Lab 4: MapReduce

Introduction

In this lab you'll build a MapReduce library as an introduction to programming in Java and to building fault tolerant distributed systems. In the first part you will write a simple MapReduce program. In the second part you will write a Master that hands out tasks to MapReduce workers. And you will complete a parallel version of Mapreduce and handle the failures of workers in part 3 and part 4. There is a optional exercise in part 5, which would you bonus if you complete a inverted index application.
The interface to the library and the approach to fault tolerance is similar to the one described in the original [MapReduce paper](#).

Copyright

This is a Java-implementation of MIT 6.824 Lab 1: MapReduce, for SDIC course lab, SJTU. Some resources directly or indirectly refer to the [original materials](#).
Notice: For educational or personal usage only.

Collaboration Policy

You must write all the code you hand in for this course, except for code that we give you as part of the assignment. You are not allowed to look at anyone else's solution, which means that you may discuss the assignments with other students, but you may not look at or copy each others' code. The reason for this rule is that we believe you will learn the most by designing and implementing your lab solution code yourself.

Software

You'll implement this lab (and all the labs) in **Java**. Java is an mature OOP language and many learning materials could be found online. We believe that most of you are able to learn it and work out this lab with it. We will grade your labs using [Java 1.8_latest](#), running on Java HostSpot VM(64 bit). You better use this too, though we don't know if there is any problem with other versions.

Java is platform-independent so you don't need to worry about the difference between OS and hardwares. But it needs to set environment variebles. Google it and learn how to

do it for your OS.

We supply you with parts of a MapReduce implementation that supports both distributed and non-distributed operation (just the boring bits). You'll fetch the initial lab software from [here](). This is a Maven project under IDEA. To learn more about Maven, look at the [Apache Maven]() or other resources. [IDEA]() is the most widely used Java IDE now, we recommend you to use this IDE because it accelerates the developing, though it is not required, some following steps are demonstrated by IDEA.

The Map/Reduce implementation we give you has support for two modes of operation, *sequential* and *distributed*. In the former, the map and reduce tasks are executed one at a time: first, the first map task is executed to completion, then the second, then the third, etc. When all the map tasks have finished, the first reduce task is run, then the second, etc. This mode, while not very fast, is useful for debugging. The distributed mode runs many worker threads that first execute map tasks in parallel, and then reduce tasks. This is much faster, but also harder to implement and debug.

Preamble: Getting familiar with the source

The mapreduce project provides a simple Map/Reduce library (in the common, core, rpc package). Applications should normally call *distributed*() [located in core/Master.java] to start a job, but may instead call *sequential*() [also in core/Master.java] to get a sequential execution for debugging.

The code executes a job as follows:

1.The application provides a number of input files, a map function, a reduce function, and the number of reduce tasks (nReduce).
2. A master is created with this knowledge. It starts an RPC server, and waits for workers to register (using the RPC call register() [defined in core/Master. java]). As tasks become available (in steps 4 and 5), schedule() [core/Scheduler. java] decides how to assign those tasks to workers, and how to handle worker failures.
3. The master considers each input file to be one map task, and calls doMap() [core/Mapper. java] at least once for each map task. It does so either directly (when using sequential()) or by issuing the doTask RPC to a worker [core/Worker. java]. Each call to doMap() reads the appropriate file, calls the map function on that file's contents, and writes the resulting key/value pairs to nReduce intermediate files. doMap() hashes each key to pick the intermediate file and thus the reduce task that will process the key. There will be nMap x nReduce files after all map tasks are done. Each file name contains a prefix, the map task number, and the reduce task number. If there are two map tasks and three reduce tasks, the map tasks will create these six intermediate files:

```
1   mrtmp.xxx-0-0
2   mrtmp.xxx-0-1
3   mrtmp.xxx-0-2
4   mrtmp.xxx-1-0
5   mrtmp.xxx-1-1
6   mrtmp.xxx-1-2
```

Each worker must be able to read files written by any other worker, as well as the input files. Real deployments use distributed storage systems such as GFS to allow this access even though workers run on different machines. In this lab you'll run all the workers on the same machine, and use the local file system.

4. The master next calls doReduce() [core/Reducer.java] at least once for each reduce task. As with doMap(), it does so either directly or through a worker. The doReduce() for reduce task r collects the r'th intermediate file from each map task, and calls the reduce function for each key that appears in those files. The reduce tasks produce nReduce result files. The master calls merge() [core/Master.java], which merges all the nReduce files produced by the previous step into a single output.

5. The master sends a Shutdown RPC to each of its workers, and then shuts down its own RPC server.

> **Note:** Over the course of the following exercises, you will have to write/modify doMap, doReduce, and schedule yourself. These are located in core/Mapper.java, core/Reducer.java, and core/Scheduler.java respectively. You will also have to write the map and reduce functions in WordCount.java.

You should not modify any other files, but reading them might be useful in order to understand how the other methods fit into the overall architecture of the system.

Part I: Map/Reduce input and output

The Map/Reduce implementation you are given is missing some pieces. Before you can write your first Map/Reduce function pair, you will need to fix the sequential implementation. In particular, the code we give you is missing two crucial pieces: the function that divides up the output of a map task, and the function that gathers all the inputs for a reduce task. These tasks are carried out by the doMap() function in Mapper.java, and the doReduce() function in Reducer.java respectively. The comments in those files should point you in the right direction.

To help you determine if you have correctly implemented doMap() and doReduce(), we have provided you with a test suite that checks the correctness of your implementation.

These tests are implemented in the file MRTest.java. To run the tests for the sequential implementation that you have now fixed, run:

```
@Test
public void testSequentialSingle() {

@Test
public void testSequentialMany() {
```

> **TASK:** You receive full credit for this part if your software passes the Sequential tests (as run by the command above) when we run your software on our machines.

If the output did not show *ok* next to the tests, your implementation has a bug in it. To give more verbose output, set debugEnabled = true in common/Utils.java. You will get much more output along the lines of:

```
✓ MRTest                    5 s 324 ms   master: Starting Map/Reduce task test
  ✓ testSequentialSingle    3 s 529 ms   Merge: read mrtmp.test-res-0
  ✓ testSequentialMany      1 s 795 ms   master: Map/Reduce task completed
```

✓ Tests passed: 2 of 2 tests – 5 s 324 ms

```
✓ MRTest                    5 s 324 ms   master: Starting Map/Reduce task test
  ✓ testSequentialSingle    3 s 529 ms   Merge: read mrtmp.test-res-0
  ✓ testSequentialMany      1 s 795 ms   Merge: read mrtmp.test-res-1
                                          Merge: read mrtmp.test-res-2
                                          master: Map/Reduce task completed
```

# Part II: Single-worker word count

Now you will implement word count — a simple Map/Reduce example. Look in WordCount.java; you'll find empty mapFunc() and reduceFunc() functions. Your job is to insert code so that WordCount.java reports the number of occurrences of each word in its input. A word is any contiguous sequence of letters, as determined by regular expression:
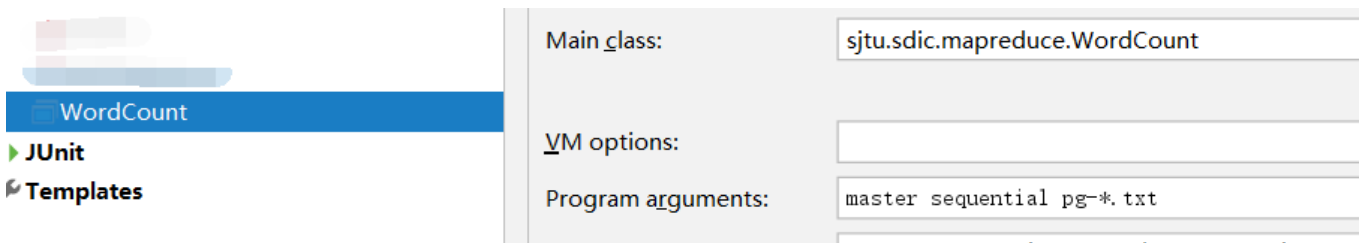
```
1  [a-zA-Z0-9]+
```

There are some input files with pathnames of the form pg-*.txt in project's root dir, downloaded from Project Gutenberg. Here's how to run WordCount.java with the input files:

1.Find "Edit Configurations" of WordCount
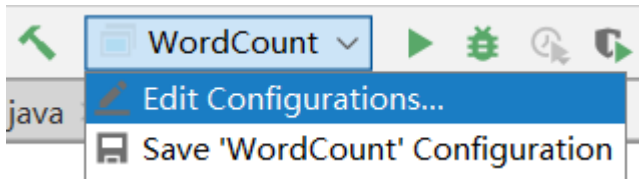
2.Edit Program arguments as below. This is same with specifying parameters in Console/shell.

Review Section 2 of the MapReduce paper. Your mapFunc() and reduceFunc() functions will differ a bit from those in the paper's Section 2.1. Your mapFunc() will be passed the name of a file, as well as that file's contents; it should split the contents into words, and return a List of common/KeyValue. While you can choose what to put in the keys and values for the mapFunc output, for word count it only makes sense to use words as the keys. Your reduceFunc() will be called once for each key, with a slice of all the values generated by mapFunc() for that key. It must return a string containing the total number of occurrences of the key.

- **Hint:** Use Pattern and Matcher to extract words.
- **Hint:** Integer.valueOf(String str) could parse a valid string to integer.

The output will be in the file "mrtmp.wcseq". Your implementation is correct if the following command produces the output shown here (under XNIX):

```
$ sort -n -k2 mrtmp.wcseq | tail -10
that: 7871
it: 7987
in: 8415
was: 8578
a: 13382
of: 13536
I: 14296
to: 16079
and: 23612
the: 29748
```

Part III: Distributing MapReduce tasks

Your current implementation runs the map and reduce tasks one at a time. One of Map/Reduce's biggest selling points is that it can automatically parallelize ordinary sequential code without any extra work by the developer. In this part of the lab, you will complete a version of MapReduce that splits the work over a set of worker threads that run in parallel on multiple cores. While not distributed across multiple machines as in real Map/Reduce deployments, your implementation will use RPC to simulate distributed computation.

The code in core/Master.java does most of the work of managing a MapReduce job. We also supply you with the complete code for a worker thread, in core/Worker.java, as well as some code to deal with RPC in package rpc.

Your job is to implement schedule() in core/Scheduler.java. The master calls schedule() twice during a MapReduce job, once for the Map phase, and once for the Reduce phase. schedule()'s job is to hand out tasks to the available workers. There will usually be more tasks than worker threads, so schedule() must give each worker a sequence of tasks, one at a time. schedule() should wait until all tasks have completed, and then return. schedule() learns about the set of workers by reading its registerChan argument. That channel yields a string for each worker, containing the worker's RPC address. Some workers may exist before schedule() is called, and some may start while schedule() is running; all will appear on registerChan. schedule() should use all the workers, including ones that appear after it starts.
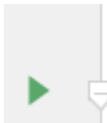
schedule() tells a worker to execute a task by calling

```
1   Call.getWorkerRpcService(worker).doTask(arg);
```

This RPC's arguments are defined by DoTaskArgs in common/DoTaskArgs.java. The file element is only used by Map tasks, and is the name of the file to read; schedule() can find these file names in mapFiles.

Your solution to Part III should only involve modifications to Scheduler.java. If you modify other files as part of debugging, please restore their original contents and then test before submitting.

The corresponding test methods are as below, defined in MRTest.java:

```java
@Test
public void testParallelBasic() {

@Test
public void testParallelCheck() {
```

The latter verifies that your scheduler causes workers to execute tasks in parallel.

TASK: You will receive full credit for this part if your software passes testParallelBasic and testParallelCheck when we run your software on our machines.

- **Hint:** RPC implementation relys on SOFARPC, documents could be found here.
- **Hint:** schedule() should send RPCs to the workers in parallel so that the workers can work on tasks concurrently. Learn how to use Java Thread API.
- **Hint:** schedule() must wait for a worker to finish before it can give it another task. Try to exploit Channel.
- **Hint:** You may find CountDownLatch is useful.
- **Hint:** To stop a thread, using interrupt(). Google it to learn more.
- **Hint:** The easiest way to track down bugs is to insert print statements (perhaps calling debug() in Utils.java, and then think about whether the output matches your understanding of how your code should behave.

Part IV: Handling worker failures

In this part you will make the master handle failed workers. MapReduce makes this relatively easy because workers don't have persistent state. If a worker fails while handling an RPC from the master, the master's call() will eventually return false due to a timeout. In that situation, the master should re-assign the task given to the failed worker to another worker.

An RPC failure doesn't necessarily mean that the worker didn't execute the task; the worker may have executed it but the reply was lost, or the worker may still be executing but the master's RPC timed out. Thus, it may happen that two workers receive the same task, compute it, and generate output. Two invocations of a map or reduce function are required to generate the same output for a given input (i.e. the map and reduce functions are "functional"), so there won't be inconsistencies if subsequent processing

sometimes reads one output and sometimes the other. In addition, the MapReduce framework ensures that map and reduce function output appears atomically: the output file will either not exist, or will contain the entire output of a single execution of the map or reduce function (the lab code doesn't actually implement this, but instead only fails workers at the end of a task, so there aren't concurrent executions of a task).

> **Note:** You don't have to handle failures of the master. Making the master fault-tolerant is more difficult because it keeps state that would have to be recovered in order to resume operations after a master failure. Much of the later labs are devoted to this challenge.

Your implementation must pass the two remaining test cases in MRTest.java. The first case tests the failure of one worker, while the second test case tests handling of many failures of workers. Periodically, the test cases start new workers that the master can use to make forward progress, but these workers fail after handling a few tasks. To run these tests:

```
    @Test
    public void testOneFailure() {

    @Test
    public void testManyFailures() {
```

> **TASK:** You receive full credit for this part if your software passes the tests with worker failures (those run by the command above) when we run your software on our machines.

Your solution to Part IV should only involve modifications to Scheduler.java. If you modify other files as part of debugging, please restore their original contents and then test before submitting.

Part V: Inverted index generation (optional, as bonus)

For this optional no-credit exercise, you will build Map and Reduce functions for generating an *inverted index*.

Inverted indexes are widely used in computer science, and are particularly useful in document searching. Broadly speaking, an inverted index is a map from interesting facts

about the underlying data, to the original location of that data. For example, in the context of search, it might be a map from keywords to documents that contain those words.

We have created a second binary in InvertedIndex.java that is very similar to the WordCount.java you built earlier. You should modify mapFunc and reduceFunc in InvertedIndex.java so that they together produce an inverted index. Running InvertedIndex.java should output a list of tuples, one per line, in the following format:

```
1  $ head -n5 mrtmp.iiseq
2  0: 1 pg-metamorphosis.txt
3  000: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
   frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
   metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
4  1: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
   frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
   metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
5  10: 4 pg-dorian_gray.txt,pg-frankenstein.txt,pg-
   huckleberry_finn.txt,pg-sherlock_holmes.txt
6  100: 1 pg-sherlock_holmes.txt
```

If it is not clear from the listing above, the format is:

```
1  word: #documents documents,sorted,and,separated,by,commas
```

The expected output should be like this (under XNIX):

```
$ LC_ALL=C sort -k1,1 mrtmp.iiseq | sort -snk2,2 | grep -v '16' |
tail -10
www: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
year: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
years: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
yesterday: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
yet: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
you: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
young: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
your: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
yourself: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
zip: 8 pg-being_ernest.txt,pg-dorian_gray.txt,pg-
frankenstein.txt,pg-grimm.txt,pg-huckleberry_finn.txt,pg-
metamorphosis.txt,pg-sherlock_holmes.txt,pg-tom_sawyer.txt
```

Handin procedure

Reoprt:
You need to finish a report before submitting. The report should describe your implementation in details.

Submit your:
Mapper.java, Reducer.java, Scheduler.java, WordCount.java, InvertedIndex.java(if you have done it) and lab report to Canvas platform.
The requirement format for naming the submitted file : Lab4_StudentID_Name.zip
DEADLINE : 23:59 of June 11, 2020

Note: You may submit multiple times. We will use the timestamp of your lastsubmission for the purpose of calculating late days.