Homework 2: due by 5pm on Wednesday, February 11, 2015 (Total: 100 points)

Instructor: Vivek Sarkar

All homeworks should be submitted in a directory named hw_2 using the turn-in script. In case of problems using the script, you should email a zip file containing the directory to `comp322-staff@mailman.rice.edu` before the deadline. See course wiki for late submission penalties.

Honor Code Policy: All submitted homeworks are expected to be the result of your individual effort. You are free to discuss course material and approaches to problems with your other classmates, the teaching assistants and the professor, but you should never misrepresent someone elses work as your own. If you use any material from external sources, you must provide proper attribution.

1 Written Assignments (50 points total)

Please submit your solutions to the written assignments in either a plain text file named `hw_2_written.txt` *or a PDF file named* `hw_2_written.pdf` *in the hw_2 directory.*

1.1 Finish Accumulators (25 points)

Consider the pseudocode shown below in Listing 1 for a Parallel Search algorithm that is intended to compute C, the number of occurrences of the pattern array in the text array. What possible values can variables *count0*, *count1*, and *count2* contain after line 13, and why? Write your answers in terms of M, N, and C.

```
1  static int count0 = 0; // static field
2  . . .
3  accumulator a = new accumulator(SUM, int.class);
4  finish (a) {
5    for (int i = 0; i <= N - M; i++)
6    async {
7      int j;
8      for (j = 0; j < M; j++) if (text[i+j] != pattern[j]) break;
9      if (j == M) { count0++; a.put(1); } // found at offset i
10     int count1 = a.get().intValue();
11   } // for-async
12 } // finish
13 int count2 = a.get().intValue();
```

Listing 1: Parallel Search using Finish Accumulators

1.2 Inverted Index using Map Reduce (25 points)

Consider the Map Reduce pattern described in the Topic 2.4 video lectures, and summarized in Section 2.4 of the Module 1 handout. Assume an input set of key-value pairs of the form (*file*, *word*). Define the map and reduce functions to obtain an inverted index consisting of (*word*, {*file1, file2, ...*}) key-value pairs *i.e.,* the inverted index should specify for each *word* the set of files (*file1, file2, ...*) that contain the word.

2 Programming Assignment (50 points)

2.1 Setup

You should checkout the `hw_2` project from your svn repository at:
`https://svn.rice.edu/r/comp322/turnin/S15/`*`NETID`*`/hw_2`.

You will notice the following Java files in your project: `GeneralizedReduce.java`, `GeneralizedReduceApp.java`, `SumReduction.java`, `SumReductionMain.java`, and `SumReductionTest.java`.

IMPORTANT: Among these files, you should edit `GeneralizedReduce.java`, and leave the other files unchanged. Further, you should not modify any of the existing declarations in `GeneralizedReduce.java`. Making additional modifications will make it difficult for the teaching staff to run automated scripts to evaluate your code, and may result in penalties to your grade.

2.2 Generalized Reduce (50 points)

The goal of this assignment is to implement a `GeneralizedReduce` *class to perform a user-specified reduction on an array of generic objects.* A skeleton for this class is available in the `GeneralizedReduce.java` file.

Your `GeneralizedReduce` class should support any class that implements the `GeneralizedReduceApp` interface with `init()` and `combine()` methods to define a reduction operation. As an example, class `SumReduction` implements the `GeneralizedReduceApp` interface to specify a sum reduction. Further, `SumReductionMain.java` contains a test harness for `SumReduction`.

You should use abstract metrics to evaluate the parallelism in your solution. Specifically, any reduction client should include a call to `doWork(1)` for each call to the `combine()` method, as in `SumReduction.java`. Thus, the total WORK and CPL (critical path length) for your program executions will be evaluated assuming that each call to `combine()` takes one unit of time. You will not be penalized if the actual execution time of your parallel program is large due to overheads, so long as the abstract metrics are correct. Note that the abstract metrics are independent of the computer that you run your program on.

Your submission should include the following in the hw_2 directory. All code should include basic documentation for each method in each class, and follow best practices for coding conventions (even if the code that we provided does not!) Please feel free to state which coding convention you use. If you do not have a preference, we recommend the Google Java Style defined at `https://google-styleguide.googlecode.com/svn/trunk/javaguide.html`.

1. (20 points) A complete parallel implementation of the `GeneralizedReduce` class. You should aim for maximum parallelism, assuming that each call to `combine()` just does 1 unit of work. The implementation can use `async`, `finish` and/or `future` constructs, but not finish accumulators. Test your implementation with the `SumReductionTest` test harness.

2. (15 points) A complete implementation of new `MaxReduction` and `MaxReductionMain` classes to test your `GeneralizedReduce` implementation for a special max-index reduction that returns both the max value and its index in an input integer array. If the max value has multiple occurrences in the input array, then you should return the index of the first occurrence. For example, if the input array is [(5,0), (1,1), (-8,2), (5,3), (0,4)] where each element consists of a value paired with its index in the array, the output should consist of (5, 0), since 5 is the largest value in the array and its first occurrence is at index 0.

3. (15 points) A report file formatted either as a plain text file named `hw_2_report.txt` or a PDF file named `hw_2_report.pdf` in the hw_2 directory. The report should summarize the design of your parallel solution, and explain why you believe that your implementation is correct, data-race-free, and maximally parallel. It should also include abstract metrics obtained from executing both SumReductionMain and MaxReductionMain with randomly-generated arrays of size n = 1, 10, 100, 1000. The abstract metrics should include the WORK, CPL, and IDEAL PARALLELISM (= WORK/CPL) values from each run.