Assignment 1 - CS 6240

- Sriharsha Srinivasa Karthik Kaipa, Sec 01

Weather Data Source Code:

The source code for this part of the assignment is available in the folder named “assignment1”. The “src” folder contains the final code. Run instructions for the code are as follows:

- For cleaning the project, building the jar and then running it from the command line, please use the following rule
  `make default input=<location of input file>`
- For cleaning the project, use the following rule
  `make clean`
- For building the jar, use the following rule
  `make jar`

Weather Data Results:

The results for various runs of the program over 10 runs for both parts B and C are as follows (all data is in milliseconds):

For Sequential run:

Without Fibonacci:
Max:2779 Min:596 Avg:827
With Fibonacci:
Max:10092 Min:9956 Avg:9999

For No Lock run:

Without Fibonacci:
Max:397 Min:186 Avg:319
With Fibonacci:
Max:5506 Min:4523 Avg:4884

For Coarse Lock run:

Without Fibonacci:
Max:967 Min:914 Avg:934
With Fibonacci:
Max:16426 Min:14437 Avg:15143

For Fine Lock run:

Without Fibonacci:

Max:581 Min:349 Avg:445
With Fibonacci:
Max:6538 Min:4568 Avg:5477

For No Sharing run:

Without Fibonacci:
Max:432 Min:324 Avg:353
With Fibonacci:
Max:6711 Min:4526 Avg:5007

For the system on which the code was run to get the above data, the number of worker threads were 4 (dual core CPU - i7-6500U)

Speed up for multithreaded implementation:

Threaded no lock:
Without FIbonacci: 2.5925
With Fibonacci: 2.0473

Threaded coarse lock:
Without Fibonacci: 0.8854
With Fibonacci: 0.6603

Threaded fine lock:
Without Fibonacci: 1.8584
With Fibonacci: 1.8256

Threaded no sharing:
Without Fibonacci: 2.3428
With Fibonacci: 1.997

1. Disregarding whether the final data is correct or not, multithreaded implementation of the program with no lock is expected to complete first. This is because, in other implementations of multithreading with locks involved, it would take time for multiple threads to simultaneously update/insert data into the shared data structure, but in a no lock scenario, the different threads have no regards to whether another thread is working on the data structure or not, leading to no wait periods for update/insert. This is observed in the reported data above.
   If correctness of the final data is to be taken into account, no sharing implementation should finish the earliest since each thread has it's own data accumulation data structure on which there is no locking required since there is only one thread working on it. The only delay it might encounter would be during the data accumulation stage but the final run times should be comparable or lesser than that seen from the next fastest, fine lock implementation. This is observed in the reported data above.
2. Irrespective of whether the final data is correct or not, sequential execution is expected to be the slowest because there is only one thread working diligently

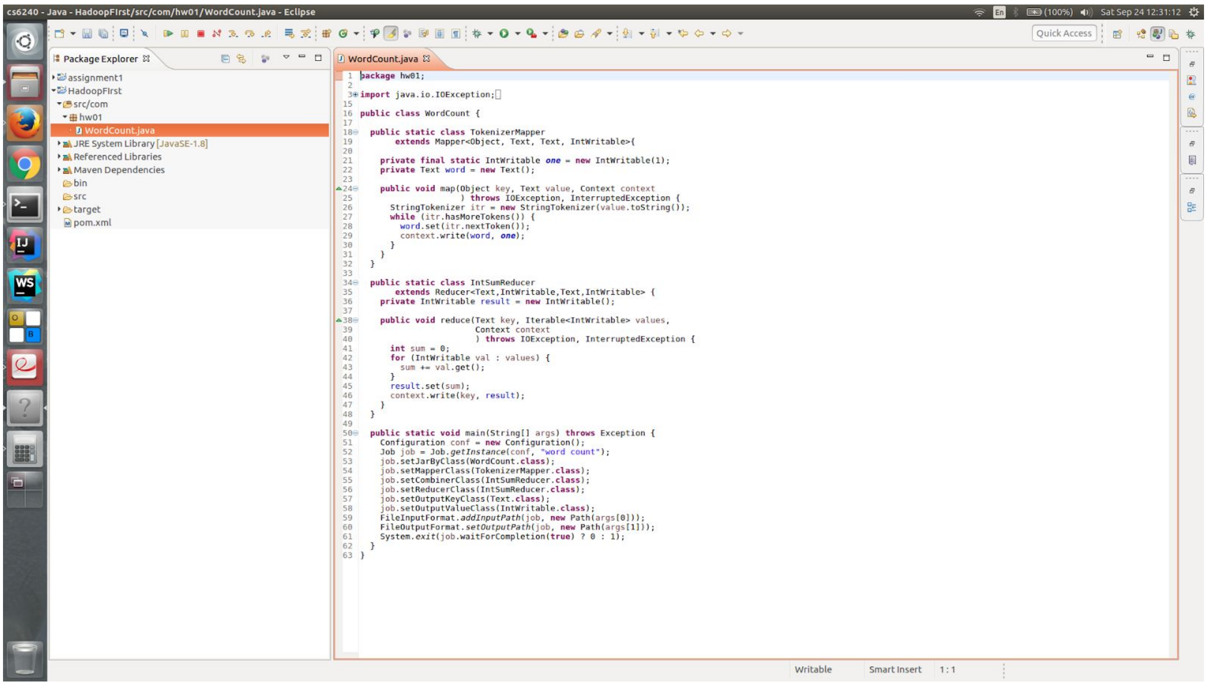
through the entire data set one after the other and updates the records based on the input data. This is not seen in the data reported above. This anomaly might be because of the overhead created because of the locking and unlocking of the data structure and also because of the overhead for thread backoff. Maybe, for considerably larger data sets, the expected run times might be seen.

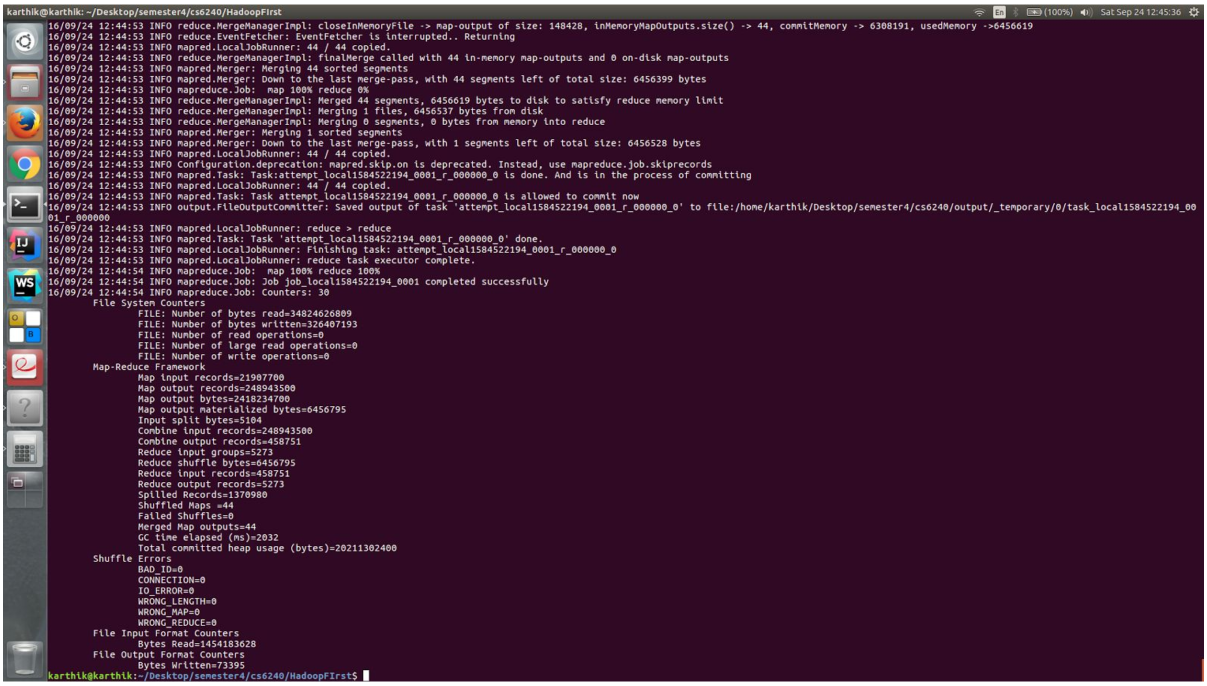
3. After examining the output data, it is clear that multithreaded implementation without locks gives false data since there would be simultaneous updates/inserts which would cause the data to be highly inconsistent.
4. As mentioned in point (2), in for this data set (1912.csv), and this particular implementation of sequential and coarse lock programs, it is seen that coarse lock is taking longer to complete when compared to sequential execution. This might be because of the overhead created from locking the data structure and thread backoff and could also be because of how the OS schedules the threads involved in result accumulation.
5. If observed from the above data, the run time for coarse lock with fibonacci is approximately 16.2131 times that of when there is no fibonacci running. The same for fine lock is seen as 12.3079. As can be seen, the higher computation of fibonacci is causing a more drastic increase in run time for coarse lock since only one thread can run the fibonacci computation at any given time but in the case of fine lock, all threads could, in theory, execute fibonacci at the same time causing for a faster completion of the code.

Word Count Local Execution:

The source code for Hadoop Word Count is available in the folder “HadoopFIrst”. The “src” folder contains the source code for WordCount and a sample input is available in the folder “input” inside this project. Output is created in a folder named “output”, alongside “input” folder. Run instructions for the code are as follows:

- For cleaning the project, building the jar and then running the jar locally, use the following code:
  `make local`
  Point to note, I have hadoop installed such that executing “hadoop” from the terminal runs hadoop.
- For cleaning the project, use the following code:
  `make clean`
- For building the jar, use the following code:
  `make jar`


cs6240 - Java - HadoopFIrst/src/com/hw01/WordCount.java - Eclipse
Package Explorer
assignment1
HadoopFIrst
src/com
hw01
WordCount.java
JRE System Library [JavaSE-1.8]
Referenced Libraries
Maven Dependencies
bin
src
target
pom.xml
WordCount.java
package hw01;
import java.io.IOException;
public class WordCount {
public static class TokenizerMapper
extends Mapper<Object, Text, Text, IntWritable>{
private final static IntWritable one = new IntWritable(1);
private Text word = new Text();
public void map(Object key, Text value, Context context
) throws IOException, InterruptedException {
StringTokenizer itr = new StringTokenizer(value.toString());
while (itr.hasMoreTokens()) {
word.set(itr.nextToken());
context.write(word, one);
}
}
}
public static class IntSumReducer
extends Reducer<Text,IntWritable,Text,IntWritable> {
private IntWritable result = new IntWritable();
public void reduce(Text key, Iterable<IntWritable> values,
Context context
) throws IOException, InterruptedException {
int sum = 0;
for (IntWritable val : values) {
sum += val.get();
}
result.set(sum);
context.write(key, result);
}
}
public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
Writable
Smart Insert
1 : 1


karthik@karthik: ~/Desktop/semester4/cs6240/HadoopFIrst
16/09/24 12:44:53 INFO reduce.MergeManagerImpl: closeInMemoryFile -> map-output of size: 148428, inMemoryMapOutputs.size() -> 44, commitMemory -> 6308191, usedMemory ->6456619
16/09/24 12:44:53 INFO reduce.EventFetcher: EventFetcher is interrupted.. Returning
16/09/24 12:44:53 INFO mapred.LocalJobRunner: 44 / 44 copied.
16/09/24 12:44:53 INFO reduce.MergeManagerImpl: finalMerge called with 44 in-memory map-outputs and 0 on-disk map-outputs
16/09/24 12:44:53 INFO mapred.Merger: Merging 44 sorted segments
16/09/24 12:44:53 INFO mapred.Merger: Down to the last merge-pass, with 44 segments left of total size: 6456399 bytes
16/09/24 12:44:53 INFO mapreduce.Job: map 100% reduce 0%
16/09/24 12:44:53 INFO reduce.MergeManagerImpl: Merged 44 segments, 6456619 bytes to disk to satisfy reduce memory limit
16/09/24 12:44:53 INFO reduce.MergeManagerImpl: Merging 1 files, 6456537 bytes from disk
16/09/24 12:44:53 INFO reduce.MergeManagerImpl: Merging 0 segments, 0 bytes from memory into reduce
16/09/24 12:44:53 INFO mapred.Merger: Merging 1 sorted segments
16/09/24 12:44:53 INFO mapred.Merger: Down to the last merge-pass, with 1 segments left of total size: 6456528 bytes
16/09/24 12:44:53 INFO mapred.LocalJobRunner: 44 / 44 copied.
16/09/24 12:44:53 INFO Configuration.deprecation: mapred.skip.on is deprecated. Instead, use mapreduce.job.skiprecords
16/09/24 12:44:53 INFO mapred.Task: Task:attempt_local1584522194_0001_r_000000_0 is done. And is in the process of committing
16/09/24 12:44:53 INFO mapred.LocalJobRunner: 44 / 44 copied.
16/09/24 12:44:53 INFO mapred.Task: Task attempt_local1584522194_0001_r_000000_0 is allowed to commit now
16/09/24 12:44:53 INFO output.FileOutputCommitter: Saved output of task 'attempt_local1584522194_0001_r_000000_0' to file:/home/karthik/Desktop/semester4/cs6240/output/_temporary/0/task_local1584522194_0001_r_000000
16/09/24 12:44:53 INFO mapred.LocalJobRunner: reduce > reduce
16/09/24 12:44:53 INFO mapred.Task: Task 'attempt_local1584522194_0001_r_000000_0' done.
16/09/24 12:44:53 INFO mapred.LocalJobRunner: Finishing task: attempt_local1584522194_0001_r_000000_0
16/09/24 12:44:53 INFO mapred.LocalJobRunner: reduce task executor complete.
16/09/24 12:44:54 INFO mapreduce.Job: map 100% reduce 100%
16/09/24 12:44:54 INFO mapreduce.Job: Job job_local1584522194_0001 completed successfully
16/09/24 12:44:54 INFO mapreduce.Job: Counters: 30
File System Counters
FILE: Number of bytes read=34824626809
FILE: Number of bytes written=326407193
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=5104
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)=2032
Total committed heap usage (bytes)=20211302400
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=1454183628
File Output Format Counters
Bytes Written=73395
karthik@karthik:~/Desktop/semester4/cs6240/HadoopFIrst$

Word Count AWS Execution: