitHub classroom will be used for all the code submissions.
- You are expected to submit a Report containing group member's names, summary of the performance evaluation results, and list of sources consulted. It should also explain your code design and the important design decisions you made.
- Remember to comment your code *as you write it* to explain how everything works.
- Please make sure you commit your final code and the Report on the GitHub repository before the deadline. Last commit at the time of deadline expiry will be used for the project evaluation and grading.

Important reminder

As with all projects in this class, you are not permitted to:

- Use code from outside sources without citing (Internet, previous class projects, etc.)
- Share code with other teams or post parts of your code publicly where others could find and use them. You are responsible for protecting your work from falling into the hands of others.
- Allow anyone outside your team to write any part of the code for this assignment.

If neither member of your team is able to explain your code and development process at the demo, this will be highly suspect. If there is any question at all about what is acceptable, ask the instructor. **Above all, be honest. This always goes a long way.**