Parallel Processing (1)

- Modern computers have multiple “cores” and are typically arrayed in clusters. This allows us (in practice, forces us) to do things *in parallel*. There are two types of parallel processing and *Scala* can be effective in both:
  - *immutable state*—the definition of the problem is known at the start and does not change until the problem is solved
    - e.g. count the total number of words in 1,000,000 documents
    - Amdahl’s law applies to this type of problem:
      - maximum possible speedup $\boldsymbol{\sigma} = 1/\boldsymbol{\alpha}$ where $\boldsymbol{\alpha}$ is the proportion of total time which is non-parallelizable
    - *Because each document is independent of the others*, a master node can divide up (this part is not parallelizable) the documents to be processed by a set of worker nodes/threads;
    - Then (in parallel) each worker counts the words in the documents it was given and returns the result, asynchronously of course, as a future value
    - Once all of these future values are realized, they can be summed to get the grand total (that final step is also non-parallelizable)
    - This is (more or less) the principle on which Map/Reduce works

Parallel Processing (2)

- continuing:
  - *mutable state*—the conditions are inherently changing at all times
    - e.g. ticket agency
      - there are two fundamental mutable states: the pool of tickets and the bank account of agency
      - it is therefore essential that these can be updated only by one thread at a time
    - there are other temporary mutable states—e.g. the status of an shopping cart—but these, if lost or corrupted, can be restarted
    - for these applications, we use *actors* (more about this later)

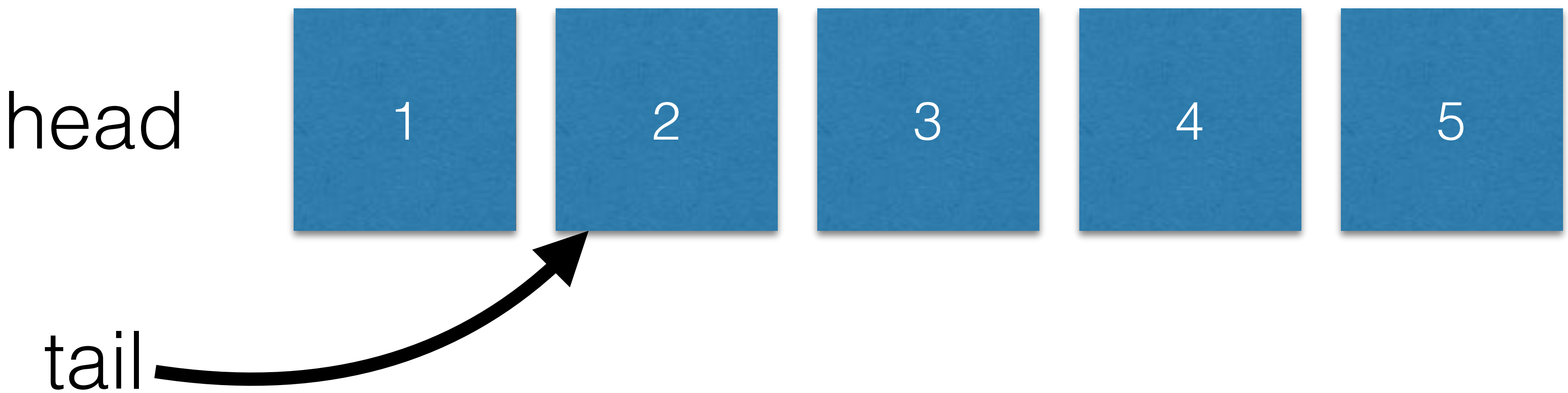
Parallel Processing (3)

- Apart from the internals of actors, is there any need to use mutable variables (vars) or mutable containers/collections?
- Caching (memoization)?
  - Maybe. But this can also be done inside an actor.
- Aggregation?
  - Normally, in functional programming we perform aggregation by recursion—no mutable state required.
- Sorting as a prelude to searching?
  - Yes—we do typically perform sorting on a mutable array but this again can/should be done inside an actor.
- Non-deterministic algorithms?
  - Yes, like the Newton Approximation or maybe genetic algorithms, etc.

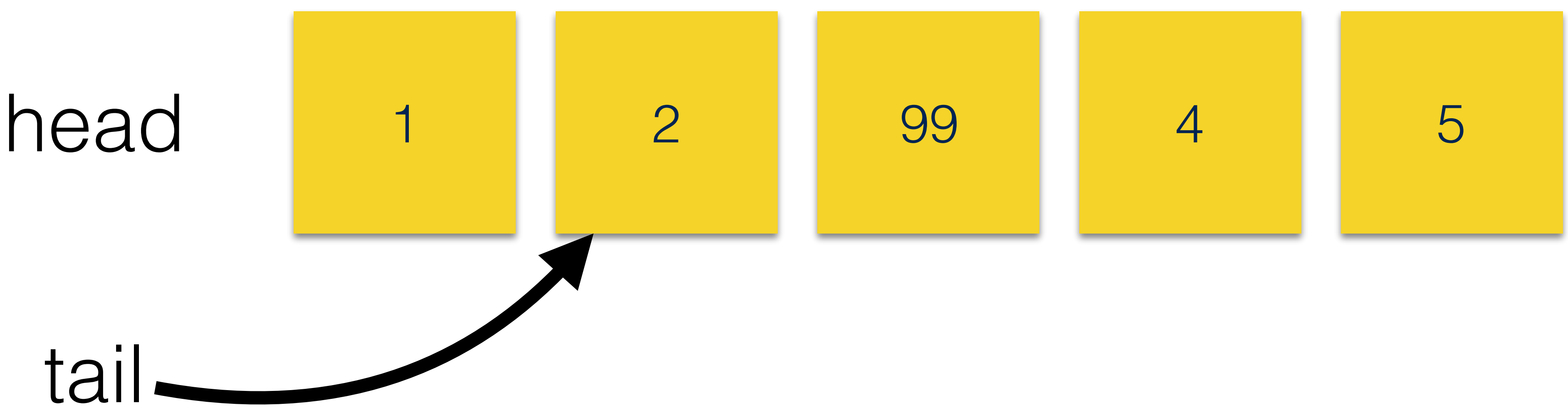
But surely we need *some* variables?

- Rarely! Although Scala provides *var* and has a library of mutable collections—you typically don't need them!
- Let's think about a box office app (see next slide):
  - if we are careful we can combine all mutable state into one *var* (the "state" of an actor);
  - when we take a list of items from another list (for example, we remove some tickets from the pool), surely we end up with lots of redundant copies of tickets! No, we don't.
  - in fact, the opposite is true: in an imperative program (e.g. Java) we typically end up doing a lot of list copying because our lists are mutable when they don't need to be!

Immutable list



Mutable list



Oops!

Ticket Agency

```
package edu.neu.coe.scala
package boxOffice
object BoxOffice {
  val initialState = State(List(Ticket(1,1,100),Ticket(1,2,100)), List())
  def makeTransaction(state: State, sale: Sale): State = ???
  def main(args: Array[String]): Unit = Tickets(initialState).start
}
case class Tickets(var state: State) extends scala.actors.Actor {
  def act() {
    while (state.availability) {
      receive {
        case sale: Sale => state = BoxOffice.makeTransaction(state,sale)
        case Status => sender ! state
        case _ => println("unknown message")
      }
    }
  }
}
case class State(tickets: List[Ticket], transactions: List[Transaction]) {
  def availability = tickets.length>0
}
case class Ticket(row: Int, seat: Int, price: Int)
case class Sale(tickets: List[Ticket], transaction: Transaction)
case class Transaction(creditCard: Long, total: Int, timestamp: Long, confirmation: String)
object Status
```

??? is a boon to designers who want to create stubs to be implemented later

Scala actors have been deprecated in favor of Akka

The field(s) of a case class can be marked "var"

Newton's Method updated

- Here's the Newton program in Scala:

```
package edu.neu.coe.scala
import scala.annotation.tailrec
import scala.util._
case class Newton(f: Double => Double, dfbydx: Double => Double) {
  private def step(x: Double, y: Double) = x - y / dfbydx(x)
  def solve(tries: Int, threshold: Double, initial: Double): Try[Double] = {
    @tailrec def inner(r: Double, n: Int): Try[Double] = {
      val y = f(r)
      if (math.abs(y) < threshold) Success(r)
      else if (n == 0) Failure(new Exception("failed to converge"))
      else inner(step(r, y), n - 1)
    }
    inner(initial, tries)
  }
}
object Newton extends App {
  val newton = Newton({ x => math.cos(x) - x }, { x => -math.sin(x) - 1 })
  newton.solve(10, 1E-10, 1.0) match {
    case Success(x) => println(s"the solution to math.cos(x) - x is $x")
    case Failure(t) => System.err.println(t.getLocalizedMessage)
  }
}
```

case class: "Newton" represents the problem domain

method "inner" marked "tailrec" and returns *Try[Double]*

companion object to Newton class extends *App*

if: chooses from expressions — like Java: (x?y:z)

Observations on Scala version

- It is a little more verbose than the original (non O-O) “Fortran” version.
- It’s functional and object-oriented.
- There are *no (!)* variables or any other mutable state.
- Iteration is replaced by recursion:
  - Oh? but isn’t that inefficient in practice and therefore to be avoided?
  - No: we use “tail recursion” (more later).

More improvements?

- There are still improvements we can make but we can come back to this later:
  - no protection against division by zero
  - doesn’t behave well for very small or very large numbers
  - generalize

The evils of mutable state when testing

```
package edu.neu.coe.scala;

import static org.junit.Assert.assertEquals;
import org.junit.BeforeClass;
import org.junit.Test;
import scala.util.Random;

public class GenericTest {
	private static Random random;
	@BeforeClass
	public static void setUpBeforeClass() throws Exception {
		random = new Random(0L);
	}
	@Test
	public void test1() {
		int x = 0;
		for (int i = 0; i < 2; i++)
			x = random.nextInt();
		assertEquals(x, -723955400);
	}
	@Test
	public void test2() {
		int x = random.nextInt();
		assertEquals(x, 1033096058);
	}
}
```

This is Java

what if we change this number? How will test2 work out?

what if the test runner decides to run test2 before test1?

- We will come back to this problem later.