Homework 4: due by 5:00pm on Monday, April 6, 2015 (Total: 100 points)

Instructor: Vivek Sarkar

All homeworks should be submitted in a directory named hw_4 using svn or turnin as before. In case of problems, you should email a zip file containing the directory to `comp322-staff@rice.edu` **before the deadline. See course wiki for slip day policy and 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 Assignment (50 points total)

Please submit your solution to this assignment in a plain text file named `hw_4_written.txt` *in the submission system. Syntactic errors in program text will not be penalized in the written assignment e.g., missing semicolons, incorrect spelling of keywords, etc. Pseudo-code is acceptable so long as the meaning of your program is unambiguous.*

1.1 Phasers and Atomic Integers (25 points)

Consider a desired inter-task synchronization pattern in Figure 1, followed by an incomplete HJ program in Listing 1.

1. (15 points) Complete the missing phaser declarations (SIG, WAIT, or SIG_WAIT) and phaser registrations in lines 3, 8, 11, 14, to obtain a complete HJ program that implements the inter-task synchronization pattern in Figure 1.
2. (10 points) Assume that `a` is a reference to a Java AtomicInteger object initialized to zero, and that each of steps A() through L() include the following code, "`int n = a.getAndAdd(1);`". What are the possible values that variable `n` can receive in step H()?

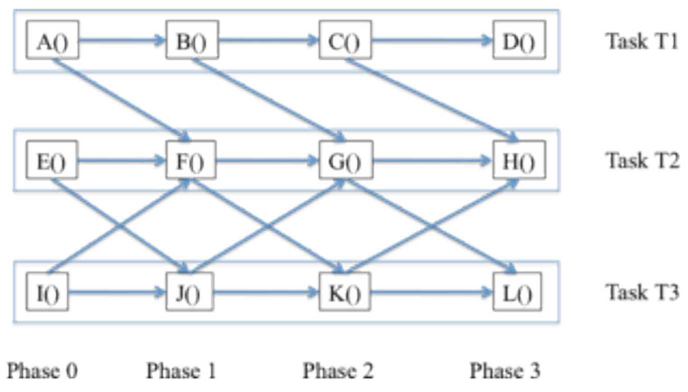

Figure 1: Desired inter-task synchronization pattern using phasers

```
finish(() -> {
  // INSERT MISSING PHASER DECLARATIONS AND INITIALIZATIONS BELOW




  // INSERT MISSING PHASER REGISTRATIONS BELOW
  asyncPhased(_______________________________________ , () -> { //Task T1
      A(); next; B(); next; C(); next; D();
  }); // Task T1
  asyncPhased(_______________________________________ , () -> { //Task T2
      E(); next; F(); next; G(); next; H();
  }); // Task T2
  asyncPhased(_______________________________________ , () -> { //Task T3
      I(); next; J(); next; K(); next; L();
  }); // Task T3
}); // finish
```

Listing 1: Incomplete HJ program with missing phaser declarations and registrations

1.2 HJ isolated constructs vs. Java atomic variables (25 points)

Many applications use Pseudo-Random Number Generators (PRNGs) as in class IsolatedPRNG in Listing 2. The idea is that the seed field takes a linear sequence of values obtained by successive calls to the nextInt() method as shown in line 6. The use of the HJ isolated construct in lines 4–9 ensures that there will be no data race on the seed field if nextSeed() is called in parallel by multiple tasks. The serialization of the isolated construct instances will determine which task obtains which seed in the sequence.

```
 1 class IsolatedPRNG {
 2   private int seed;
 3   public int nextSeed() {
 4     final int retVal = isolatedWithReturn(() -> {
 5       final int curSeed = seed;
 6       final int newSeed = nextInt(curSeed);
 7       seed = newSeed;
 8       return curSeed;
 9     });
10     return retVal;
11   } // nextSeed()
12   . . . // Definition of nextInt(), constructors, etc.
13 } // IsolatedPRNG
```

Listing 2: Concurrent PRNG implemented using isolated construct

Since the isolated construct is not available in standard Java, class AtomicPRNG in Listing 3 attempts to implement the same functionality by using Java atomic variables instead.

1. (15 points) Assuming a scenario where nextSeed() is called by multiple tasks in parallel on the same PRNG object, state if the implementation of AtomicPRNG.nextSeed() has the same semantics as that of IsolatedPRNG.nextSeed(). If so, why? If not, why not?

   By "same semantics", we mean that for every IsolatedPRNG execution, we can find an equivalent AtomicPRNG execution that results in the same answer, and for every AtomicPRNG execution, we can find an equivalent IsolatedPRNG execution that results in the same answer.

2. (10 points) Why is the "while (true)" loop needed in line 5 of AtomicPRNG.nextSeed()? What would happen if the while(true) loop was replaced by a loop that executes for only iteration?

```
1  class AtomicPRNG {
2    private AtomicInteger seed;
3    public int nextSeed() {
4      int retVal;
5      while (true) {
6        retVal = seed.get();
7        int nextSeedVal = nextInt(retVal);
8        if (seed.compareAndSet(retVal, nextSeedVal)) break;
9      } // while
10    return retVal
11   } // nextSeed()
12   . . . // Definition of nextInt(), constructors, etc.
13 } // AtomicPRNG
```

Listing 3: Concurrent PRNG implemented using Java's AtomicInteger class

2 Programming Assignment (50 points total)

Many problems from artificial intelligence can be defined as combinatorial optimization problems. For example, Branch and Bound (BnB) is a widely used tool for solving large scale NP-hard combinatorial optimization problems. A BnB algorithm searches the complete space of solutions for a given problem for the best solution. Subproblems are derived from the originally given problem through the addition of new constraints. An objective function computes the lower/upper bounds for each subproblem. The upper bound is the worst value for the potential optimal solution; the lower bound is the best value. The entire tree maintains a global upper bound (GUB): this is the best upper bound of all nodes. Nodes with a lower bound higher than the GUB are eliminated from the tree because branching these sub-problems will not lead to the optimal solution. In many practical cases, the amount of pruning that occurs in this type of BnB algorithm can be very significant.

In parallel implementations, pruning the branches of the search tree may lead to terminating existing computations. The structure of the BnB search requires the ability to terminate individual subtrees of the search tree. A BnB version of our array search example is where we are interested in finding the lowest index of the goal item if it exists in the array. We can achieve this by using a `MinimaEureka` instance. In our EuPM, the GUB is available in the `MinimaEureka` instance, `eu`, that a speculative task is registered on and can be retrieved by a call to `eu.get()`. Calls to `check` and `offer` pass the current known bounds or solution, respectively, as the argument. If the argument in the `offer` call is lower than the GUB, the GUB is updated in the `MinimaEureka` instance, otherwise the current task is terminated. Similarly, calls to `check` terminate a task if the argument is larger than the currently known GUB in `eu`.

2.1 Constraint Satisfaction Search algorithms

Constraint-satisfaction problems arise frequently in several applications areas including puzzle-solving and engineering design. These problems are computationally intensive and well suited for speedup through parallel processing. This assignment explores parallelization of *constraint-satisfaction search* algorithms that use a game-tree search. In some puzzle-solving algorithms, the puzzle is represented as a tree of game states. Different search strategies are possible when exploring the game-tree search space, for BnB algorithms it is required to explore some monotonic property while exploring game states to ensure optimality in solutions.

An intermediate state of a constraint-satisfaction search is characterized by a *partial Problem State* in which some variables have a single assigned value, and a *Feasible Value Table* (FVT), that provides a set of possible values for the remaining *free* variables. If the set becomes empty for any variable, then it implies that no feasible solution can be derived from the given intermediate state. If an FVT has exactly one value per variable, then it can be combined with the *partial Problem State* to obtain a *complete Problem State*.

Many puzzles can be represented by a set of rules that, applied on the current state of the puzzle, decide what are the possible actions that can be performed, which lead to a new puzzle state (with an assignment of values to a subset of free variables), thereby making them amenable to constraint-satisfaction search. This assignment will focus on the use of constraint-satisfaction search in puzzle-solving, with Sudoku puzzles.

2.2 The Sequential Constraint-Satisfaction Solver

```
public ProblemState search(ProblemState rootState) {

  final Queue<ProblemState> queue = Sorted-Queue();
  queue.push(rootState);

  while (!queue.isEmpty()) {
    final ProblemState loopState = queue.poll();

    if (loopState.isSolution()) {
      return loopState;
    } // if

    final List<ProblemState> neighbors = loopState.neighbors();
    for (final ProblemState neighbor : neighbors) {
      queue.add(neighbor);
    } // for
  } // while
}
```

Listing 4: Game tree search in constraint-satisfaction

We have provided you an implementation of a sequential constraint-satisfaction search algorithm. In the sequential code given to you, you can find the constraint-satisfaction search code in method `computeSudoku()` of `SequentialBenchmark.java`, which is also shown in Listing 4. In this method, parameter `state` (of type `ProblemState`) contains the partial problem statement on entry. The solution maintains a queue of problem states to explore in sorted order to ensure that when a solution state is found, it is the best solution. As each state is explored, its children in the game tree are found and added into the work queue.

Sudoku is a popular puzzle game that requires players to fill in missing numbers from 0 to N-1 on a square N×N board, taking into account the following constraints:

- No square contains more than a number
- Every number appears only once on each column of the board.
- Every number appears only once on each row of the board.
- Every number appears only once in each individual region of the board. Regions are usually rectangular areas of size $\sqrt{N} \times \sqrt{N}$ size.

Although Sudoku games are usually 9×9 with 3×3 regions, as in the 9x9.txt file, there are also variations that take larger board sizes as input, such as 16x16.txt with 4x4 regions. If 9x9 boards use the digits 0..8

to fill the board, larger sizes use 0..9, A, B, C, etc for the same purpose. Furthermore, some variations of Sudoku allow for multiple solutions, and the solver provided can finds any possible solution. Using a comparison function, the solver can find the "cheapest" solution.

2.3 Your Assignment: Parallel Constraint-Satisfaction Search

Your assignment is to design and implement a parallel algorithm for constraint-satisfaction search, using the provided sequential implementation as a starting point. Your homework deliverables are as follows.

1. [**Implementation of utility function in `SudokuUtils` (10 points)**]
   Implement the following functions in `SudokuUtils`: `setPossibleValue`, `countUnsolved`, `isSolution`, `isConsistent`, `newWorkQueue`, and `boardComparator`. Documentation is provided in each of these methods to explain their desired functionality. Correctly implementing these functions correctly will allow the unit tests in `Homework4SudokuUtilsTest` to pass.

2. [**Implementation of parallel search that find a valid solution of the Sudoku board (15 points)**]
   Create a new parallel version of `SequentialBenchmark` that uses `async` and `finish` constructs, and (possibly) `Eureka` or `isolated` constructs, to find a valid solution to the Sudoku problem in the `ParallelSearchBenchmark.java` file. You will be graded on the correctness and speedup of your parallel version, relative to the sequential version. You can focus your attention on parallelizing the `computeSudoku()` method.

3. [**Implementation of parallel search that find a solution with the minimum lexicographic order of the Sudoku board (10 points)**]
   Create a new parallel version of `SequentialBenchmark` that uses `async` and `finish` constructs, and (possibly) `Eureka` or `isolated` constructs, to find a solution with the minimum lexicographic order to the Sudoku problem in the `ParallelLexicalBenchmark.java` file. While one approach is to simply reuse the solution from Part 2 above and return the solution with lowest cost, you can be smarter and reduce the work done by pruning the exploration of partial solutions that are guaranteed to never lead to a solution lower than the current best solution.

4. [**Homework report (15 points)**]
   You should submit a brief report summarizing the design of your parallel algorithms in Parts 2 and 3 above, explaining why you believe that each implementation is correct and data-race-free. Your report should also include the following measurements for both parts 2 and 3:

   (a) Performance of the sequential version with the default inputs.

   (b) Performance of the parallel versions (`ParallelSearchBenchmark` and `ParallelLexicalBenchmark`) with the inputs `25x25-1.txt` and `25x25-2.txt`, executed with the "`-Dhj.numWorkers=1`", "`-Dhj.numWorkers=4`" and "`-Dhj.numWorkers=8`" options on a STIC compute node to run with 1, 4 and 8 workers.

   The timings can be obtained by running the following commands: `mvn clean compile exec:exec -PSudoku1` and `mvn clean compile exec:exec -PSudoku2`. You are also provided a sample `myjob.slurm` to help you run the programs on STIC. Please place the report file(s) in the top-level `hw_4` directory. Please commit all files related to the homework into your turnin svn repository. (You are not allowed to add or modify method definitions in any class that contains the following string in their documentation: "`This class should not be modified`".)