Name: Atindra Mardikar
Class: Nat Tuck (Tue/Fri 1:35-3:15pm)
HW-01 Report

Weather Data Source Code:

In the submitted HW folder there is a folder called source code inside which the complete package folder (avgtmax) for the java programs is present.
In this package folder (avgtmax) there are all the .java files, the makefile and manifest.MF

** There is also a small input file for testing.
** To test larger data put the file in same location as the java and makefile and change the filename in AvgTmax.java

Weather Data Results:

Report of the runtime (minimum, maximum, average) for each of the sequential and multithreaded program:

Minimum for Sequential: 702
Maximum for Sequential: 1151
Average for Sequential: 759

Minimum for No Lock: 365
Maximum for No Lock: 417
Average for No Lock: 376

Minimum for Coarse Lock: 1451
Maximum for Coarse Lock: 1558
Average for Coarse Lock: 1512

Minimum for Fine Lock: 346
Maximum for Fine Lock: 399
Average for Fine Lock: 365

Minimum for No Sharing: 331
Maximum for No Sharing: 413
Average for No Sharing: 352

----------------------With Fibonacci----------------------

Minimum for Sequential with Fibonacci: 661
Maximum for Sequential with Fibonacci: 723
Average for Sequential with Fibonacci: 685

Minimum for No Lock with Fibonacci: 344
Maximum for No Lock with Fibonacci: 457
Average for No Lock with Fibonacci: 369

Minimum for Coarse Lock with Fibonacci: 1321
Maximum for Coarse Lock with Fibonacci: 1841
Average for Coarse Lock with Fibonacci: 1621

Minimum for Fine Lock with Fibonacci: 347
Maximum for Fine Lock with Fibonacci: 451
Average for Fine Lock with Fibonacci: 369

Minimum for No Sharing with Fibonacci: 337
Maximum for No Sharing with Fibonacci: 419
Average for No Sharing with Fibonacci: 354

For the Multithreaded programs I have used **2 Threads** as I am having a dual core processor. Also when I tested using 3 Threads the programs were taking slightly more time than 2 threads.
The multithreaded programs (except the Coarse Lock) were significantly faster than the sequential (almost twice as fast as sequential), as seen from the times above.
Surprisingly, the SEQ version with Fibonacci was faster than the non-Fibonacci version of it. Same was the case with NO-LOCK.

Analysis:

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.**

   The fastest to finish should be the NO-SHARING version. This is because it is running on multiple threads and every thread class has its own copy of the accumulation data structure. So every thread is working seperately and parallely with its own accumulation data structure.
   Yes, the experiments confirm my expectations. As you can see from the runtimes NO-SHARING is faster than rest of the versions.

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**

   The Slowest to finish should be the COARSE-LOCK version. This is because though it is running on multiple threads, there is a shared accumulation data structure and to update this data structure you need to have a lock. So technically, it is being executed sequentially as the call to the update function are synchronized that is, only one call from one thread at a time. All the other calls have to wait until the lock from current call is released. Also this version is slower than SEQ because there is an extra waiting time involved. As every call is waiting for the lock to be released.
   Yes, the experiments confirm my expectations. As you can see from the runtimes COARSE-LOCK is slower than rest of the versions.

3. **Compare the temperature averages returned by each program version. Report if any of them is incorrect.**

   Experiments show that all though the NO-LOCK version is fast but it gives incorrect values for the temperature average. This is because the accumulation data structure is shared and all the threads are accessing this data simultaneously. So while adding temperature to the running sum, some call may still have older sum value which may result is incorrect answers.
   All the other versions give consistent and correct values of the temperature average.

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.)**

   The COARSE-LOCK is slower than the SEQ version. This is because, though COARSE_LOCK is running on multiple threads, there is a shared accumulation data structure and to update this data structure you need to have a lock. So technically, it is being executed sequentially as the call to the update function are synchronized that is, only one call from one thread at a time. All the other calls have to wait until the lock from current call is released. The SEQ version reads the input file sequntially and updates the accumulation data structure. So there is no waiting time to get a lock. Thus COARSE-LOCK version is slower than SEQ because there is an extra waiting time involved. As every call is waiting for the lock to be released.
   This is the case with the fibonacci version of both the program as well. As the call to the fibonaci takes place just before the updation of the accumulation data structure.

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

   The experiment shows that the difference in time (fibonacci-noFibonacci) for COARSE-LOCK is higher than the FINE-LOCK.
   The reason could be in FINE-LOCK instead of locking the complete call to the function we just lock the value part of the accumulation data structure. So other call start the execution of the function and wait just for updating the values of the data structure.

Word Count Local Execution:

Fig. Project Directory structure with WordCount.java:

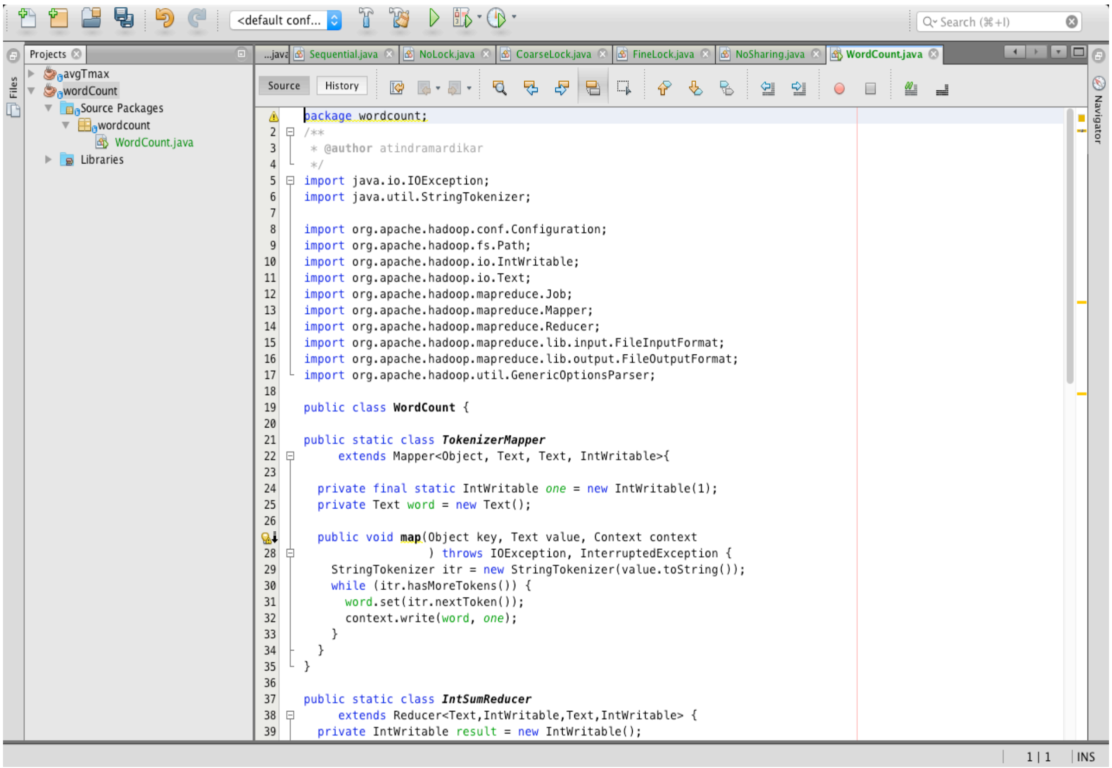
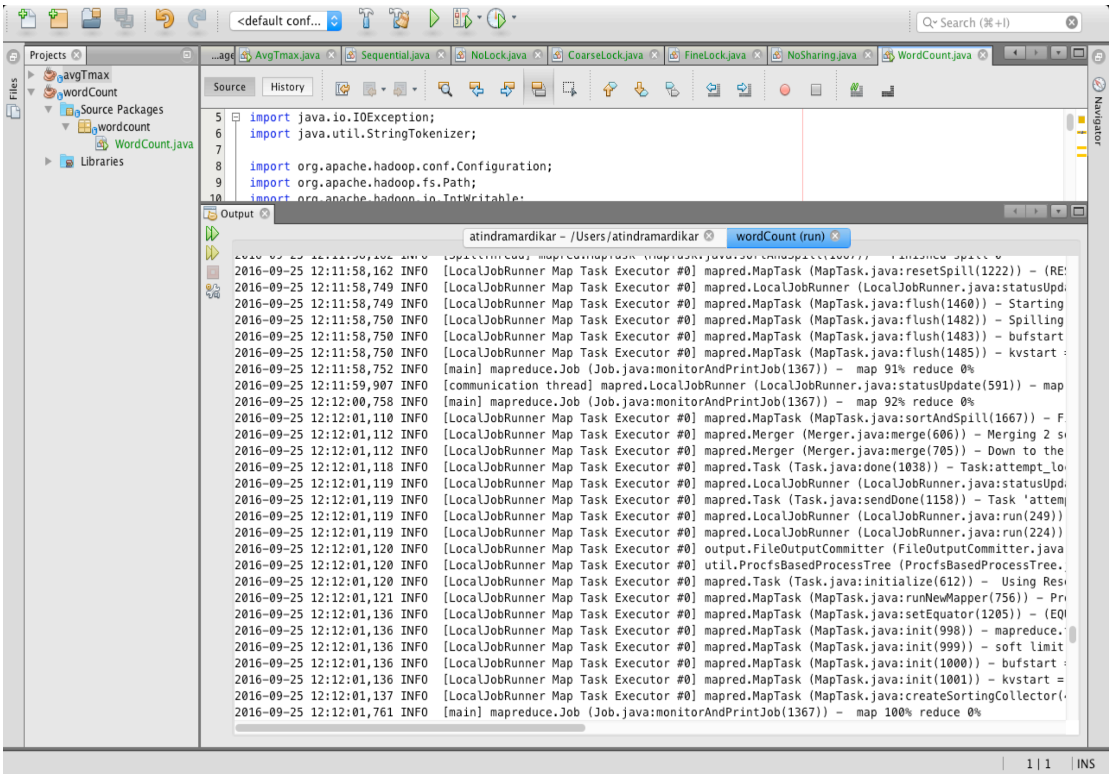
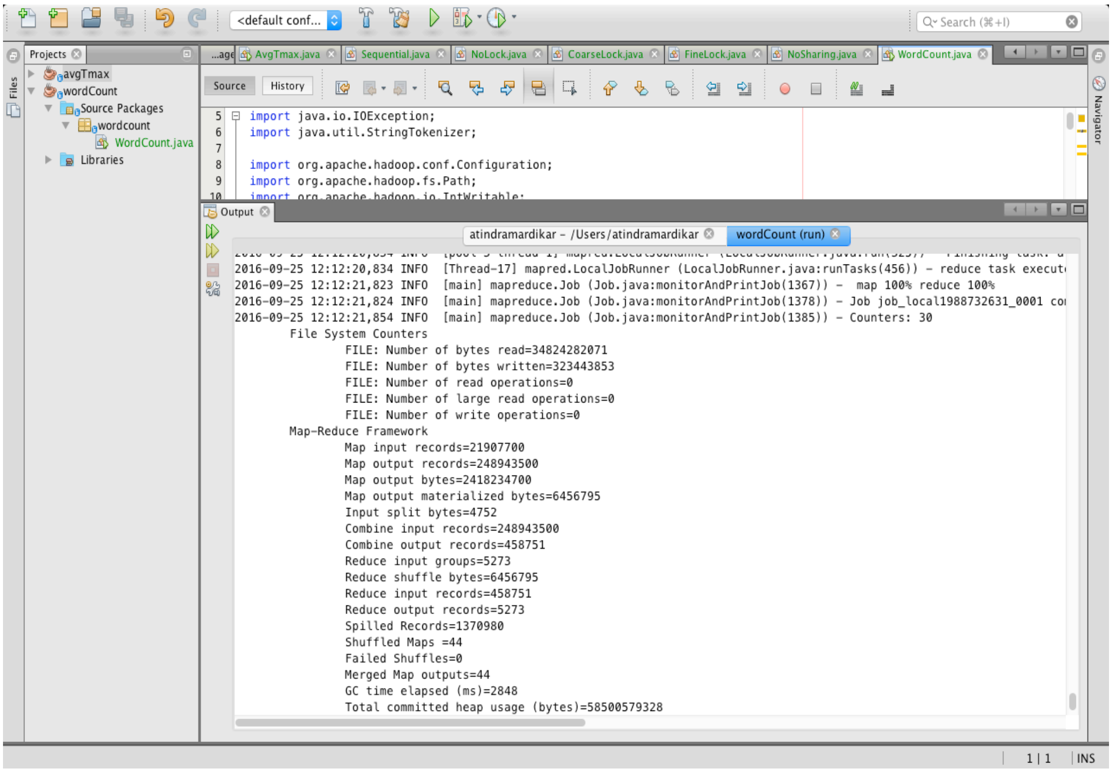


Fig. Console output for a successful run of the WordCount.java:

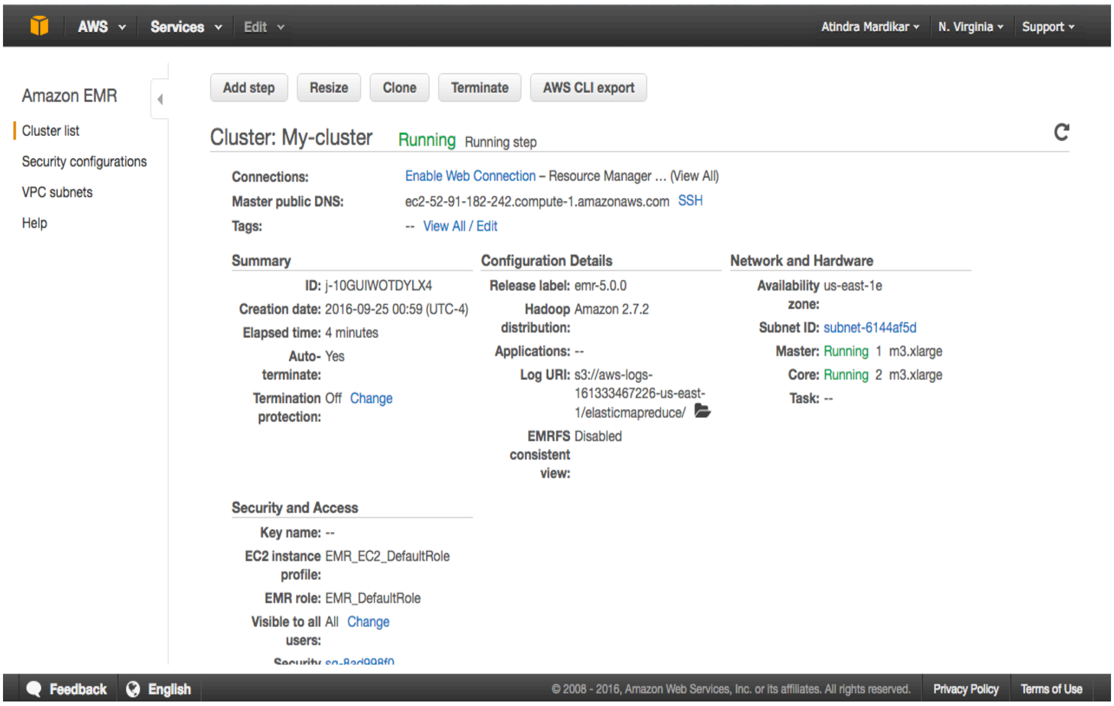


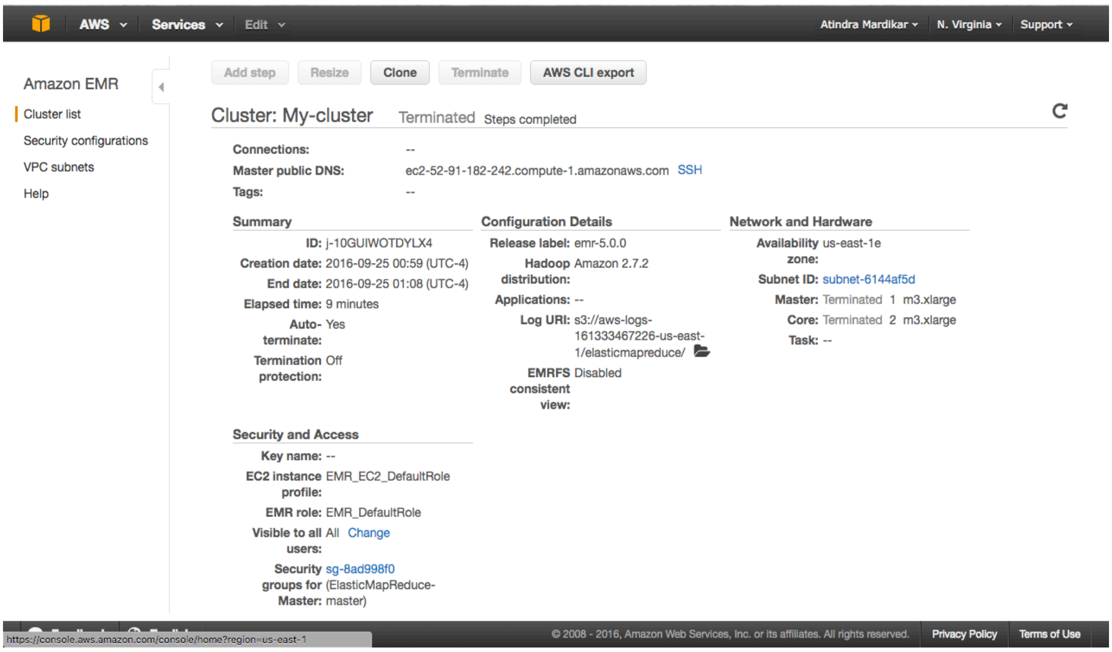


<default conf...
Search (⌘+I)
Projects
avgTmax
wordCount
Source Packages
wordcount
WordCount.java
Libraries
AvgTmax.java
Sequential.java
NoLock.java
CoarseLock.java
FineLock.java
NoSharing.java
WordCount.java
Source
History
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
Output
atindramardikar – /Users/atindramardikar
wordCount (run)
2016-09-25 12:12:20,834 INFO [Thread-17] mapred.LocalJobRunner (LocalJobRunner.java:runTasks(456)) – reduce task executo
2016-09-25 12:12:21,823 INFO [main] mapreduce.Job (Job.java:monitorAndPrintJob(1367)) – map 100% reduce 100%
2016-09-25 12:12:21,824 INFO [main] mapreduce.Job (Job.java:monitorAndPrintJob(1378)) – Job job_local1988732631_0001 co
2016-09-25 12:12:21,854 INFO [main] mapreduce.Job (Job.java:monitorAndPrintJob(1385)) – Counters: 30
File System Counters
FILE: Number of bytes read=34824282071
FILE: Number of bytes written=323443853
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
Map-Reduce Framework
Map input records=21907700
Map output records=248943500
Map output bytes=2418234700
Map output materialized bytes=6456795
Input split bytes=4752
Combine input records=248943500
Combine output records=458751
Reduce input groups=5273
Reduce shuffle bytes=6456795
Reduce input records=458751
Reduce output records=5273
Spilled Records=1370980
Shuffled Maps =44
Failed Shuffles=0
Merged Map outputs=44
GC time elapsed (ms)=2848
Total committed heap usage (bytes)=58500579328
1 | 1
INS

Word Count AWS Execution:

Fig. WordCount run on AWS:





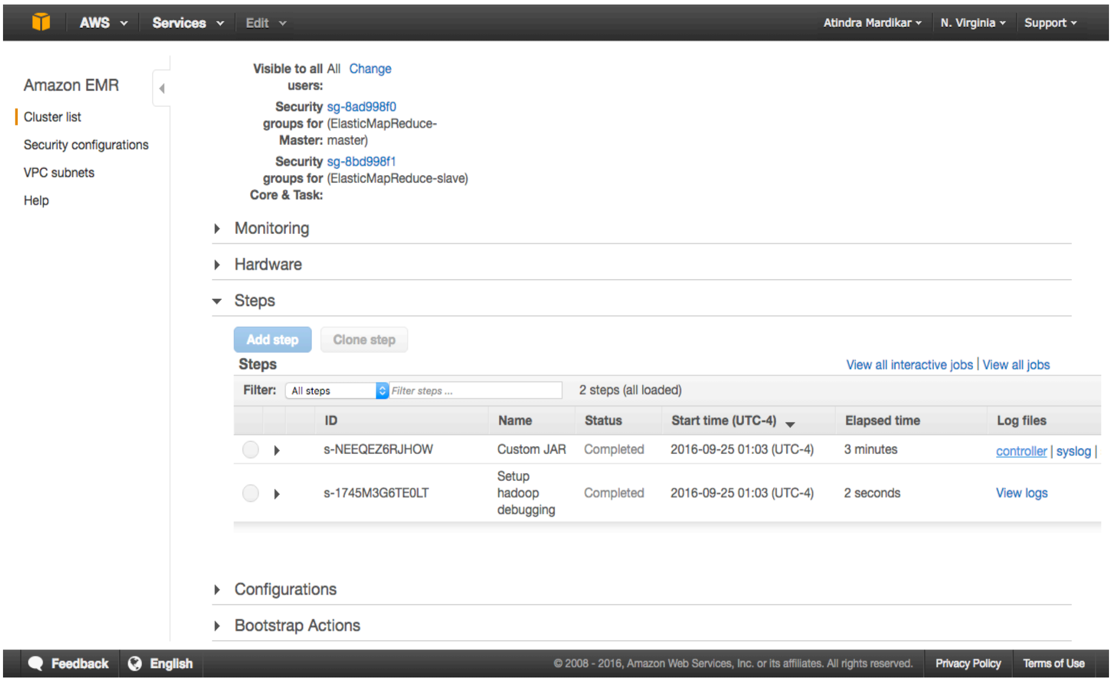


Fig. Output folder in S3 after EMR execution: