# Concurrent Programming - Sheet 1

George Andrews

February 18, 2019

## 1

## 2

For this, to have a $log(n)$ running time, we want a Tree-based protocol. Calling Barrier(p), we make the function use a tree protocol of size p, which will, instead of using a passed value, will use 'me' as a value, since the actual minimum and maximum value don't matter. So each sync calls apply, which waits for the top node to receive the overall minimum and maximum, and then re-send these values to the other nodes - which will have all called sync at this point - so that simply allows them to continue after that. As the tree takes around $log(n)$ rounds, this takes $log(n)$ time.

```
class TreeSync(n:Int) extends MinMax{
    private val up,down=Array.fill(n)(OneOne[IntPair])

    def apply(me:Int,x:Int)={
        val child1=2*me+1;val child2=2*me+2
        var min=x; var max=x
        if(child1 < n){
            val (min1,max1)=up(child1)?()
            if(min1 < min)min=min1; if(max1 > max)max=max1
        }
        if(child2 < n){
            val (min1,max1)=up(child2)?()
            if(min1 < min)min=min1; if(max1 > max)max=max1
        }
        if(me!=0){
            up(me)!(min,max)
            val pair=down(me)?;min=pair._1;max=pair._2
        }
        if(child1 < n)down(child1)!(min,max)
        if(child2 < n)down(child2)!(min,max)
        (min,max)
    }
}

class Barrier(p:Int){
    val t=new Tree(p)

    def sync(id:Int)=proc{
        t(id,id)
    }
}
```

## 3

We first need to $n$ shared variables to represent the threads. These should only ever be accessed by me and me+gap on each iteration. Summer $[n - gap, .., n - 1]$ will always simply await synchronisation with its pair, the rest will set the variable to the value it needs to send, and then sync with me-gap. Summers greater than or equal to gap will then wait for sync and continute as in the original program.

```
import io.threadcso._
import scala.util.Random

object CPSheet3Q3{

    /** Calculate prefix sums of an array a of size n in poly-log n (parallel)
  *  steps.  Based on Andrews Section 3.5.1. */
    class PrefixSums(n: Int, a: Array[Int]){
      require(n == a.size)

      /** Shared array, in which sums are calculated. */
      private val sum = new Array[Int](n)

      /** Barrier synchronisation object. */
      private val barrier = new Barrier(n)
      private val summerBarriers = Array.fill(n)(new Barrier(2))

      /** Channels on which values are sent, indexed by receiver's identity. */
      private val toSummers =
        Array.fill(n)(0)
        //Array.fill(n)(N2NBuf[Int](size = 1, readers = 1, writers = n-1))

      /** An individual thread.  summer(me) sets sum[me] equal to sum(a[0..me]). */
      private def summer(me: Int) = proc("summer"+me){
        // Invariant: gap = 2^r and s = sum a(me-gap .. me]
        // (with fictious values a(i) = 0 for i < 0).  r is the round number.
        var r = 0; var gap = 1; var s = a(me)
        while(gap < n){
          if(me+gap < n){
            toSummers(me+gap)=s; summerBarriers(me+gap).sync()
            // pass my value up the line
          }
          if(gap <= me){                   // receive from me-gap,
            summerBarriers(me).sync()
            val inc = toSummers(me) // inc = sum a(me-2*gap .. me-gap]
            s = s + inc                // s = sum a(me-2*gap .. me]
          }
          r += 1; gap += gap              // s = sum a(me-gap .. me]
          barrier.sync()
        }
        sum(me) = s
      }

      /** Calculate the prefix sums. */
      def apply(): Array[Int] = {
        (|| (for (i <- 0 until n) yield summer(i)))()
        sum
      }
    }
```

```
  val reps = 10000

    /** Do a single test. */
  def doTest = {
      // Pick random n and array
      val n = 1+Random.nextInt(20)
      val a = Array.fill(n)(Random.nextInt(100))
      // Calculate prefix sums sequentially
      val mySum = new Array[Int](n)
      var s = 0
      for(i <- 0 until n){ s += a(i); mySum(i) = s }
      // Calculate them concurrently
      val sum = new PrefixSums(n, a)()
      // Compare
      assert(sum.sameElements(mySum),
             "a = "+a.mkString(", ")+"\nsum = "+sum.mkString(", ")+
               "\nmySum = "+mySum.mkString(", "))
  }

  def main(args : Array[String]){
      for(r <- 0 until reps){ doTest; if(r%100 == 0) print(".") }
      println; exit
  }


}
```

# 4

```
import io.threadcso._
import scala.collection.mutable.Queue

object CPSheet3Q4{
}
```

## 5

For this process we will split the rows between the workers. Each worker will process an entire row, accessing (and only accessing) the original picture, and returning an updated picture. A number of barriers will be used in sequence, equal to the number of workers, which allow them access to the global 'new picture' variable and a 'changes made' variable to update it one at a time. Once all barriers have been synchronised, the controller will send the next set of rows. This repeats until all rows have been completed. At this point the picture is updated, if no changes have been made or the maximum number of rounds has been completedthe system returns the nwe picture and terminates. Otherwise it repeats this.

```
import io.threadcso._
import scala.util.Random

object CPSheet3Q5{
    type Row = Array[Boolean]
    type Image = Array[Row]
    val numWorkers = 10
    val cutoffValue = 1000
    class smoothing{
        private var rounds = 0
        private var baseImage:Image=Array.fill(100,100)(false)
        private var updatedImage:Image=Array.fill(100,100)(false)
        val finalBarrier=new Barrier(2)
        val roundBarrier=new Barrier(numWorkers+1)
        val workerBarriers=Array.fill(numWorkers)(Barrier(1))
        var w=0
        var xsize=0
        var ysize=0
        var changeOccured=false
        for(w<-0 to numWorkers+1) workerBarriers(w)=new Barrier(numWorkers-w)


        def apply(image: Image)={
                baseImage=image
                xsize=baseImage(0).length
                ysize=baseImage.length
                changeOccured=false
                (|| (for (i <- 0 until numWorkers) yield worker(i))
                    || controller)()


            finalBarrier.sync()
            baseImage
        }
        private def worker(me:Int)=proc{
                var rowProgress=0
                while(rounds< cutoffValue){
                        roundBarrier.sync()
                        rowProgress=0
                        while(rowProgress*numWorkers+me < ysize){
                                val rownum=rowProgress*numWorkers+me
                                val currentRow=baseImage(rownum)
                                var newRow=new Row(xsize)
                                var i=0
                                var changeInRow=false
                                for(i <- 0 to xsize-1){
                                        val shouldset = shouldSet(rownum,i)
```

```
                                        changeInRow = changeInRow
                                            || shouldset==baseImage(rownum)(i)
                                        baseImage(rownum)(i)=shouldset
                                }
                                for (i<-0 to me){
                                        workerBarriers(i).sync()
                                }
                                updatedImage(rownum)=newRow
                                changeOccured = changeOccured || changeInRow
                                workerBarriers(me).sync()
                                rowProgress+=1
                        }

                }
        }
        private def shouldSet(r:Int,c:Int):Boolean={
                var x=0
                var tot=1
                if(baseImage(r)(c))x+=1
                if(r!=0){
                        tot+=1
                        if(baseImage(r-1)(c))x+=1
                }
                if(c!=0){
                        tot+=1
                        if(baseImage(r)(c-1))x+=1
                }
                if(r!=0&&c!=0){
                        tot+=1
                        if(baseImage(r-1)(c-1))x+=1
                }
                if(r!=ysize-1&&c!=0){
                        tot+=1
                        if(baseImage(r+1)(c-1))x+=1
                }
                if(r!=0&&c!=xsize-1){
                        tot+=1
                        if(baseImage(r-1)(c+1))x+=1
                }
                if(r!=ysize-1&&c!=xsize-1){
                        tot+=1
                        if(baseImage(r+1)(c+1))x+=1
                }
                if(r!=ysize-1){
                        tot+=1
                        if(baseImage(r+1)(c))x+=1
                }
                if(c!=xsize-1){
                        tot+=1
                        if(baseImage(r)(c+1))x+=1
                }
                (2*x>tot)
        }

        private def controller=proc{
                while(rounds< cutoffValue || !changeOccured){
                        var rowProgress=0
                        while(rowProgress< ysize/numWorkers){
```

```
                        var i=0
                        for (i<-0 to numWorkers){
                            workerBarriers(i).sync()
                        }
                        rowProgress+=1
                    }
                    baseImage=updatedImage
                }
                finalBarrier.sync()
            }
    }

    def test()={
        val s=new smoothing
        val tests =1000
        var i=0
        for(i<-0 to tests){
            val baseImage = Array.fill(100,100)(Random.nextBoolean())
            s(baseImage)
            print(".")
        }
    }

    def main(args : Array[String])={
        test()
    }
}
```

# 6

## 7

```
import io.threadcso._
import scala.collection.mutable.Queue

import io.threadcso._
import scala.util.Random

object CPSheet3Q7{

  class threeDivSync{

    var totalIds=0

    def enter(id: Int)=synchronized{
      while(totalIds % 3 != 0) wait()
      assert(totalIds % 3 == 0)
      totalIds+=id
    }

    def exit(id: Int)=synchronized{
      totalIds-=id
      notifyAll()
    }

  }

  def main(args : Array[String]){
    val tds = new threeDivSync

    def accessor(me: Int) = proc{
      while(true){
        Thread.sleep(scala.util.Random.nextInt(500))
        tds.enter(me)
        println(me+": Accessing")
        Thread.sleep(scala.util.Random.nextInt(500))
        tds.exit(me)
      }
    }

    (|| (for (i <- 0 until 20) yield accessor(i)))()
    println; exit
  }
}
```

The overall process is quite simple, if a thread is between the enter and exit commands for its id then it is accessing, and the assert statement ensures that it cannot access it unless the divisible by 3 condition is met. The testing simply runs a lot of threads repeatedly to show the assert is always true, and that the system does not deadlock.