Concurrent Programming

Bernard Sufrin and Gavin Lowe

Hilary Term 2016-17

8: Patterns of Concurrent Programming

Patterns of concurrent programming

In this chapter we will look at various patterns of concurrent programming.

Process Farm (aka Bag of tasks with replacements)

We have already seen the *bag of tasks* pattern:

* A controller process (the farmer) holds a bag of tasks that need completing;
* Worker processes repeatedly obtain a task from the controller, and complete it.

We saw this in the numerical integration problem, where a task represented an interval whose integral needed estimating; no tasks were ever returned to the bag, so this made the controller very simple.

We will now look at a more interesting example, where sub-tasks are returned by the workers to the farmer for further work to be done on them.

Case-Study: Magic squares

A *magic square* of size n is an n by n square, containing the integers from 1 to n^2, such that the sums of all the rows, columns and diagonals are the same. For example:

16	4	13	1
2	5	12	15
9	14	3	8
7	11	6	10

We will design a concurrent program, using the bag of tasks pattern, to find magic squares.

Similar techniques can be used for many other search problems.

Partial solutions

Each task will correspond to a partial solution of the puzzle, i.e. where some, but not necessarily all, of the locations have been filled in.

A worker process will obtain a partial solution, and, if it is not complete:

* choose an empty square;
* create all the partial solutions obtained by filling that square with a value that *might* lead to a complete solution (there might be none);
* return those new partial solutions to the bag.

Representing partial solutions

We will represent partial solutions using objects of the following class.

```
class PartialSoln(size : Int)
{
  // empty this partial solution then return it
  def empty : PartialSoln = {··· return this}

  // Is the partial solution completed?
  def finished : Boolean = {···}

  // Is it legal to play piece k in position (i,j)
  def isLegal(i: Int, j: Int, k: Int) : Boolean = {···}

  ...
}
```

```
class PartialSoln(size : Int)
{ ...
  // Return new partial solution obtained by placing k at (i,j)
  def doMove(i: Int, j: Int, k: Int) : PartialSoln = {...}

  // Choose a position in which to play
  def choose : (Int,Int) = {...}

  // Print this solution
  def print = {...}
}
```

* The implementation of `PartialSoln` (especially `choose`) makes a big difference to the efficiency of the program.
* But here we will concentrate on organising a concurrent solution.

System Architecture

* The solver is composed of workers and a controller.
* The controller keeps a bag of partial solutions and distributes them to workers by their individual channels.
* A worker sent an incomplete solution will send back its legal developments to the controller, followed by `None`.
* A worker sent a *complete* solution outputs it to `solutions`.

```
def Solver(size: Int, workers: Int, solutions: ![PartialSoln]): PROC =
{ val fromW = N2NBuf[Option[PartialSoln]](BUFSIZE, workers, 1, "fromW")
  val toW   = for (w←0 until workers) yield OneOne[PartialSoln](s"toW($w)")

  val Workers =
      || (for (w ← 0 until workers) yield Worker(w, size, toW(w), fromW, solutions))

  (  Workers || Controller(n, toW, fromW)   )
}
```

A worker

```
  def Worker(me: Int,
             size:        Int,
             tasksIn:     ?[PartialSoln],
             tasksOut:    ![Option[PartialSoln]],
             solutions:   ![PartialSoln]) = proc(s"Worker␣$me")
{ repeat                                            // (until there are no more tasks)
  { val partial = tasksIn?()                        // acquire a task
    if (partial.finished) solutions!partial         // it may be a solution
    else
    // choose an empty square; generate all subtasks that legally fill it
    {  val (i,j) = partial.choose
       for (k ← 1 to size*size)
           if (partial.isLegal(i,j,k)) tasksOut!Some(partial.doMove(i,j,k))
    }
    tasksOut!None                                   // ask controller for more
  }
  solutions.closeOut                               // no more solutions from me
}
```

The Controller

* The controller passes tasks to the workers, and receives new tasks from them.

* It stores tasks that (may) still need work.

* It can terminate when it is holding no partial solutions and none of the workers is busy.

* So it must keep track of the number of busy workers.

* A worker signals that it has finished working on one task (and is ready to work on a new task) by sending back `None` on the `tasksIn` channel.

```
def Controller(  size:          Int,
              ,  tasksOut:       Seq[![PartialSoln]],
              ,  tasksIn:        ?[Option[PartialSoln]],
              ) =
   proc("Controller")
   {
     var busyWorkers = 0
     val tasks = new scala.collection.mutable.Stack[PartialSoln];
     // The initial task is an empty board
     val init = new PartialSoln(size).empty
     tasks.push(init)



     ... Main serve loop
   }
```

```
def Controller(···) = proc
{
 ···
 // Main serve loop
 serve ( // send a task to a listening worker
       | (for (out←tasksOut) yield
               (!tasks·isEmpty && out) =!⇒
                     {  busyWorkers += 1
                        counter += 1
                        tasks·pop
                     })
        // receive a response from a busy worker
       | (busyWorkers>0 && tasksIn)    =?⇒
            { case Some(partial) ⇒ tasks·push(partial)  // another sub-task
              case None          ⇒ busyWorkers -= 1     // a task was completed
            }
       )
 // serve loop terminates when tasks.isEmpty and busyWorkers==0
 for (out←tasksOut) out·closeOut
 tasksIn·closeIn
}
```

Question: can the `tasksOut` channels be buffers? What happens?

* Performance: Puzzle size 3: different numbers of workers; time to find all (8) solutions

```
Size #W   ms          Size #W   ms
   3 001 148             3 005  10
   3 001  45             3 005   8
   3 001  43             3 005  12
   3 001  46             3 005   9
   3 001  48             3 005  10
   3 002  40             3 006   9
   3 002  30             3 006   8
   3 002  28             3 006   8
   3 002  25             3 006   8
   3 002  22             3 006  10
   3 003  18             3 007   8
   3 003  18             3 007  10
   3 003  21             3 007   9
   3 003  29             3 007   9
   3 003  18             3 007   7
   3 004  11             3 008   8
   3 004  16             3 008   7
   3 004  11             3 008   7
   3 004  11             3 008   8
   3 004   8             3 008   8
```

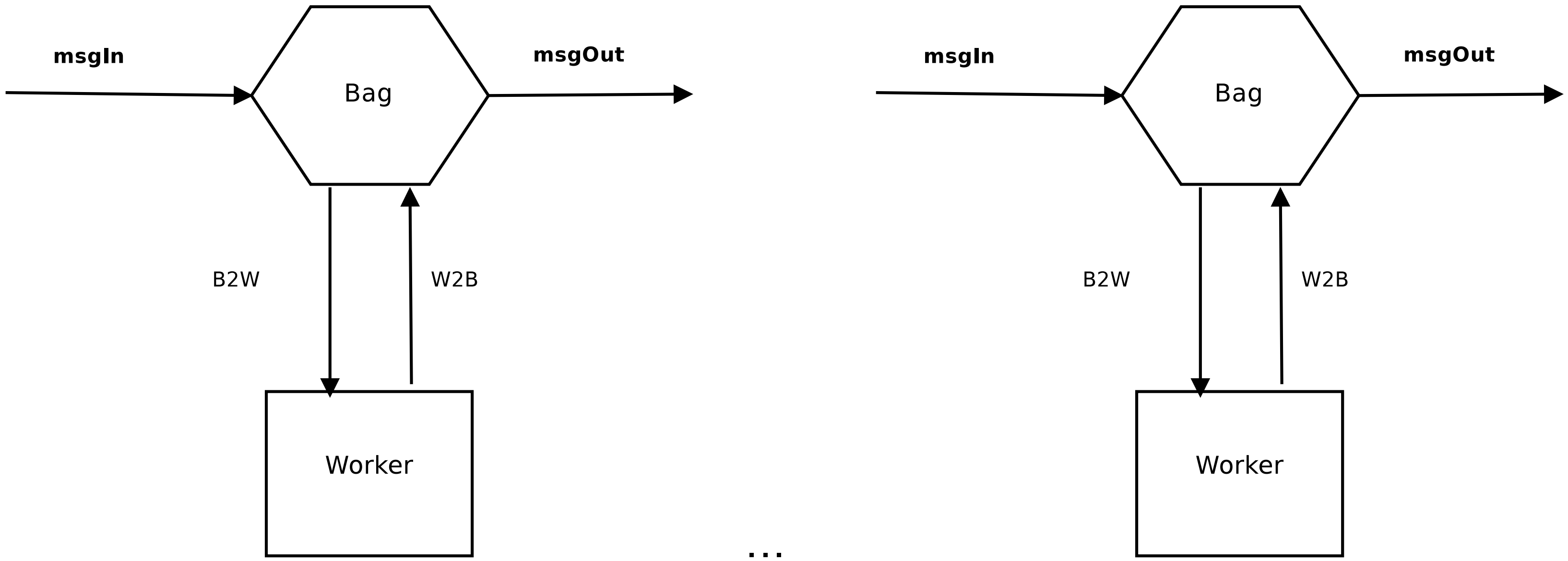
* Note the effect of JIT compilation after the first trial

Removing the bottleneck from Bag of Tasks

* Bag of tasks usually held by a *single, centralized* farmer process.
* The farmer process can become a bottleneck.
* One alternative solution distributes the bag of tasks among a collection of working nodes.
  ◦ Each working node holds its own part of the bag.
  ◦ Working nodes pass tasks among themselves as their load varies.

Working Node = Bag ∥ Worker

* Nodes will form a (logical) ring
* When busy, a node can delegate a task to its clockwise neighbour.
* When not busy, a node can ask its anticlockwise neighbour for a task.



* The Worker does the computation.
* The Bag process deals with work-sharing (and termination of the ring).

* A task may or not be finished. If it isn't finished then it may need subtasks to be performed.

```
trait Task[T]
{ def finished: Boolean
  def subtasks: Iterable[T]
}
```

* A Worker with an unfinished task sends its subtasks to its Bag as a stream of `Some[Task]`.

* Sending `None` to its Bag indicates that the worker is ready for another task.

```
val Worker: PROC = proc(s"Worker$me")
{ repeat
  { val task = B2W?()
    if (task·finished)
       out!task                             // export finished task
    else
       for (st←task·subtasks)
           W2B!Some(st)                     // send subtasks to my bag
    W2B!None                                // signal readiness for more
  }
}
```

* Nodes circulate Messages
  - A Message may contain work in the form of a Task

```
abstract class Message[T]
case class Work[T](task: T) extends Message[T]
```

  - Or it may be a message (definite or tentative) about terminating the whole ring.

```
case class Terminating[T](definite: Boolean) extends Message[T]
```

```
def Node[T <: Task[T]](me: Int, msgIn:  ?[Message[T]],
                                msgOut: ![Message[T]],
                                out:    ![T],
                                first: T = null): PROC =
{  val B2W = OneOne[T](s"B$me␣to␣W$me")
   val W2B = OneOneBuf[Option[T]](0, s"W$me␣to␣B$me") // unbounded
   val Bag: PROC =
       ‘‘Bag definition’’

   val Worker: PROC =

       ‘‘Worker definition’’
   (Bag || Worker)
}
```

```
val Bag: PROC =

   proc(s"Bag$me")
   { object TerminationState extends Enumeration
     {  // 3 possible states of master node
        // RECOVERING means recovering from an aborted termination
        val RUNNING, TERMINATING, RECOVERING, NONMASTER = Value
     }
     import TerminationState._
     var state = if (me==0) RUNNING else NONMASTER

     // Local collection of tasks to be done
     val work = new Stack[T]

     // Start with the first task, if given
     if (first!=null) work.push(first)

     // The bag records whether its worker is working
     var workerBusy = false

     ‘‘The Bag Serve-Loop’’
   }
```

The Bag Serve-Loop

```
priserve
{(  // busy worker, work to be done, successor solicits a task
    (workerBusy && !work.isEmpty && msgOut) =!=> { Work(work.pop) }

    // busy worker finishes a task or generates a new subtask
 |  (workerBusy && W2B) =?=>
               { case None           => workerBusy = false
                 case Some(subtask)  => work.push(subtask)
               }

    // work to be done, (idle) worker solicits a task
|   (!workerBusy && !work.isEmpty && B2W) =!=> { workerBusy=true; work.pop }

    // idle worker, work to be done, successor solicits work
|   (!workerBusy && !work.isEmpty && msgOut) =!=> { Work(work.pop) }
|   orelse ==>
          ''handle the soliciting of tasks and the termination protocol''
)} // priserve
B2W.closeOut
```

Soliciting Tasks and the Termination Protocol

* The master node is in a $RUNNING$, a $RECOVERING$, or a $TERMINATING$ state.

  This state is maintained by its `Bag`, and reflects the status of the ring as a whole.

  (a) A $RUNNING$ master node starts a tentative (non-definite) `Terminating` message circulating when it has nothing else to do; and becomes $TERMINATING$.

  (b) A $TERMINATING$ master node converts a tentative `Terminating` message from its predecessor into a definite one if it has received no work from its predecessor since (a).

  (c) A $TERMINATING$ master node that receives work from its predecessor becomes $RECOVERING$ (its tentative termination attempt has failed).

  (d) A $RECOVERING$ master node suppresses all `Terminating` messages.

  (e) All nodes terminate in response to a definite `Terminating` message.

```
| orelse ⟹
        ‘‘handle the soliciting of tasks and the termination protocol’’

  { if (!workerBusy && work·isEmpty)  // Node is idling
    {  if (state==RUNNING)            // Master node ...
       { state=TERMINATING            // ...  starts termination
         msgOut!Terminating(false)    // ...  tentatively
       }

       // Await a message from predecessor (there will certainly be one)
       val message = msgIn?()
       message match
       { case Work(task) ⇒
              workerBusy=true; B2W!task
              // abandon termination:  jettison future Terminating messages
              if (state==TERMINATING) state=RECOVERING

         case Terminating(definite) ⇒
              ‘‘process a termination message’’
       }
    }
  }
```

```
case Terminating(definite) ⇒ ‘‘process a termination message’’

     state match
     { case TERMINATING  ⇒   // convert tentative to definite
             msgOut!Terminating(true)
             msgIn?()         // await rearrival of termination
             out·closeOut     // close the output port
             stop             // terminate the serve loop
       case RECOVERING   ⇒
             state=RUNNING    // recovery is complete
       case NONMASTER ⇒
             msgOut!message         // pass on the termination message
             if (definite) stop     // terminate serve loop if definite
     }
```

Putting it all together: Magic Squares Revisited

* We will wrap up (Gavin's) original magic square logic of partial solutions

... not necessary had partial solutions been formulated as Tasks in the first place

... but that's the reality of programming with other folks' code!

```
class MagicSquare(size: Int, wrap: PartialSoln = null)
extends DBOT.Task[MagicSquare]
{  val soln = if (wrap==null) new PartialSoln(size).empty else wrap
   def finished: Boolean = soln.finished
   def subtasks  =
   { val (i,j) = soln.choose
     for (k ← 1 to size*size if (soln.isLegal(i,j,k))) yield
         new MagicSquare(size, soln.doMove(i,j,k))
   }
   override def toString = soln.toString
   def print = soln.printSolution
}
```

```
object DistBagSquares
{ import DBOT._; import io.threadcso._
  def main(args: Array[String])
  { var N = 4
    val solns = N2N[MagicSquare](1, 1, "Solutions")

    val link = // links between nodes (indexed by destination)
        for (i←0 until N) yield OneOne[Message[MagicSquare]](s"${(N+i-1)%N}->$i")

    val ring = // The ring of nodes
        (  Node(0, link(0), link(1), solns, new MagicSquare(4))
        || ||(for (i ← 1 until N) yield Node(i, link(i), link((i+1)%N), solns)))

    val report = proc ("report") // print the finished tasks -- the solutions
    { repeat { val b = solns?(); b.print } }

    println(debugger)
    (ring || report)()
    exit
  }
}
```

Recursive parallelism

Many sequential programs use recursive procedures.

The idea of recursive parallelism is that (some) recursive calls are replaced by spawning parallel processes that have the same effect. This is particularly useful where a procedure would make two or more recursive calls to itself that are independent (operate on disjoint data): the corresponding recursive parallel processes can then run concurrently.

A typical pattern for recursive parallelism is:

* In some base case(s), calculate the result directly;
* In other cases, spawn a parallel process corresponding to each recursive call, together with a controller (if necessary) to coordinate distribution of arguments and collection of results.

Case Study: Recursively-Parallel in-place Quicksort

Sequential algorithm to recursively sort a segment of an array

* If the segment is large enough
  - Partition it about a value it contains
  - Recursively sort its smaller and larger segments
* Otherwise use a simpler (iterative) method

```
def seqSort(a: Array[Int], l: Int, r: Int)
{ if (r-l>K)
  {  val (p, q) = partition(a, l, r)
     seqSort(a, l, p); seqSort(a, q, r)
  }
  else
  { ··· sort iteratively ··· }
}
```

* The smaller and larger subsegments `a[l:p)` and `a[q:r)` are independent
* The obvious parallel decomposition is

```
def parSort(a: Array[Int], l: Int, r: Int)
{ if (r-l>K)
  {  val (p, q) = partition(a, l, r)
     (parSort(a, l, p) || parSort(a, q, r)) ()
  }
  else
  { ··· sort iteratively ··· }
}
```

```
      +--------------------------------------------+
   a: | ... | <v .... | =v .... | >v .... | .... |
      +--------------------------------------------+
            l         p         q          r
```

Limits of recursive parallelism

Unbounded recursive parallelism can lead to many more processes than there are processors. The overheads involved in spawning new processes, and context-switching between processes mean that this is very inefficient.

A more efficient way is to limit the number of processes. There are a number of ways to do this. The simplest is for each recursive call to be passed a parameter `maxleaves` that says how many leaf (i.e. base case) processes it is allowed to run. If the value of `maxleaves` is 1 then the process switches to the sequential algorithm.

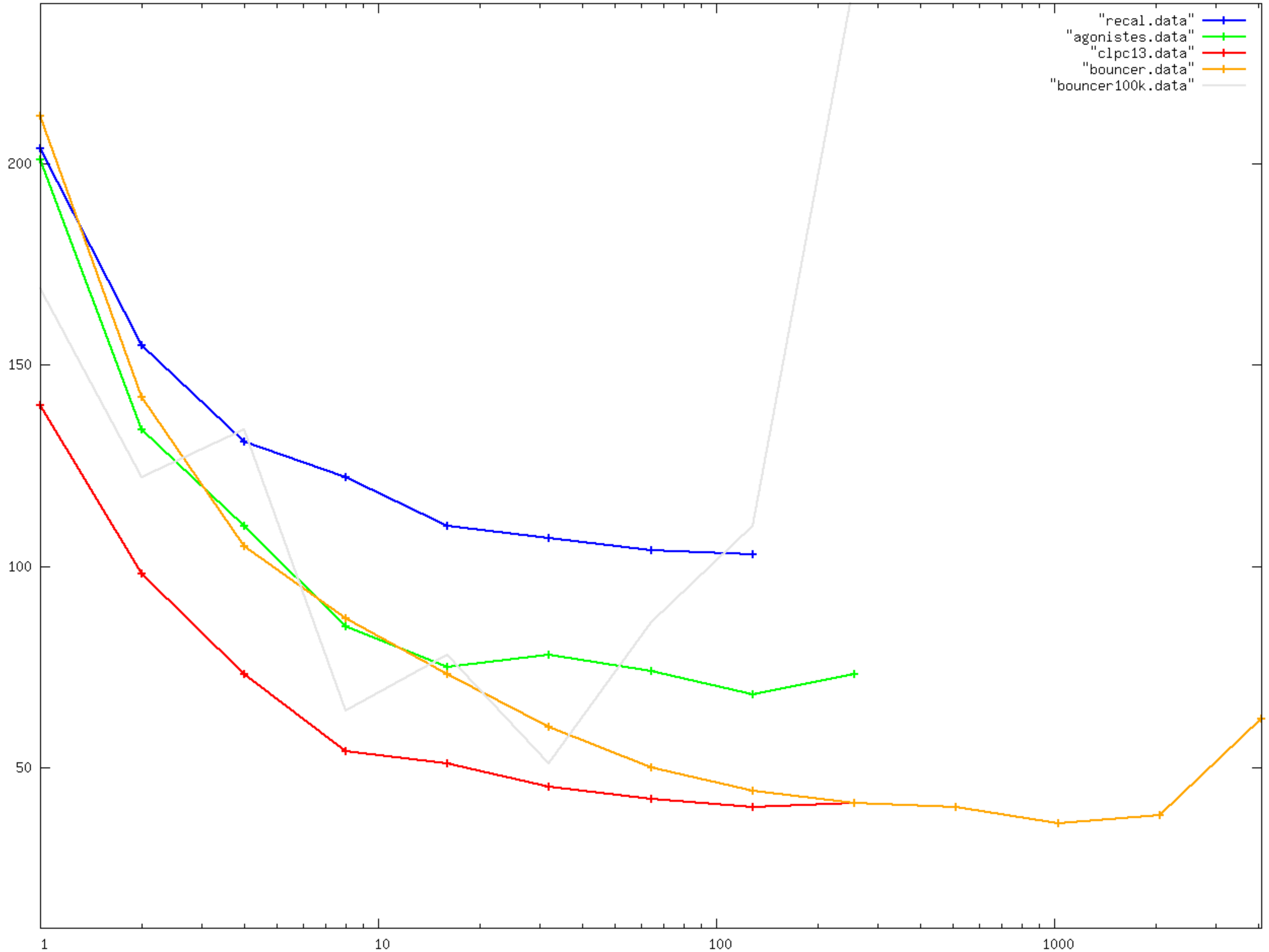
Limiting the number of processes

* Once the number of permitted leaves is 1 switch to a sequential sort
* Otherwise allocate (about) half the number of leaves to each concurrent sub-sort

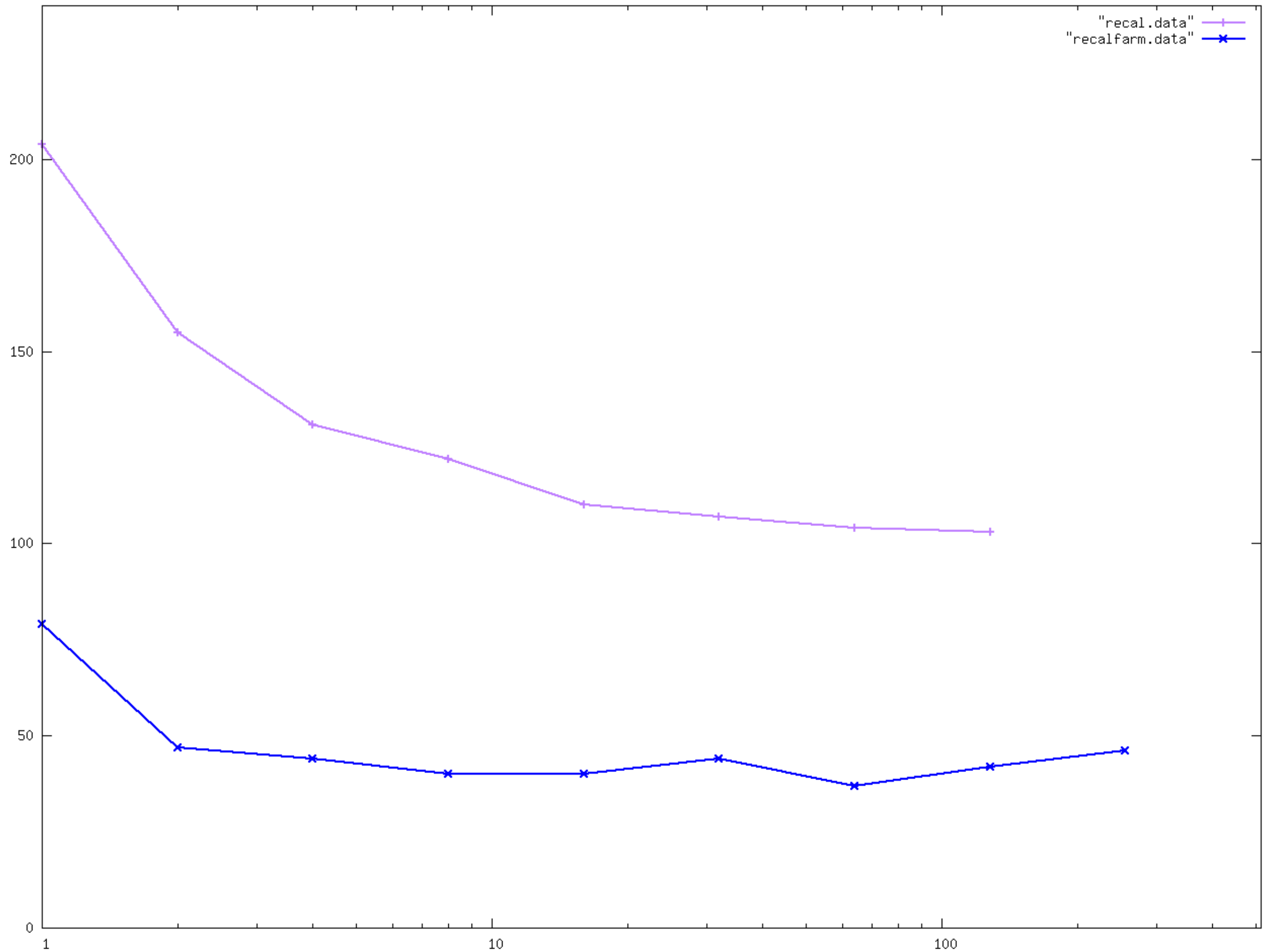
```
def quickSort(a: Array[Int], l: Int, r: Int, maxleaves: Int)
{ if (l>=r) return
  if (maxleaves<=1)
     seqSort(a, l, r)
  else
  { val (p,q) = partition(a, l, r)
    val ml = maxleaves/2
    (quickSort(a, l, p, ml) || quickSort(a, q, r, maxleaves-ml))()
  }
}
```

* Implicit assumption here is that the sizes of the two segments are similar
* Exercise: what should be done when they are not?

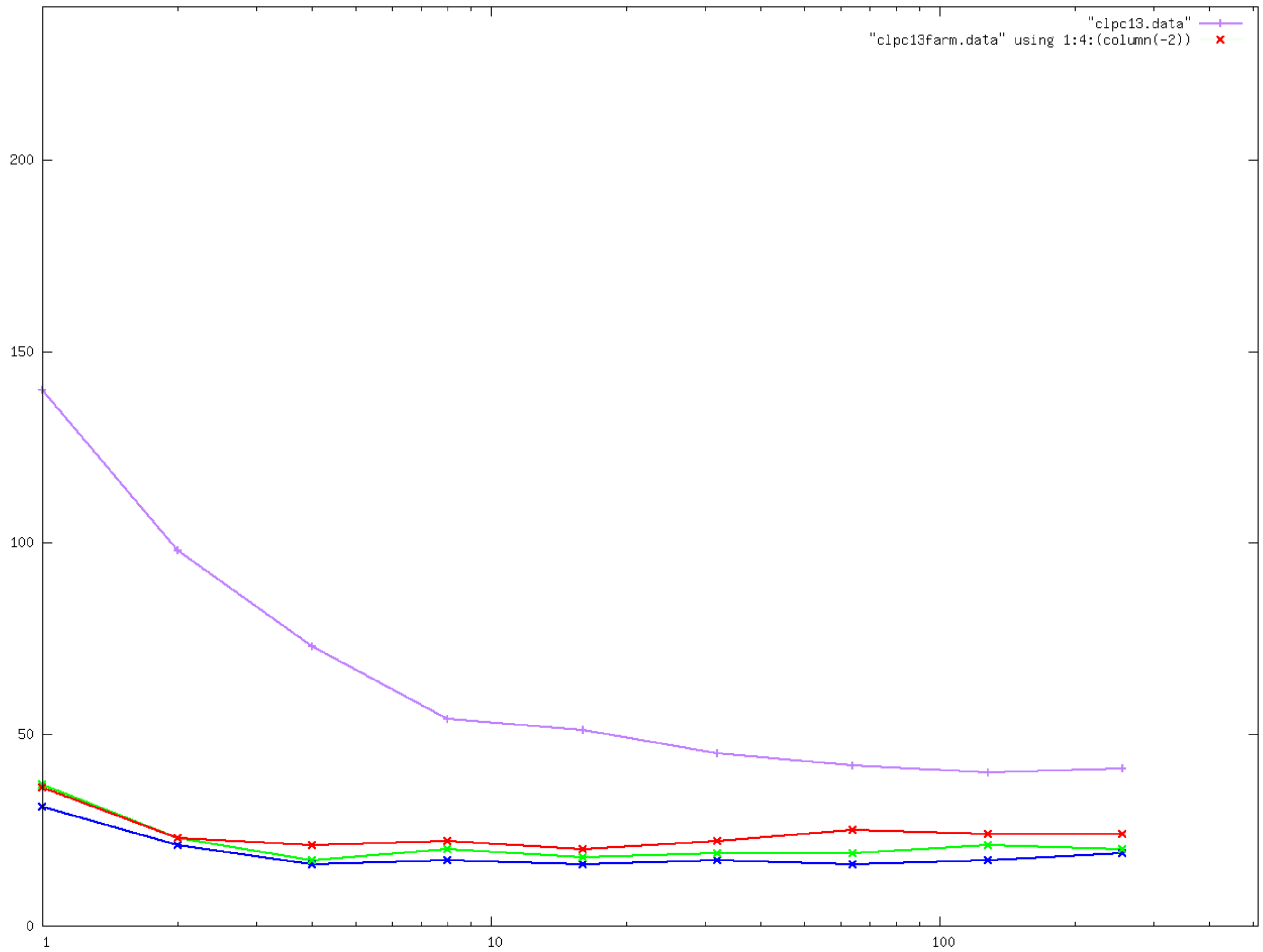
Performance: $ns/number$ vs. $log\ maxleaves$ array size 10^7, $K = 6$



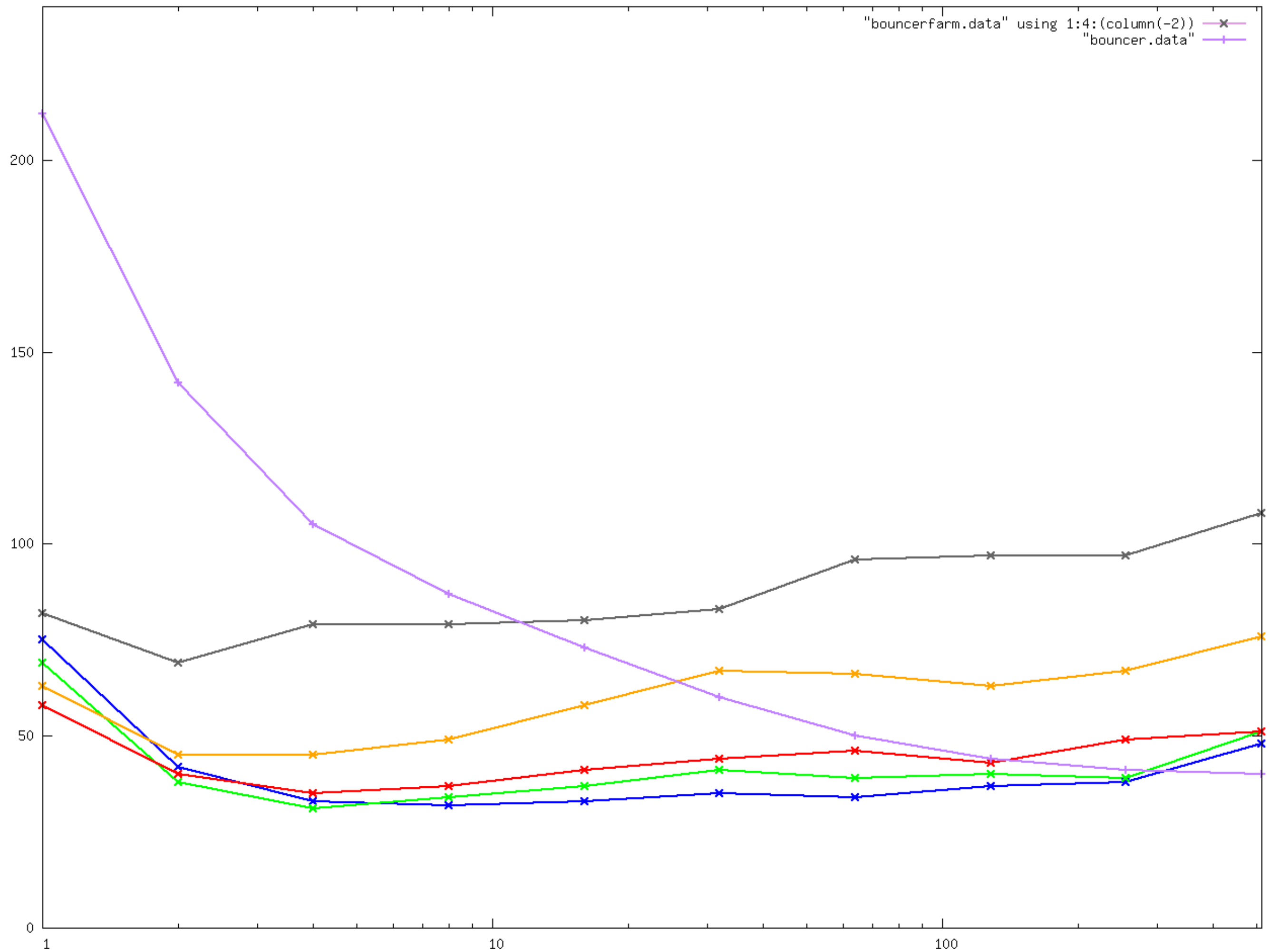
Quicksort: process-farming appears to beat recursive parallelism



Recalcitrant: NS/Number *vs* log workers for arrays of size 10^7. Sequential threshold = 1024



Clpc13: NS/Number *vs* log workers for arrays of size 10^7
Sequential threshold: 4096, 1024, 512



Bouncer: NS/Number *vs* log workers for arrays of size 10^7
Sequential threshold: 8192, 4096, 2048, 1024, 512

Tentative conclusion

* When using the JVM, recursive parallelism seems not to be the best way of reliably benefiting from concurrency on multiple CPUs to improve the performance of *a compute-bound* task.

* Process-farming seems better (a tutorial question asks you to implement a quicksort farm)

* Recursive parallelism is reputed to yield more reliable speed-ups when applied to *input/output-bound* tasks.

* **But** we have not yet explored the performance of similar programs in `Go`.

Competition parallel

The idea of competition parallel is to use two (or more) different algorithms for a problem, run them independently in parallel, and take the answer of the one that finishes first.

Among the other things for which it is useful, this approach works well on the boolean satisfiability problem.

The problem is NP-complete. However, there are a number of SAT-solving algorithms that work well in many cases. But all known algorithms have cases where they perform badly, and different algorithms perform badly on different cases.

So if the computational resources are available, running two or more algorithms in competition with one another can give better average results than parallelising a single algorithm.

* An *overly* simplistic implementation of competition parallel is

```
def compete[T](alg₁: => T, alg₂: => T) : T =
{ val r₁, r₂ = OneOne[T]
  var r = null
  proc { r₁!alg₁ } . fork
  proc { r₂!alg₂ } . fork
  alt ( r₁ =?=> { res => r = res} | r₂ =?=> { res => r = res} )
  return r
}
```

* The problem is that the algorithm that loses the race continues to use up computational resources that could be put to better use.

* As the basis for a better implementation we use two facts
  - A running CSO process that is performing input-output or waiting for channel or semaphore activity will receive a `java.lang.InterruptedException` exception when its handle's `interrupt` method is called.
  - The method `java.lang.Thread.interrupted` returns `true` iff the currently-running process was interrupted since the previous call of the same method in the same running process.

* A less simplistic implementation of competition parallel has the winner interrupt the loser (using its thread handle)

```
def compete[T](alg₁: => T, alg₂: => T) : T =
{ val r₁, r₂ = OneOne[T]
  var r = null
  val h₁ = proc { r₁!alg₁ } . fork
  val h₂ = proc { r₂!alg₂ } . fork
  alt ( r₁ =?=> { res => r = res ; h₂ . interrupt }
      | r₂ =?=> { res => r = res ; h₁ . interrupt }
      )
  return r
}
```

* This implementation imposes on the competing algorithms the requirements
  - that they periodically check to see if they have been interrupted, and
  - that they handle `InterruptedException`s appropriately.

Task parallel programming

Most of the concurrent programs we have seen so far have been *data parallel*: the data has been split up between different processes, each of which have performed the same task on its data.

An alternative is *task parallel programming*[1], where different processes have performed different operations on the same data, typically in some kind of pipeline.

Examples:

* Compilers typically operate in a number of stages, e.g., lexical analysis, syntactical analysis, semantic analysis, type checking, code generation, optimisation. Each stage can be implemented by a separate process, passing its output to the next process.
* Unix pipes, e.g. `ls -R | grep elephant | more`.

[1] a different meaning of the word “task” than in the bag of tasks pattern

* Example: parallelizing relational (natural join) queries

```
Table   t₁: (forename, surname, address)
Table   t₂: (forename, surname, employer)
Query:  from t₁ ⊗ t₂ select (employer, address) where surname="sufrin"
```

Direct translation

```
     tableReader(t₁, chan₁)
  || tableReader(t₂, chan₂)
  || natJoin(chan₁, chan₂, chan₃)
  || filter({(_, surname, _, _) ⇒ surname=="sufrin"})(chan₃,chan₄)
  || select({(_, _, address, employer)⇒(employer, address)})(chan₄,out)
```

A better translation can drastically reduce the load on natural join

```
     tableReader(t₁, chan₁)
  || filter({_, surname, _)⇒ surname=="sufrin"})(chan₁, chan₁s)
  || tablereader(t₂, chan₂)
  || filter({_, surname, _)⇒ surname=="sufrin"})(chan₂, chan₂s)
  || natJoin(chan₁s, chan₂s, chan₃)
  || select({(_, _, address, employer)⇒(employer, address)})(chan₃, out)
```

Futures

A *future* (or *promise*) is a value that is computed in parallel with the main computation, to be used at some time in the future. For example:

```
val x = scala.concurrent.ops.future( <some lengthy computation> )
··· do something else ···
val y = f(x());
```

The expression `x()` returns the result of the computation, waiting until it is complete if necessary. In CSO we can simulate this as follows:

```
def phuture[T](computation: ⇒ T) : () ⇒ T
{ val result  = OneOne[T]
  val compute = proc { result!computation }
  compute.fork
  return { () ⇒ result? }
}

val x = phuture( <some lengthy computation> )
··· do something else ···
f(x())
```

Contents

Note 1: 4 ☞

For example, given the partial solution

16	4	11	3
2	12		

The next position has to be at least 5 to make it possible for that row to add up to 34; so could be 5, 6, 7, 8, 9, 10, 13, 14, 15.

If 5 is selected, the next value has to be 15.

Alternatively, if 8 were selected, no value is posible.

Note 2: Why an individual channel from the controller to each worker? 7 ☞

Early on in the development of this program we made the mistake of having a single shared channel from controller to workers. Although this looks very elegant, it can (and nearly always does) lead to the worker-end of the channel being used in more than a single alternation simultaneously: something that was forbidden in order to simplify the implementation of alternations.

Note 3: Putting it all together: Eight Queens Solved by a Distributed Bag of Tasks 21 ☞

```
object EightQueens
{ import DBOT._
  import io.threadcso._

  class Partial(N: Int) extends Task[Partial] {
    // Represent a partial solution (a task) by a list of Ints whose
    // i'th entry represents the row number of the queen in column i
    private var board: List[Int] = Nil
    private var len = 0

    def finished: Boolean = (len==N)

    private def isLegal(j: Int) = {
      // is piece (i1,j1) on different diagonal from (len,j)?
```

```
    def otherDiag(p:(Int, Int)): Boolean= {
      val (i1, j1) = p
      i1-len!=j1-j && i1-len!=j-j1
    }

    (board forall ((j1: Int) ⇒ j1 != j)) &&       // row j not already used
    (List.range(0,len) zip board forall otherDiag) // diagonals not used
  }

  // New partial solution resulting from playing in row j
  private def doMove(j: Int): Partial = {
    val newPartial = new Partial(N)
    newPartial.board = this.board ::: (j :: Nil)
    newPartial.len = this.len+1
    newPartial
  }



  // Every subtask is an legal extension of this partial solution
  def subtasks =
  {   for (j ← 0 until N if (isLegal(j))) yield doMove(j)
  }

  override def toString: String = {
    var st = "/";
    for (i ← 0 until len) st = st + (i, board(i))+"/";
    return st
```

```
      }
    }

    def main(args: Array[String])
    { var N = 3
      val solns = N2N[Partial](1, 1, "Solutions")

      // Channels that form the ring
      val link =
          for (i←0 until N) yield
              OneOne[Message[Partial]](s"${(N+i-1)%N}->$i") // indexed by recipient's id

      // The ring of Nodes
      val ring =
          (  Node(0, link(0), link(1), solns, new Partial(8)) ||
          || ( for (i ← 1 until N) yield
                    Node(i, link(i), link((i+1)%N), solns) ))

      println(debugger)
      (ring || component·console(solns))()
      exit
    }
  }
```

Note 4: 23 ☞

You are invited to instrument this code's performance, for comparison with that of the centralized bag of tasks.

Note 5: Partition 25 ☞

```
       +------------------------------------------+
    a: | ... | ......................... | .... |
       +------------------------------------------+
              l                              r
                       PARTITION
       +------------------------------------------+
    a: | ... | <v .... | =v .... | >v .... | .... |
       +------------------------------------------+
              l         p         q          r
```

```
def partition(a: Array[Int], l: Int, r: Int) : (Int, Int) =
{ var pr = r-1
  val m  = (l+r)/2
  // reorder a(l), a(m), a(pr) so that a(m) is the median
  if (a(pr)<a(l)) swap(a, l, pr)
  if (a(m)<a(l))  swap(a, m, l)
  if (a(pr)<a(m)) swap(a, m, pr)
  // Make a(l) the pivot
  swap(a, l, m)

  val v  = a(l)       // pivot value
  var pl = l
  var pe = pl+1
  // [  <v |   =v  |   ...   |   >v  ]
  //  l     pl      pe     pr          r
  //  l <=  pl  <   pe <=     pr+1<=  r
  while(pe<=pr)
  { val ve=a(pe)
    if (ve<v)  { a(pl) = ve; a(pe) = v; pl=pl+1; pe=pe+1 }
    else
    if (ve>v)  { a(pe) = a(pr); a(pr) = ve; pr=pr-1 }
    else { pe=pe+1 }
  }
  return (pl, pe)
}
```

Note 6: Quicksort performance graph interpreted 29 ☞

Notice that none of the machines have a speedup larger than about 4. The very best performance for arrays of size 10^7 comes from Bouncer (with $maxLeaves = 1000$), but remember that it has 80 CPUs (40 cores each with 2 hardware threads) and a cache capable of holding the whole 10^7 words of the array. Clpc13 does nearly as well on 8 CPUs (4 cores each with 2 hardware threads) with $maxLeaves = 128$. The very worst, and least consistent, performance also comes from Bouncer – for arrays of size 10^5. It starts to deteriorate exponentially at $maxLeaves = 32$.

Note 7: Process farmed quicksort 30 ☞

The sequential $Threshold$ parameter in a process-farmed quicksort is the size of the segments below which a worker process will switch to sequential quicksort without returning a new job.

Note 8: 34 ☞

Boolean satisfiability problem: given a boolean formula such as:

$$(b_1 \lor b_2 \lor \neg\, b_3) \land (\neg\, b_1 \lor b_3) \land (\neg\, b_1 \lor b_2 \lor b_3)$$

is there a way of choosing values for the boolean variables to make the formula true?

Lots of problems can be mapped onto SAT-solving, e.g. constraint satisfaction, model checking.

Note 9: 38 ☞

Here we use a very informal notation for relational database tables and assume all fields in the rows of a table are represented as strings.

The natural join operator makes rows from the fields of all combinations of the rows of its operands whose correspondingly-named fields are identical. Thus in the example $\mathtt{t}_1 \otimes \mathtt{t}_2$ means something like

```
[(forename, surname, address, employer) |
                 (forename, surname, address) <- t₁,
                 (forename₂, surname₂, employer)   <- t₂,
                 forename=forename₂, surname=surname₂ ]
```

Exercise 1: Magic Squares (Slide 12) (Ans 1 ☞)

Show how to replace the single `Option[PartialSoln]` channel – that communicates *both* new tasks and task completions from workers back to the controller – with a `PartialSoln` channel that communicates new tasks and a `Unit` channel that communicates task completions. The complete Scala code for the former can be found on the course website; and only a small number of alterations will be needed.

Answer 1: Magic Squares

Ex 1 ☞