CS 6240: Assignment 1

Goals: Gain hands-on experience with parallel computation and synchronization primitives in a single-machine shared-memory environment. Set up Hadoop on your development machine, become familiar with AWS, and start working with MapReduce.

This homework is to be completed individually (i.e., no teams). You have to create all deliverables yourself from scratch. In particular, you are not allowed to look at or copy someone else's code/text and modify it. (If you use publicly available code/text, you need to cite the source in your report!)

Please submit your solution through Blackboard by the due date shown online. For late submissions you will lose one percentage point per hour after the deadline. This HW is worth 100 points and accounts for 15% of your overall homework score. To encourage early work, you will receive a 10-point bonus if you submit your solution on or before the early submission deadline stated on Blackboard. (Notice that your total score cannot exceed 100 points, but the extra points would compensate for any deductions.) Always package all your solution files, including the report, into a single standard ZIP file. Make sure your report is a **PDF** file.

For each program submission, include complete source code, build scripts, and small output files. Do not include input data, output data over 1 MB, or any sort of binaries such as JAR or class files.

The following is *optional* for this assignment, but will be required starting with the next: To enable the graders to run your solution, make sure you include a standard Makefile with the same top-level targets (e.g., *alone* and *cloud*) as the one Joe presented in class (see the Extra Material folder in the Syllabus and Course Resources section). You may simply copy Joe's Makefile and modify the variable settings in the beginning as necessary. For this Makefile to work on your machine, you need Maven and make sure that the Maven plugins and dependencies in the pom.xml file are correct. Notice that in order to use the Makefile to execute your job elegantly on the cloud as shown by Joe, you also need to set up the AWS CLI on your machine.

We strongly recommend using a Makefile already for this assignment's MapReduce part. Try to make it work as shown by Joe in class. This will save you time in the long run and prepare you for the next assignments; and using such build tools is an important skill in industry. If you are familiar with Gradle, you may also use it instead. However, we do not provide examples for Gradle.

As with all software projects, you must include a README file briefly describing all of the steps necessary to build and execute both the standalone and AWS Elastic MapReduce (EMR) versions of your program. This description should start with unzipping the original submission, include the build commands, and fully describe the execution steps. This README will also be graded and you will be able to reuse it on all of this semester's assignments with little modification (assuming you keep your project layout the same).

You have about 2 weeks to work on this assignment. Section headers include recommended timings, e.g., “complete in week 1”, to help you schedule your work. Of course, the earlier you work on this, the better.

Analyze Data with Sequential and Concurrent Java Programs (Complete in Week 1)

For this assignment we will be working with climate data from NOAA. You can get the data here:

ftp://ftp.ncdc.noaa.gov/pub/data/ghcn/daily/by_year/

You may develop your program using a smaller input file but final performance tests must use the **1912.csv.gz** file as input. The goal is to compute the **average TMAX temperature for each station ID, using a single multi-core machine (e.g., your personal laptop)**.

Program Requirements

A. To eliminate the effect of I/O, create a 1-to-1 in-memory copy of the input file as follows: Write a loader routine that takes an input filename, reads the file, and returns a String[] or List<String> containing the lines of the file. Do not change the content or order of the lines in any way. This array or list is the starting point for all of the versions of the program described in B and C.

B. Write five versions of the program. All of them will parse the lines of the file and then calculate the average TMAX temperature by station. A record will usually have format (StationId, Date, Type, Reading,…), but, as in any real data set, there could be errors or missing values. (We recommend you try to find documentation for the data and consult it.) Note that the computation should ignore all records (e.g., TMIN, PRCP) that are not TMAX records.

   1. SEQ: Sequential version that calculates the average of the TMAX temperatures by station Id. Pick a data structure that makes sense for grouping the accumulated temperatures and count of records by station. We will refer to this as the **accumulation data structure**.

   For the threaded versions, spawn exactly the maximum number of worker threads that your processor can schedule concurrently and assign about the same amount of work to each.

   2. NO-LOCK: Multi-threaded version that assigns subsets of the input String[] (or List<String>) for concurrent processing by separate threads. This version should use a single shared accumulation data structure and should use **no locks or synchronization** on it, i.e., it completely ignores any possible data inconsistency due to parallel execution.
   3. COARSE-LOCK: Multi-threaded version that assigns subsets of the input String[] (or List<String>) for processing by separate threads. This version should also use a single shared accumulation data structure and can only use the single **lock** on the **entire data structure**. Design your program to ensure (1) correct multithreaded execution and (2) minimal delays by holding the lock only when absolutely necessary.

4. FINE-LOCK: Multi-threaded version that assigns subsets of the input String[] (or List<String>) for processing by separate threads. This version should also use a single shared accumulation data structure, but should **lock only the accumulation value objects and not the whole data structure.** Design your program to ensure (1) correct multithreaded execution and (2) minimal delays by holding the locks only when absolutely necessary. Try to accomplish this using a data structure which will avoid data races.
5. NO-SHARING: Per-thread data structure multi-threaded version that assigns subsets of the input String[] (or List<String>) for processing by separate threads. Each thread should work on its own separate instance of the accumulation data structure. Hence **no locks** are needed. However, you need a barrier to determine when the separate threads have terminated and then reduce the separate data structures into a single one using the main thread.

For each of the above versions, use System.currentTimeMillis() within your code to time its execution. Note that you should **not** time the file loading routine (i.e., step A above), nor should you time the printing of results to stdout. Carefully use barriers for multi-threaded code to time only the record parsing and per station average calculations. Time the execution of your calculation code 10 times in a loop within the same execution of the program, after loading the data. Output the average, minimum, and maximum running time observed.

C. Now we want to see what happens when computing a more expensive function. For simplicity, we use the following trick to simulate this with minimal programming effort: Modify the value accumulation structure to slow down the updates by executing Fibonacci(17) whenever a temperature is added to a station's running sum. If this update occurs in a synchronized method or while holding a lock, this Fibonacci evaluation should also occur in that method / with that lock. Time all five versions exactly as in B above.

Sign Up For an AWS Account (Complete in Week 2)

Go to Amazon Web Services (AWS) and create an account. You should be able to find out how to do this on your own. Enter the requested information and you are ready to go. Make sure you apply for the education credit.

Set Up the Local Development Environment (Complete in Week 2)

We recommend using Linux for Hadoop development. (We had problems with Hadoop on Windows.) If your computer is a Windows machine, you can run Linux in a virtual machine. We tested Oracle VirtualBox: install VirtualBox (free) and create a virtual machine running Linux, e.g., Ubuntu (free). (If you are using a virtual machine, then you need to apply the following steps to the virtual machine.)

Disclaimer: The following procedure is just one way for setting up your environment. You are free to use a different approach, as long as you are able to develop, test, and then run your code on AWS. For

example, some people find the Cloudera distribution of Hadoop and Spark useful. Also consider already installing both Hadoop and Spark, as we will use Spark in a future assignment.

Download a Hadoop 2 distribution, e.g., version 2.7.3, directly from http://hadoop.apache.org/ and unzip it in your preferred directory, e.g., /usr/local. That's almost all you need to do to be able to run Hadoop code in standalone (local) mode from your IDE, e.g., Eclipse or IntelliJ. Make sure your IDE supports development in Java. Java 1.7 and 1.8 should both work.

In your IDE, you should create a Maven project. This makes it simple to build "fat jars", which recursively include dependent jars used in your MapReduce program. There are many online tutorials for installing Maven and also creating Maven projects via archetypes. These projects can be imported into your IDE or built from a shell. The provided example pom.xml file is sufficient for this WordCount assignment.

Running Word Count (Complete in Week 2)

Find the example Word Count code in the Hadoop release you downloaded. Get it to run in the IDE on your development machine. Notice that you will need to provide an input directory containing text files and a path to an output directory (that directory should not exist). Once the program runs fine, look at it closely and see what Map and Reduce are doing. Use the debugging perspective, set breakpoints in the map and reduce functions, then experiment by stepping through the code to see how it is processing the input file. Make sure you work with a small data sample.

To run your MapReduce program on EMR, follow the instructions in the AWS Setup Google Doc, linked below. In short, you need to tell EMR what jar file you want to run, what its input parameters are, where the input directory is located, and where the output should go (some location on S3). Use the input data file for Assignment 1 at http://www.ccs.neu.edu/course/cs6240f14/. (Ignore the other data files there—they are from previous years.) Unzip it and upload into an S3 bucket in your Amazon account. To access the data from Elastic MapReduce, provide the bucket location, e.g., s3n://myPath/hw1.in, as the commandline parameter for the input path of your job. Before writing any MapReduce code, try to look at the content of the file. It is in plain text format.

If you are working with Maven and have set up the CLI for AWS, then you can deploy your job from the commandline using "make cloud" (see the Makefile we distributed.) No more clicking and remembering parameter lists for the GUI! However, you need to make sure that the variables in the beginning of the Makefile are set according to your environment, e.g., bucket name on S3 etc. Note: it may be necessary to run an EMR job manually one time in order for AWS to create the default security roles and to determine your region's subnet-id.

WARNING: Leaving large data on S3 and using EMR will cost money. Read carefully what Amazon charges and estimate how much you would pay per hour of computation time before running a job. Use only the small or medium machine instances. And remember to test your code as much as possible on your development machine. When testing on AWS, first work with small data samples.

Efficient Way of Using AWS

Joe created this useful document to help you get started with an efficient industry-style setup:
https://docs.google.com/document/d/1-UjNVFasTSzhAaqLtKSmeie6JZMinhtVEqCEZwUkxeE/edit?usp=sharing

Report

Write a brief report about your findings, using the following structure.

Header

This should provide information like class number, HW number, and your name.

Weather Data Source Code

Make sure the source code for each of the program versions for the weather data analysis is easy to find in your submitted source code files. With proper project structure and naming, this should be fairly straightforward.

IMPORTANT: Please ensure that your code is properly documented. In particular, there should be comments concisely explaining the role/purpose of a class, and comments that clarify what is locked. But do not over-comment! For example, a line like "SUM += val" does not need a comment. As a rule of thumb, you want to add a brief comment for a block of code performing some non-trivial step of the computation. You also need to add a brief comment about the role of any major data structure you introduce.

Weather Data Results

For each of the versions of your sequential and multithreaded program detailed in B and C, report the minimum, average, and maximum running time observed over the 10 runs. (5 points)

Report the number of worker threads used and the speedup of the multithreaded versions based on the corresponding average running times. (5 points)

Answer the following questions in a brief and concise manner: (4 points each)

1. **Which** program version (SEQ, NO-LOCK, COARSE-LOCK, FINE-LOCK, NO-SHARING) would you normally expect to finish fastest and **why**? Do the experiments confirm your expectation? If not, try to **explain** the reasons.
2. **Which** program version (SEQ, NO-LOCK, COARSE-LOCK, FINE-LOCK, NO-SHARING) would you normally expect to finish slowest and **why**? Do the experiments confirm your expectation? If not, try to explain the reasons.
3. Compare the temperature averages returned by each program version. Report if any of them is **incorrect** or if any of the programs **crashed** because of concurrent accesses.
4. Compare the running times of SEQ and COARSE-LOCK. Try to explain **why** one is slower than the other. (Make sure to consider the results of both B and C—this might support or refute a possible hypothesis.)

5. **How** does the higher computation cost in part C (additional Fibonacci computation) affect the difference between COARSE-LOCK and FINE-LOCK? Try to **explain** the reason.

Word Count Local Execution

Show a screenshot of your IDE window. It should contain the following information:

- Project directory structure, showing that the WordCount.java file is somewhere in the src directory. (10 points)
- The console output for a successful run of the WordCount program inside the IDE. The console output refers to the job summary information Hadoop produces, not the output your job emits. Show at least the last 20 lines of the console output. (10 points)

Word Count AWS Execution

Show a similar screenshot that provides convincing evidence of a successful run of the Word Count program on AWS. Make sure you run the program using at least three machines, i.e., one master node and two workers. (10 points)

Once the execution is completed, look for the corresponding log files, in particular controller and syslog, and save them.

Deliverables

1. Source code for your sequential and parallel versions of the Java programs, including build scripts and their configuration files (optional) and a single Makefile (optional). (20 pts)
2. The report as discussed above. (1 PDF file)
3. The log files (controller and syslog) for a successful run on AWS. (plain text files) (10 points)
4. The final result file(s) produced on AWS. (plain text file) (10 points)