CPSC 5600/4600
Exam #2

Practice Exam

Solution

So that you may receive partial credit, remember to show all your work. **PRINT CLEARLY** – if it is unreadable, it is incorrect.

This test is closed-book and closed-notes except for a 8½ x 11" hand-written one-sided "cheat sheet". You are allowed the full class period to complete the exam, 2 hour and 5 minutes.

DO NOT comment extensively **OR** give elaborate I/O messages due to the time constraint of this test. Exact Java/C++ syntax is preferred but not required.

Question	Points	Score
I. Tally (p. 2)	20	
II. Recursive (p. 4)	20	
III. Non-Recursive (p. 6)	20	
IV. Dependency Analysis (p. 7)	20	
V. Time Analysis (p. 8)	20	
Total	**100**	

I. Tally Class

Write a parallel uniquifying method that takes a sequence of integers and produces a set of the unique integers in the sequence. You must use one of the generalized reducers or scanners that we have discussed (either the recursive or the non-recursive versions), and you can assume that the code for the general algorithm already exists. You need to write the uniquifying method and any specialized tally class (or classes) referenced there. You can use any language that you like and you can use generic libraries from your chosen language (e.g., STL **`std::set`** in C++ or **`HashSet<E>`** in Java).

[As an example, if you chose to use **`GeneralScan3`** in Java you would likely write a class **`Uniquify`** that has a nested class **`Tally`**, a nested class **`Reducer`** that inherits from **`GeneralScan3`**, and a static method **`uniquify`** that takes a **`List<Integer>`** and returns a **`Set<Integer>`**.]./////kl,

```
public class Uniquifier {
   static class Tally {
      public Set<Integer> set;
      public Tally() {
         set = new HashSet<>();
      }
      public void accum(int i) {
         set.add(i);
      }
      public static Tally combine(Tally a, Tally b) {
         Tally combo = new Tally();
         for (int i : a.set)
            combo.set.add(i);
         for (int i : b.set)
            combo.set.add(i);
         return combo;
      }
   }

   static class Reducer extends GeneralScan3<Integer, Tally> {
      public Reducer(List<Integer> data) {
         super(data);
      }
      protected Tally init() {
         return new Tally();
      }
      protected void accum(Tally tally, Integer i) {
         tally.accum(i);
      }
      protected Tally combine(Tally a, Tally b) {
         return Tally.combine(a, b);
      }
   }

   public static Set<Integer> uniquify(List<Integer> data) {
      Reducer reducer = new Reducer(data);
      Tally tally = reducer.getReduction();
      return tally.set;
   }
}
```

II. Recursive Parallelism

Write a recursive parallel algorithm in Java to evaluate a spreadsheet workbook. A workbook is composed of cells each of which represents a formula typed in by the application user. The formulas together with their **`args`** form a DAG. You need to write the **`evaluate`** method for the **`Cell`** class. You may also add additional components to the class. You can assume that all the cells are set up including their references and functors, but some or all of them may be currently invalid. Make sure that you prevent any concurrency conflicts on cells that may be referred to by multiple dependent cells.

```
public class Cell {
    /**
     * Row and column of this cell in spreadsheet.
     */
    private int r, c;

    /**
     * Evaluation result of this cell, or null if currently invalid.
     */
    public Object value;

    /**
     * Evaluation functor. You guarantee to only call this
     * if all the cells referenced by args are already valid.
     * The functor's results must be placed into value upon its
     * return with: this.value = fn.invoke()
     */
    private Callable fn;

    /**
     * Thread pool to use for recursive evaluation.
     */
    private ForkJoinPool pool;

    /**
     * Dependencies of this cell (items that fn needs in order to
     * produce a valid value for this cell).
     */
    private List<Cell> args;

    // FIXME – your code for the evaluate method and anything else you require for it
```

```
// How evaluate is called by SpreadSheet:
public class SpreadSheet {
   private List<Cell> cells;

   public void evaluateSheet() {
      for (Cell cell : cells)
         cell.evaluate();
   }
}
```

```
    /**
     * Evaluate this cell in a thread.
     */
    public void evaluate() {
        pool.invoke(new CellEvaluator(this));
    }

    /**
     * Evaluator to run in a thread.
     */
    class CellEvaluator extends RecursiveAction {
        private Cell cell;

        public CellEvaluator(Cell cell) {
            this.cell = cell;
        }

        /**
         * Thread-safe compute method first evaluates cell's arguments,
         * then invokes this cell's functor.
         */
        public void compute() {
            synchronized (cell) {  // need to use cell as a monitor
                if (value == null) {
                    List<CellEvaluator> dependencies = new ArrayList<>();
                    for (Cell arg : args)
                        dependencies.add(new CellEvaluator(arg));
                    invokeAll(dependencies);
                    value = fn.invoke();
                }
            }
        }
    }
}
```

III. Non-Recursive Parallelism

Write an algorithm to non-recursively evaluate a reduction where the tally's **`combine`** method takes *three* parameters instead of two. For this you should visualize a ternary tree (a tree with three children per interior node). Base your answer on a modified version of the non-recursive version of the general reduction (what we called **`Reduce.java`** in the classroom and with a similar algorithm in the POPP textbook in Chapter 5). Write just the thread's **`Runnable`**. You may assume the number of threads is a power of three.

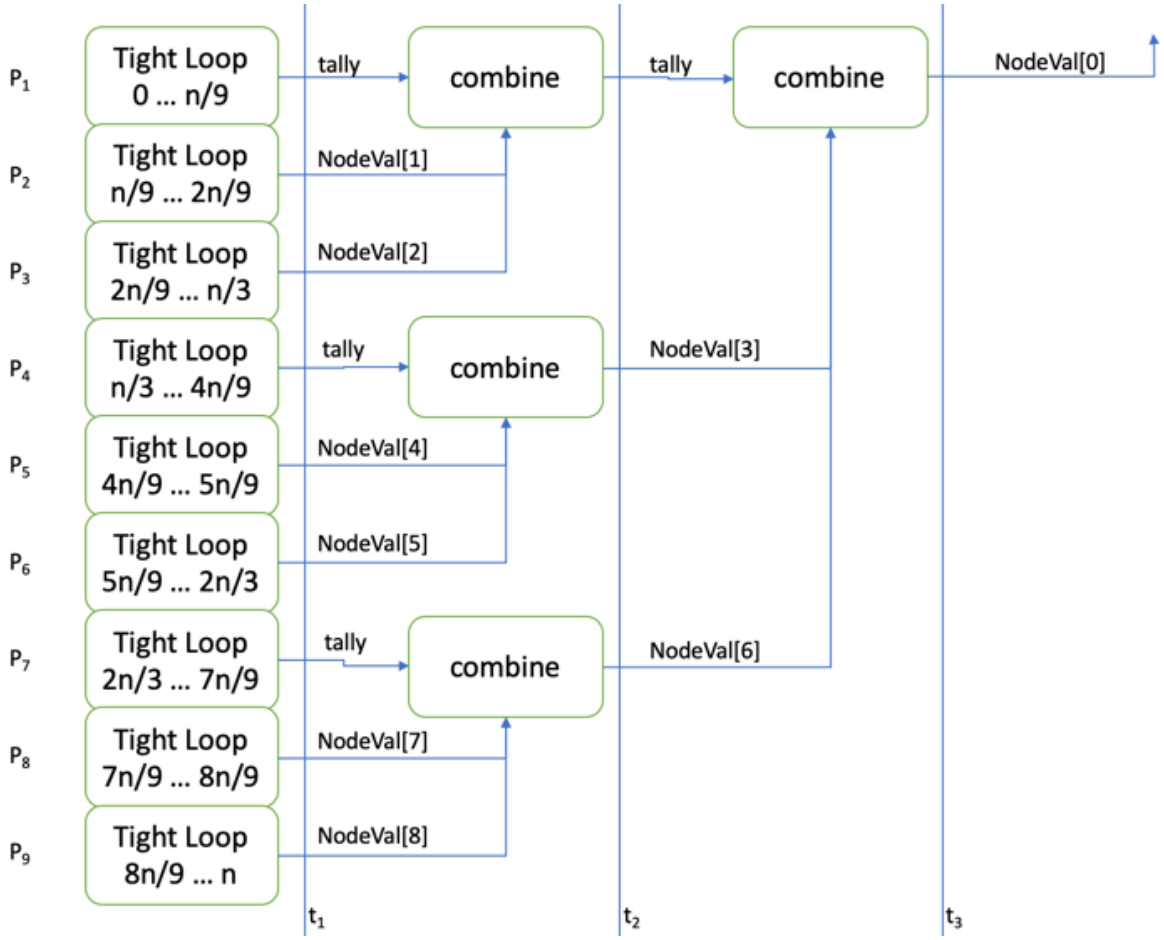
```
class Task implements Runnable {
   public void run() {
      try {
         TallyType tally = newTally();
         for (int i = start; i < end; i++)
            tally.accum(data.get(i));

         for (int stride = 1;
               stride < threadP && index % (3 * stride) == 0;
               stride *= 3)
            tally.combine(getNode(index + stride),
                          getNode(index + stride * 2));
         setNode(index, tally);
         return;

      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
   public Task(int threadi) {
      index = threadi;
      size = n / threadP; // n and threadP are in the enclosing class
      start = size * threadi;
      end = (threadi == threadP - 1 ? n : start + size);
   }
   private int index, size, start, end;
}
```

IV. Dependency Analysis

Show a data flow diagram for your ternary non-recursive algorithm in Part III above where *P = 9* including time steps and labeled inter-thread communication.



V. Time Analysis

1. For Part II above, let *v* be the number of cells, *e* be the number of references (in all the **`Cell.args`** in total), and *d* be the depth of the graph (maximum length of a reference chain), give the time-efficiency class using big-O notation of both the work and latency, in terms of *v*, *e*, and/or *d*. Assume unlimited parallelism.

Work: $\Theta(v + e)$ – assuming all cells are invalid, then we call **`fn.invoke`** exactly once for each cell and **`evaluate`** exactly once for each edge.

Latency: $\Theta(d)$ – each **`fn.invoke`** has to wait for all its dependencies to finish first.

2. For Part III above, what is the time-efficiency of the work and latency, in terms of the number of data elements, *n*. Assume unlimited parallelism.

Work: $\Theta(n)$ – specifically, $n - 1$ **`accum`** calls and $< n/3$ **`combine`** calls.

Latency: $\Theta(\log n)$ – specifically, it's $\log_3 n + 1$ time steps with unlimited parallelism (with $P = n$).

(Without unlimited parallelism, you'd have a specific P, and so latency would be about $\frac{n}{P} + \log_3 P$, assuming the **`accum`** and **`combine`** methods take equivalent time.)