Lecture 10: Parallel Programming in Scala

Computer Science Department, University of Crete

Multicore Processor Programming

Based on slides by P. Haller, material by scala-lang.org
The Actor Model

- A model of concurrent computation
- Introduced in 1973 (Lisp, Simula)
- Main idea: *Everything is an Actor*
  - Similar to OO idea that *Everything is an Object*
- An actor can:
  - Send messages to other actors
  - Create new actors
  - React to messages it receives
- There is no constraint on order between these:
  - Can occur in parallel across actors, also for any actor
  - Parallel computation and communication
Actors in Scala

- **send, receive** constructs adopted from Erlang
- **send** is asynchronous
  - Incoming messages buffered in actor’s *mailbox*
- **receive** picks the first message in the mailbox that matches one of the patterns `msg_pat_i`
- If no pattern matches, the actor suspends
Goals of Scala

- Create a language with better support for component software

Hypotheses:

- Programming language for component software should be scalable
  - The same concepts describe small and large parts
  - Rather than adding lots of primitives, focus on abstraction, composition, decomposition
- Language that unifies OOP and functional programming can provide scalable support for components

// Asynchronous message send
actor ! message

// Message receive
receive {
  case msg_pat_1 => action_1
  ...
  case msg_pat_i => action_i
}
Goals of Scala

- Create a language with better support for component software
- Hypotheses:
  - Programming language for component software should be scalable
    - The same concepts describe small and large parts
    - Rather than adding lots of primitives, focus on abstraction, composition, decomposition
  - Language that unifies OOP and functional programming can provide scalable support for components

```scala
// Asynchronous message send
actor ! message

// Message receive
receive {
  case msg_pat_1 => action_1
  ...
  case msg_pat_i => action_i
}
```

Partial Function of type `PartialFunction[Msg, Action]`
val summer = actor {
  var sum = 0
  loop {
    receive {
      case ints: Array[Int] =>
        sum += ints.reduceLeft((a, b) => (a+b) % 7)
      case from: Actor =>
        from ! sum
    }
  }
}
Goals of Scala Actors

- Offer high scalability on mainstream platforms
- Integrate well with thread-based code
- Provide safe, intuitive, efficient message passing
Actor Implementation with Threads

- One thread per actor
- Rely on JVM to map threads to OS processes and HW cores
- `receive` blocks the actor’s thread while waiting for a message

Pros:
- No inversion of control
- Interoperability with threads

Cons:
- High memory consumption
- Context switching overhead
Event-Based Actors

- Problem of thread-based actors
  - Actors consume lots of resources
  - Waiting for messages is expensive
- Idea: Suspend actors, save continuation closure and release current thread
- Transparent thread pooling

```
def act() {
  react {
    case Put(x) =>
      react {
        case Get(from) =>
          from ! x
          act()
      }
    }
  }
```
Invocations do *not* return!
  ▶ Must provide continuation in the body of `react`

No need to write code in continuation-passing style (CPS)
  ▶ Use *control-flow combinators* to enable composition

```
a andThen b // runs a followed by b

def loop(body: => Unit) = body andThen loop(body)
```
Thread-based Programming

- Actors should be able to block their thread temporarily:
  - When interacting with thread-based code
  - When it is difficult to provide the continuation

```scala
val tasks: List[Task]
tasks foreach { task => worker ! task }
val results = tasks map { task =>
  receive {
    case Done(result) => result
  }
}
```
Actors should be able to block their thread temporarily:

- When interacting with thread-based code
- When it is difficult to provide the continuation

```scala
val tasks: List[Task] = tasks foreach { task => worker!task }
val results = tasks map { task =>
  receive {
    case Done(result) => result
  }
}
```

Blocks the current thread if the actor has to wait for a message.
Any object can be a message
  ▶ Including Actor objects

// use singleton objects for messages
case object Ping
case object Pong
case object Stop

// import actors
import scala.actors.Actor
import scala.actors.Actor._
Example: Thread-based Actors (2)

- Actor objects inherit from `Actor` class

```scala
class Ping(count: Int, pong: Actor) extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! Ping
    while (true) {
      receive {
        case Pong =>
          if (pingsLeft % 1000 == 0) {
            Console.println("Ping: pong")
            if (pingsLeft > 0) {
              pong ! Ping
              pingsLeft -= 1
            } else {
              Console.println("Ping: stop")
              pong ! Stop
              exit()
            }
          }
      }
    }
  }
}
```
class Pong extends Actor {
    def act() {
        var pongCount = 0
        while (true) {
            receive {
                case Ping =>
                    if (pongCount % 1000 == 0)
                        Console.println("Pong: ping " + pongCount)
                    sender ! Pong
                    pongCount = pongCount + 1
                case Stop =>
                    Console.println("Pong: stop")
                    exit()
            }
Example: Thread-based Actors (3)

```scala
class Pong extends Actor {
  def act() {
    var pongCount = 0
    while (true) {
      receive {
        case Ping =>
          if (pongCount % 1000 == 0)
            Console.println("Pong: ping " + pongCount)
          sender ! Pong
          pongCount = pongCount + 1
        case Stop =>
          Console.println("Pong: stop")
          exit()
      }
    }
  }
}
```

Method of the **Actor** class, returns reference to sender of message
Example: Thread-based Actors (4)

```scala
object pingpong extends App {
  val pong = new Pong
  val ping = new Ping(100000, pong)
  ping.start
  pong.start
}
```
object pingpong extends App {
    val pong = new Pong
    val ping = new Ping(100000, pong)
    ping.start
    pong.start
}
class Pong extends Actor {
    def act() {
        var pongCount = 0
        loop {
            react {
                case Ping =>
                    if (pongCount % 1000 == 0)
                        Console.println("Pong: ping "+pongCount)
                    sender ! Pong
                    pongCount = pongCount + 1
                case Stop =>
                    Console.println("Pong: stop")
                    exit()
            }
        }
    }
}
class PreOrder(n: Tree) extends Producer[int] {
    def produceValues = traverse(n)
    def traverse(n: Tree) {
        if (n != null) {
            produce(n.elem)
            traverse(n.left)
            traverse(n.right)
        }
    }
}
abstract class Producer[T] {
  protected def produceValues: Unit

  protected def produce(x: T) {
    coordinator! Some(x)
    receive { case Next => }
  }

  private val producer: Actor = actor {
    receive {
      case Next =>
        produceValues
        coordinator! None
    }
  }

  ...
}
private val coordinator: Actor = actor {
    loop {
        react {
            case Next =>
                producer ! Next
            reply {
                receive {
                    case x: Option[_] => x
                }
            }
            case Stop => exit('stop)
        }
    }
}
def iterator = new Iterator[T] {
  private var current: Any = Undefined
  private def lookAhead = {
    if (current == Undefined) current = coordinator !? Next
    current
  }

  def hasNext: Boolean = lookAhead match {
    case Some(x) => true
    case None => { coordinator ! Stop; false }
  }

  def next: T = lookAhead match {
    case Some(x) => current = Undefined; x.asInstanceOf[T]
  }
}