CS2030 Programming Methodology

Semester 1 2019/2020

1 November 2019

Problem Set #9

Fork/Join Framework

1. Run the following program and observe which worker is running which task.

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

class A {
    static class Task extends RecursiveTask<Integer> {
        int count;
        Task(int count) {
            this.count = count;
        }

        public Integer compute() {
            System.out.println(Thread.currentThread().getName()
                    + " " + this.count);
            if (this.count == 4) {
                return this.count;
            }
            Task t = new Task(this.count + 1);
            t.fork();
            return t.join();
        }
    }
```

Task s = new task (this.count ++);
Task t = new Task(this.count ++);
s.fork()
t.fork()
s.join() # this makes the s1 never be executed and main keeps waiting on s1
The fork/join queue can be thought of as a stack
return t.join()

task 0 is processed by main, task 1 is added to deque by main and added to queue, task 2 waits for task 1 to complete,… until task 4, then task 4 is done, -> task 3… task 2, task 1, returns task 0

```
    public static void main(String[] args) {
        ForkJoinPool.commonPool().invoke(new Task(0));
    }
}
```

not parallel!, invoke forks then joins the whole thing immediately

Suppose the program is invoked with a maximum of three additional workers. What can you observe about the behaviour of a worker when the task that it is running blocks at the call to join?

2. Given below is the classic recursive method to obtain the n^{th} term of the Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$ *without memoization*

```
static int fib(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

(a) Parallelize the above implementation by transforming the above to a recursive task and inherit from `java.util.concurrent.RecursiveTask`

(b) Explore different variants and combinations of `fork`, `join` and `compute` invocations.

3. Consider the following `RecursiveTask` called `BinSearch` that finds an item within a sorted array using binary search.

```
class BinSearch extends RecursiveTask<Boolean> {
    int low;
    int high;
    int toFind;
    int[] array;

    BinSearch(int low, int high, int toFind, int[] array) {
        this.low = low;
        this.high = high;
        this.toFind = toFind;
        this.array = array;
    }

    protected Boolean compute() {
        if (high - low <= 1) {
            return array[low] == toFind;
        } else {
            int middle = (low + high)/2;
            if (array[middle] > toFind) {
                BinSearch left = new BinSearch(low, middle, toFind, array);
                left.fork();
                return left.join();
            } else {
                BinSearch right = new BinSearch(middle, high, toFind, array);
                return right.compute();
            }
        }
    }
}
```

As an example,

```
int[] array = {1, 2, 3, 4, 6};
new BinSearch(0, array.length, 3, array).compute(); // return true
new BinSearch(0, array.length, 5, array).compute(); // return false
```

Assuming that we have a large number of parallel processors in the system and we never run into stack overflow, comment on how `BinSearch` behaves.

4. Many ways have been devised to multiply two large integers. One of these ways is attributed to Anatoly Karatsuba in 1960 and is described below using the example $1234 \times 567 = 699678$.

   Step 1. If necessary, pad the smaller number with leading zeros to make two numbers of the same length L, i.e. 1234 and 0567.

   Step 2. Divide the two numbers into equal left and right portions and label them a, b, c, d

$a = 12$	$b = 34$
$c = 05$	$d = 67$

   Step 3. Calculate $ac = 12 \times 5 = 60$

   Step 4. Calculate $bd = 34 \times 67 = 2278$

   Step 5. Calculate $(a + b)(c + d) = 46 \times 72 = 3312$

   Step 6. Calculate the result of step (5) - step (4) - step (3) $= 3312 - 2278 - 60 = 974$

   Step 7. Add the partial results with zero padding

600000	from step (3) by padding L trailing zeroes
2278	from step (4) with no additional padding of trailing zeroes
97400	from step (6) by padding $L/2$ trailing zeroes
699678	

   Notice that multiplying two large numbers require three smaller multiplications which can be done independently in steps (3), (4) and (5).

   Your task is to define a `Task` class that extends `RecursiveTask<BigInteger>` and computes the multiplication in parallel. The following methods from the `BigInteger` class may be useful to you.

   - `public BigInteger(String val)`
     Translates the decimal `String` representation of a `BigInteger` into a `BigInteger`.
   - `public BigInteger add(BigInteger val)`
     Returns a BigInteger whose value is `this + val`.
   - `public BigInteger subtract(BigInteger val)`
     Returns a BigInteger whose value is `this - val`.
   - `public BigInteger multiply(BigInteger val)`
     Returns a BigInteger whose value is `this * val`. Use this when the numbers to be multiplied are of length less than 2.
   - `public BigInteger pow(int exponent)`
     Returns a BigInteger whose value is `this` raised to the power of `exponent`.
   - `public String toString()` Returns the String representation of this `BigInteger`.

   You may also use other methods from the Java API.