CS 551 Systems Programming, Summer 2024
Programming Project 2

Out: 07/19/2024 Fri.
Due: 07/28/2024 Sun. 23:59:59

In this project we are going to simulate the MapReduce framework on a single machine using multi-process programming.

1 Introduction

In 2004, Google (the paper “MapReduce: Simplified Data Processing on Large Clusters” by J. Dean and S. Ghemawat) introduced a general programming model for processing and generating large data sets on a cluster of computers.

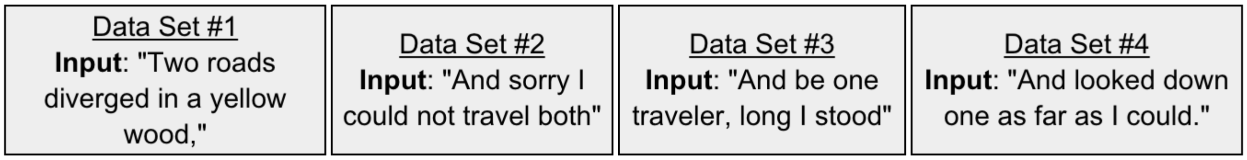
The general idea of the MapReduce model is to partition large data sets into multiple splits, each of which is small enough to be processed on a single machine, called a worker. The data splits will be processed in two phases: the map phase and the reduce phase. In the map phase, a worker runs user-defined map functions to parse the input data (i.e., a split of data) into multiple intermediate key/value pairs, which are saved into intermediate files. In the reduce phase, a (reduce) worker runs reduce functions that are also provided by the user to merge the intermediate files, and outputs the result to result file(s).

We now use a small data set (the first few lines of a famous poem by Robert Frost, see Figure 1) to explain to what MapReduce does.

```
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could.
```

Figure 1: A small data set to be processed by MapReduce.

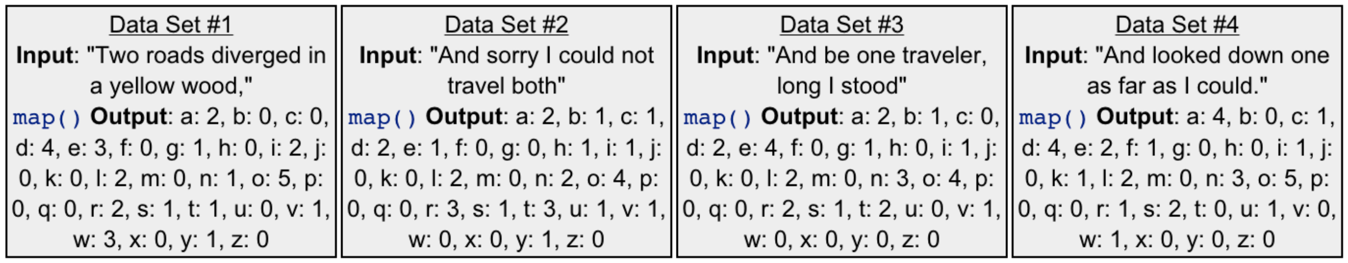
To run MapReduce, we first split the dataset into small pieces. For this example, we will split the dataset by the four lines of the poem (Figure 2).



Data Set #1	Data Set #2	Data Set #3	Data Set #4
Input: "Two roads diverged in a yellow wood,"	**Input**: "And sorry I could not travel both"	**Input**: "And be one traveler, long I stood"	**Input**: "And looked down one as far as I could."

Figure 2: Partitioning the input data set into multiple splits.

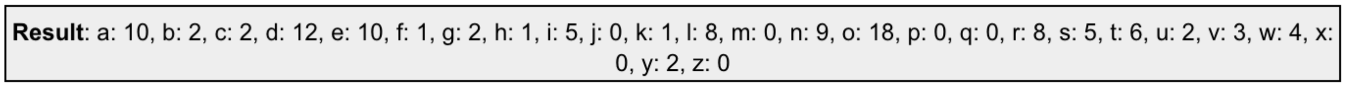
The MapReduce framework will have four workers (in our project, the four workers are four **processes** that are forked by the main program. In reality, they will be four independent machines) to work on the four splits (each worker is working on a split). These four map worker will each run a user-defined map function to process the split. The map function will map the input into a series of (key, value) pairs. For this example, let the map function simply count the number of each letter (A-Z) in the data set.



Data Set #1	Data Set #2	Data Set #3	Data Set #4
Input: "Two roads diverged in a yellow wood,"	**Input**: "And sorry I could not travel both"	**Input**: "And be one traveler, long I stood"	**Input**: "And looked down one as far as I could."
`map()` **Output**: a: 2, b: 0, c: 0, d: 4, e: 3, f: 0, g: 1, h: 0, i: 2, j: 0, k: 0, l: 2, m: 0, n: 1, o: 5, p: 0, q: 0, r: 2, s: 1, t: 1, u: 0, v: 1, w: 3, x: 0, y: 1, z: 0	`map()` **Output**: a: 2, b: 1, c: 1, d: 2, e: 1, f: 0, g: 0, h: 1, i: 1, j: 0, k: 0, l: 2, m: 0, n: 2, o: 4, p: 0, q: 0, r: 3, s: 1, t: 3, u: 1, v: 1, w: 0, x: 0, y: 1, z: 0	`map()` **Output**: a: 2, b: 1, c: 0, d: 2, e: 4, f: 0, g: 1, h: 0, i: 1, j: 0, k: 0, l: 2, m: 0, n: 3, o: 4, p: 0, q: 0, r: 2, s: 1, t: 2, u: 0, v: 1, w: 0, x: 0, y: 0, z: 0	`map()` **Output**: a: 4, b: 0, c: 1, d: 4, e: 2, f: 1, g: 0, h: 0, i: 1, j: 0, k: 1, l: 2, m: 0, n: 3, o: 5, p: 0, q: 0, r: 1, s: 2, t: 0, u: 1, v: 0, w: 1, x: 0, y: 0, z: 0

Figure 3: The outputs of the map phase, which are also the inputs to the reduce phase.

The map outputs in our example are shown in Figure 3. They are also the inputs for the reduce phase. In the reduce phase, a reduce worker runs a user-defined reduce function to merge the intermediate results output by the map workers, and generates the final results (Figure 4).



Result: a: 10, b: 2, c: 2, d: 12, e: 10, f: 1, g: 2, h: 1, i: 5, j: 0, k: 1, l: 8, m: 0, n: 9, o: 18, p: 0, q: 0, r: 8, s: 5, t: 6, u: 2, v: 3, w: 4, x: 0, y: 2, z: 0

Figure 4: The final result

2 Simulating the MapReduce with multi-process programming

2.1 The base code

Download the base code from the Brightspace. You will need to add your implementation into this base code. The base code also contains three input data sets as examples.

2.2 The working scenario

In this project, we will use the MapReduce model to process large text files. The input will be a file that contains many lines of text. The base code folder contains three example input data files. We will be testing using the example input data files, or data files in similar format.

A driver program is used to accept user inputs and drive the MapReduce processing. The main part of driver program is already implemented in `main.c`. You will need to complete the `mapreduce()` function, which is defined in `mapreduce.c` and is called by the driver program. A Makefile has already been given. Running the Makefile can give you the executable of the driver program, which is named as "run-mapreduce". The driver program is used in the following way:

```
./run-mapreduce "counter"|"finder" file_path split_num [word_to_find]
```

where the arguments are explained as follows.

- The first argument specifies the type of the task, it can be either the "Letter counter" or the "Word conter" (explained later).
- The second argument "`file_path`" is the path to the input data file.
- The third argument "`split_num`" specifies how many splits the input data file should be partitioned into for the map phase.

- The fourth argument is used only for the “Word finder” task. This argument specifies the word that the user is trying to find in the input file.

The `mapreduce()` function will first partition the input file into `N` roughly equal-sized splits, where `N` is determined by the `split_num` argument of the driver program. Note that the sizes of each splits do not need to be exactly the same, otherwise a word may be divided into two different splits.

Then the `mapreduce()` forks one worker process per data split, and the worker process will run the user-defined map function on the data split. After all the splits have been processed, *the first worker process forked* will also need to run the user-defined reduce function to process all the **intermediate** files output by the map phase. Figure 5 below gives an example about this process.

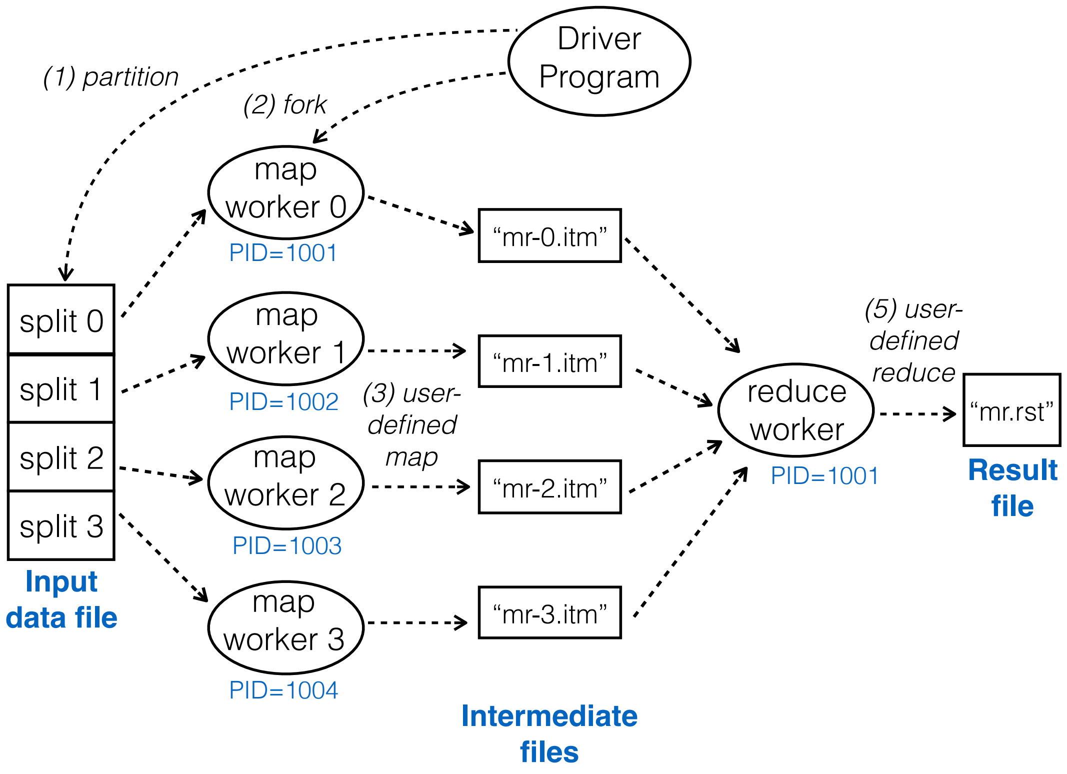


Figure 5: An example of the working scenario.

2.3 The two tasks

The two tasks that can be performed by the driver program are described as follows.

The “Letter counter” task is similar to the example we showed in Section 1, which is counting the number of occurrence of the 26 letters in the input file. The difference is the intermediate file and the final result file should be written in the following format:

```
A number-of-occurrences
B number-of-occurrences
...
Y number-of-occurrences
Z number-of-occurrences
```

Bonus(15 points): The "Word finder" task is to find the word provided by user (specified by the "`word_to_find`" argument of the driver program) in the input file, and outputs to the result file all the lines that contain the target word in the same order as they appear in the input file. For this task, you should implement the word finder as a whole word match, meaning that the function should only recognize complete words that match exactly with the specified search terms. And if multiple specified words are found in the same line, you only need to output that line once.

For the base implementation, you are only required to handle "Letter counter" task. The "Word finder" task is designed as bonus. Correctly implemented it will guarantee an extra 20 points.

2.4 Other requirements

- Besides the `mapreduce()` function defined in `mapreduce.c`, you will also need to complete the map/reduce functions of the two tasks (in `usr_functions.c`.)
- About the interfaces listed in "user_functions.h" and "mapreduce.h":
  - Do not change any function interfaces.
  - Do not change or delete any fields in the structure interfaces (but you may add additional fields in the structure interface if necessary).

  The above requirements allow the TA to test your implementations of worker logic and user map/reduce functions separately. Note that violation to these requirements will result in 0 point for this project.
- Use `fork()` to spawn processes.
- Be careful to avoid fork bomb (check on Wikipedia if you are not familiar with it). A fork bomb will result in 0 point for this project.
- The `fd` in the `DATA_SPLIT` structure should be a file descriptor to the original input data file.
- The intermediate file output by the first map worker process should be named as "mr-0.itm", the intermediate file by the second map worker process should be named as "mr-1.itm", ... The result file is named as "mr.rst" (already done in `main.c`).
- Do not delete the intermediate files. They will be checked when grading.

3 Log and submit your work

3.1 Log and submit your work

Log your work: besides the files needed to build your project, you must also include a README file which minimally contains your name and B-number. Additionally, it can contain the following:

- The status of your program (especially, if not fully complete).
- Bonus is implemented or not.

- A description of how your code works, if that is not completely clear by reading the code (note that this should not be necessary, ideally your code should be self-documenting).
- Possibly a log of test cases which work and which don't work.
- Any other material you believe is relevant to the grading of your project.

Compress the files: compress your README file, all the files in the base code folder, and any additional files you add into a ZIP file. Name the ZIP file based on your BU email ID. For example, if your BU email is "abc@binghamton.edu", then the zip file should be "proj2_abc.zip".

Submission: submit the ZIP file to Brightspace before the deadline.

3.2 Grading guidelines

(1) Prepare the ZIP file on a Linux machine. If your zip file cannot be uncompressed, 5 points off.

(2) If the submitted ZIP file/source code files included in the ZIP file are not named as specified above (so that it causes problems for TA's automated grading scripts), 10 points off.

(3) If the submitted code does not compile:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(4) If the code is not working as required in this spec, the TA should take points based on the assigned full points of the task and the actual problem.

(5) Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with your fellow students, but code should absolutely be kept private.