The Many Flavors of Parallel Programming in Scala

Philipp Haller, Stanford University and EPFL
Scala’s Toolbox for Parallel Programming

- Actors
- Parallel Graph Processing
- STM
- Parallel DSLs
- Futures
- Parallel Collections
- Distributed
Scala’s Toolbox for Parallel Programming

- Actors
- Parallel Graph Processing
- STM
- Parallel DSLs
- Futures
- Parallel Collections
- Distributed
ACTORS
in Scala

Sunday, July 17, 2011
Scala Actors.

Send/receive constructs adopted from **Erlang**

Send is asynchronous: messages are buffered in actor’s **mailbox**

Receive picks the first message in the mailbox that matches one of the patterns msgpat_i

If no pattern matches the actor suspends

```
// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat_1 => action_1
  ...
  case msgpat_n => action_n
}
```
A Simple Actor.

```scala
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
    }
  }
}
```
Erlang-style Actors.
Erlang-style Actors.

✗ No inversion of control
- Message reception is explicit and blocking
Erlang-style Actors.

- No inversion of control
  - Message reception is explicit and blocking

- Fine-grained message filtering
  - Messages are filtered upon reception
Erlang-style Actors.

- No inversion of control
  - Message reception is explicit and blocking
- Fine-grained message filtering
  - Messages are filtered upon reception
- **Not** Erlang-style actors: E, Kilim, (Akka)
Implementing Actors.

Thread-based implementation:
Implementing Actors.

Thread-based implementation:

- One thread per actor
Implementing Actors.

Thread-based implementation:

- One thread per actor
- JVM maps threads to OS processes
Implementing Actors.

Thread-based implementation:

- One thread per actor
- JVM maps threads to OS processes
- Receive blocks thread while waiting for message
Implementing Actors.

Thread-based implementation:

- One thread per actor
- JVM maps threads to OS processes
- Receive blocks thread while waiting for message

**Pros**
- No inversion of control
- Interoperability with threads

**Cons**
- High memory consumption
- Context switching overhead
Event-Based Actors.
Event-Based Actors.

**Main Problem** of thread-per-actor model:

Actors consume a lot of resources while waiting for messages.
Event-Based Actors.

**Main Problem** of thread-per-actor model:

Actors consume a lot of resources while waiting for messages.

**Idea:** Suspend actor by saving continuation closure and releasing current thread.
Event-Based Actors.

**Main Problem** of thread-per-actor model:

Actors consume a lot of resources while waiting for messages.

**Idea:** Suspend actor by saving continuation closure and releasing current thread

```scala
def act() {
  react { case Put(x) =>
    react { case Get(from) =>
      from ! x
      act()
    }
  }
}
```
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
  }
}
```

Blocks current thread if actor has to wait for a message
Managing Blocking.

Actor (many)

Thread Pool

task queue

worker threads (few)
Managing Blocking.

Actors (many)

Thread Pool

worker threads (few)
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  receive {
    case Next =>
  }

Thread Pool

- worker threads (few)
- task queue
- task queue
- task queue
- task queue

Actors (many)
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  
```scala
receive {
  case Next =>
}
```

Thread Pool

Worker threads (few)

Actors (many)
Managing Blocking.

Actors (many)

Actor A:

• Start 3 actors
• Then:
  receive {
    case Next =>
  }

Thread Pool

worker threads (few)
Managing Blocking.

- Start 3 actors
- Then:
  - receive {
    - case Next =>
  }

Actor A:

```scala
receive {
  case Put(x) =>
}
```

- Thread Pool
- Actors (many)
- Worker threads (few)
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  receive {
    case Next =>
  }

Thread Pool:

- Task queues
- Worker threads (few)

Actors (many)

receive {
  case Put(x) =>
}

Sunday, July 17, 2011
Managing Blocking.

Actor A:
- Start 3 actors
- Then:
  receive {
    case Next =>
  }

Thread Pool:
- worker threads (few)
- task queue
- task queue
- task queue
- task queue

receive {
  case Put(x) =>
}

Actors (many)
Managing Blocking.

Actor A:
- Start 3 actors
- Then:
  receive {
    case Next =>
  }

Thread Pool:
- worker threads (few)
- task queue
- Actors (many)

receive {
  case Put(x) =>
}

receive {
  case Put(x) =>
}
Managing Blocking.

Actor A:
• Start 3 actors
• Then:
  receive {
    case Next =>
  }

receive {
  case Put(x) =>
}

receive {
  case Put(x) =>
}

Thread Pool:
worker threads (few)

Actors (many)
Managing Blocking.

Actor A:
- Start 3 actors
- Then: receive { case Next => }

```scala
actor {
  A ! Next
} receive {
  case Put(x) =>
}
```

Thread Pool:
- worker threads (few)
  - task queue

Actors (many):
- receive {
  case Put(x) =>
}
- receive {
  case Put(x) =>
}

Sunday, July 17, 2011
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  receive { case Next => }

Thread Pool:

<table>
<thead>
<tr>
<th>Task Queue</th>
<th>Task Queue</th>
<th>Task Queue</th>
<th>Task Queue</th>
</tr>
</thead>
</table>

worker threads (few)

Actors (many)

- receive { case Put(x) => }
- receive { case Put(x) => }
- receive { case Put(x) => }

Scala | Scalathon, Philadelphia, PA. July 16-17, 2011. 10
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  - receive { case Next => }
  - receive { case Put(x) => }

### Thread Pool

- worker threads (few)
- task queue

### Actors (many)

- receive { case Put(x) => }
- receive { case Put(x) => }
- receive { case Put(x) => }

```scala
actor {
  A ! Next
} receive {
  case Put(x) =>
} `} receive {
  case Put(x) =>
} `} receive {
  case Put(x) =>
} `} receive {
  case Put(x) =>
} ```

Actor A:

Managing Blocking.
Managing Blocking.

**Actor A:**
- Start 3 actors
- Then:
  - receive { **case** Next => }

**Thread Pool**
- worker threads (few)
- task queue

**Actors (many)**
- receive { **case** Put(x) => }

**Scalathon, Philadelphia, PA. July 16-17, 2011.**
Managing Blocking.

Actor A:
- Start 3 actors
- Then: receive { case Next => }

Thread pool locked up!

- Actors (many)
- Worker threads (few)
Managing Blocking.

Actor A:

- Start 3 actors
- Then: receive { case Next => }

Thread pool locked up!

MUST AVOID situation where:
- all worker threads blocked.
- there is a task in some task queue.

worker threads (few)
def receive[R](f: PartialFunction[Any, R]): R = {
  ...
  val elem = mailbox.extractFirst(msg => f.isDefinedAt(msg))
  if (elem == null) {
    synchronized {
      waitingFor = f
      isSuspended = true
      scheduler.managedBlock(blocker)
    }
  } else {
    // process message...
  }
  ...
}
def receive[R](f: PartialFunction[Any, R]): R = {
  ...
  val elem = mailbox.extractFirst(msg => f.isDefinedAt(msg))
  if (elem == null) {
    synchronized {
      waitingFor = f
      isSuspended = true
      scheduler.managedBlock(blocker)
    }
  } else {
    // process message...
  }
}

object blocker extends ManagedBlocker {
  def block() = {
    Actor.this.suspendActor()
  }
  def isReleasable = !Actor.this.isSuspended
}

Sunday, July 17, 2011
There is more.

- Continuations
  - Can use them once the continuations plugin is enabled by default (probably in Scala 2.10)
- Akka
  - Part of the Typesafe stack
  - We are working on merging them with scala.actors
The definitive guide to actors in the standard library
Not (only) an API reference
Language support for actors
Principles, patterns
Covers Akka’s actors

2nd preprint published Mar 2011, print release (planned for) end of September
Parallel Graph Processing

Joint work with Heather Miller
Data is growing.

At the same time, there is a growing desire to do MORE with that data.
Menthor...
Menthor... 

is a framework for parallel graph processing. 

(But it is not limited to graphs.)
Menthor...

- is a framework for parallel graph processing.
  (But it is not limited to graphs.)

- is inspired by BSP.
  With functional reduction/aggregation mechanisms.
Menthor...

- is a framework for parallel graph processing. (But it is not limited to graphs.)
- is inspired by BSP.
  With functional reduction/aggregation mechanisms.
- avoids an inversion of control
  of other BSP-inspired graph-processing frameworks.
**Menthor...**

- is a framework for parallel graph processing. (But it is not limited to graphs.)

- is inspired by BSP.
  With functional reduction/aggregation mechanisms.

- avoids an inversion of control of other BSP-inspired graph-processing frameworks.

- is implemented in Scala, and there are preliminary experimental results.
Menthor’s Model of Computation.
Data.
Data.

Split into data items managed by *vertices*. and sizes range from primitives to large matrices.
Data.

Split into data items managed by vertices.
Relationships expressed using edges between vertices.
Algorithms.
Algorithms.

Data items stored inside of vertices iteratively updated.
Algorithms.

Data items stored inside of vertices iteratively updated. Iterations happen as SYNCHRONIZED SUPERSTEPS.

(inspired by the BSP model)
Algorithms.

Data items stored inside of vertices \textit{iteratively} updated. Iterations happen as \textsc{Synchronized Supersteps}.
Algorithms.

Data items stored inside of vertices \textit{iteratively} updated. Iterations happen as \textsc{Synchronized Supersteps}.

I. update each vertex in parallel.
Algorithms.

Data items stored inside of vertices *iteratively* updated. Iterations happen as **SYNCHRONIZED SUPERSTEPS**.

1. update each vertex in *parallel*.
2. update produces *outgoing* messages to other vertices
Algorithms.

Data items stored inside of vertices *iteratively* updated. Iterations happen as **Synchronized Supersteps**.

1. update each vertex in *parallel*.
2. *Update* produces *outgoing* messages to other vertices
3. incoming messages available at the beginning of the next **Superstep**.
Substeps. (and Messages)

SUBSTEPS are computations that,
Substeps. (and Messages)

**SUBSTEPS** are computations that,

1. update the value of *this Vertex*
**Substeps. (and Messages)**

**SUBSTEPS** are computations that,

1. update the value of *this* Vertex
2. return a list of messages:
   ```scala
   case class Message[Data](source: Vertex[Data], dest: Vertex[Data], value: Data)
   ```
**Substeps. (and Messages)**

**Substeps** are computations that,

1. update the value of this Vertex
2. return a list of messages:
   
   ```scala
case class Message[Data](source: Vertex[Data], dest: Vertex[Data], value: Data)
```

**Examples...**

```scala
{
  value = ...
  List()
}
```
Substeps. (and Messages)

**SUBSTEPS** are computations that,

1. update the value of this Vertex
2. return a list of messages:

```
case class Message[Data](source: Vertex[Data],
                         dest: Vertex[Data],
                         value: Data)
```

EXAMPLES...

```
{  
  value = ...
  List()
}
```

```
{  
  ...
  for (nb <- neighbors)
  yield Message(this, nb, value)
}
```
Substeps. (and Messages)

**Substeps** are computations that,

1. update the value of this Vertex
2. return a list of messages:

    ```scala
case class Message[Data](source: Vertex[Data],
                           dest: Vertex[Data],
                           value: Data)
```

**Examples...**

```scala
{  
    value = ...
    List()  
}
```

```scala
{  
    for (nb <- neighbors)  
    yield Message(this, nb, value)  
}
```

Each is *implicitly* converted to a **Substep[Data]**
class PageRankVertex extends Vertex[Double](0.0d) {
  def update() = {
    var sum = incoming.foldLeft(0)(_ + _.value)
    value = (0.15 / numVertices) + 0.85 * sum

    if (superstep < 30) {
      for (nb <- neighbors) yield
        Message(this, nb, value / neighbors.size)
    } else
      List()
  }
}
Implementation Principles.
Implementation Principles.

❌ A pure Scala library

- No staging and code generation.
- No dependency on language virtualization.
Implementation Principles.

✗ A pure Scala library
- No staging and code generation.
- No dependency on language virtualization.

✗ Benefits
- Compatible with mainline Scala compiler.
- Fast compilation.
- Simple debugging and troubleshooting.
Implementation Principles.

✗ A pure Scala library
- No staging and code generation.
- No dependency on language virtualization.

✗ Benefits
- Compatible with mainline Scala compiler.
- Fast compilation.
- Simple debugging and troubleshooting.

✗ Drawbacks
- No aggressive optimizations.
- No support for heterogeneous hardware platforms.
Conclusions
Conclusions

Can avoid inversion of control in vertex-based BSP using closures.
Conclusions

✗ Can avoid inversion of control in vertex-based BSP using closures.
✗ Higher-order functions useful for reductions, in an imperative model.
Conclusions

- Can avoid inversion of control in vertex-based BSP using closures.
- Higher-order functions useful for reductions, in an imperative model.
- Explicit parallelism feasible if computational model simple (cf. MapReduce)
Conclusions

- Can avoid inversion of control in vertex-based BSP using closures.
- Higher-order functions useful for reductions, in an imperative model.
- Explicit parallelism feasible if computational model simple (cf. MapReduce)
- The puzzle pieces are there to make analyzing big data much easier.

http://lcavwww.epfl.ch/~hmiller/menthor/
Heterogeneous Parallel DSLs

Based on the work at Stanford University’s PPL and EPFL
Heterogeneous Parallel Programming
Heterogeneous Parallel Programming
Heterogeneous Parallel Programming
Heterogeneous Parallel Programming
Heterogeneous Parallel Programming

- Pthreads
- OpenMP
- CUDA
- OpenCL
- Verilog
- VHDL
- MPI
- Sun T2
- Nvidia Fermi
- Cray Jaguar
Heterogeneous Parallel Programming

Applications

- Scientific Engineering
- Virtual Worlds
- Personal Robotics
- Data informatics

Too many different programming models

Programming Models

- Sun T2
- Nvidia Fermi
- CUDA
- OpenCL
- Verilog
- VHDL
- MPI
- Pthreads
- OpenMP

Sunday, July 17, 2011
Hypothesis and New Problem

Q: Is it possible to write one program and run it on all these targets?
Hypothesis and New Problem

Q: Is it possible to write one program and run it on all these targets?

HYPOTHESIS: Yes, but need domain-specific languages

THOUGH, IT’S QUITE DIFFICULT TO CREATE DSLS USING CURRENT METHODS.
Lightweight Modular Staging.

Typical Compiler

- Lexer
- Parser
- Type checker
- Analysis
- Optimization
- Code gen
Lightweight Modular Staging.

Embedded DSL gets it all for free, but can’t change any of it

Typical Compiler

Lexer → Parser → Type checker → Analysis → Optimization → Code gen
Lightweight Modular Staging.

Typical Compiler

- Lexer
- Parser
- Type checker
- Analysis
- Optimization
- Code gen

Stand-alone DSL implements everything
Lightweight Modular Staging.

Modular Staging provides a hybrid approach

Typical Compiler

Lexer → Parser → Type checker → Analysis → Optimization → Code gen
Lightweight Modular Staging.

Modular Staging provides a hybrid approach

DSLs adopt front-end from highly expressive embedding language but can customize IR and participate in backend phases

Typical Compiler

Lexer ➔ Parser ➔ Type checker ➔ Analysis ➔ Optimization ➔ Code gen
Lightweight Modular Staging.

Modular Staging provides a hybrid approach

Typical Compiler

Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs by Tiark Rompf, Martin Odersky (GPCE’10)
Linear Algebra Example.

```scala
object TestMatrix {
    def example(a: Matrix, b: Matrix, c: Matrix, d: Matrix) = {
        val x = a*b + a*c
        val y = a*c + a*d
        println(x+y)
    }
}
```

Targeting heterogeneous HW requires changing

- how data is represented
- how operations are implemented
Abstracting Matrices.

Use abstract type constructor

- Do not fix a specific implementation, yet
- Operations work on abstract matrices

```scala
type Rep[T]

def infix_+(x: Rep[Matrix], y: Rep[Matrix]): Rep[Matrix]

def example(a: Rep[Matrix], b: Rep[Matrix], c: Rep[Matrix], d: Rep[Matrix]) = {
  val x = a*b + a*c
  val y = a*c + a*d
  println(x+y)
}
```

**IMPLEMENTATION DOESN’T CHANGE!**
Staging.

Programming using only \texttt{Rep[Matrix]}, \texttt{Rep[Vector]} etc. allows different implementations for \texttt{Rep}

\textbf{Example}: expression trees

\begin{verbatim}
abstract class Exp[T]
case class Const[T](x: T) extends Exp[T]
case class Symbol[T](id: Int) extends Exp[T]
abstract class Op[T]
\end{verbatim}

Matrix implementation:

\begin{verbatim}
type Rep[T] = Exp[T]

def infix_+(x: Exp[Matrix], y: Exp[Matrix]) = 
    new PlusOp(x, y)

class PlusOp(x: Exp[Matrix], y: Exp[Matrix])
    extends DeliteOpZip[Matrix]
\end{verbatim}
Staging.

Programming using only $\text{Rep}[\text{Matrix}]$, $\text{Rep}[\text{Vector}]$ etc. allows different implementations for $\text{Rep}$

**Example:** expression trees

```scala
abstract class Exp[T]
case class Const[T](x: T) extends Exp[T]
case class Symbol[T](id: Int) extends Exp[T]
abstract class Op[T]
```

Matrix implementation:

```scala
type Rep[T] = Exp[T]

def infix_+(x: Exp[Matrix], y: Exp[Matrix]) =
  new PlusOp(x, y)

class PlusOp(x: Exp[Matrix], y: Exp[Matrix])
  extends DeliteOpZip[Matrix]
```
The Delite DSL Framework

- Provides IR with parallel execution patterns
  **Example:** DeliteOpZip[T]

- Parallel optimization of IR graph

- Compiler framework with support for heterogeneous hardware platforms

- DSL extends parallel operations
  **Example:** class Plus extends DeliteOpZip[Matrix]

- Domain-specific analysis and optimization
The Delite IR Hierarchy

Domain User Interface

Domain Analysis & Opt.

Parallelism Analysis & Opt.

Code Generation

Generic Analysis & Opt.

Application

M1 = M2 + M3

V1 = exp(V2)

s = sum(M)

C2 = sort(C1)

DSL User

DSL Author

Delite

Delite

Op

Matrix Plus

Vector Exp

Matrix Sum

CollectionQuick sort

ZipWith

Map

Reduce

Divide & Conquer

Generic analysis & Opt.
Delite DSL Compilers.

Diagram:
- Liszt program
- OptiML program
- Scala Embedding Framework
- Delite Parallelism Framework
- Intermediate Representation (IR)
  - Base IR
  - Delite IR
  - DS IR
- Generic Analysis & Opt.
- Parallelism Analysis, Opt. & Mapping
- Domain Analysis & Opt.
Contributing to Delite

- Lots of cool things to work on
- New applications using existing DSLs
  - Example: recommender engine using OptiML
- New tools: scripts (delitec), profilers, debuggers, visualizers, ...
- New data input sources (cluster runtime!)
- Expand Getting Started guide, documentation, ...
Scala’s Collections.

Collections are organized in two packages.
Scala’s Collections.

Collections are organized in two packages.

- scala.collection.mutable
- scala.collection.immutable
Scala’s Collections.

Collections are organized in two packages.

- **scala.collection.mutable**
  Can change, add, or remove elements in place **as a side effect**

- **scala.collection.immutable**
Scala’s Collections.

Collections are organized in two packages.

**scala.collection.mutable**

Can change, add, or remove elements in place as a side effect

**scala.collection.immutable**

Methods that transform an immutable collection return a new collection and leave the old collection unchanged
Scala’s Collections.

Collections are organized in two packages.

`scala.collection.mutable`  `scala.collection.immutable`

Abstract classes in `scala.collection`
Parallel Collections.

Scala 2.9 introduces *Parallel Collections*, based on the idea that many operations can safely be performed in parallel.
Parallel Collections.

Scala 2.9 introduces *Parallel Collections*, based on the idea that many operations can safely be performed in parallel.

**Just add `.par`**

And the same operation is performed in parallel:

```scala
myCollection.par.foldLeft(0)((a,b) => a+b)
```
Parallel Collections.

Scala 2.9 introduces *Parallel Collections*, based on the idea that many operations can safely be performed in parallel.

Just add `.par`

And the same operation is performed in parallel:

```
myCollection.par.foldLeft(0)((a,b) => a+b)
```

```
6 9 = 15
```

```
1 2 3
4 5
```
New **method** added to regular collections

Returns a **parallel version of the collection** pointing to the same underlying data

Use **.seq** to go back to the sequential collection

Parallel sequences, maps, and sets defined in separate hierarchy
The Collections Hierarchy.
The Collections Hierarchy.

Immutable parallel collections:
- ParRange
- ParVector
- ParHashMap
- ParHashSet
The Collections Hierarchy.

Based on hash tries

Traversable

Iterable

Seq

GenTraversable

GenIterable

GenSeq

Irremutatle parallel collections:

ParRange

ParVector

ParHashMap

ParHashSet

ParIterable

ParSeq
The Collections Hierarchy.
The Collections Hierarchy.

Mutable parallel collections:
- ParArray
- ParHashMap
The Collections Hierarchy.
The Collections Hierarchy.

Why isn’t a ParSeq a Seq?
Implementing Parallel Collections.
Implementing Parallel Collections.

**Goal:** define operations in terms of *a few common abstractions*

- Typically, in terms of a foreach method or iterators
- However, their sequential nature makes these approaches *ill-suited for parallel execution!*
Implementing Parallel Collections.

**Goal:** define operations in terms of *a few common abstractions*

- Typically, in terms of a foreach method or iterators
- However, their sequential nature makes these approaches **ill-suited for parallel execution!**

**Instead:** *abstractions for splitting and combining*

- Split collection into non-trivial partition
- Iterate over disjunct subsets in parallel
- Combine partial results computed in parallel
Splitters and Combiners.
A splitter is an iterator that can be recursively split into disjoint iterators:

```scala
trait Splitter[T] extends Iterator[T] {
  def split: Seq[Splitter[T]]
}
```
Splitters and Combiners.

A splitter is an iterator that can be recursively split into disjoint iterators:

```scala
trait Splitter[T] extends Iterator[T] {
  def split: Seq[Splitter[T]]
}
```

A combiner combines partial results

- The combine method returns a combiner containing the union of its argument elements
- Results from different tasks are combined in a tree-like manner

```scala
trait Combiner[T, Coll] extends Builder[T, Coll] {
  def combine(other: Combiner[T, Coll]): Combiner[T, Coll]
}
```
Summary.
Summary.

Simple transition from regular collections to parallel collections ("just add .par!")

- If access patterns aren’t inherently sequential
Summary.

Simple transition from regular collections to parallel collections ("just add .par!")
- If access patterns aren’t inherently sequential

Parallel collections are implemented in terms of splitters and combiners
- Parallel collections must provide efficient implementations of those
Summary.

❯ Simple transition from regular collections to parallel collections ("just add .par!")
  ✔ If access patterns aren’t inherently sequential

❯ Parallel collections are implemented in terms of splitters and combiners
  ✔ Parallel collections must provide efficient implementations of those

❯ Collection-based programming is easy and powerful
  ✔ Can we make it work for more applications and for distribution?
What’s Next

We only scratched the surface:

• Debugging, Testing
• Combining parallel and concurrent collections
• More programming models/synchronizers
  • X10-style async/finish, phasers in JDK7, ...
  • Pipelines, streaming, data flow, ...
• Determinism, side effects, thread locality, ...
• Exploiting the Java Memory Model
How?

- Scala great vehicle for pushing cutting-edge research into practice

- Extractors, continuations, named and default arguments, implicits, parallel collections, ...

- Industrial practice demands stability, backward compatibility

- Another good research topic: API migration

- But: this doesn’t hinder research on concurrency libraries!
THANK YOU.
Questions?