HU Extension Assignment 11 E-90 Cloud Computing

Handed out: 11/15/2015 Due by 11:59 PM on Friday, 11/20/2015

Place all of your narratives and illustrations in a single Word or PDF document named E90_LastNameFirstNameHW11.docx [.pdf]. Use this assignment as the initial template. Please implement code solutions as a separate class in one (Java, C#, Ruby, Python,..) project. Add description of your steps and your code below problem statements used for each problem.
Upload your homework file and your working code (e.g., Filename.java) into your Assignment 11 folder. Do not include executables. Please also do not zip your files.

Please use any programming language you are comfortable with.

Problem 1: Implement the Monte Carlo algorithm for determining the value of number Π (pi) as described on pages 12 to 14 of the lecture slides on Parallel Programming. Make sure that your code accepts as input parameters a seed for your random number generator and the number of darts you are shooting on the square. Once you have the basic algorithm working organize execution of your code in a master-worker arrangement. The master would provide the worker with the seed value for the random number generator and the number of dart throws. The worker would return an approximate value of Π. In a parallel universe the master would instantiate 10 or 100 workers, instruct each of them to do the work, collect results and perform final aggregation of those results. Do not attempt to do that. Simply have your master repeatedly call the same worker 10 or 100 times. Compare the error of the aggregate value produced by the "final" master's result with the error each worker made. Experiment with the number of dart throws to judge how big that number has to be to achieve 1% accuracy.
Total Points: 50

In a class named CalculatePi I created two functions, a main function and a function called FindPi. The main function acts as a Master and FindPi is the worker function. The main function's processing loops through calls to the worker function *numWorkers* times allowing for the ability to test with any number of workers.

The Main Function calls the worker by sending a starting seed number to generate a random numbers for x,y coordinates to simulate attempts at throwing darts within a circle that is within a square in order to calculate pi. The master function tracks all results and finds an average of the results and determines its accuracy in value in comparison to pi. After adjusting the number of darts, regardless of the number of workers, it takes about 1000 darts to have about 70-90% of the workers accurately calculating pi. About 20000 darts provides a consistent accuracy within 1% for all workers each time.

The main Function (The Master):

```
// Master Loops through a number that represents
// Number of workers
for (int x = 0; x < numWorkers; x++) {
     SecureRandom random = new SecureRandom();
     double seedNum = random.nextDouble() * numWorkers;
// Add to an array the value returned from
// Sending seedNum, numTotalDarts to worker to calculate pi
     double currentPi = findPi(seedNum,numTotalDarts);
     pies.add(currentPi);
     if (Math.abs((Math.PI-currentPi)) <= .0314){
          successCount++;
     }
// Sum up all values in array and divide by array length
// To determine average result
     double sum = 0;
     for (double p : pies) {
          sum += p;
     }
     double average = 1.0d * sum / pies.size();

System.out.println("Results after " + count + " workers:
");
System.out.println("Current value of average of master pi:
" + formatter.format(average));

System.out.println(successCount + " successful workers: " +
"of " + count + " workers with " +numTotalDarts + " darts
thrown.");

double diff = Math.abs((Math.PI-average));

System.out.println("Overall difference from pi: " +
formatter.format(diff));

if (diff <= .0314){
     System.out.println("Overall Calculated pi is within 1%
accuracy");
} else {
     System.out.println("Overall Calculated pi is not
accurate");
}
```

Worker Function:

```
// Receives Seed Number and Number of Throws from Master
private static double findPi(double seedNum, double
numTotalDarts) {
     int numDartsIn = 0;
     // Throwing darts and finding number hit in circle
          for (int i = 0; i < numTotalDarts; i++) {
               double xcoord = Math.random();
               double ycoord = seedNum ;

               // if dart is in circle
               if (xcoord*xcoord + ycoord*ycoord <= 1) {
                    numDartsIn++;
               }
               SecureRandom random = new SecureRandom();
               seedNum = random.nextDouble();
          }
     double approx_pi = (numDartsIn * 4)/numTotalDarts;
     return approx_pi;
}
```

Sample Output:

```
Results after 9 workers:
Current value of average of master pi: 3.143
9 successful workers: of 9 workers with 20000.0 darts
thrown.
Overall difference from pi: 0.001
Overall Calculated pi is within 1% accuracy

Results after 10 workers:
Current value of average of master pi: 3.142
10 successful workers: of 10 workers with 20000.0 darts
thrown.
Overall difference from pi: 0.001
Overall Calculated pi is within 1% accuracy
```

Problem 2: Implement algorithm for counting words in a text described on page 44 with two functions:

a) map() that would take a file with a page of text in plain .txt format and produce a collection of name value pairs where the name is the word itself, and the value is number 1.
b) reduce() that would take any collection of (word, 1) pairs and aggregate it into an ordered word counts (word, count) collection.

Your code will need a master role as well. Namely, the master would fetch a file from the file system, pass it to the map() function, receive the results of map() calculation and pass them to the reduce() function.
Experiment with several pages of random text you copy from the internet.
Find an automated way to verify your results.
Total Points: 50

This program consists of three functions within a java class named CountWords.java. The text file is plain text of Abraham Lincoln's Gettysburg Address.

The main function is the Master Function.

```
// master()
//   * Reads a file from the file system
//   * Passes the file to map() one line at a time
//   * Receives the results of map()
//   * Passes the results of map() to reduce()
//   * Receives and displays the aggregated results
of reduce()
public static void main(String[] args) {
     try {
          String path = "/Users/marnie/Desktop/
gettysburg.txt";
          FileReader file_reader = new
FileReader(path);
          BufferedReader buff_reader = new
BufferedReader(file_reader);
          String line =  " ";
          ArrayList<Map<String,Integer>>
wordMapsList = new
ArrayList<Map<String,Integer>>();
          ArrayList<Map<String,Integer>>
wordCountList = new
ArrayList<Map<String,Integer>>();
```

```
while ((line = buff_reader.readLine()) != null) {
    // For every line of text Send it to map()
        wordMapsList= map(line);
    // Then send that to reduce()
        wordCountList.add(reduce(wordMapsList));
}
buff_reader.close();
// Create a new TreeMap of sorted words and total
counts
TreeMap<String, Integer> wordCount = new
TreeMap<String, Integer>();
for (Map<String,Integer> wordMap : wordCountList){
    for (Object  key : wordMap.keySet()) {
        String word = key.toString();
        Integer val = (Integer) wordMap.get(word);
        if (wordCount.containsKey(word)){
            val = (Integer) wordCount.get(word);
            wordCount.remove(word);
            wordCount.put(word, (val+ 1));
        } else {
            wordCount.put(word,val);
        }
    }
}
int totalWords= 0;
for (Object  key : wordCount.keySet()) {
    System.out.println(key.toString() + ", " +
wordCount.get(key.toString()));
    totalWords = totalWords +
wordCount.get(key.toString());
}
System.out.println("Total Words counted: " +
totalWords);}
```

The map function:

```
// map()
//   * Receives a line of text from master()
//   * Splits the file up on "" and places each word
//   * Into a HashMap <word,1> all HashMaps into a
List
//   * Return this list of HashMaps back to master()
public static ArrayList<Map<String, Integer>>
map(String line){
//   * Splits the line up on "" and places each word
//     into a HashMap <word,1> all HashMaps into a
List
List<String> list = Arrays.asList(line.split("\\W
+"));
ArrayList<Map<String, Integer>> wordMapList = new
ArrayList<Map<String,Integer>>();
for (String word : list){
    // creating new HashMap
    HashMap<String, Integer> map = new
HashMap<String, Integer>();
    word = word.toLowerCase();
    map.put(word,1);
    wordMapList.add(map);
}
return wordMapList;
}
```

The reduce function:

```
// reduce()
//   * Receives a List of HashMaps from master()
//   * Parses list and counts occurrences of words
//   * Return HashMap of words with counts to
master()
public static Map<String,Integer>
reduce(ArrayList<Map<String,Integer>> wordList){
    Map<String, Integer> wordCount = new
HashMap<String, Integer>();
```

```
// Loop through the list
// For each HashMap in List
for (Map<String,Integer> wordMap : wordList) {
    for (Object  key : wordMap.keySet()) {
        String word = key.toString();
        Integer val = (Integer) wordMap.get(word);
    // Is this word in HashMap already?
    // If it already exists, then add 1 to the
value
    if (wordCount.containsKey(word)){
        val = (Integer) wordCount.get(word);
        wordCount.remove(word);
        wordCount.put(word, (val+ 1));
    // if not then add to a local HashMap as
key,value
    } else {
        wordCount.put(word,val);
    }
}
}
// Return HashMap to master()
return wordCount;
}
```

Verified at http://www.csgnetwork.com/documentanalystcalc.html

Output:

```
a, 7
above, 1
add, 1
advanced, 1
ago, 1
all, 1
altogether, 1
and, 6
any, 1
are, 3
as, 1
battle, 1
be, 2
```

```
before, 1
birth, 1
brave, 1
brought, 1
but, 2
by, 1
can, 5
cause, 1
civil, 1
come, 1
conceived, 2
consecrate, 1
consecrated, 1
continent, 1
created, 1
dead, 3
dedicate, 2
dedicated, 4
detract, 1
devotion, 2
did, 1
died, 1
do, 1
earth, 1
endure, 1
engaged, 1
equal, 1
far, 2
fathers, 1
field, 2
final, 1
fitting, 1
for, 5
forget, 1
forth, 1
fought, 1
four, 1
freedom, 1
from, 2
full, 1
```

```
gave, 2
god, 1
government, 1
great, 3
ground, 1
hallow, 1
have, 5
here, 8
highly, 1
honored, 1
in, 4
increased, 1
is, 3
it, 5
larger, 1
last, 1
liberty, 1
little, 1
live, 1
lives, 1
living, 2
long, 2
measure, 1
men, 2
met, 1
might, 1
nation, 5
never, 1
new, 2
nobly, 1
nor, 1
not, 5
note, 1
now, 1
of, 5
on, 2
or, 2
our, 2
people, 3
perish, 1
```

```
place, 1
poor, 1
portion, 1
power, 1
proper, 1
proposition, 1
rather, 2
remaining, 1
remember, 1
resolve, 1
resting, 1
say, 1
score, 1
sense, 1
seven, 1
shall, 3
should, 1
so, 3
struggled, 1
take, 1
task, 1
testing, 1
that, 13
the, 11
their, 1
these, 2
they, 3
this, 4
those, 1
thus, 1
to, 8
under, 1
unfinished, 1
us, 3
vain, 1
war, 2
we, 10
what, 2
whether, 1
which, 2
```

```
who, 3
will, 1
work, 1
world, 1
years, 1
Total Words counted: 272
```