Concurrent Programming - Sheet 1

George Andrews

January 19, 2019

1

Using concurrency in the implementation of a multi-tab web browser is useful for multiple reasons: Firstly it will reduce latency and increase throughput as a sequential implementation may have a tab not currently in use taking up a large portion of processing time, making the tab in use take much longer. In addition if many tabs are used the longer processes will dominate thread time, leading to an overall reduction in throughput, though this is less prominent than the latency issue. A further benefit is the development will be easier - a single program would have to be made to deal with all the tabs at once, with complicated interaction and processor and resource sharing, whereas with a concurrent program this is much easier as each tab can run on a different thread. This would still work on a Uni-processor computer, as the processor can distribute time accordingly to have the tabs each run on different threads on the same processor (each essentially time-shifted). The problem with this is race conditions leading to unexpected outcomes if shared variables are accessed/changed at the same time by multiple threads. This can be avoided by separating variables, or by explicitly disallowing access to variables when it is already in use - however this then slows the program down further.

2

The number of interleavings is the number of ways m actions in sequence can be interleaved within n actions in sequence. This is equivalent to picking m actions from $n+m$ actions, which is $\binom{n+m}{m}$

3

The possible values for x are 3,4 and 7. Each process only loads and saves the variable once, at the start and end of running, so p increases x by 3 and q by 4.

4

First we must assume that each pass of the while loop checks each variable in turn (specifically not at the same time). Now suppose $x = y - 1$, which is a perfectly reasonable state (and in fact guaranteed to occur at some point, though perhaps not the following situation). From this, the while loop in p accesses x and y to find them not equal. Immediately after this (before incrementing occurs) q does the same. Now both the increment and decrement occur increasing both so that $x = y + 1$. At this point the program can now never terminate (ignoring the integer overflow that will presumably eventually occur). If this same sitatuion occurs at $x = y - 2$ however, the process will terminate. So it can either terminate or not terminate, effectively at random.

5

If doDebit is called twice at the same time, the balance can become negative. If canDebit is run for both values, neither of which individually exceeds balance but their sum does, canDebit will return true to both and since no further checks are made both debits can proceed, reducing the balance to below 0.

There are quite a few solutions to this. The first one to access canDebit can lock the account temporarily, bouncing other attempts until the debit is complete, or simply delaying the doDebit execution until the previous doDebit is fully complete. These, however would block perfectly valid transactions which is bad design and tie up (potentially a lot of) processor time respectively. Alternatively changes can be allowed regardless, but reversed if the balance goes below 0 (this however would generally violate invariants and have the same problems as the locking approach - should be avoided). Any approach should ensure the first process to get a "true" from canDebit should (almost) always be allowed to proceed, the balance should never go below 0, valid transactions shouldn't be blocked, and shouldn't use busy waiting (though the busy waiting could be replaced with an interrupt this method has other issues, as it vastly reduces throughput).

We update the Account class to

```
class Account{
    private var balance = 0

    def credit(value: Int) = balance += value

    def canDebit(value: Int): Boolean = balance >= value

    private def debit(value: Int) = balance -= value

    def doDebit(value: Int) = {
        if(account.canDebit(value)) account.debit(value)
        else println("Debit not allowed!")
    }
}
```

We create the new function doDebit, in which we call canDebit and debit automatically, and also make debit private so that cannot be altered without first checking. So long as (we assume that) doDebit can be made atomic (presumably using the same method as the other functions) we no longer run into the race conditions, as the debit must be complete before another doDebit can be run, and the meaningful check of balance occurs then. Since debit must always be immediately preceded by canDebit, there is no opportunity for the balance to be incorrectly reduced. The only remaining issue is that canDebit can return true, followed by a doDebit printing "Debit not allowed". However, in a situation where race conditions are able to exist, this is unavoidable without removing canDebit as a public function, or using a completely different class.

6

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

object CPSheet1Q6{

        def main(args:Array[String])= {
                Sort(args(0))
        }


        private def Sort(n: Int){
                var a: Array[Int] = Seq.fill(n)(util.Random.nextInt).toArray

                var startTime=System.nanoTime()

                var x=0

                def arrayitems(out: ![Int])=proc{
                        var i=0;
                        while(i < n){
                                out!a(i)
                                i+=1
                        }
                }

                def channels = new Array[Chan[Int]](n)

                var chanIn=OneOne[Int]
                var chanOut=OneOne[Int]

                var system=arrayitems(chanOut)

                chanIn=chanOut
                chanOut=OneOne[Int]
                var j=0
                for (j <- 0 to n){
                        def pr(in: ?[Int], out: ![Int])=proc{
                                var count=0
                                var value =0
                                var cval =0
                                value=in?()
                                while(true){
                                        cval=in?()
                                        if(cval>value){
                                                out!(value)
                                                value=cval
                                        }else{
                                                out!(cval)
                                        }
                                        count+=1
                                        if(count==n-1){
                                                out!(value)
                                        }
                                }
                        }
```

```
                                def fin(in: ?[Int])=proc{
                                        var count=0
                                        while(true){
                                                //println(in?())
                                                count+=1
                                                if (count==n) {
                                                        println((System.nanoTime-startTime)/10
                                                        exit()
                                                }
                                        }
                                }
                                if(j==n){
                                        system = system || fin(chanIn)
                                }else{
                                        system = system || pr(chanIn,chanOut)
                                }

                                chanIn=chanOut
                                chanOut=OneOne[Int]
                        }
                        run(system)
                }
        }
```

We can test it by navigating to the correct directory and using the following commands:
scalac CPSheet1Q6.scala
scala CPSheet1Q6 x
Where x can be replaced with any number to produce a random array of integers to test. The program only works for inputs up to about 2800 due to the load of having a thread per integer, and each of those inputs is too small to obtain a realistic measure of time, given overheads - 10 items takes about 0.13 seconds, while 2750 takes 0.58 seconds, implying a much faster than linear relationship which is simply false. The number of operations taken for sorting n objects is $O(n^2)$ however, since there are n threads taking a similar load, each only has $O(n)$ sequential operations.

7

For this problem, the algorithm can be reduced to multiple uses of a dot product on two n dimensional vectors. By distributing these to separate workers, and collecting the results and placing them in the new array we can split the workload quite well.

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

object CPSheet1Q7{

        var numWorkers=200                  //Value doesn't matter, is overwritten

        def main(args:Array[String])= {
                numWorkers=args(0).toInt
                var x=args(1).toInt
                var y=args(1).toInt                  //These are only for testing
                var z=args(1).toInt

                val random = new java.security.SecureRandom
                var A: Array[Array[Double]] = Array.fill(y, x) { (random.nextDouble()
                var B: Array[Array[Double]] = Array.fill(z, y) { (random.nextDouble()
                var i=0;
                startTime=System.nanoTime()
                Multiply(A,B)
                exit()
        }

        private type Task = (Int,Int)
        private val toWorkers=OneMany[Task]
        private type Result = (Double,Int,Int)
        private val toController=ManyOne[Result]
        private var MatrixA:Array[Array[Double]]=null
        private var MatrixB:Array[Array[Double]]=null
        private var MatrixC:Array[Array[Double]]=null

        private var startTime=0L

        private def Multiply(A: Array[Array[Double]],B: Array[Array[Double]]){
                var l=B.length
                var d=A(0).length
                MatrixC=Array.ofDim[Double](l,d)
                MatrixA=A
                MatrixB=B

                var system=controller || starter

                for(i<-0 until numWorkers){
                        var w = worker
                        system = system || w
                }

                run(system)

                exit()
        }
        private def worker=proc("worker"){
                var n = MatrixA.length
                while(true){
                        val (x,y)=toWorkers?()
                        var total:Double=0
                        for(i <- 0 until n){
                                total=total+MatrixA(i)(y)*MatrixB(x)(i)
                        }
                        toController!(total,x,y)
                }
        }

        private def starter=proc{
                var l=MatrixB.length
                var d=MatrixA(0).length
                for(i<-0 until l*d){
                        toWorkers!(i % l,i/l)
                }
        }

        private def controller=proc{
                val size=MatrixA(0).length*MatrixB.length
                var returned=0

                while(returned< size){
                        val (tot,x,y) = toController?()
                        MatrixC(x)(y)=tot.toInt
                        returned+=1
                }
                println((System.nanoTime()-startTime)/1000000+"ms")
                println(MatrixC.deep.mkString("\n"))
                exit()
        }
}
```

Technically this algorithm will run in $O(n^3)$ time (or $O(xyz)$ time technically), as the controller has a single thread to build the full matrix. However in practice, since the number of threads is bounded above (on this device) by about 2000, and the cost of a simple write is quite low (indeed, removing that line entirely makes effectively no difference to the running time) so this is not a particularly great loss. After 2000 threads has been reached AND the threads are being used fully (not with matrices with $n < 2000$ at least) the complexity should return to $O(n^3)$, being effectively $O(\frac{n^3}{2000})$. However, this appears to occur at quite a substantial size, instead increasing quadratically up to the largest size (time) I can reasonably test. The reason for this is unclear.

```
D:\Computer_Science>scala CPSheet1Q7 2000 800 800 800
18301ms

D:\Computer_Science>scala CPSheet1Q7 1 800 800 800
18255ms

D:\Computer_Science>scala CPSheet1Q7 2000 1600 1600 1600
60555ms

D:\Computer_Science>scala CPSheet1Q7 2000 400 400 400
4675ms

D:\Computer_Science>scala CPSheet1Q7 2000 200 200 200
1469ms
```

```
D:\Computer_Science>scala CPSheet1Q7 2000 100 100 100
707ms

D:\Computer_Science>scala CPSheet1Q7 2000 50 50 50
528ms
```

As can be seen, the program time seems to be about $O(n^2)$ for problem sizes below 2000, due to there being at least 1 thread for each column/row, with the exception of some overhead at smaller sizes. The most interesting part of these tests, however, is that reducing the number of threads to 1 did not increase the time whatsoever. Whether this is due to some caching behavior, or the use of channels and processes being inefficient, or some inherent multi-threading in the operating system/scala itself is unclear.

After having written this, my running of a matrix beyond the size 2000 completed.

```
D:\Computer_Science>scala CPSheet1Q7 2000 3200 3200 3200
337604ms
```

As expected this has given more than a fourfold increase, (about 5.6) and matches up quite closely with $O(n^2)$ behaviour up to 2200-2300 and $O(n^3)$ beyond. Considering overhead and some OS optimization this seems quite reasonable, however still goes no way towards exploring why reducing to 1 thread makes so little difference.

8

We first assume a and b are larger than the number of threads available. If we have more threads than items in a and b there is literally no use for the excess threads, as we can have each thread check a single item. Even this is a little excessive and whether the overhead is worth it is a serious concern.
The basic function is to split a into small parts and send each of them to an available thread to be processed. In this instance processed means iterated through and each checked for containment in b. While a contains would take $O(n)$ time for an arbitrary array, in this case we have an array of integers, and space to abuse, so we can instead use a linear function in the controller to create a hash-table (or, if we are dead-set on making this run as quickly as possible, we can use the threads to divide up the work of populating the hash-table), allowing contains to be completed in $O(1)$ time (note that this would allow the problem to be completed in $O(n)$ time with a sequential algorithm). We then have each thread run on it's segment and keep going until all the segments have been processed. Each thread returns a value and the controller adds them up to get a final result.
The fastest complexity possible for this algorithm is in fact $O(n^{\frac{1}{2}})$ as the controller sends $n^{\frac{1}{2}}$ problems of size $n^{\frac{1}{2}}$. If either is smaller the other must be larger. Faster complexities may be possible using nested processes (Some form of divide-and-conquer) but the overhead makes this unlikely to be worth it, except for with very extreme sizes/number of arrays - however if this is the case there are far more effective approaches to be had anyway.