Introductory concurrency

Mark Armstrong

December 7, 2020

Contents

1 Introduction

Here we briefly explore (some of) the techniques to take advantage of concurrency and parallelism in the languages we have used throughout the course.

Our context for this discussion will be the problem of computing them maximum element of an (unsorted) tree of integers. This naturally leads

to a divide and conquer approach, which can easily take advantage of concurrency, while also avoiding the issues that can arise from concurrent execution. We do give a brief discussion of these issues before starting our exploration.

We justify the use of this problem to frame our exploration with a discussion at the end of the notes.

2 Some brief background

The following should be review from other courses, most notably from "Concurrent Systems". If you are not familiar with these topics, this will be a sufficient introduction for our purposes here.

2.1 Concurrency vs. parallelism

To define concurrency and parallelism, we must refer to "tasks being executed".

Two tasks `A` and `B` are said to be *concurrently executed* if the execution may be carried out in any of these orders:

- Part of `A` is executed, then the remainder of `A` and `B` are executed concurrently.
- Part of `B` is executed, then the remainder of `B` and `A` are executed concurrently.
- Part of `A` and part of `B` are executed *at the same time*, then the remainder of `A` and `B` are executed concurrently.

Note that tasks can be run concurrently *even if* it is not possible for the underlying system to execute two tasks at the same time.

On the other hand, two tasks are said to be *executed in parallel* if their execution takes place (at least partly) at the same time.

Parallelism then requires hardware support to run two tasks at the same time. In the past, this implied distributed systems, but on modern computers, the presence of multiple *cores* of the processor allow for local parallel computation.

As the notion of concurrency is more general, we usually talk about concurrent execution, unless we specifically mean that tasks are executed at the exact same time.

2.2 Process vs. thread

A *process* is an instance of a program being executed on a machine. The machine, usually through an operating system, grants each process its own memory space, and handles the scheduling of the execution of this and all other processes.

A *thread* is a single sequence of instructions to be executed by the machine. Threads are not granted their own memory space by the machine (but the process may handle assigning memory to threads.)

A single process process may be made up of several threads; so long as the process is being executed, there is at least one thread (of execution.)

So, this gives rise to two approaches to concurrency: using multiple processes or using multiple threads. Processes are more independent, and usually more costly to create (because the system must allocate them memory.) Threads may be more dangerous, because they inherently share memory, which can lead to errors.

2.3 Race condition vs. deadlock vs. starvation

Race conditions, deadlock and starvation are all classes of error that can occur when using concurrency.

A *race condition* refers to an error that arises when one thread of execution performs a task before another thread of execution expected it to be done. Common instances include

- writing to memory while a thread expected memory to remain the same,
- :TODO:

A *deadlock* refers to an error that arises when multiple threads of execution are all waiting for access or control of some resources, and each holds a resource that the other is waiting for. This means no progress can be made until one thread relinquishes control of the resource it is holding.

A task is said to *starve* when it is waiting for access or control of some resource, but it does not get that access or control due to other threads "locking it out" (either indefinitely, or for an unacceptable amount of time.)

3 Creating threads with futures in Clojure

Clojure has support for several approaches to concurrency, including but not limited to:

- a STM (Software Transactional Memory) system,
- an *agent* system (somewhat related to the *actor model*), and
- an *atom* system which avoids race conditions regarding changes to data.

Refer to the documentation for more information about these.

For today, we are well served with a simpler approach: the `future` macro. A sequence of expressions wrapped in a `future` are evaluated in a new thread.

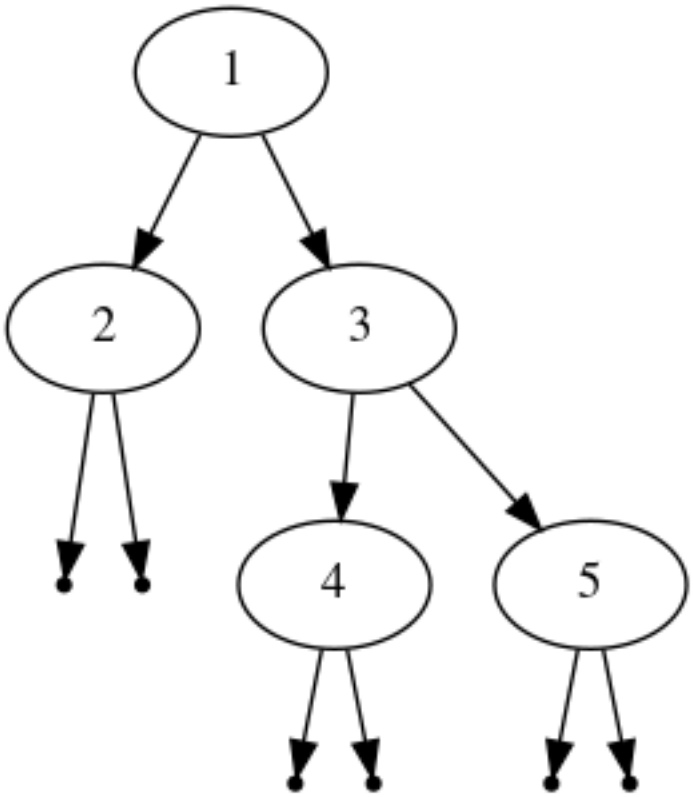
```
(future (Thread/sleep 4000) ;; 4 seconds
        (println "Computation on this thread paused and this
         ↪  printed after 4 seconds."))

(println "Computation on the main thread continued and this
 ↪  printed immediately.")
```

Before we look closer at futures, we must discuss how we represent these trees in Clojure. Then we will show a single-threaded implementation of the maximum function, and finally rework it into an approach using threads via futures.

3.1 Representing trees in Clojure

Quite simply, we will consider *nested sequences* to be trees. We will use the convention (at least for these notes) that the first element of the list is the *label* of a node, and the next two elements of the list gives the *children* of that node.



For instance, the tree can be written

```
'(1 (2) (3 (4) (5)))
```

Note that in the case that a tree has no children, we omit the rest of the list.

3.2 Finding the maximum of a tree

```
(defn maximum-tree
  "Find the maximum element of a tree of integers `t`.
`t` is assumed to be a list whose first element is an integer
(the label of the root) and whose remaining elements are lists
representing trees containing integers."
  [t]
  (if
    (empty? t) ##-Inf ;; Negative infinity
    ;; Use list deconstruction to separate the `label` and the
     ↪  list of `children`.
```

```
    (let [[label left right] t]
      (max label (maximum-tree left) (maximum-tree right)))))

(maximum-tree [1 [2] [3 [4 [10] [20]] [5]]])
```

3.3 Using futures

```
(defn maximum-tree
  "Find the maximum element of a tree of integers `t`.
`t` is assumed to be a list whose first element is an integer
(the label of the root) and whose remaining elements are lists
representing trees containing integers."
  [t]
  (print   "Finding the maximum of ")
  (println t)
  (if
    (empty? t) ##-Inf ;; Negative infinity
    ;; Use list deconstruction to separate the `label` and the
     ↪  list of `children`.
    (let [[label left right] t]
      (let [max-left  (future (maximum-tree left))
            max-right (future (maximum-tree right))]
        ;; Need to dereference the reference of the futures
        ;; either using the `deref` form or an `@`.
        (max label @max-left @max-right)))))
```

4 Forking processes in Ruby

4.1 Introduction

In Ruby, we can generate new *processes* easily using the `fork` method.

```
10.times do |i|
  fork do
    sleep 3
    puts "I am process #{i}"
  end
end

puts "All started."
```

```ruby
Process.waitall
puts "All done."
```

The `Process.waitall` method can be used to wait for all "child" processes to finish their execution.

Watch out for "zombie" processes if you do not wait!

```ruby
10.times do |i|
  fork do
    sleep 3
    puts "I'm zombie number #{i}!"
  end
end
abort "The main process has aborted, but the children might
 ↪  still act."
```

4.2 Summing trees

Now, consider an implementation of these trees with a `sumtree` operation in Ruby.

```ruby
class Tree
  def sumtree()
    case self
    when Leaf
      0
    when Branch
      # Non-concurrent solution:
      # self.data + self.lchild.sumtree + self.rchild.sumtree

      # Concurrent solution:
      # To communicate betwen processes, we use an IO pipe.
      reader, writer = IO.pipe
      fork do
        sum = self.lchild.sumtree
        writer.puts(sum) # Print the result to the pipe.
      end
      fork do
        sum = self.rchild.sumtree
        writer.puts(sum) # Print the result to the pipe.
      end
```

```
      Process.waitall
      sum1 = reader.gets.to_i   # Read one result and convert
       ↪  it to an integer.
      sum2 = reader.gets.to_i   # Read the other result and
       ↪  convert it.

      self.data + sum1 + sum2
    end
  end
end

class Leaf < Tree
end

class Branch < Tree
  attr_reader :lchild
  attr_reader :rchild
  attr_reader :data

  def initialize(l,v,r)
    @lchild = l
    @rchild = r
    @data = v
  end
end

x = Leaf.new
puts x.sumtree

x = Branch.new(Branch.new(Branch.new(Leaf.new,
                                     5,
                                     Leaf.new),
                          3,
                          Leaf.new),
               1,
               Branch.new(Leaf.new,
                          2,
                          Leaf.new))
puts x.sumtree
```

5 The `thread` library in Prolog

In (SWI) Prolog, one method to run goals concurrently is the thread library, which provides high-level threading primitives.

(There may be other methods, and there may be different methods in non-SWI implementations.)

The simplest predicate in the library is `concurrent`. `concurrent(N,Goals,Options)` runs

- each goal in the list of goals `Goals`,
- using `N` threads, and
- initialising each thread with the list of options `Options`.

```
sumtree(empty,0).
sumtree(branch(T1,V,T2),N) :-
        % sumtree(T1,N1),
        % sumtree(T2,N2),
        concurrent(2,[sumtree(T1,N1),sumtree(T2,N2)],[]),
        N is V + N1 + N2.
```

6 Futures in Scala

```
sealed trait Tree
case object Leaf extends Tree
case class Branch(left: Tree, n: Int, right: Tree) extends
 ↪  Tree

def sumtree(t: Tree): Int = t match {
  case Leaf => 0
  case Branch(l,n,r) =>
    sumtree(l) + n + sumtree(r)
}

import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Try, Success, Failure}

def sumtree(t: Tree): Int = t match {
  case Leaf => 0
```

```
  case Branch(l,v,r) =>
    val l_sum = Future { sumtree(l) }
    val r_sum = Future { sumtree(r) }
    l_sum + r_sum + v
}

def sumtree_future(t: Tree): Future[Int] = t match {
  case Leaf => Future { 0 }
  case Branch(l,n,r) =>
    val l_sum: Future[Int] = sumtree_future(l)
    val r_sum: Future[Int] = sumtree_future(r)

    l_sum.flatMap(the_l_sum =>
      r_sum.flatMap(the_r_sum =>
        Future { n + the_l_sum + the_r_sum }))
}
```

7 Justifying our example problem

:TODO: