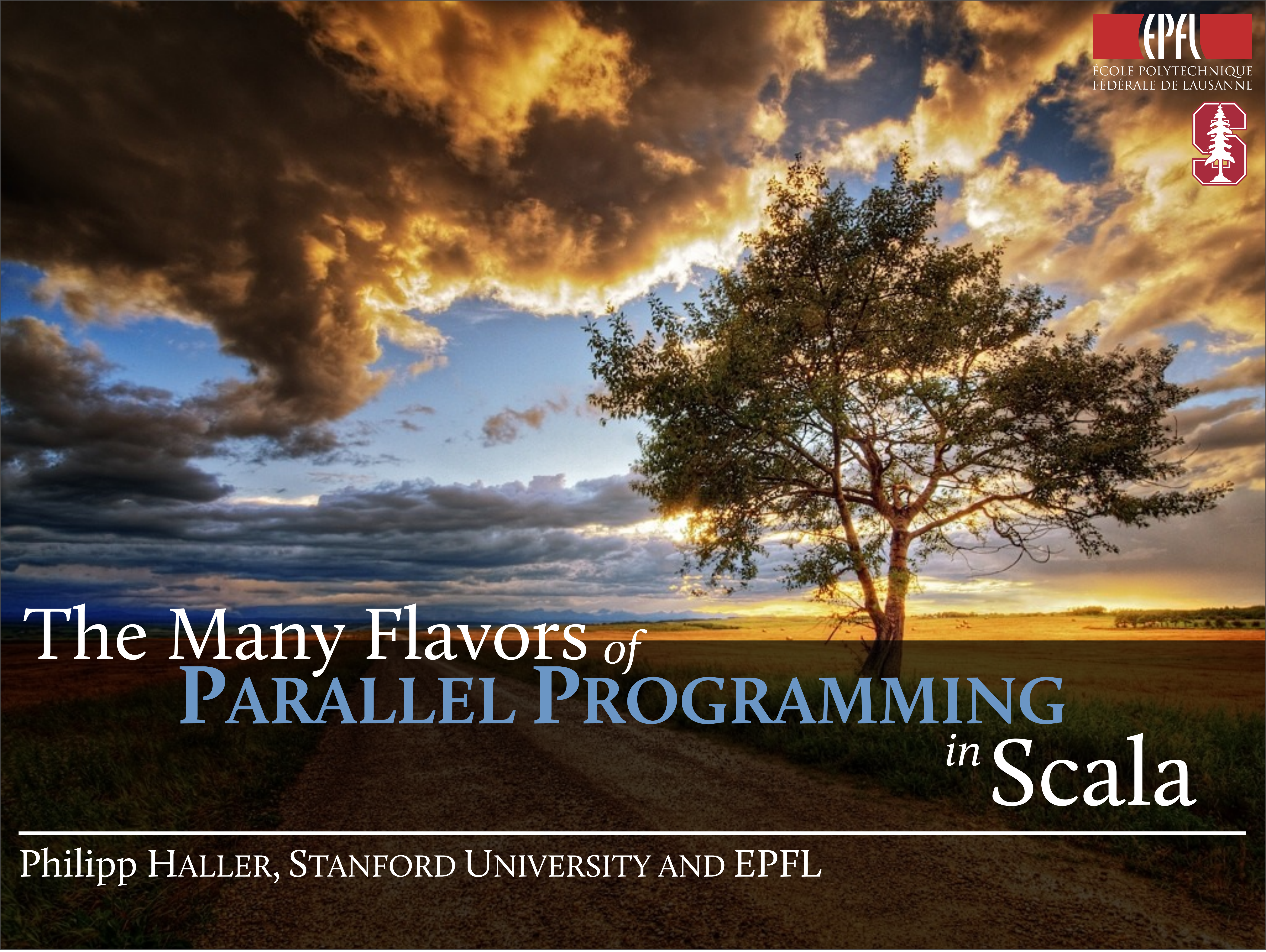


The Many Flavors *of* Parallel Programming *in* Scala

Philipp Haller, Stanford University and EPFL

Sunday, July 17, 2011

Scala's Toolbox for Parallel Programming

ACTORS

PARALLEL GRAPH PROCESSING

STM

PARALLEL DSLs

FUTURES

COLLECTIONS → PARALLEL

COLLECTIONS → DISTRIBUTED

Scala's Toolbox for Parallel Programming

- ACTORS
- PARALLEL GRAPH PROCESSING
- STM
- PARALLEL DSLs
- FUTURES
- COLLECTIONS → PARALLEL, DISTRIBUTED


ACTORS
in Scala

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 msgpat₁ => action₁
  …
  case msgpatₙ => actionₙ
}
```

A Simple Actor.

```
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	Cons
– No inversion of control	– High memory consumption
– Interoperability with threads	– 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

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

Thread-based Programming

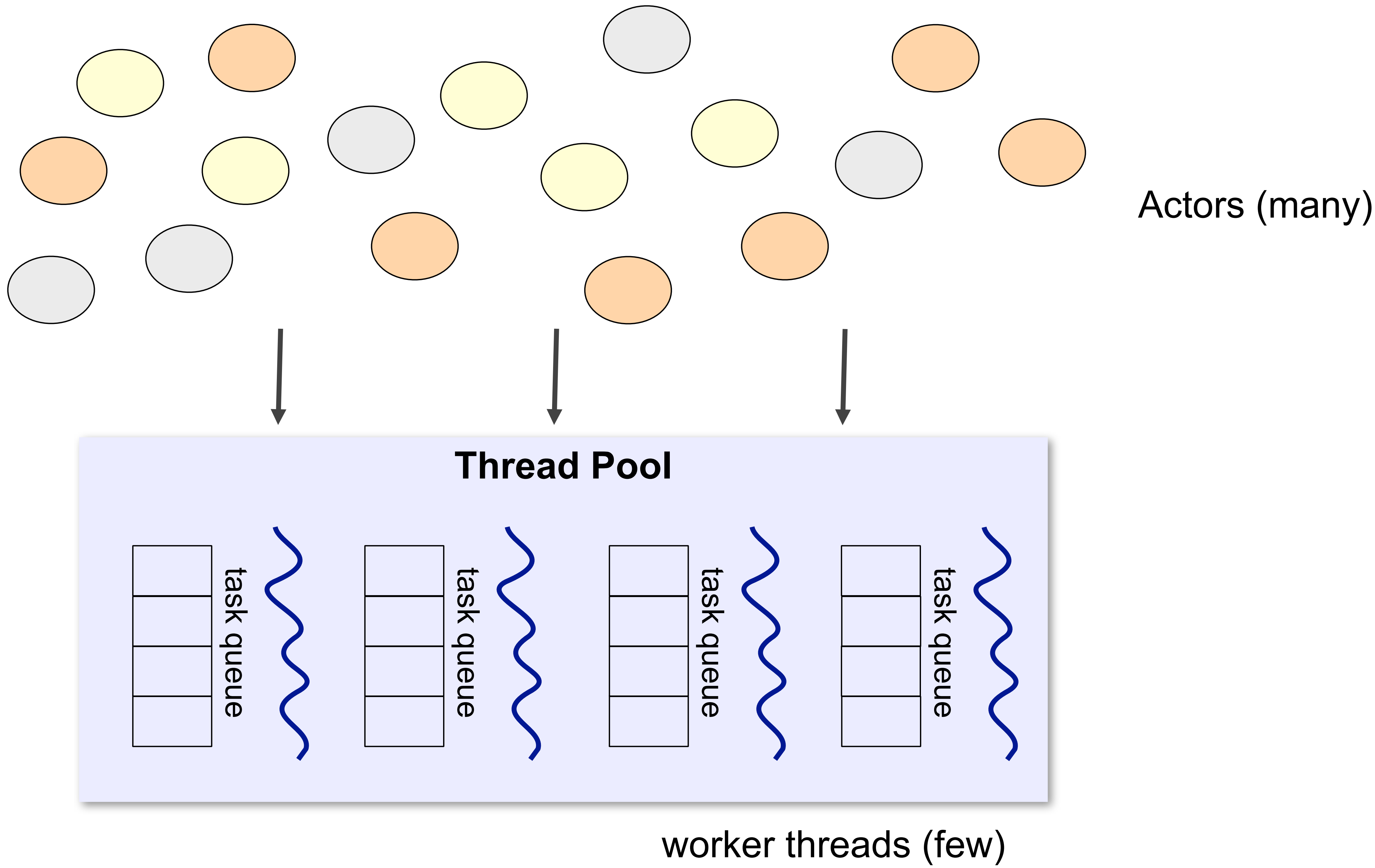
Actors should be able to block their thread temporarily:

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

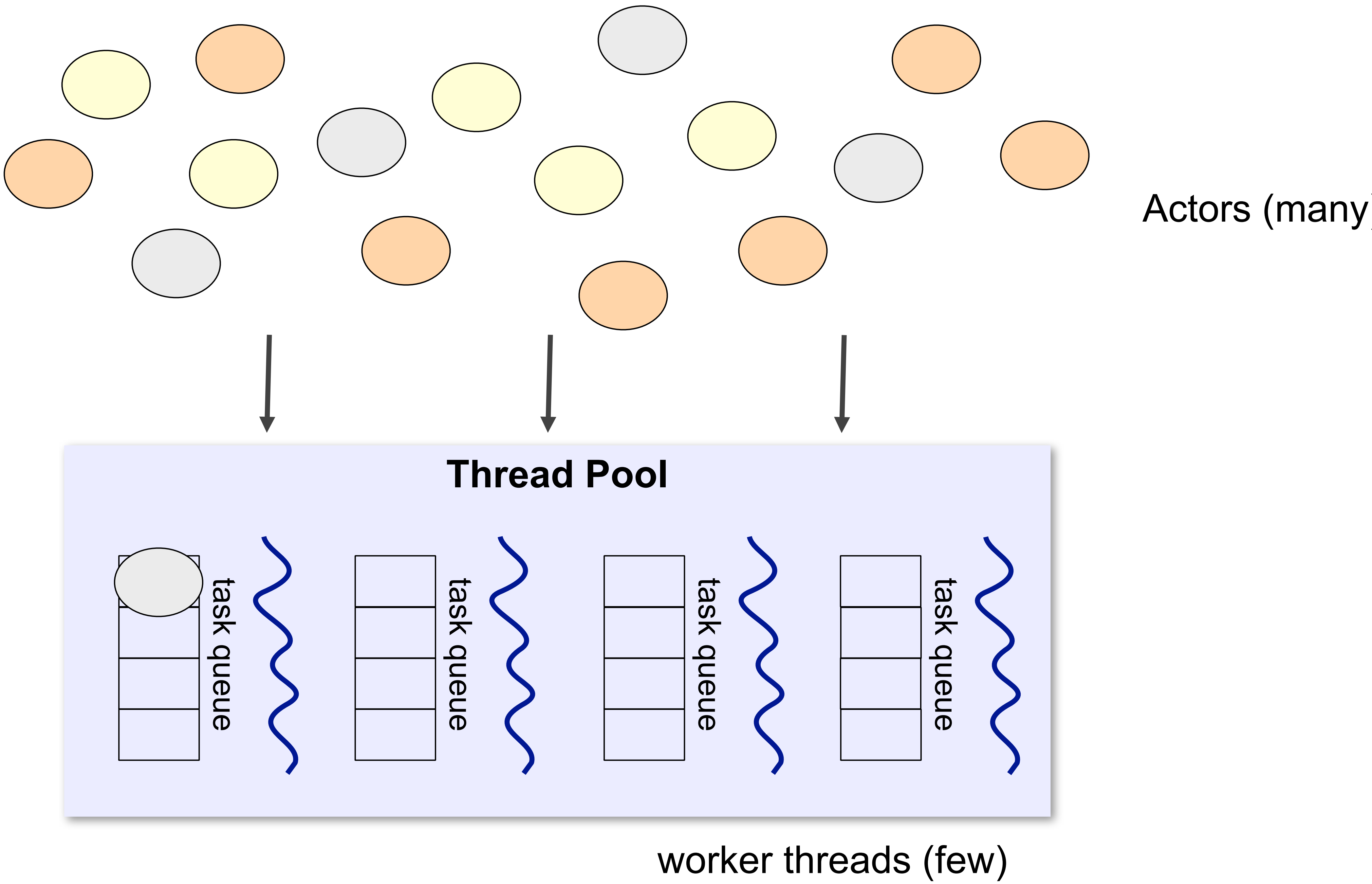
```
val tasks: List[Task]
tasks foreach { task => worker ! task }
val results = tasks map { task =>
  receive {
    case Done(result) => result
  }
}
```

Blocks current thread if actor has to wait for a message

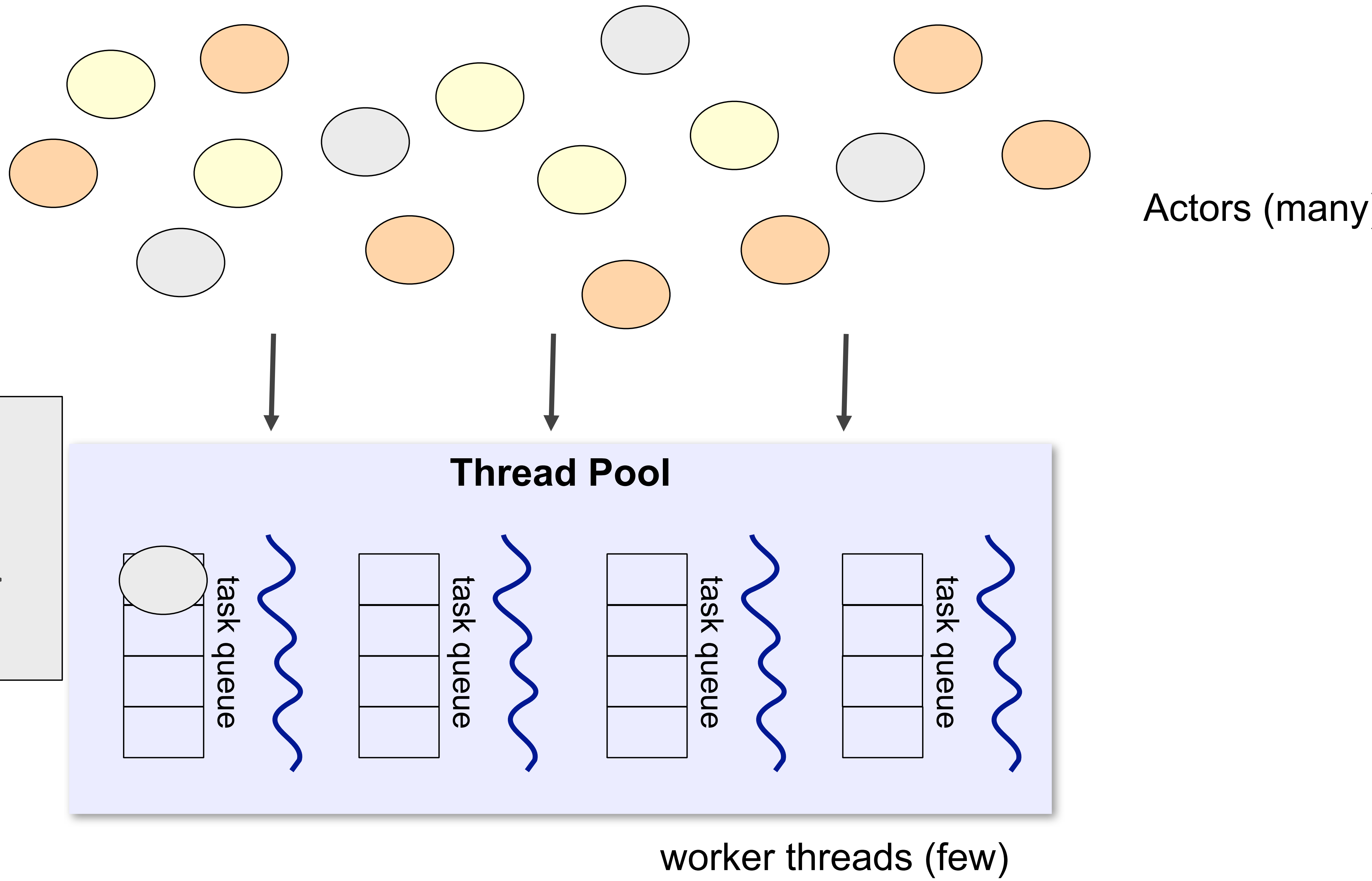
Managing Blocking.



Managing Blocking.



Managing Blocking.

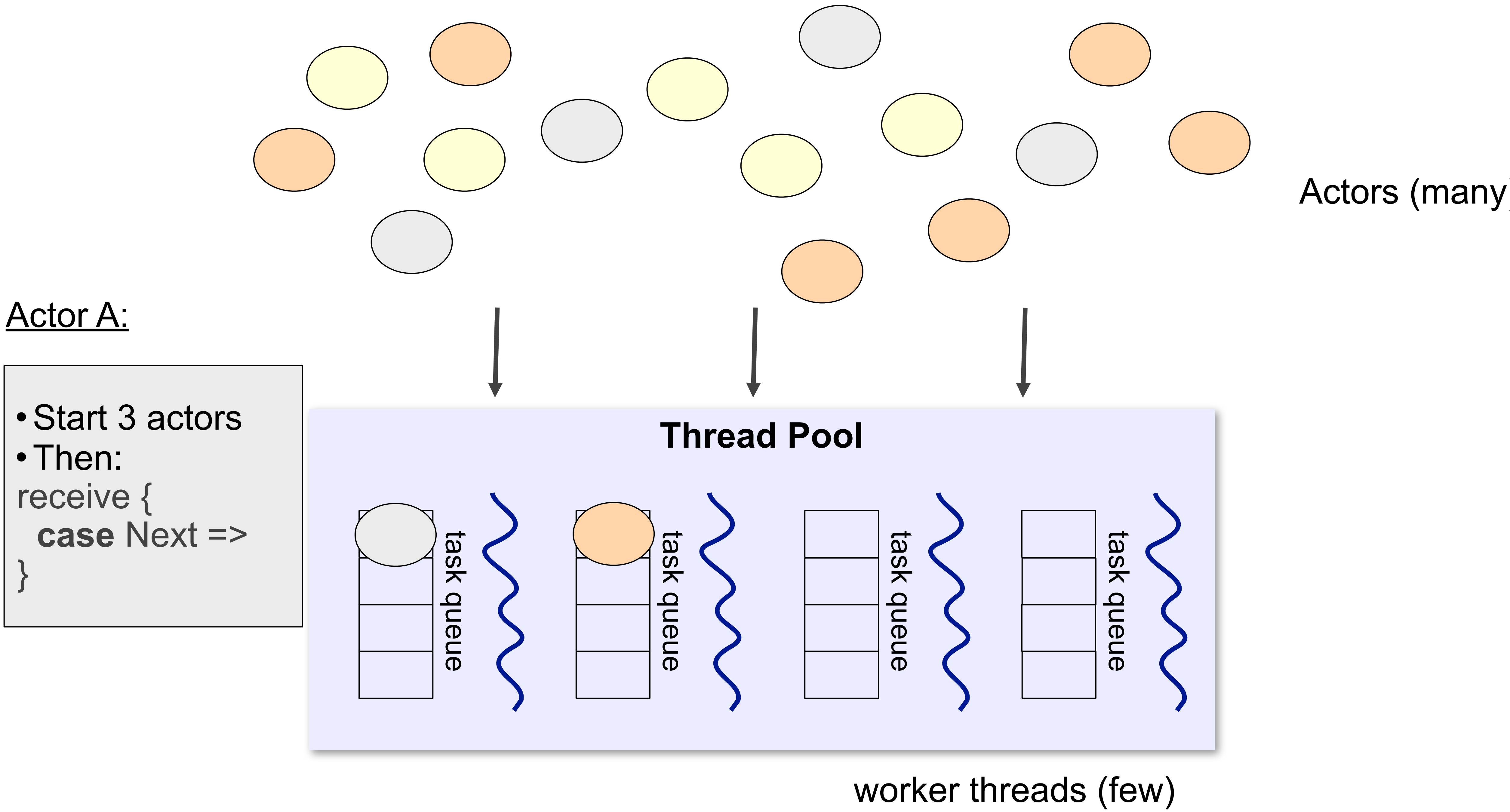


Actor A:

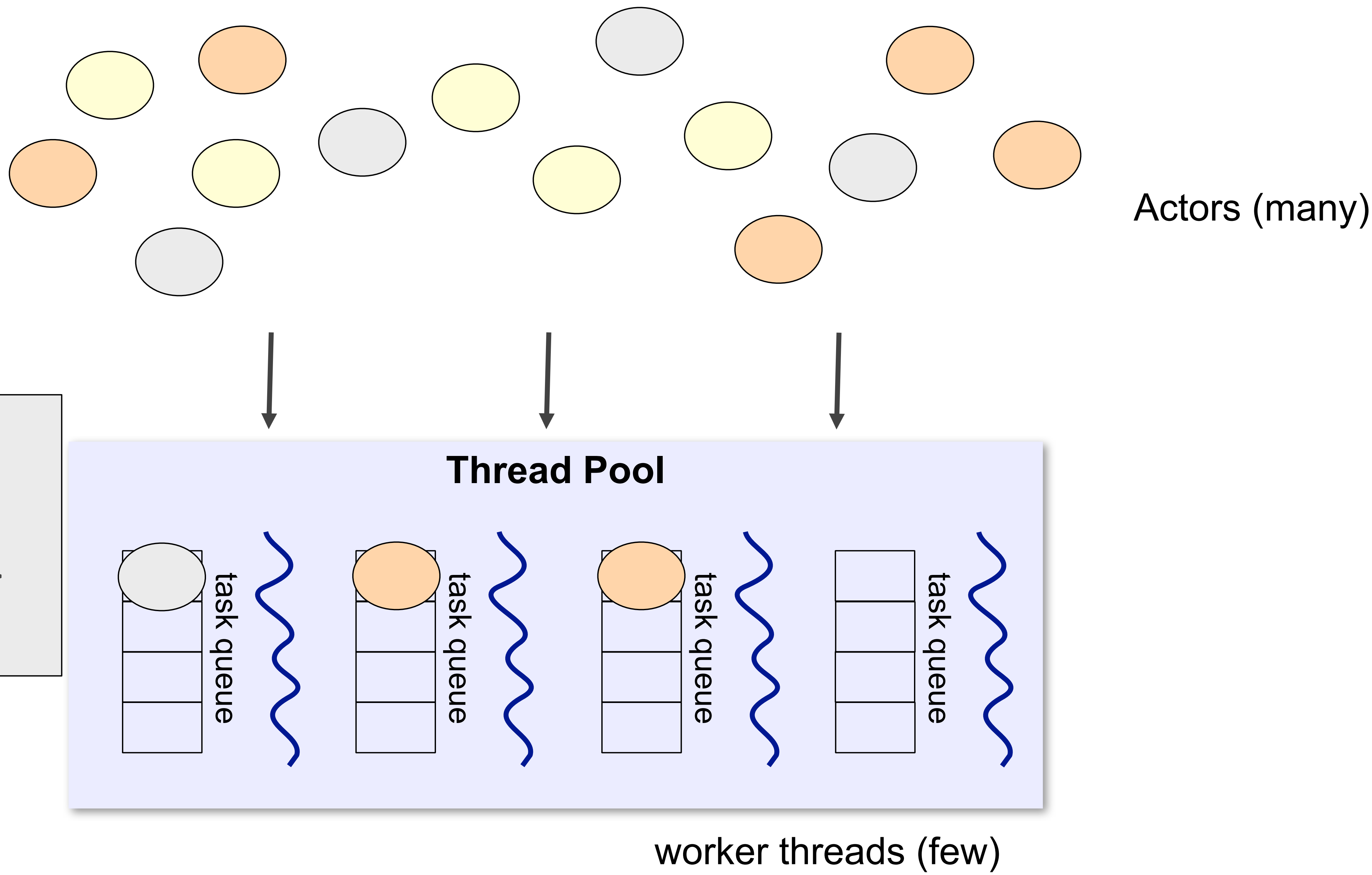
- Start 3 actors
- Then:

```
receive {
  case Next =>
}
```

Managing Blocking.



Managing Blocking.

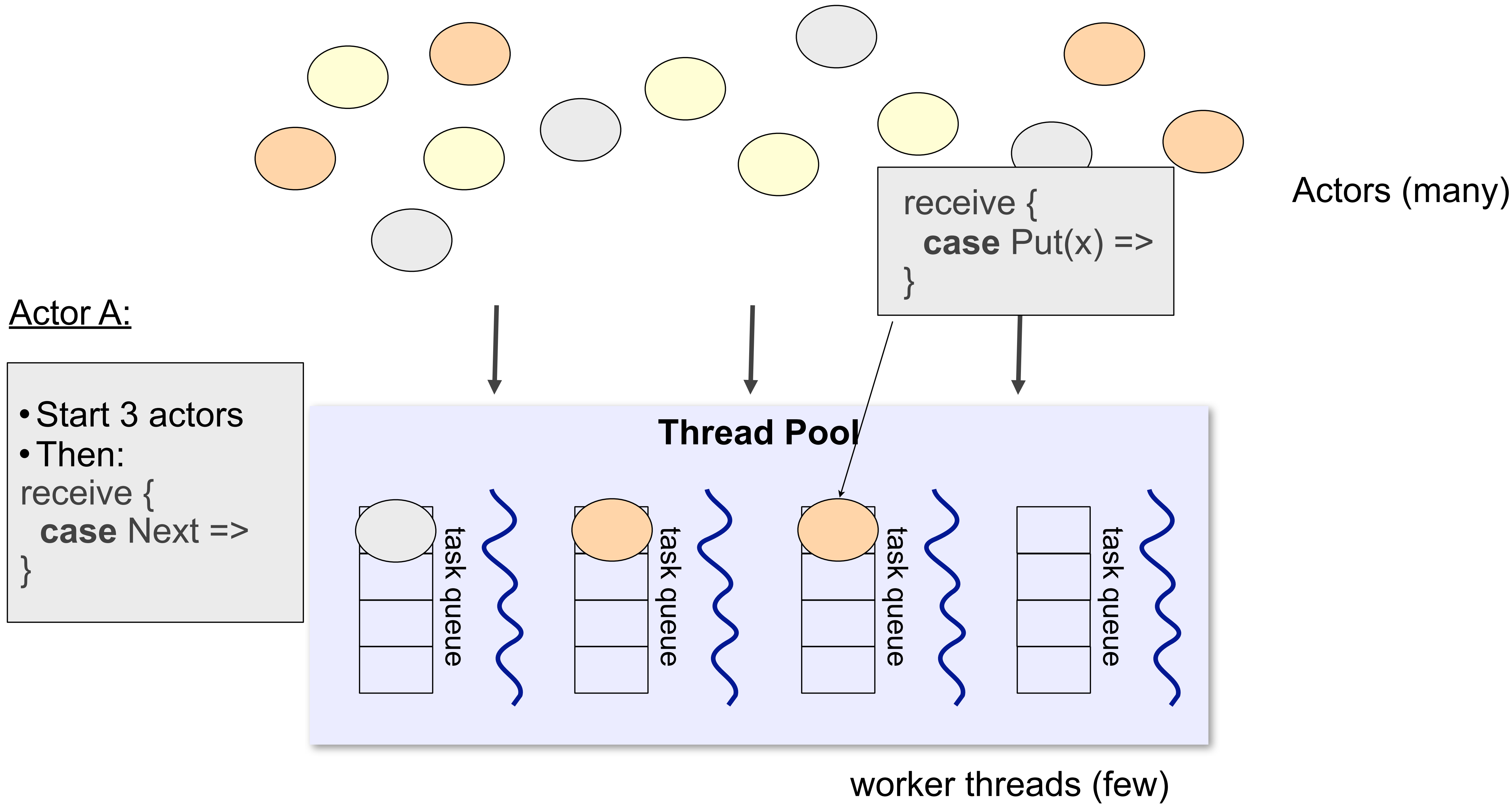


Actor A:

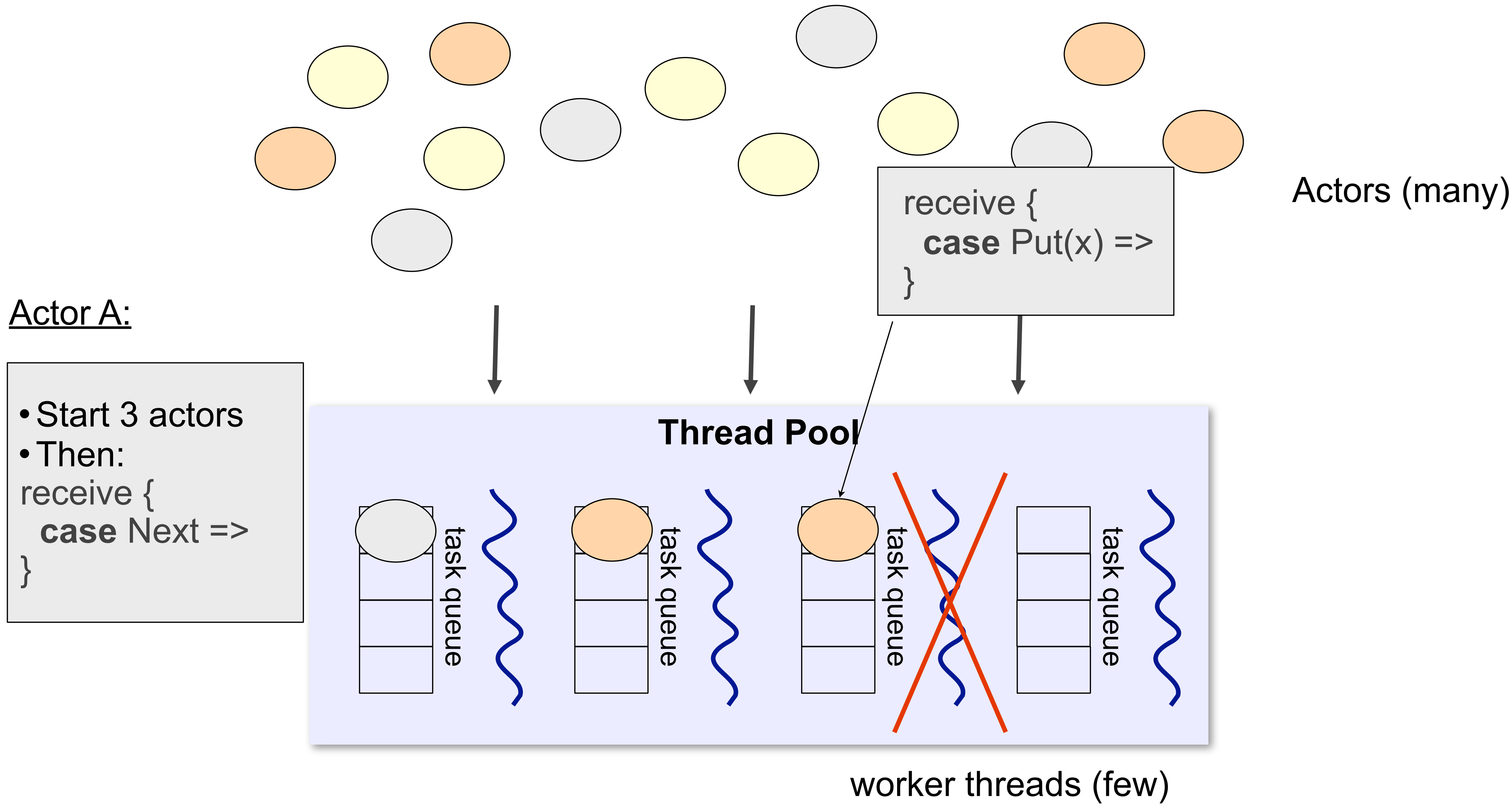
- Start 3 actors
- Then:

```
receive {
  case Next =>
}
```

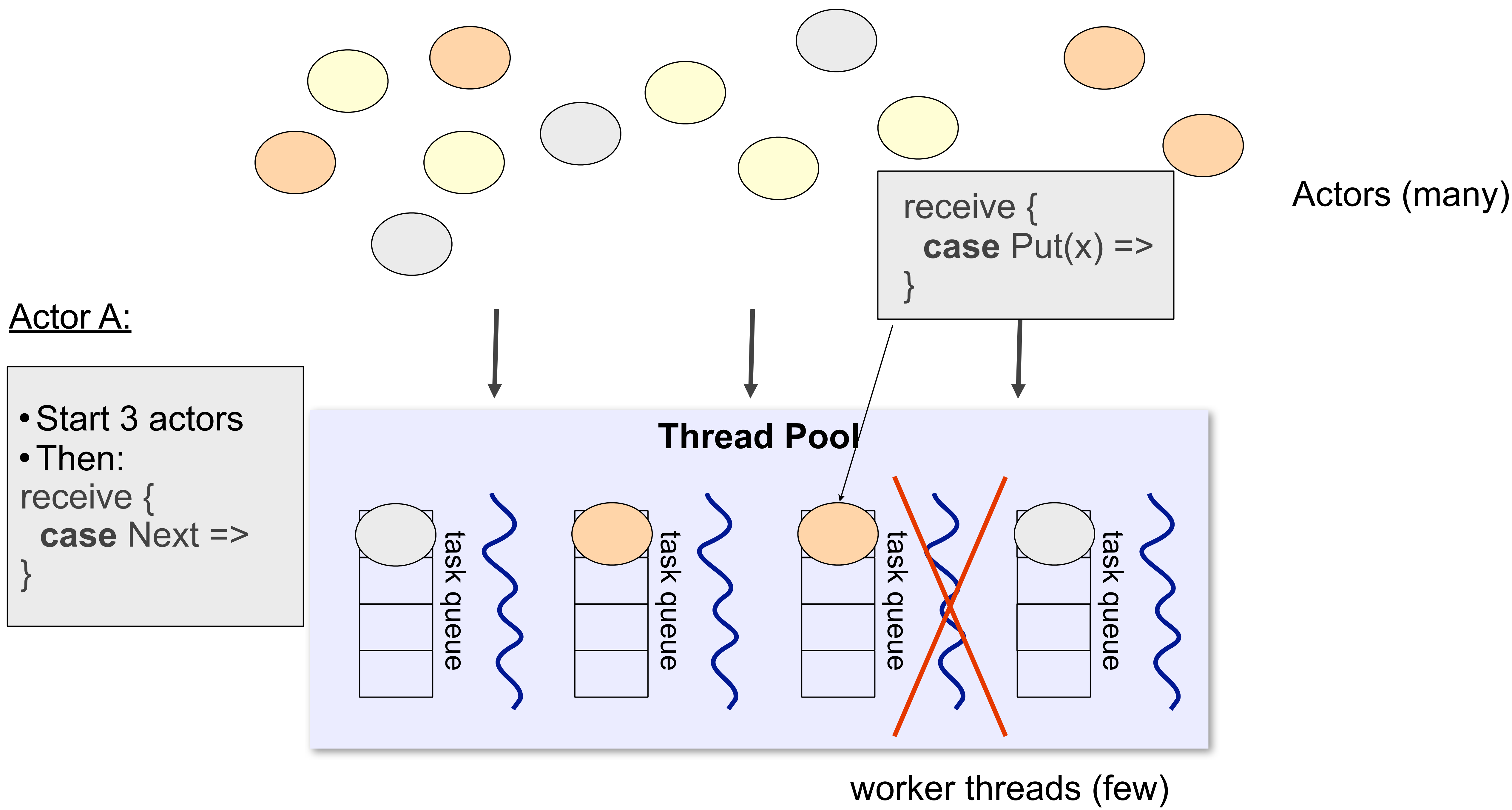
Managing Blocking.



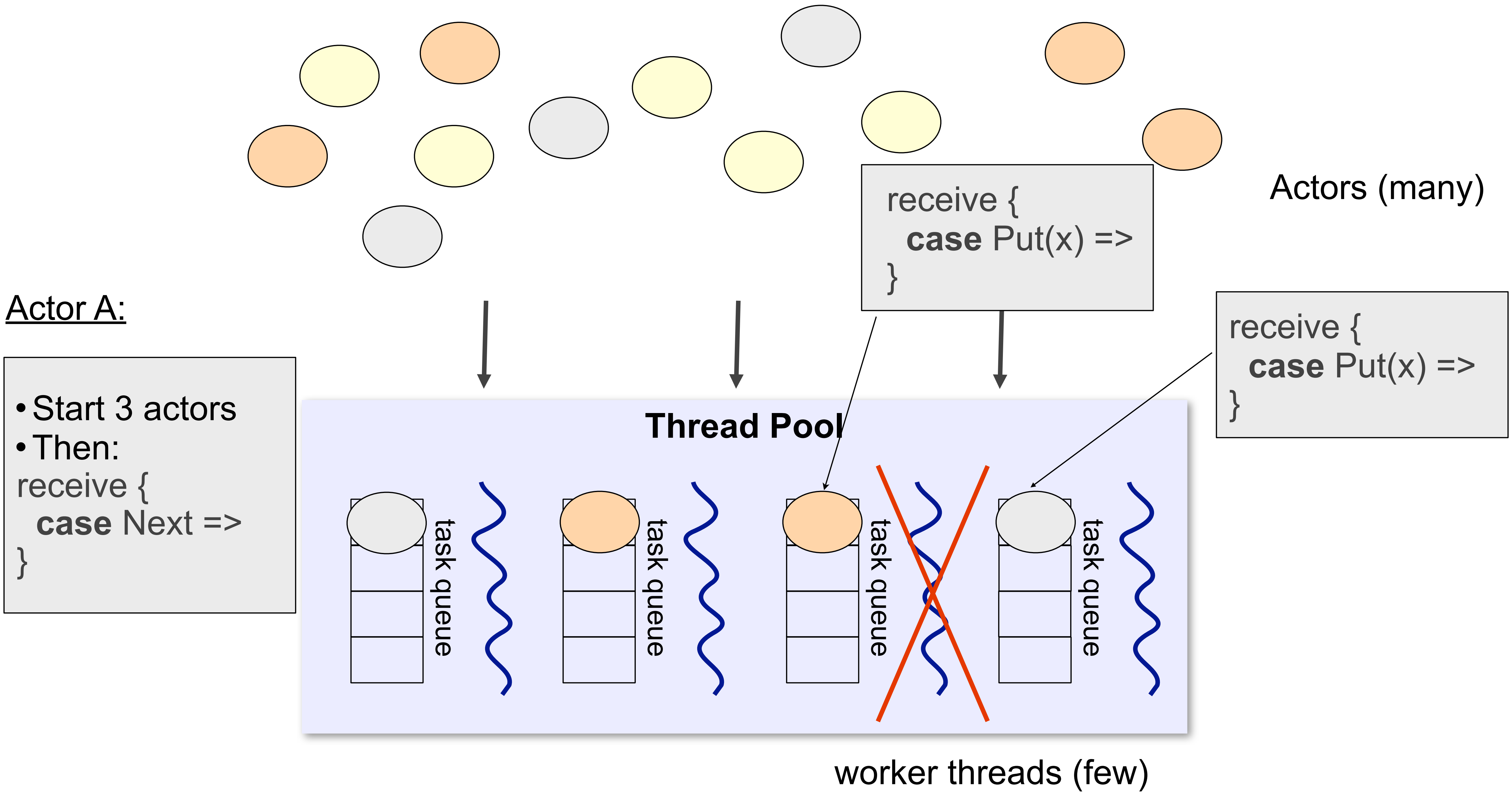
Managing Blocking.



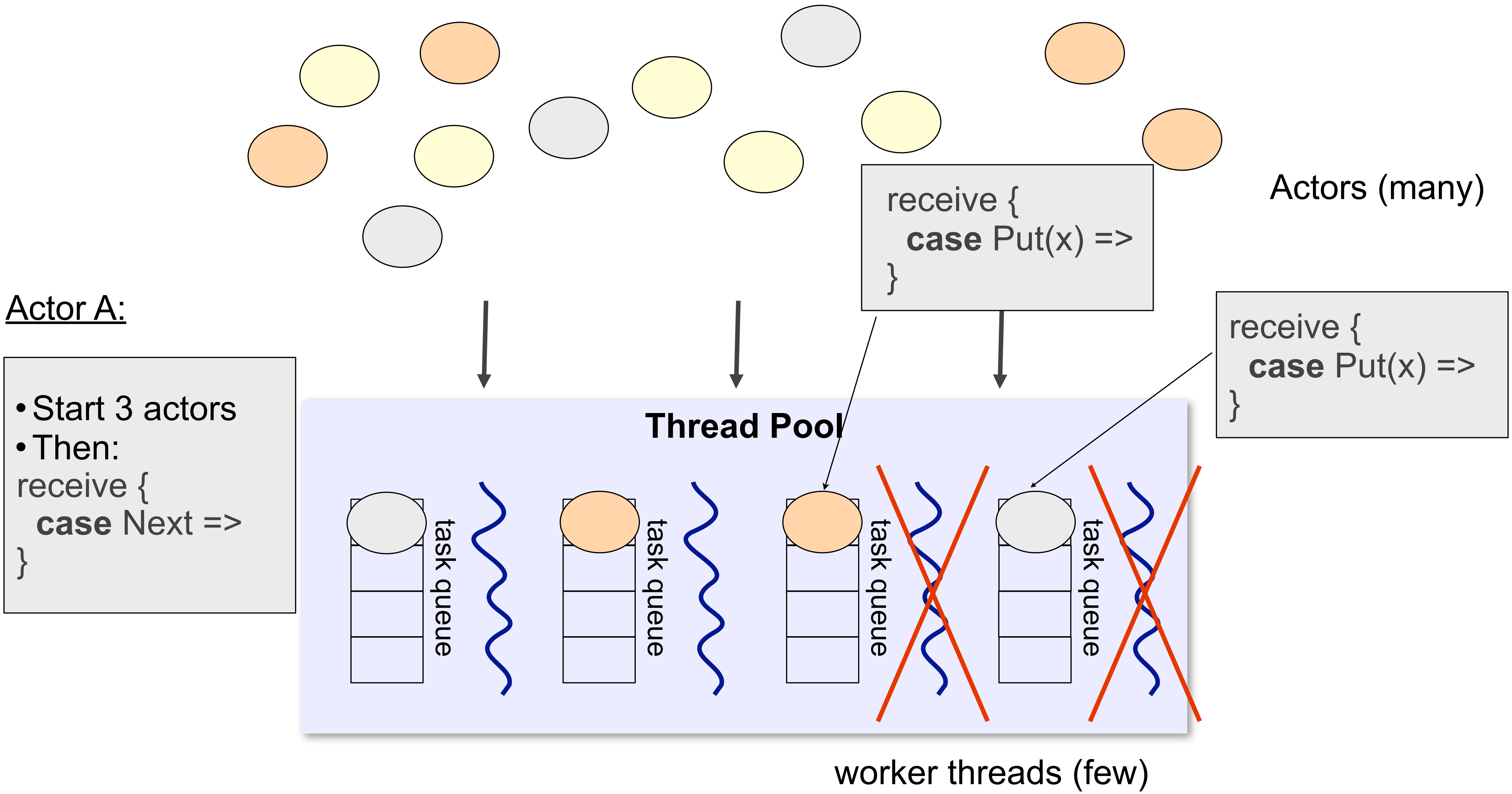
Managing Blocking.



Managing Blocking.



Managing Blocking.

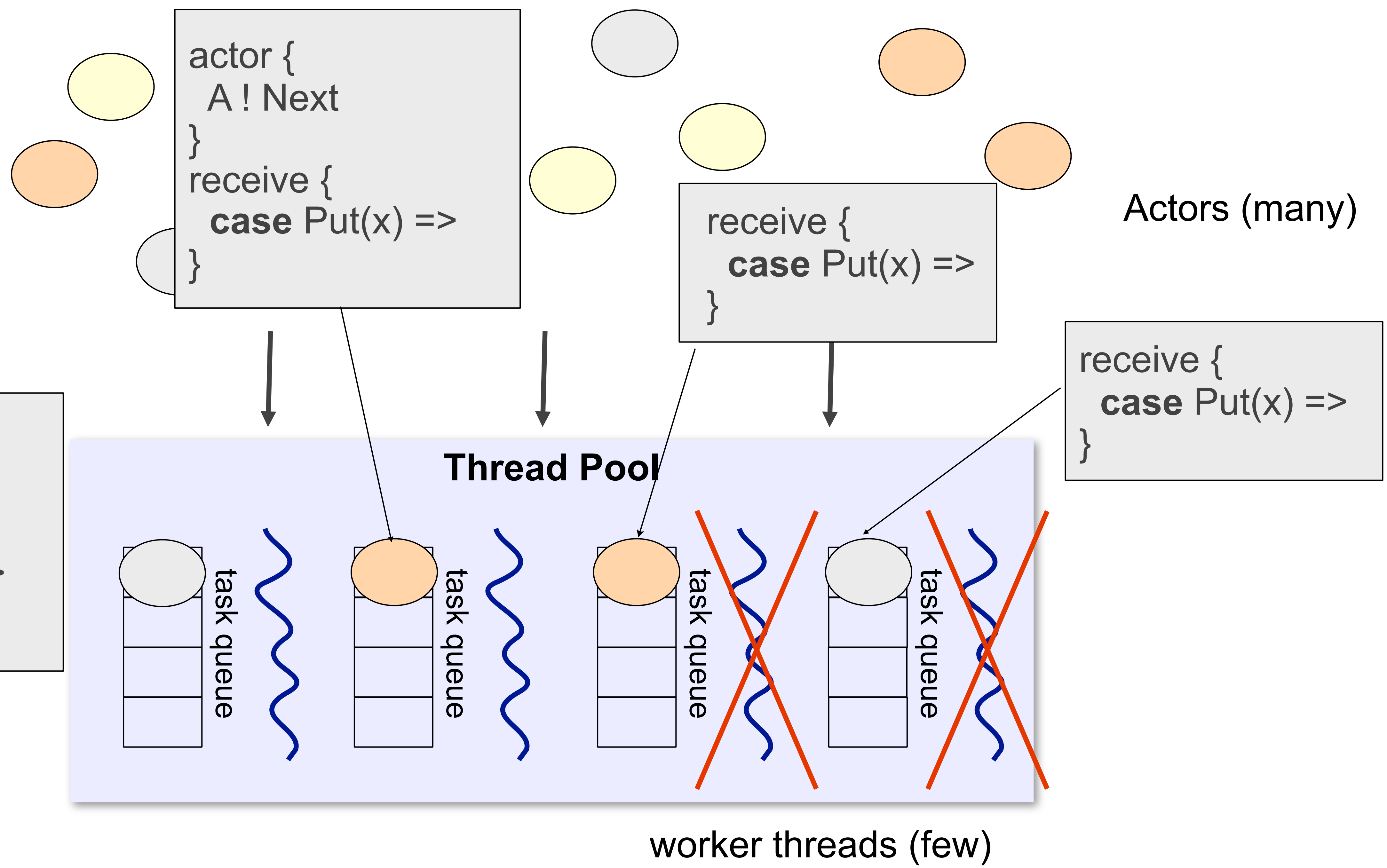


Managing Blocking.

<u>Actor A:</u>

- Start 3 actors
- Then:

```
receive {
  case Next =>
}
```

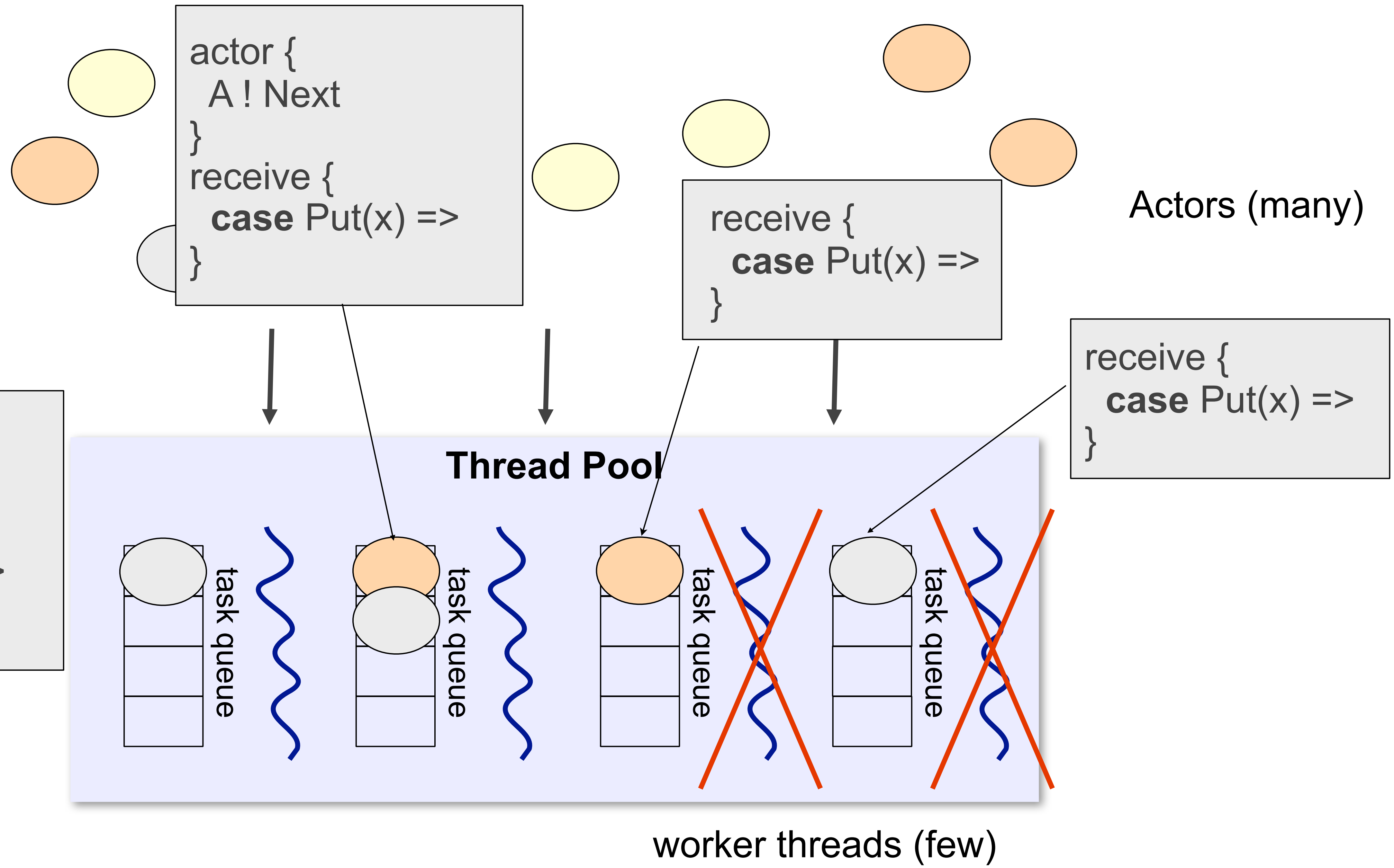


Managing Blocking.

Actor A:

- Start 3 actors
- Then:

receive {
  case Next =>
}

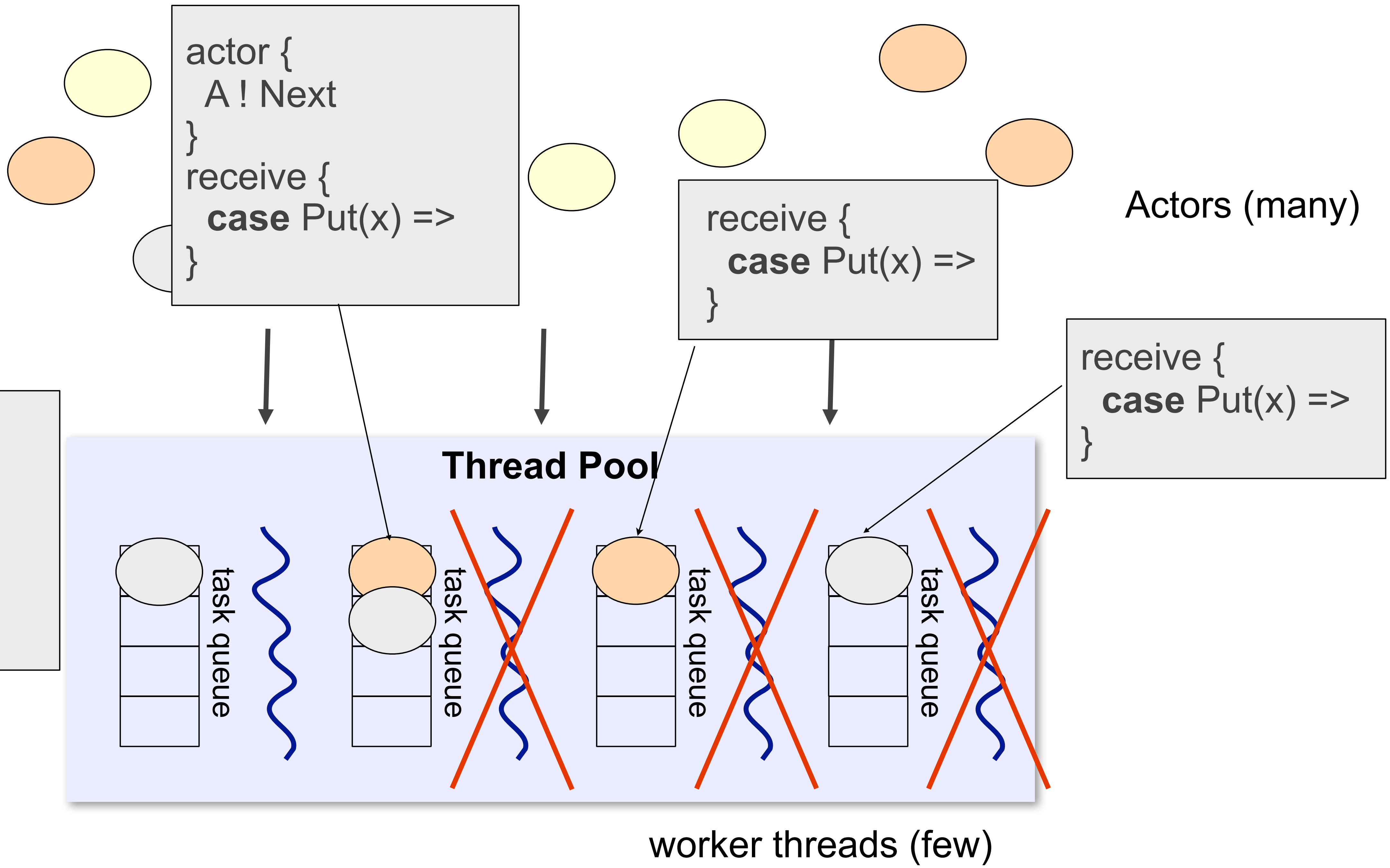


Managing Blocking.

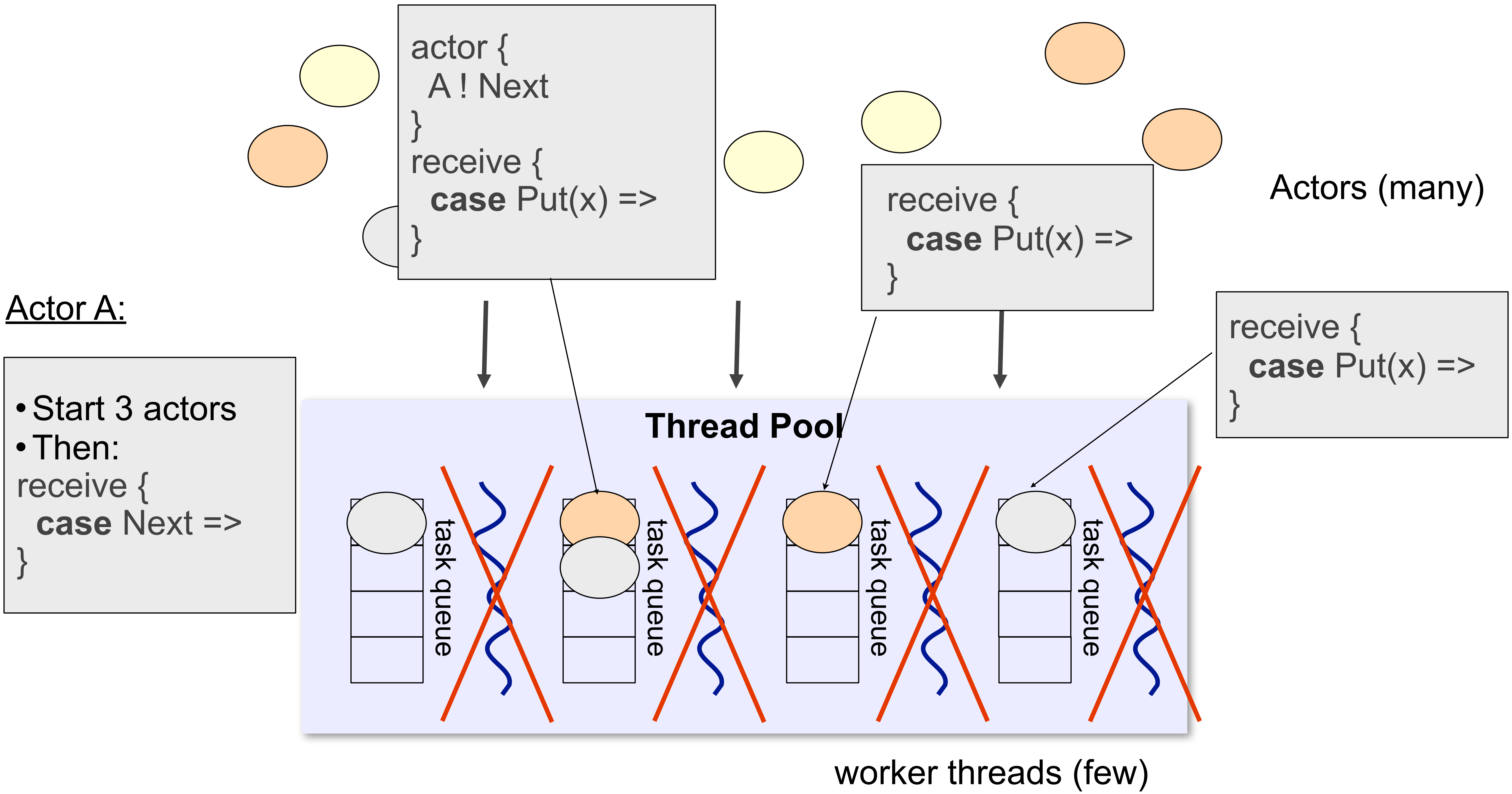
Actor A:

- Start 3 actors
- Then:

```
receive {
  case Next =>
}
```



Managing Blocking.

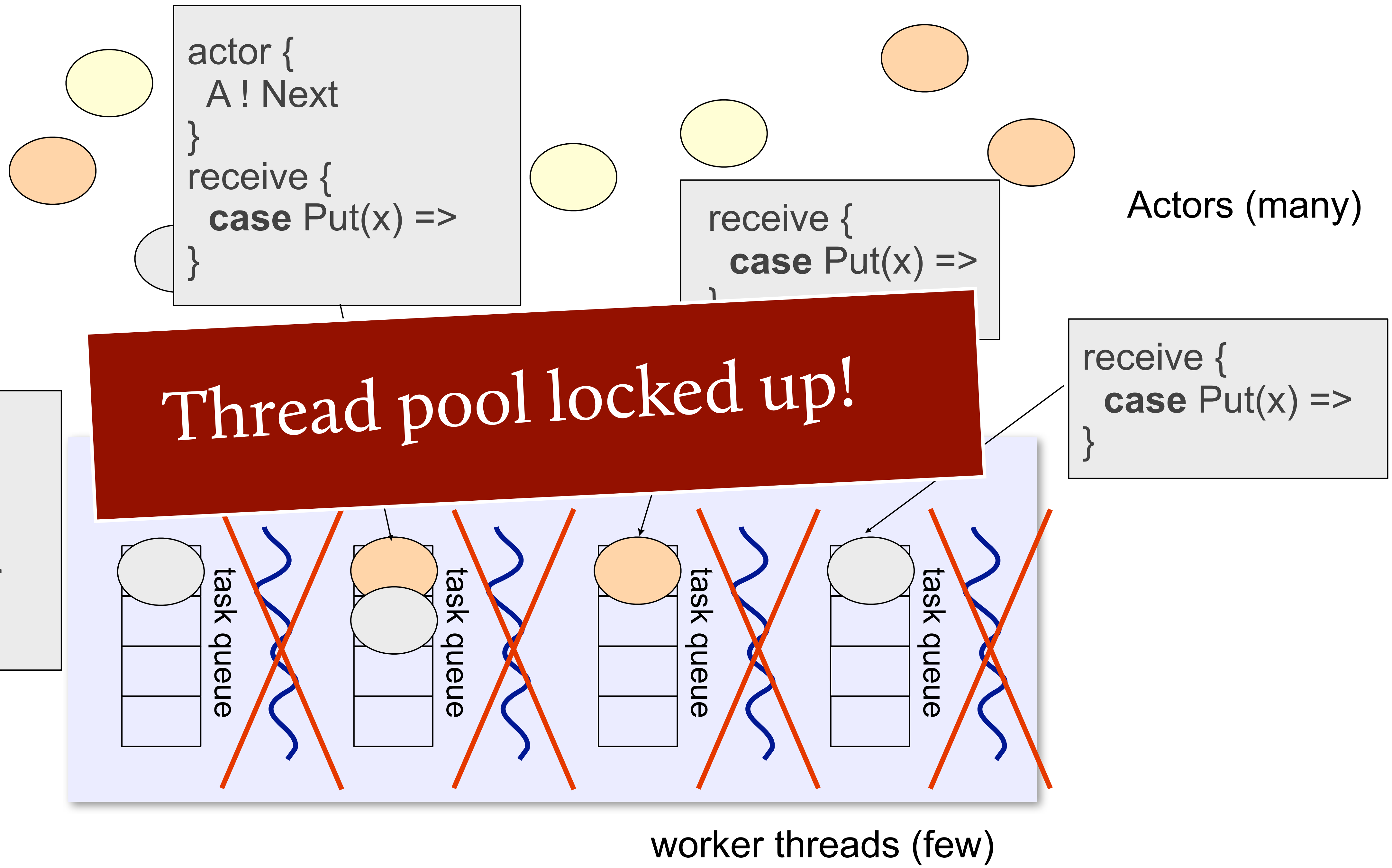


Managing Blocking.

Actor A:

- Start 3 actors
- Then:

```
receive {
  case Next =>
}
```

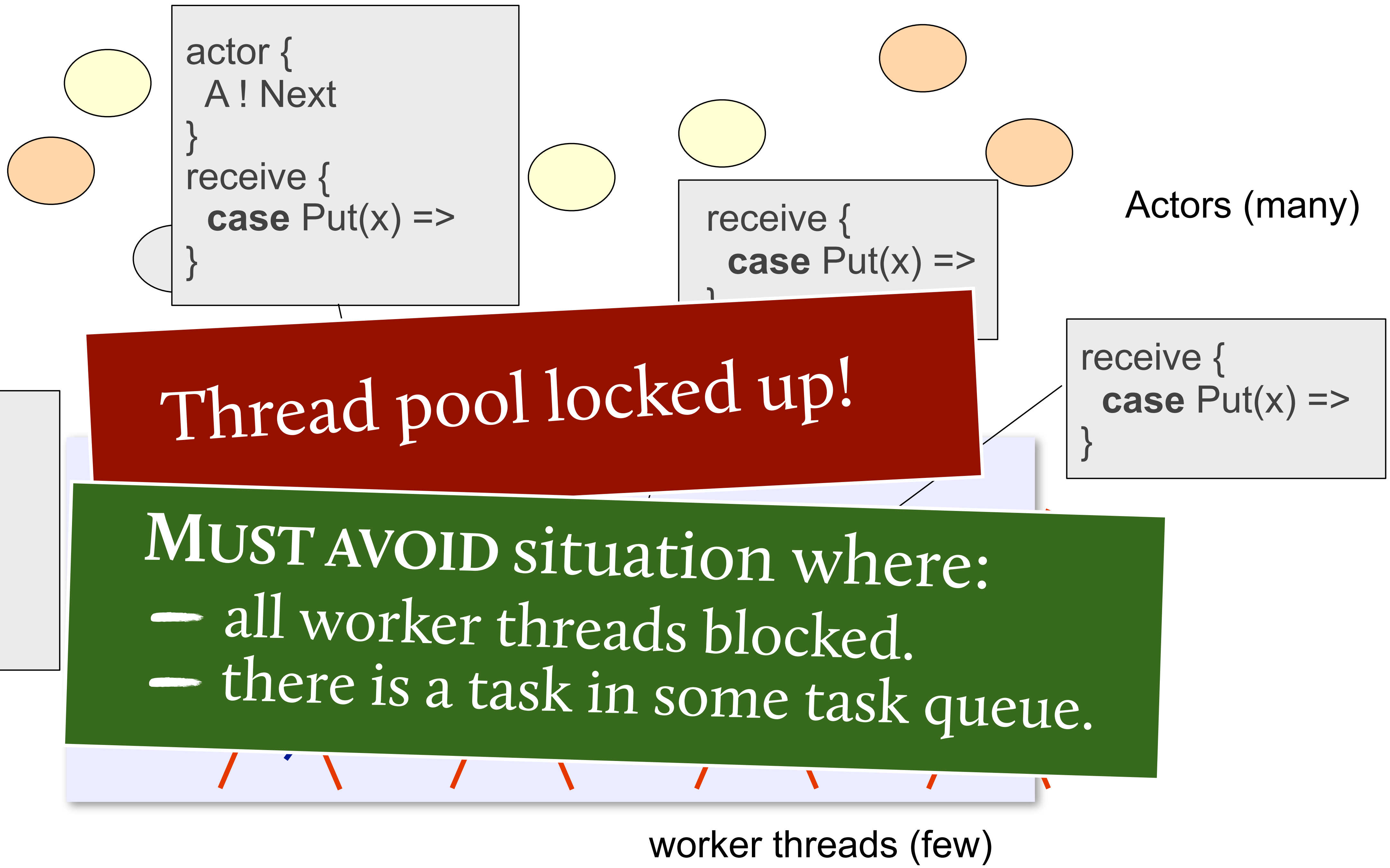


Managing Blocking.

Actor A:

- Start 3 actors
- Then:

```
receive {
  case Next =>
}
```



Under the Hood.

```
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...
  }
  ...
}
```

Under the Hood.

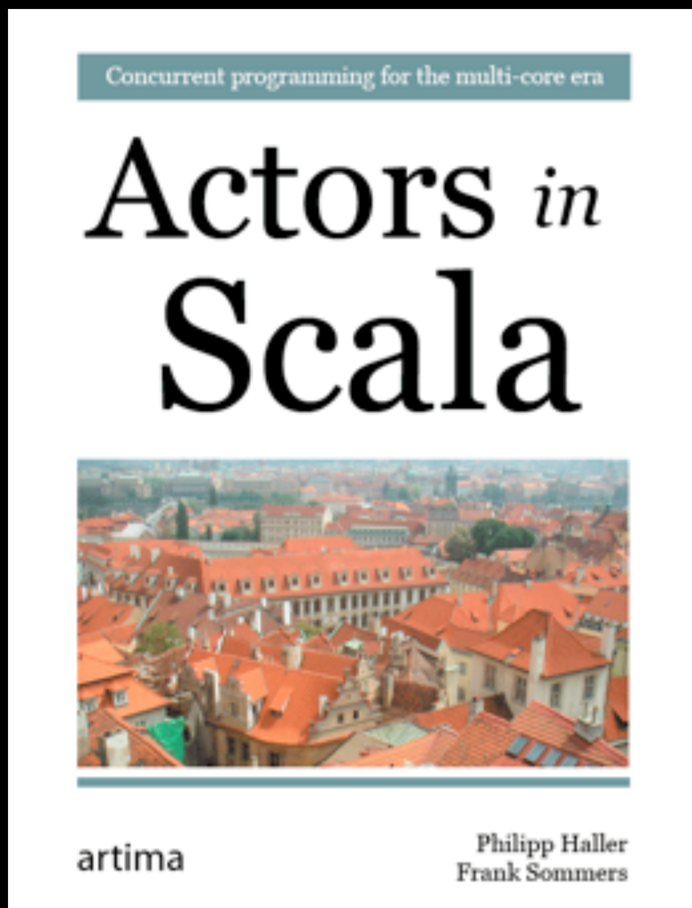
```
def receive[R](f: PartialFunction[Any, R]): R = {
  ...
  val elem = mailbox.extractFirst(msg => f.isDefinedAt(msg))
  if (elem == null) {
    synchronized {
      waitingFor = f
      isSuspended = true
      scheduler.managedBlock(blocker)
    }
  }
  else {
    // process message...
  }
  ...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

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

There is more.

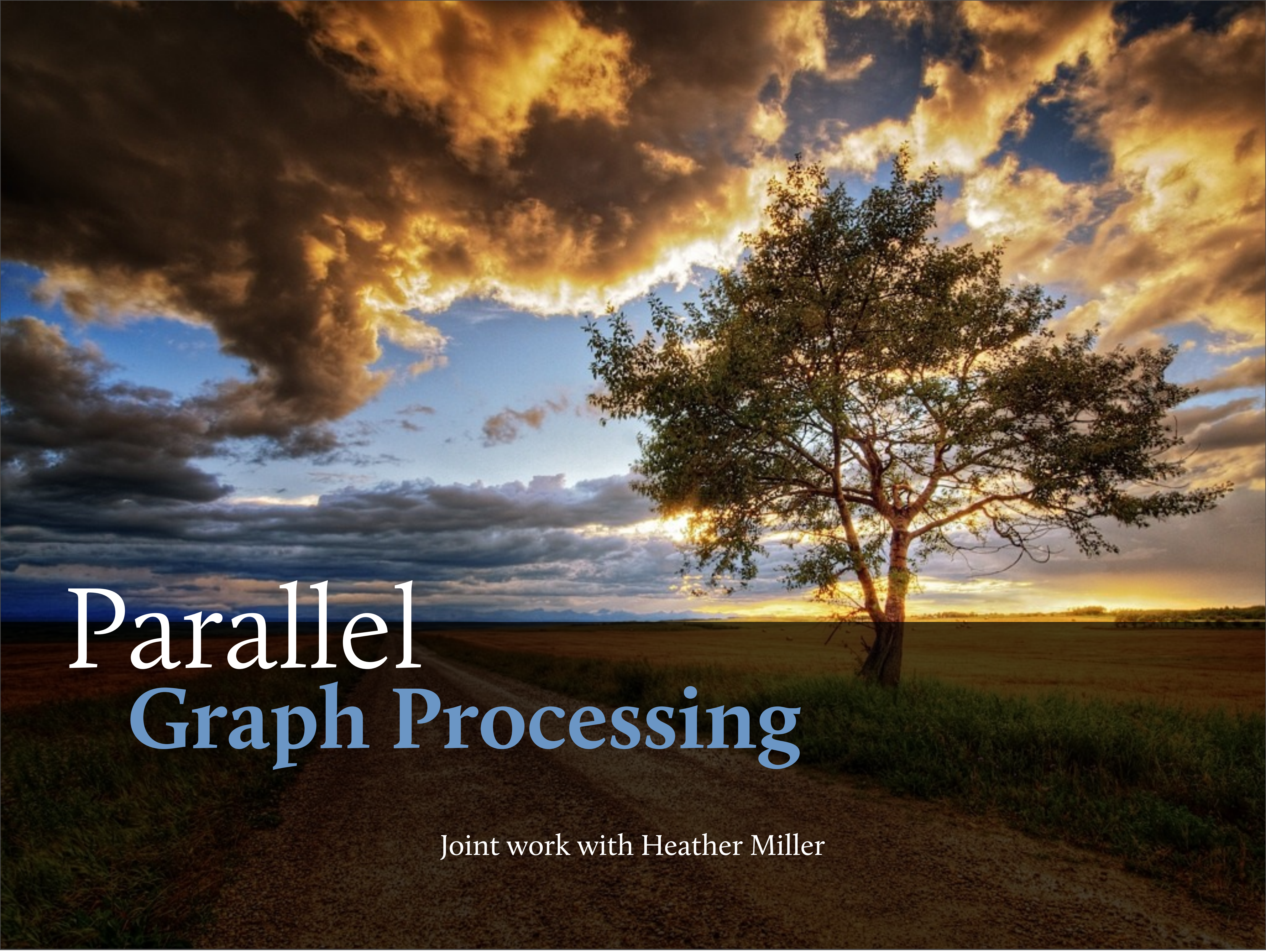
- Continuations
  - Can use them once the continuations plugin is enabled by default (probably in Scala 2.10)
- Akka
  - Part of the Typesafe stack
  - We are working on merging them with scala.actors

The Book.



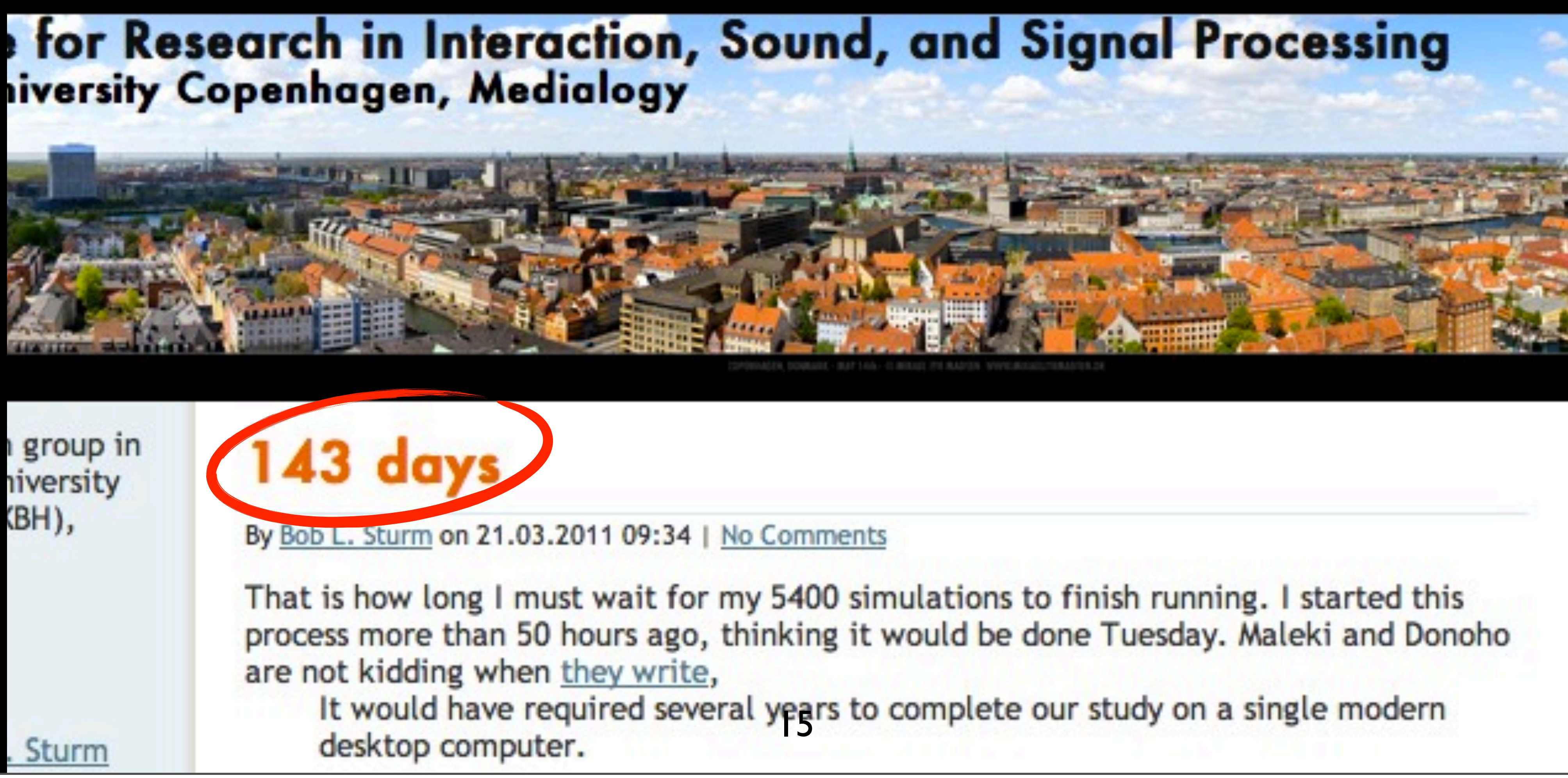
- The definitive guide to actors in the standard library
- Not (only) an API reference
- Language support for actors
- Principles, patterns
- Covers Akka's actors

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.

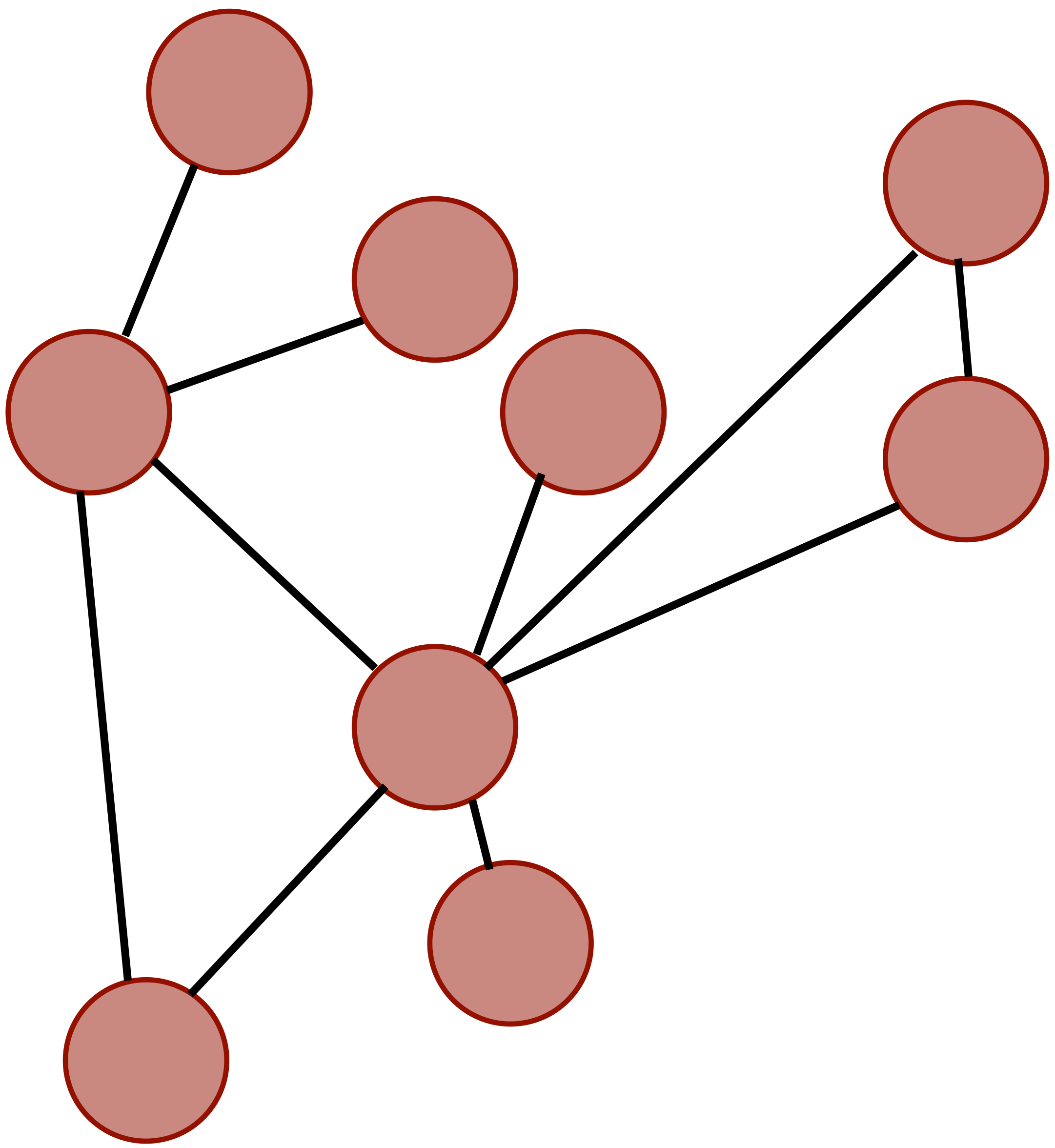
Menthor...

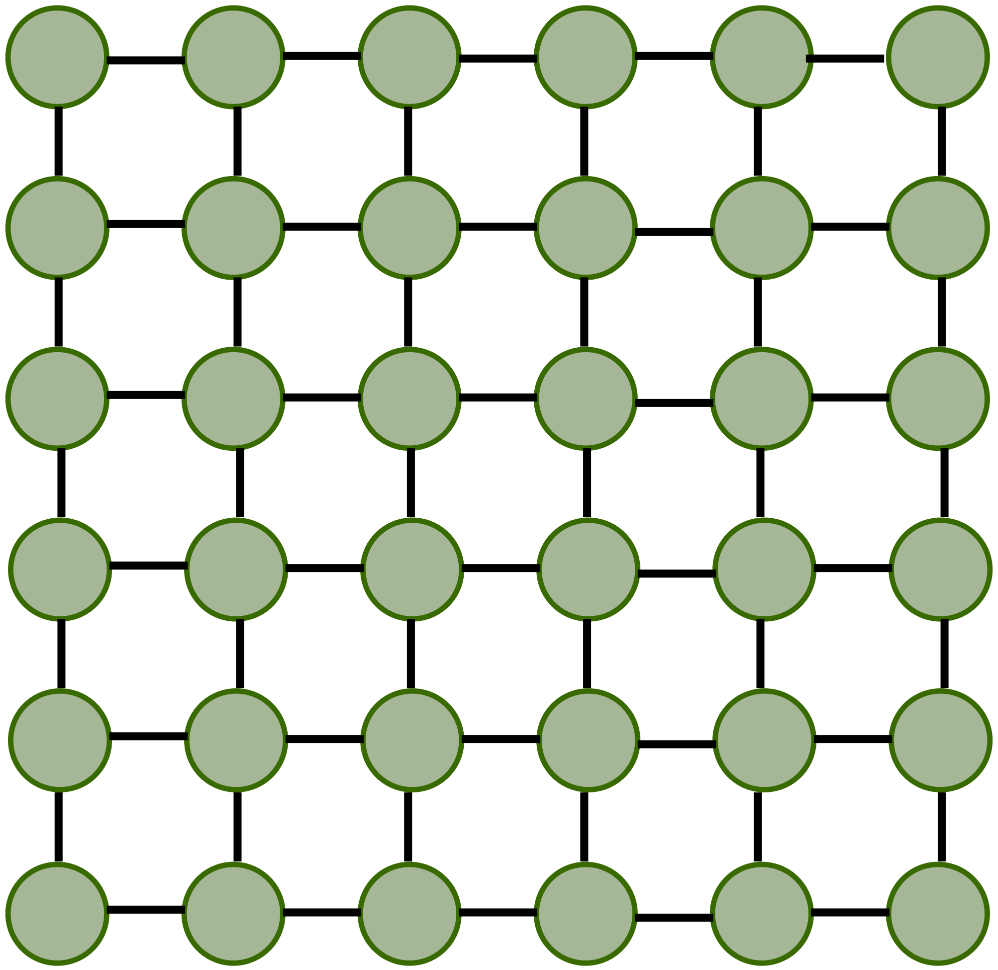
- is a framework for parallel graph processing.
  (But it is not limited to graphs.)
- is inspired by BSP.
  With functional reduction/aggregation mechanisms.
- avoids an inversion of control
  of other BSP-inspired graph-processing frameworks.
- is implemented in Scala,
  and there are preliminary experimental results.

Menthor's

Model of Computation.

Data.

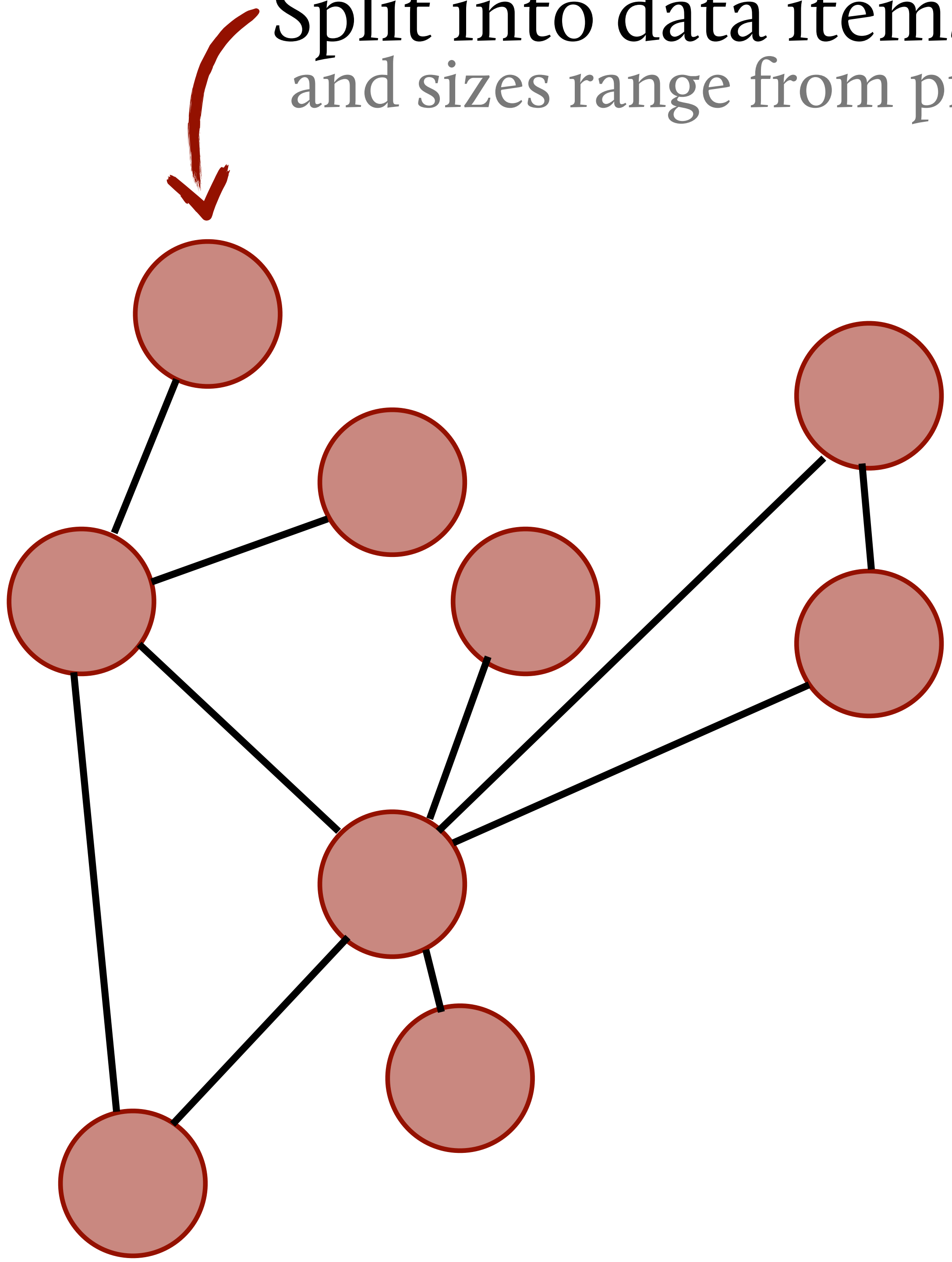


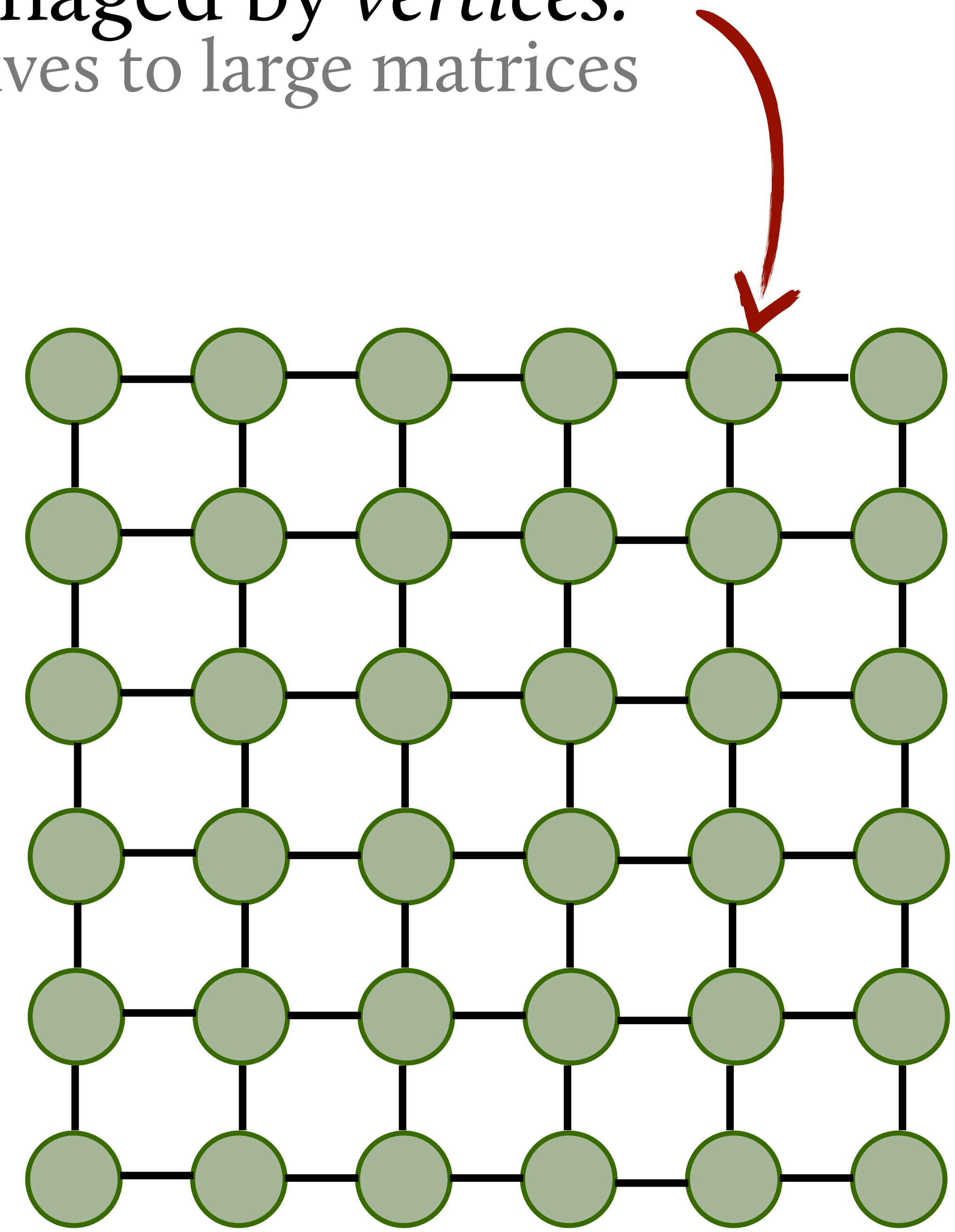


Data.

Split into data items managed by *vertices.*

and sizes range from primitives to large matrices

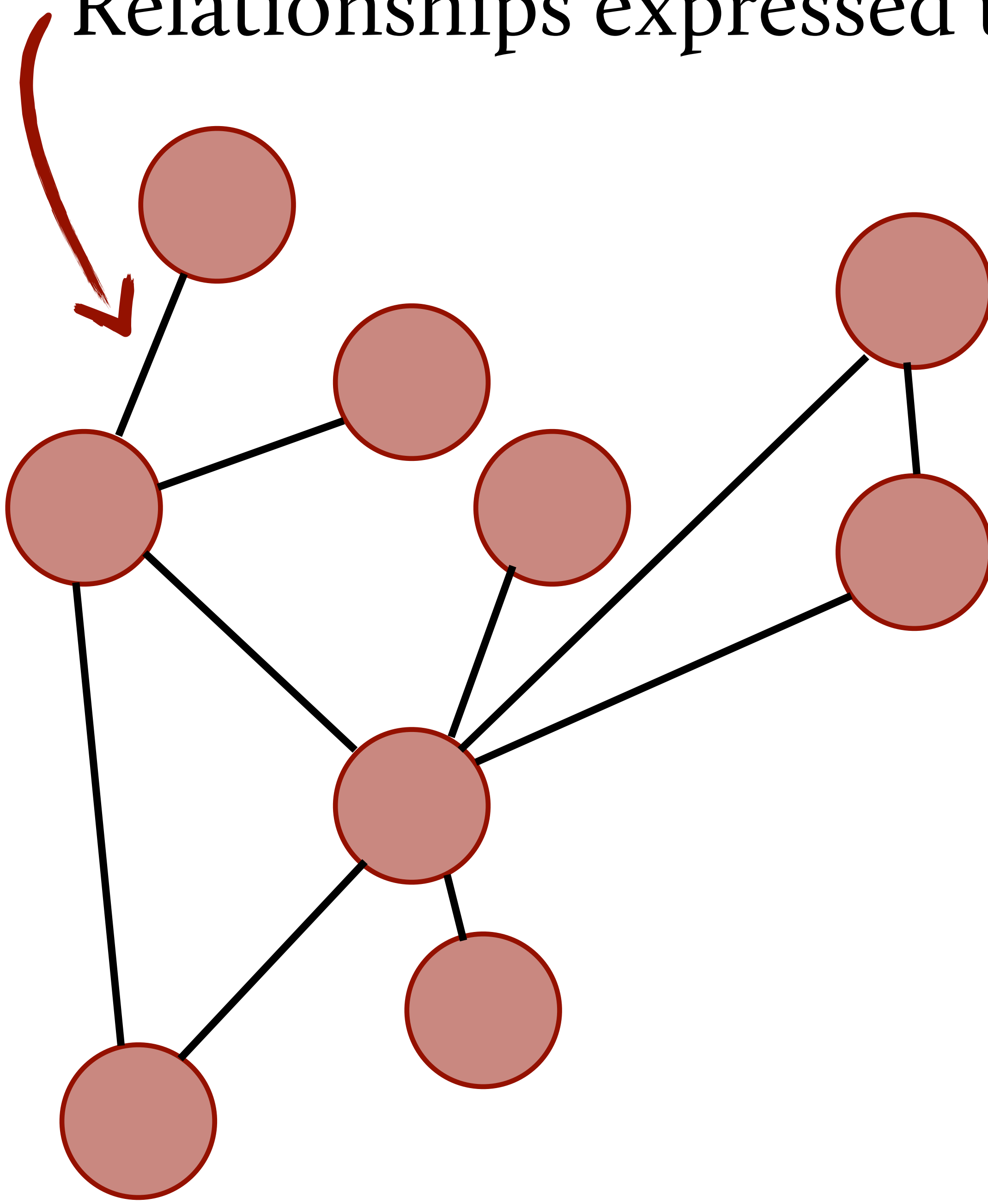


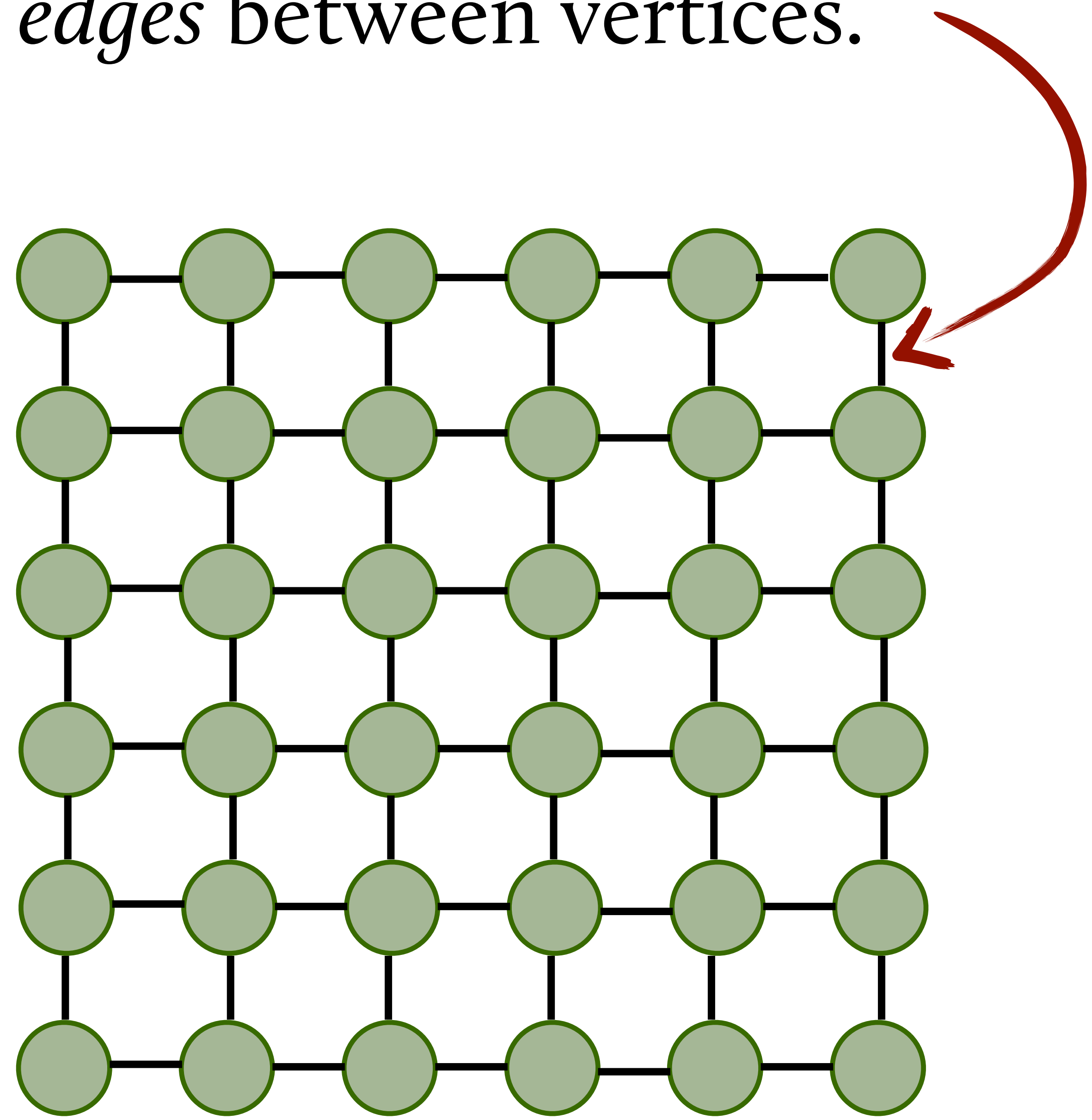


Data.

Split into data items managed by *vertices.*

Relationships expressed using *edges* between vertices.





Algorithms.

Algorithms.

- Data items stored inside of vertices *iterative[l]y* updated.

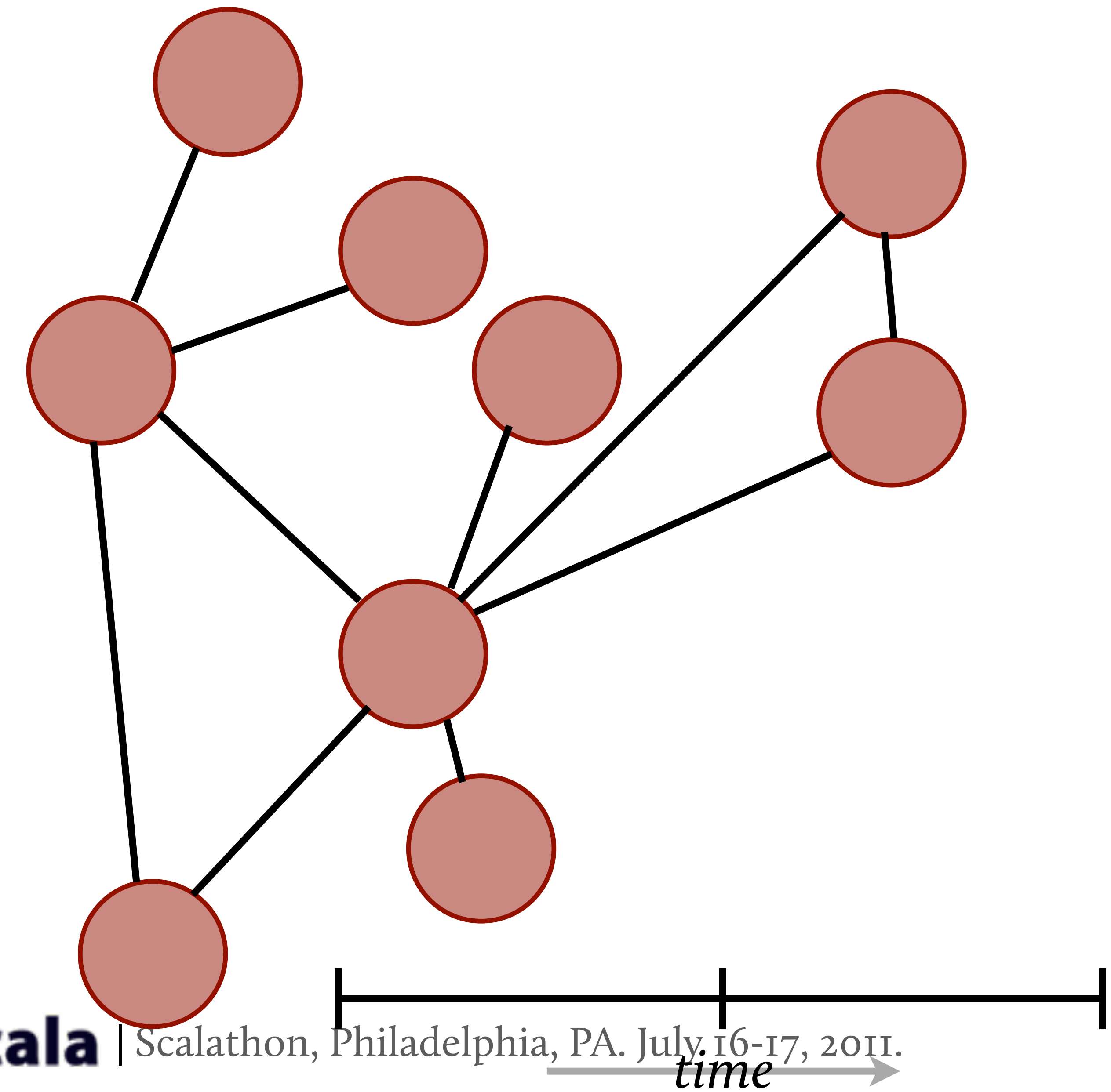
Algorithms.

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

(inspired by the BSP model)

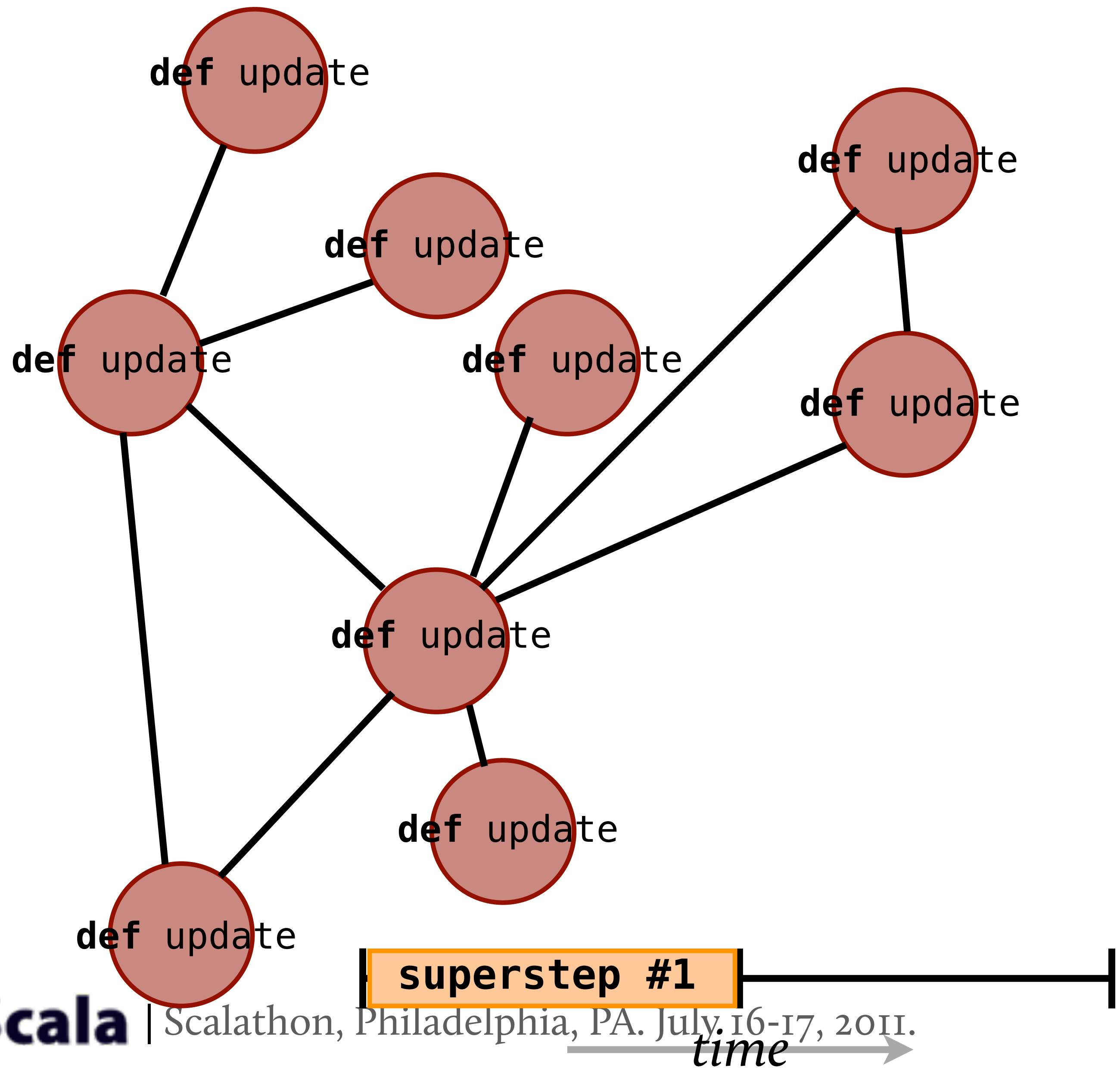
Algorithms.

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



Algorithms.

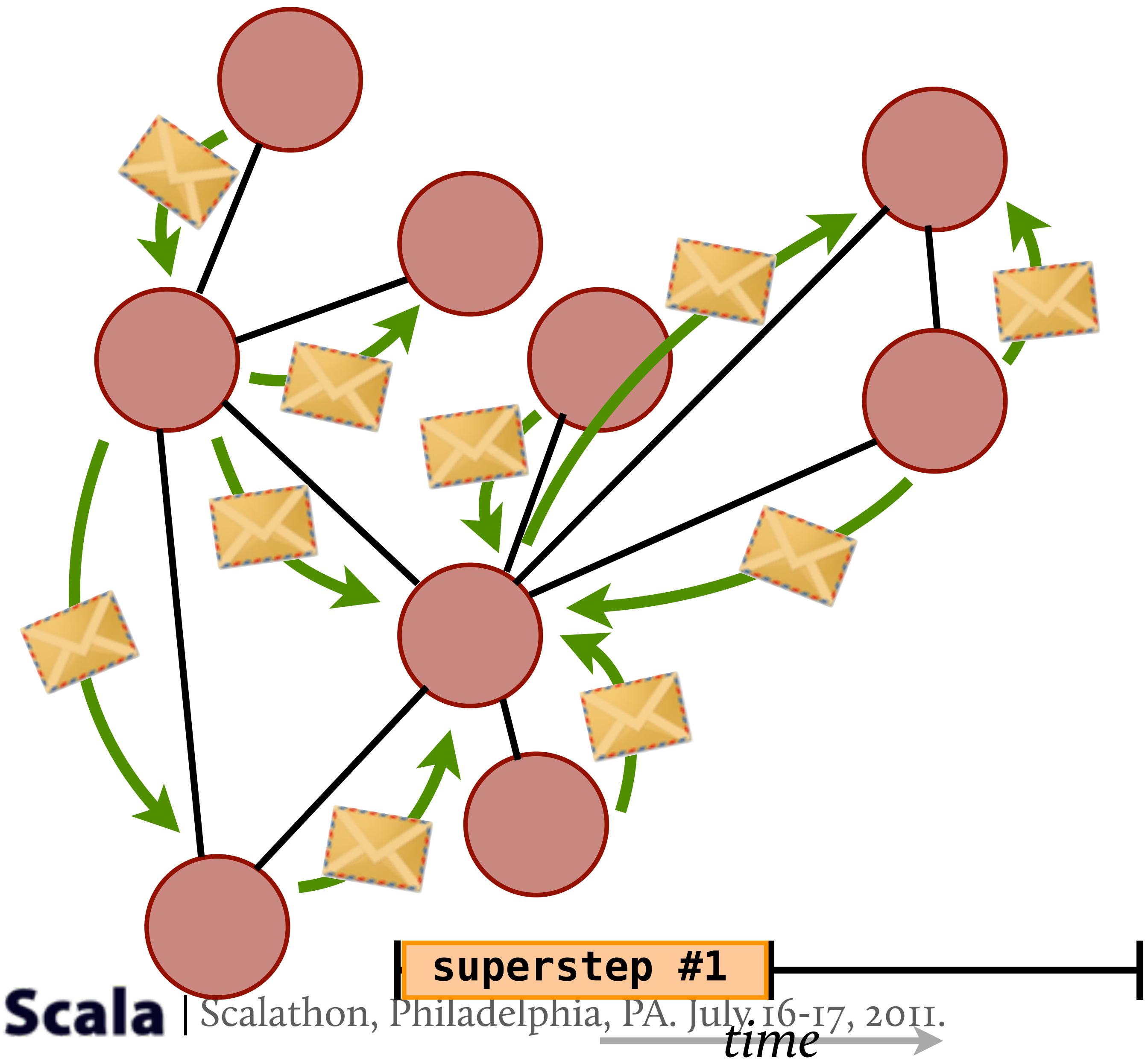
- Data items stored inside of vertices *iteratively* updated.
- Iterations happen as **SYNCHRONIZED SUPERSTEPS.**



1. | update each vertex in *parallel.*

Algorithms.

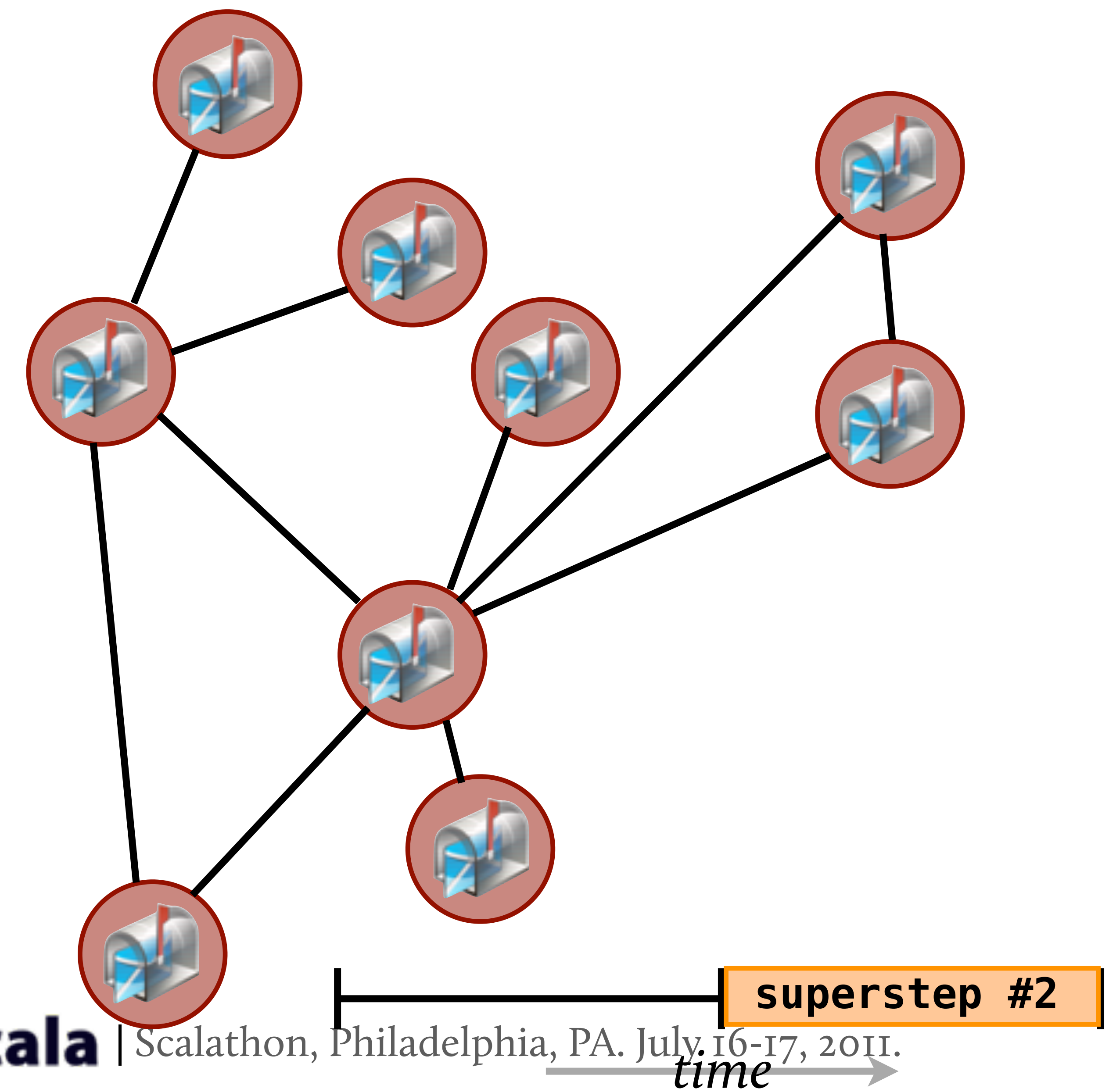
- Data items stored inside of vertices *iteratively* updated.
- Iterations happen as **SYNCHRONIZED SUPERSTEPS.**



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:

```
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:

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

Examples...

```
{
  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:

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

EXAMPLES...

```
{
}
```

```
{
  yield Message(this, nb, value)
}
```

Each is *implicitly* converted to a `Substep[Data]`

PageRank.

```
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.
  - Framework developer-friendly.

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.
  - Framework developer-friendly.
- 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.

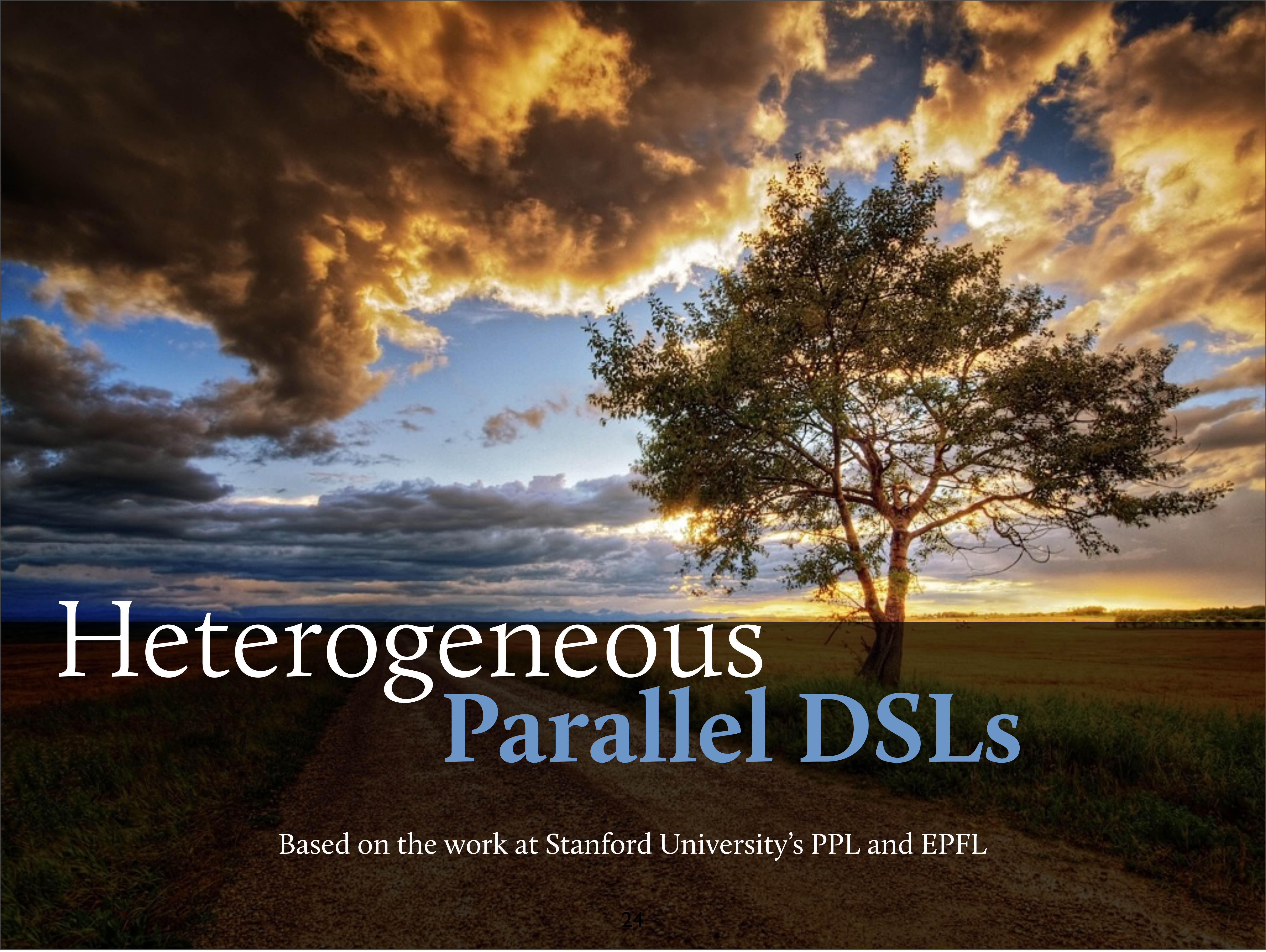
Conclusions

- 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.

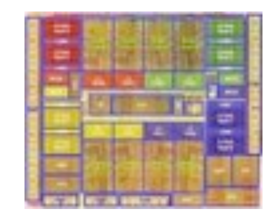
http://lcavwww.epfl.ch/~hmiller/menthor/



Heterogeneous Parallel DSLs

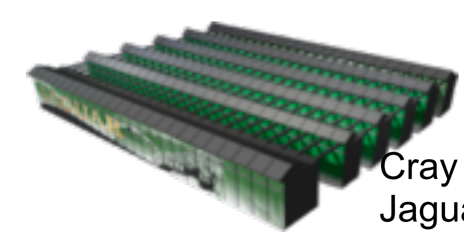
Based on the work at Stanford University's PPL and EPFL

Heterogeneous Parallel Programming



Sun
T2

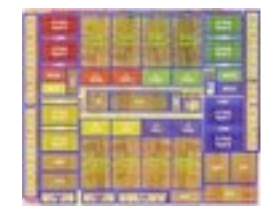
Nvidia
Fermi



Cray
Jaguar

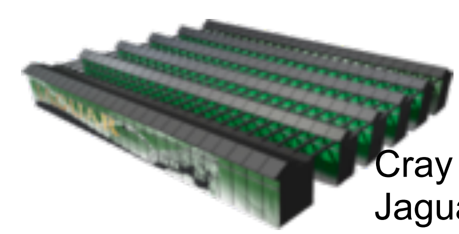
Heterogeneous Parallel Programming

Pthreads
OpenMP



Sun
T2

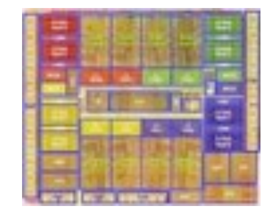
Nvidia
Fermi



Cray
Jaguar

Heterogeneous Parallel Programming

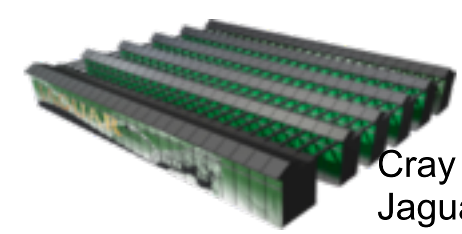
Pthreads
OpenMP



Sun
T2

CUDA
OpenCL

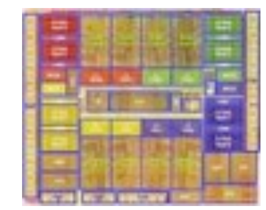
Nvidia
Fermi



Cray
Jaguar

Heterogeneous Parallel Programming

Pthreads
OpenMP

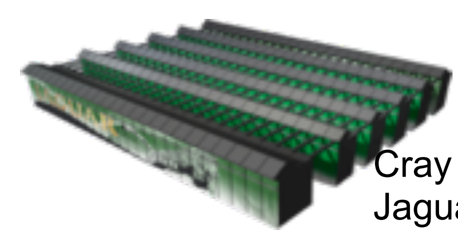


Sun
T2

CUDA
OpenCL

Nvidia
Fermi

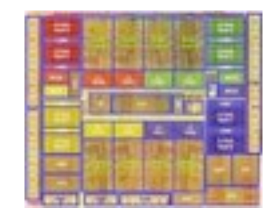
Verilog
VHDL



Cray
Jaguar

Heterogeneous Parallel Programming

Pthreads
OpenMP



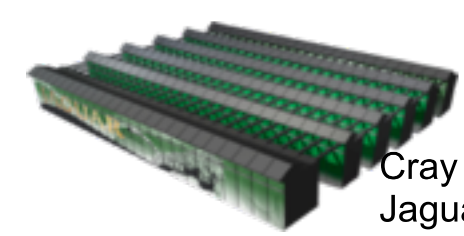
Sun
T2

CUDA
OpenCL

Nvidia
Fermi

Verilog
VHDL

MPI



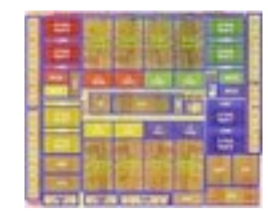
Cray
Jaguar

Heterogeneous Parallel Programming

Applications

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

Pthreads
OpenMP



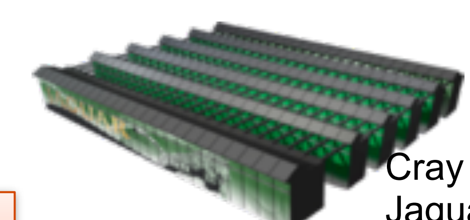
Sun
T2

CUDA
OpenCL

Nvidia
Fermi

Verilog
VHDL

MPI



Cray
Jaguar

Too many different programming models

Hypothesis and New Problem

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

Hypothesis and New Problem

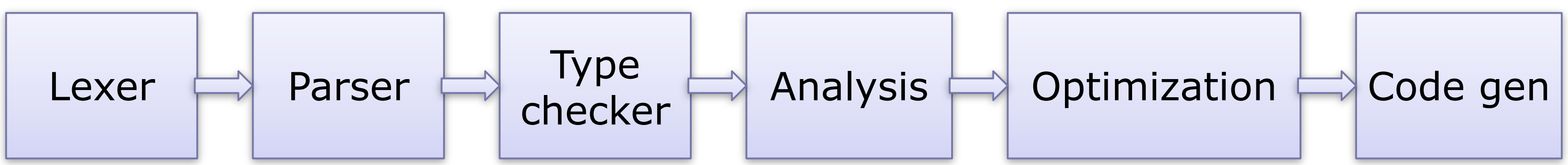
Q: Is it possible to write one program and run it on all these targets?

HYPOTHESIS: Yes, but need domain-specific languages

THOUGH, IT'S QUITE DIFFICULT TO CREATE DSLS USING CURRENT METHODS.

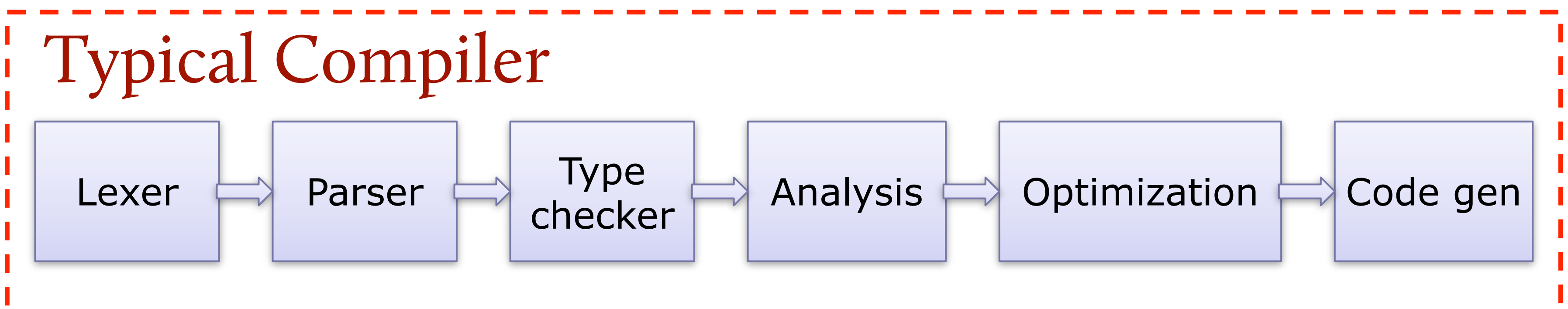
Lightweight Modular Staging.

Typical Compiler



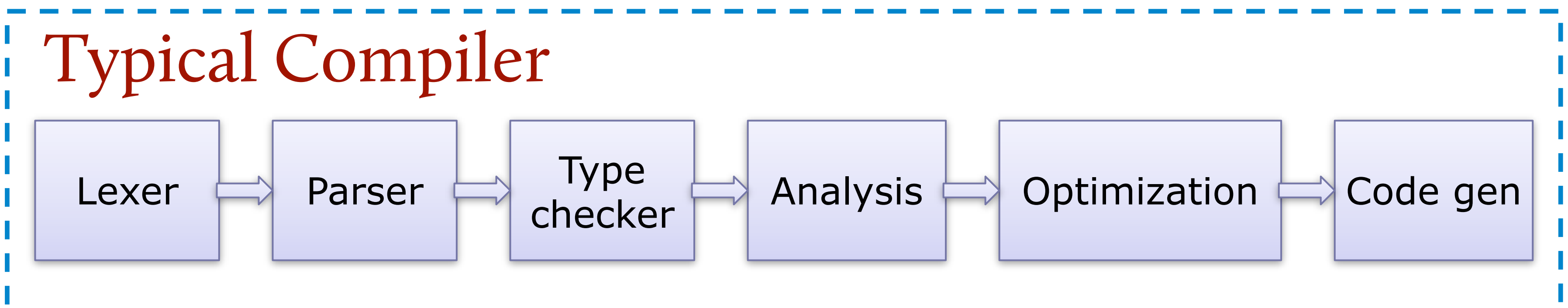
Lightweight Modular Staging.

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



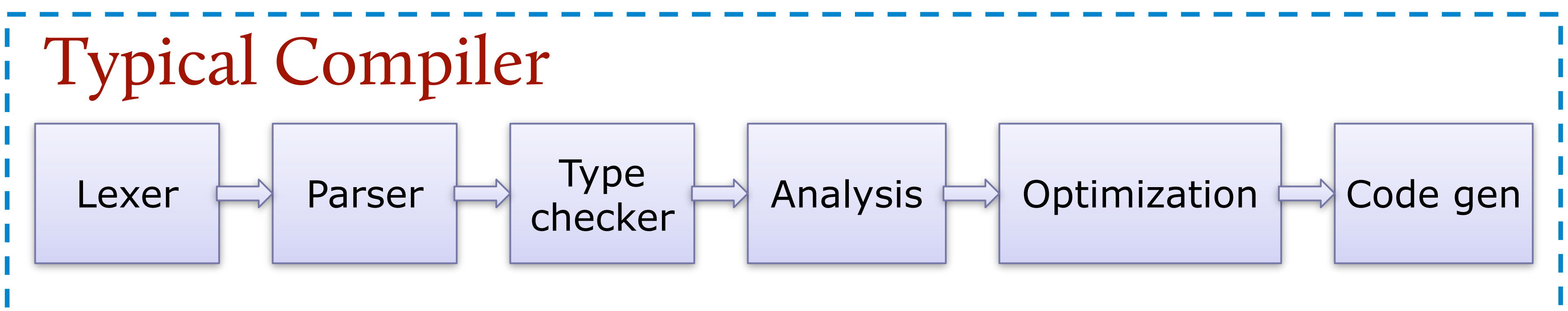
Lightweight Modular Staging.

Stand-alone DSL implements everything

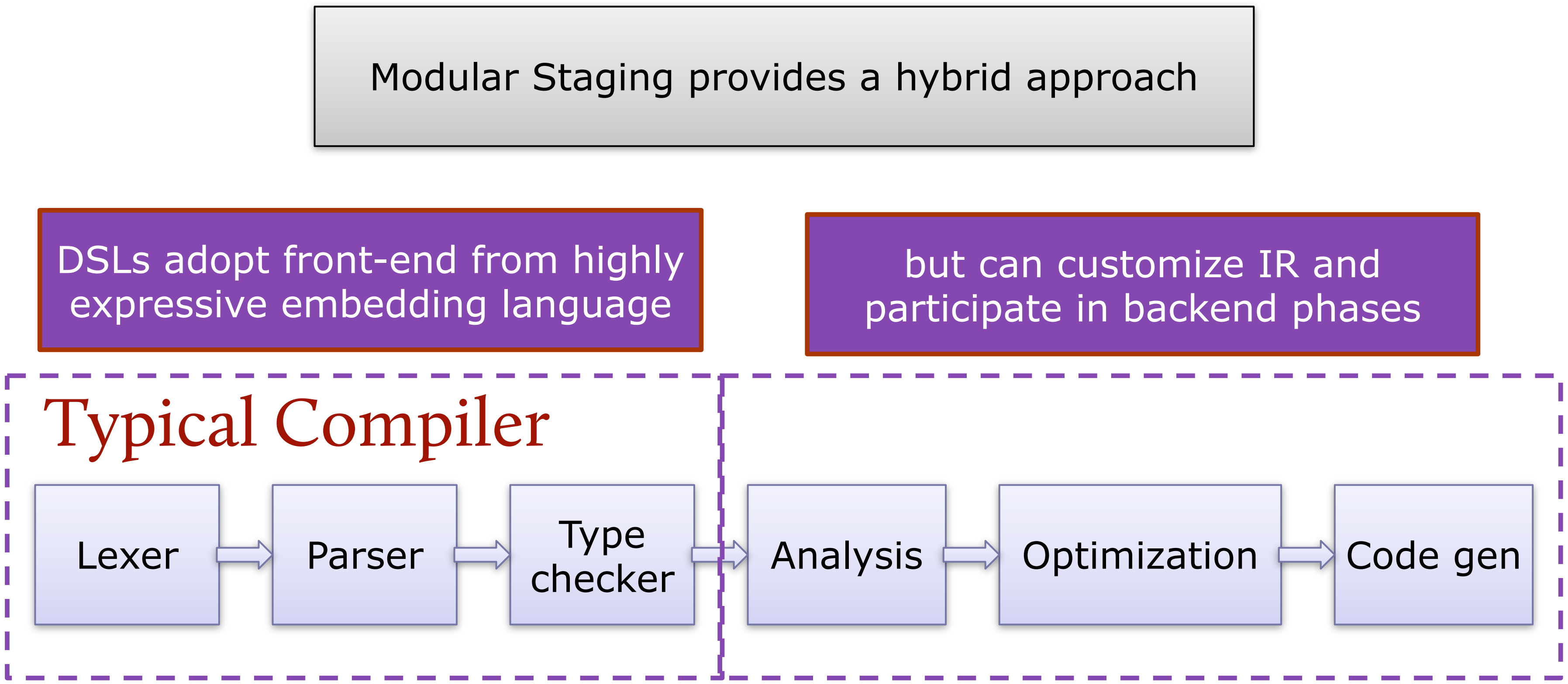


Lightweight Modular Staging.

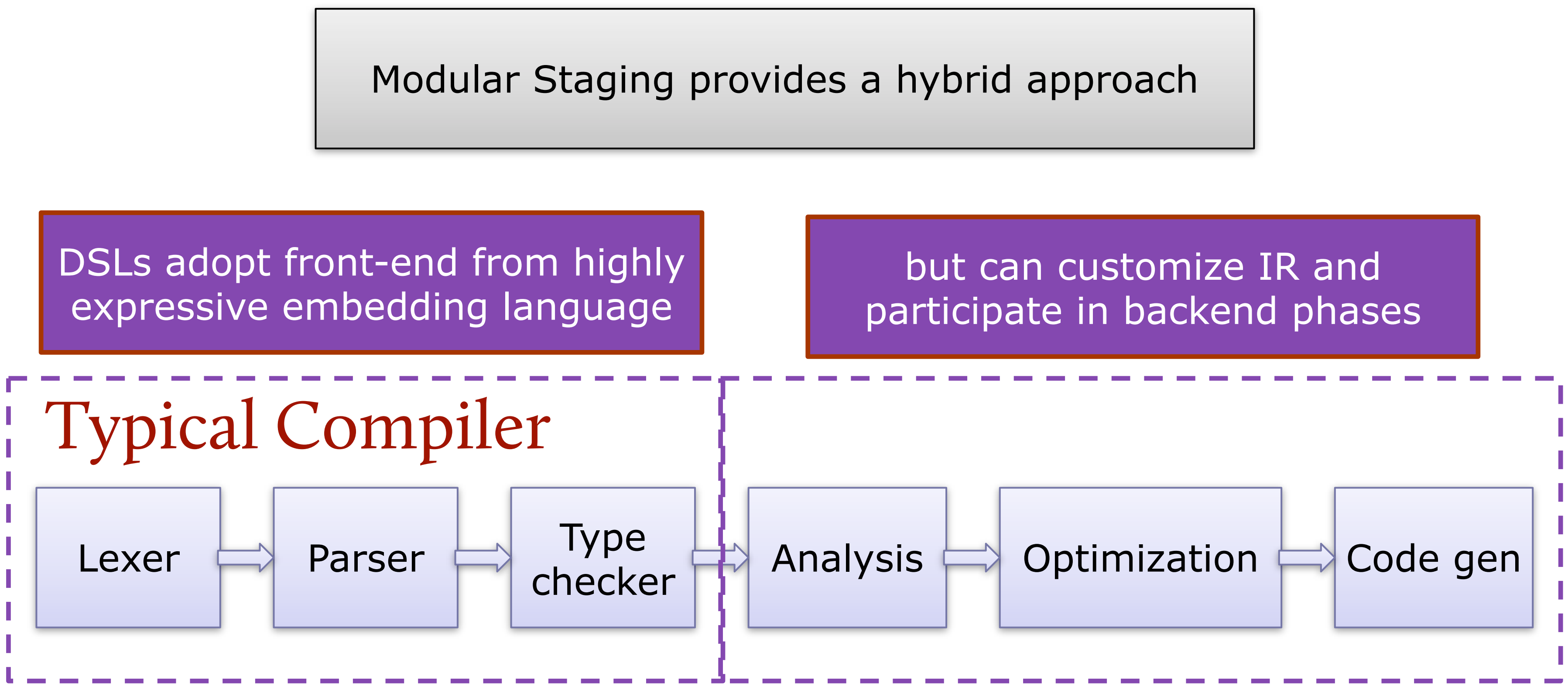
Modular Staging provides a hybrid approach



Lightweight Modular Staging.



Lightweight Modular Staging.



Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs
by Tiark Rompf, Martin Odersky **(GPCE'10)**

Linear Algebra Example.

```
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

```
  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 `Rep[Matrix]`, `Rep[Vector]` etc. allows different implementations for `Rep`

EXAMPLE: expression trees

```
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:

```
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]
```

Staging.

Programming using only `Rep[Matrix]`, `Rep[Vector]` etc. allows different implementations for `Rep`

EXAMPLE: expression trees

```
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:

```
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

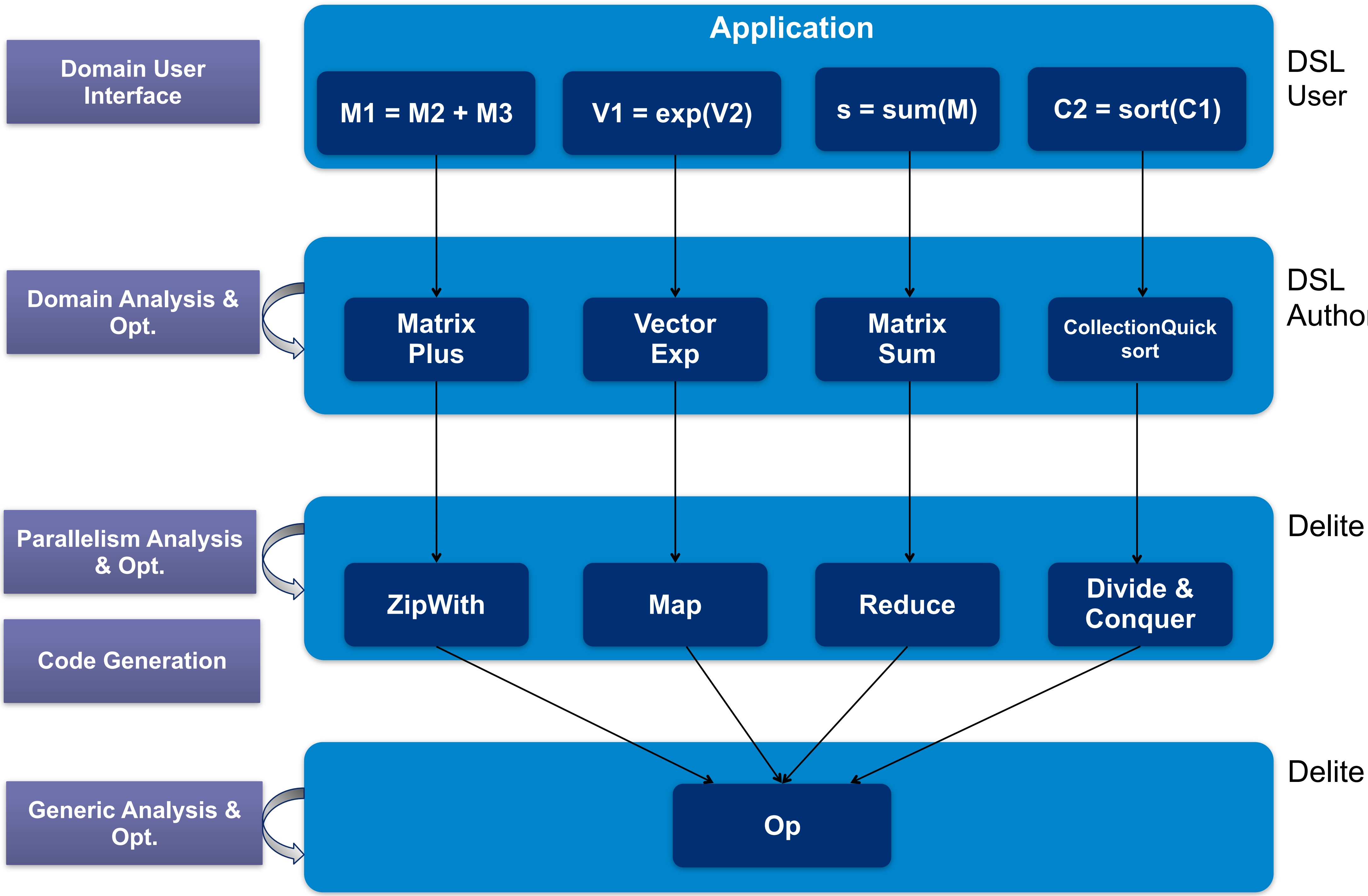
  EXAMPLE: `DeliteOpZip[T]`

- Parallel optimization of IR graph
- Compiler framework with support for heterogeneous hardware platforms
- DSL extends parallel operations

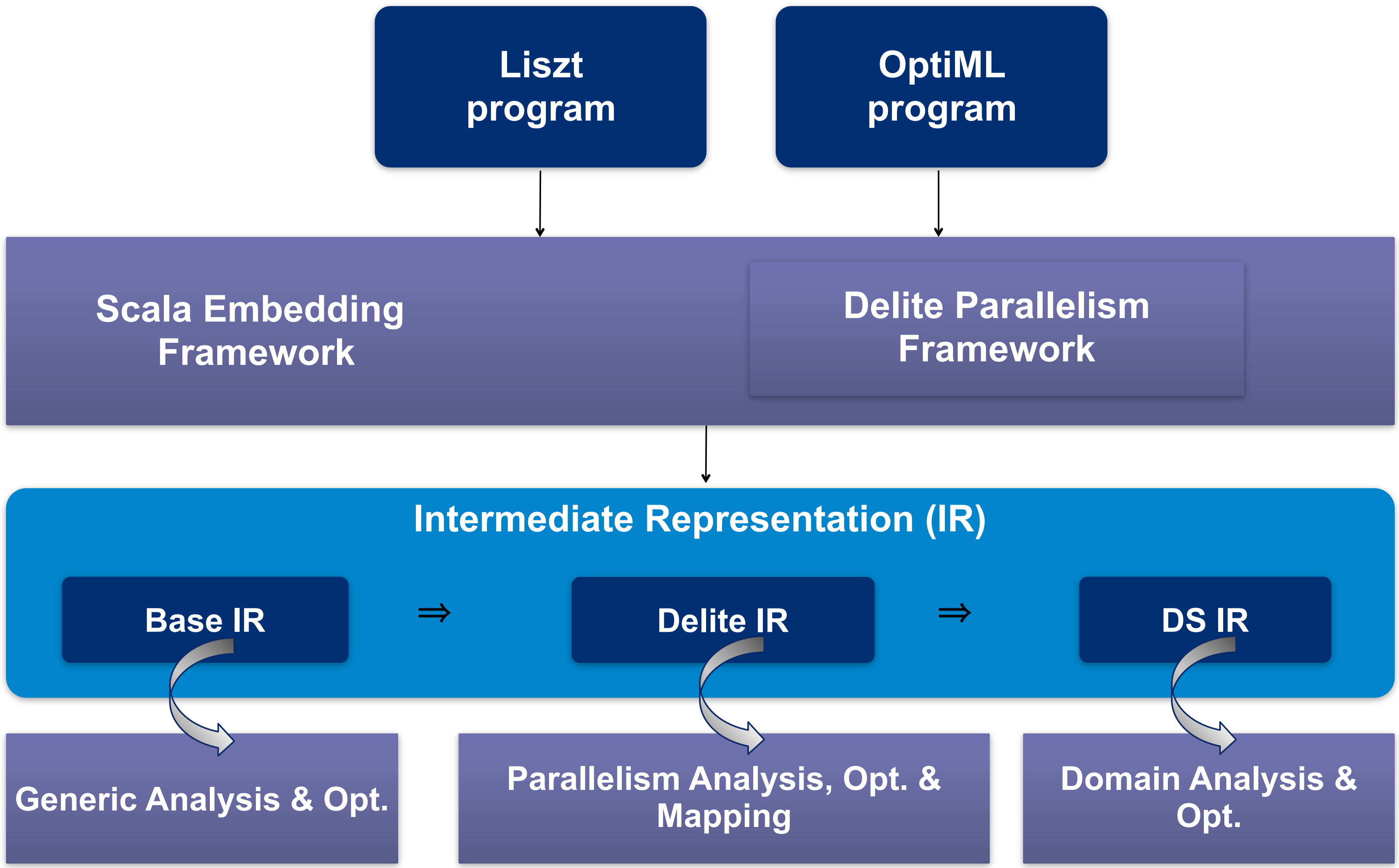
  EXAMPLE: `class Plus extends DeliteOpZip[Matrix]`

- Domain-specific analysis and optimization

The Delite IR Hierarchy



Delite DSL Compilers.



Contributing to Delite

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

- **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, ...
- http://stanford-ppl.github.com/Delite/

Parallel Collections

Based on the work by Aleksandar Prokopec, Tiark Rompf, and Martin Odersky

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

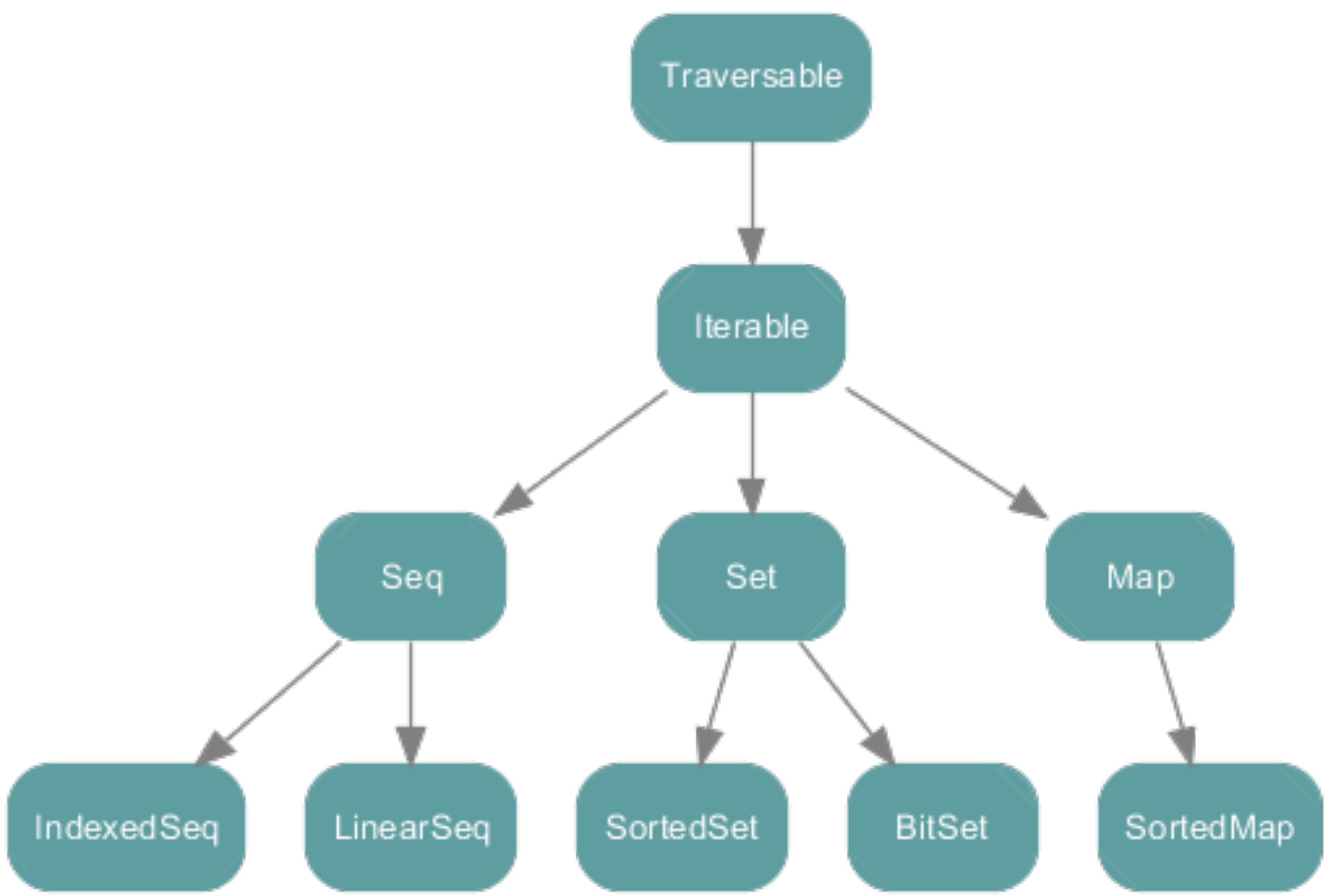
Methods that transform an immutable collection **return a new collection** and leave the old collection unchanged

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:

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

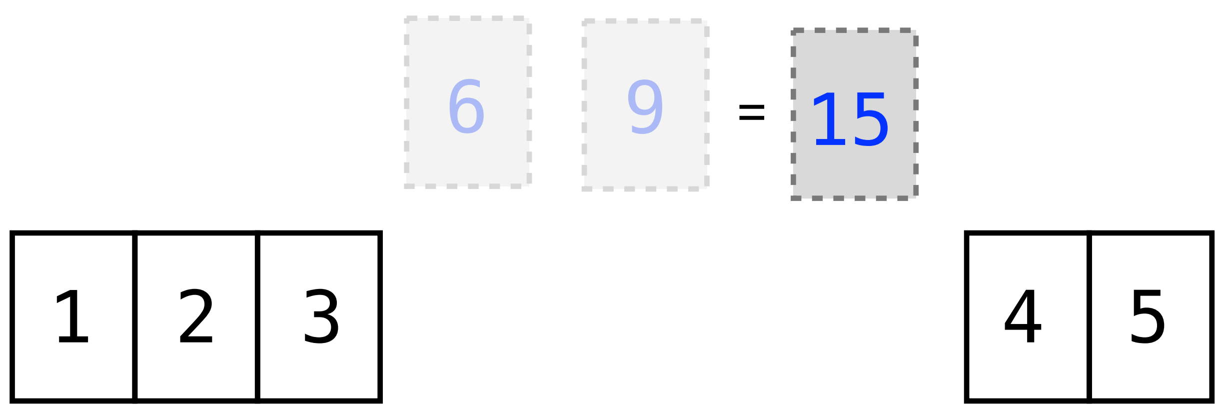
Parallel Collections.

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:

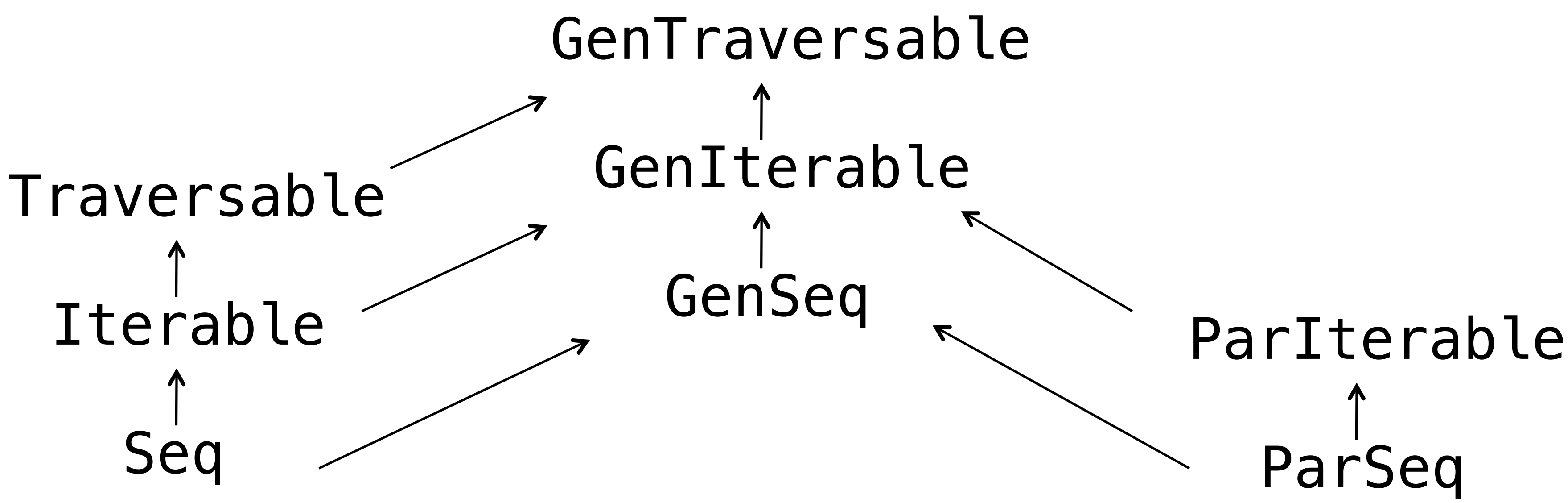
```
myCollection.par.foldLeft(0)((a,b) => a+b)
```



`.par`

- **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.

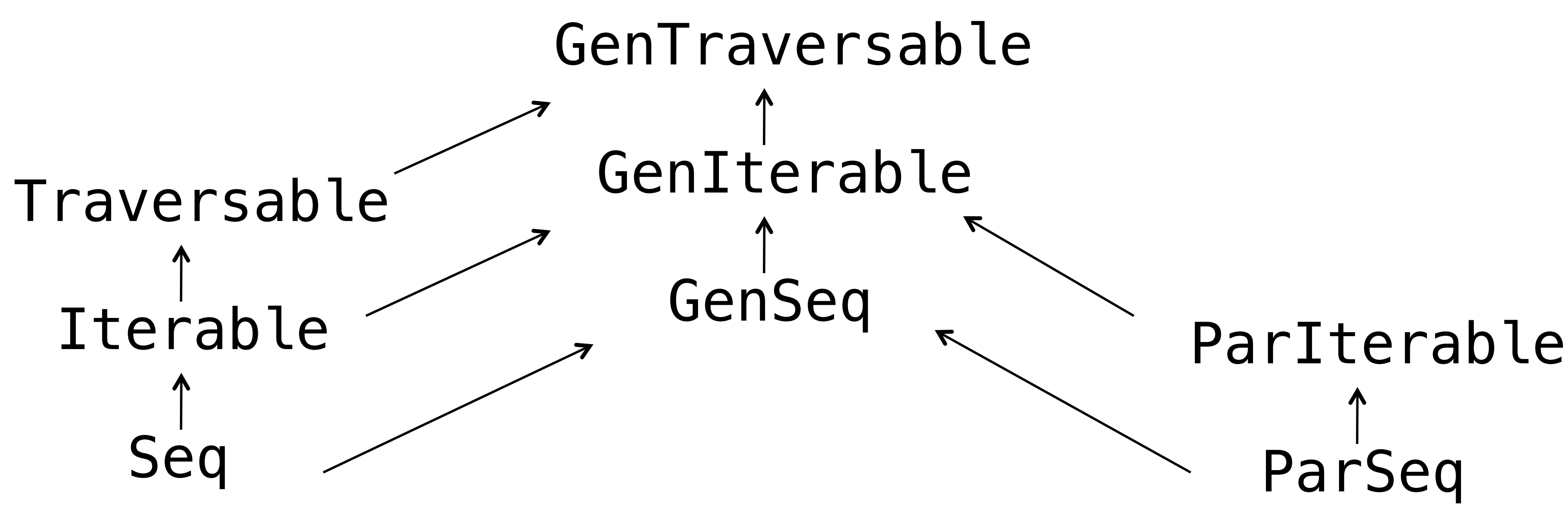


Sunday, July 17, 2011

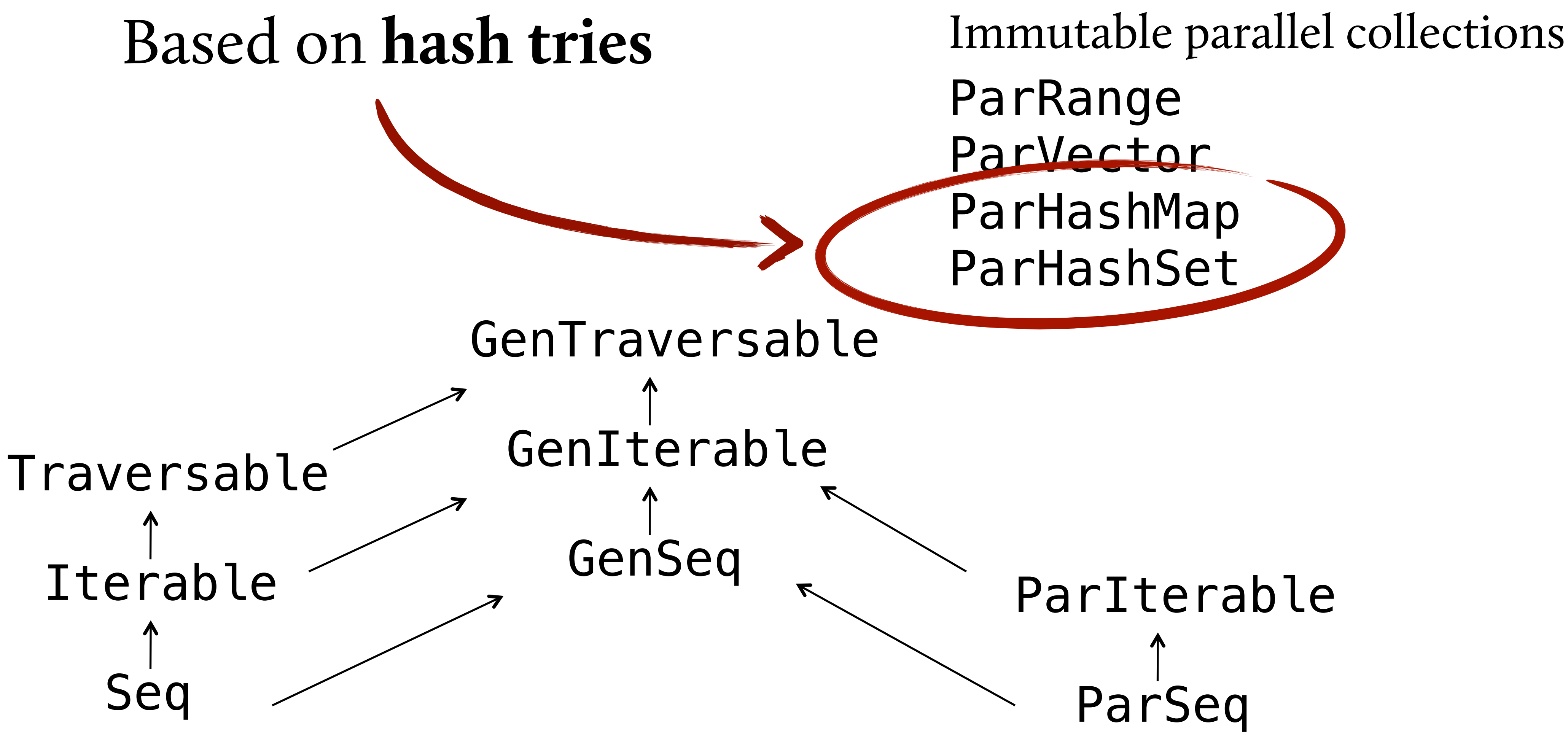
The Collections Hierarchy.

Immutable parallel collections:

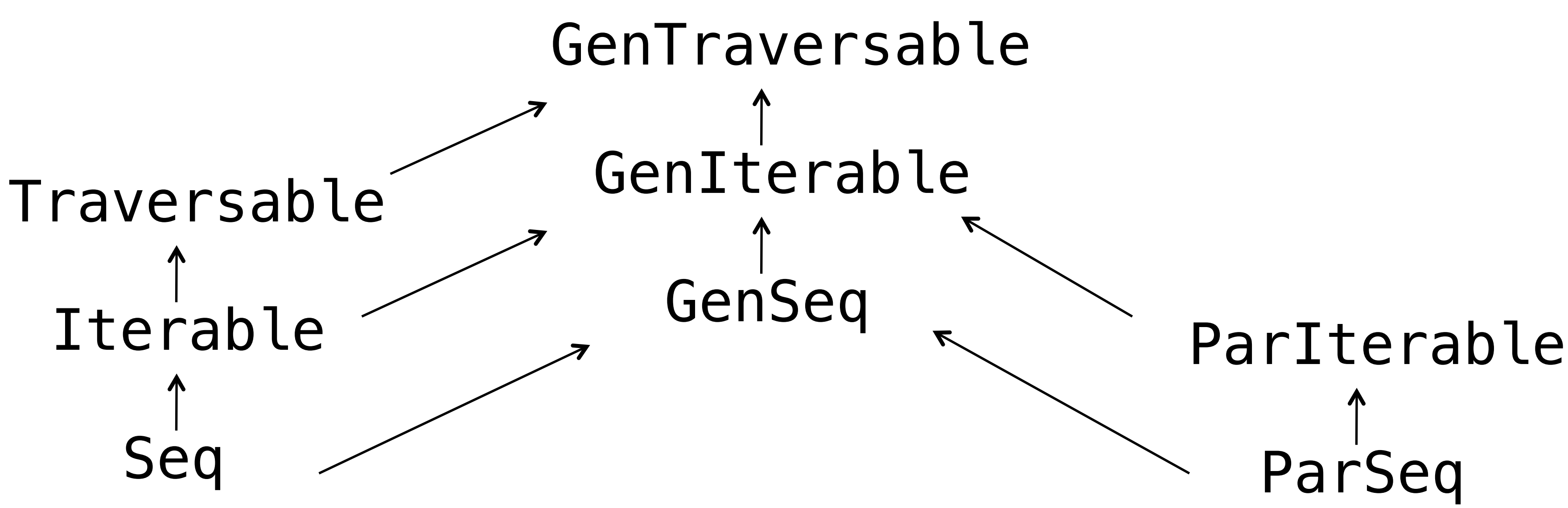
`ParRange`
`ParVector`
`ParHashMap`
`ParHashSet`



The Collections Hierarchy.



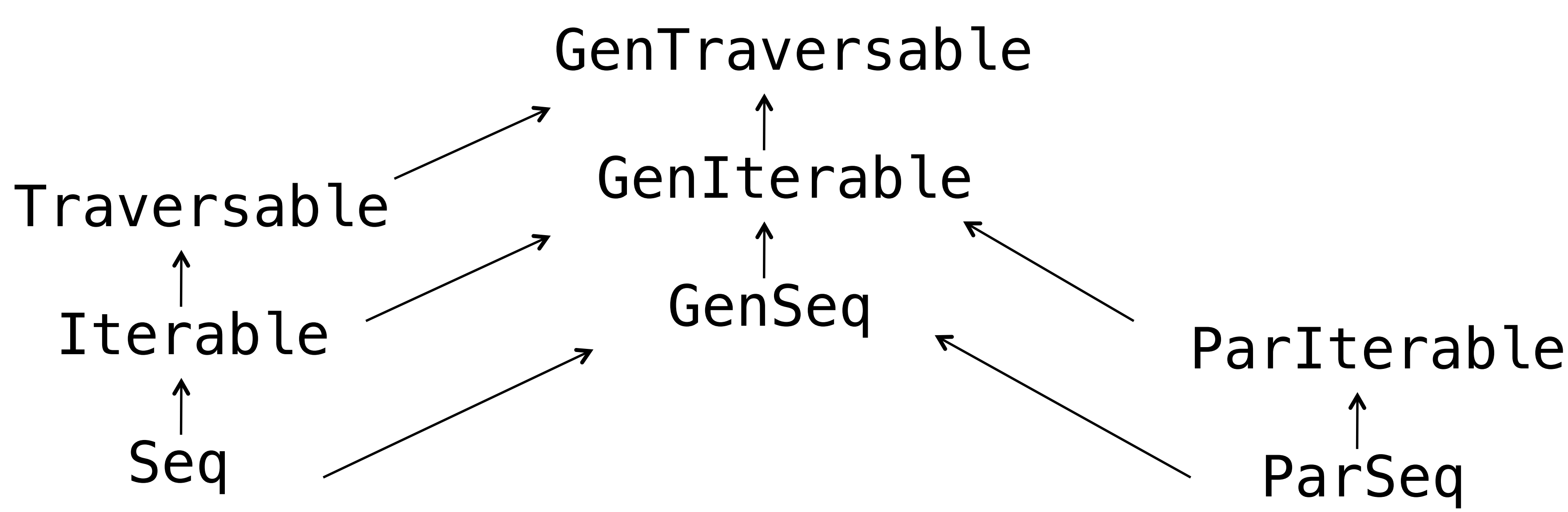
The Collections Hierarchy.



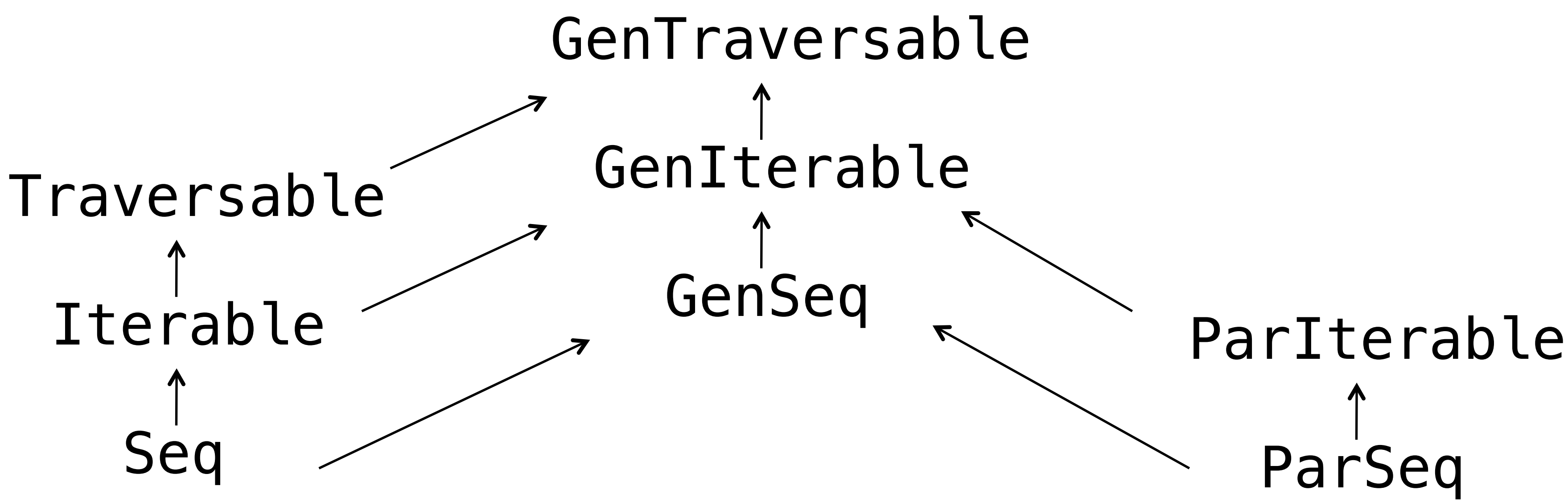
The Collections Hierarchy.

Mutable parallel collections:

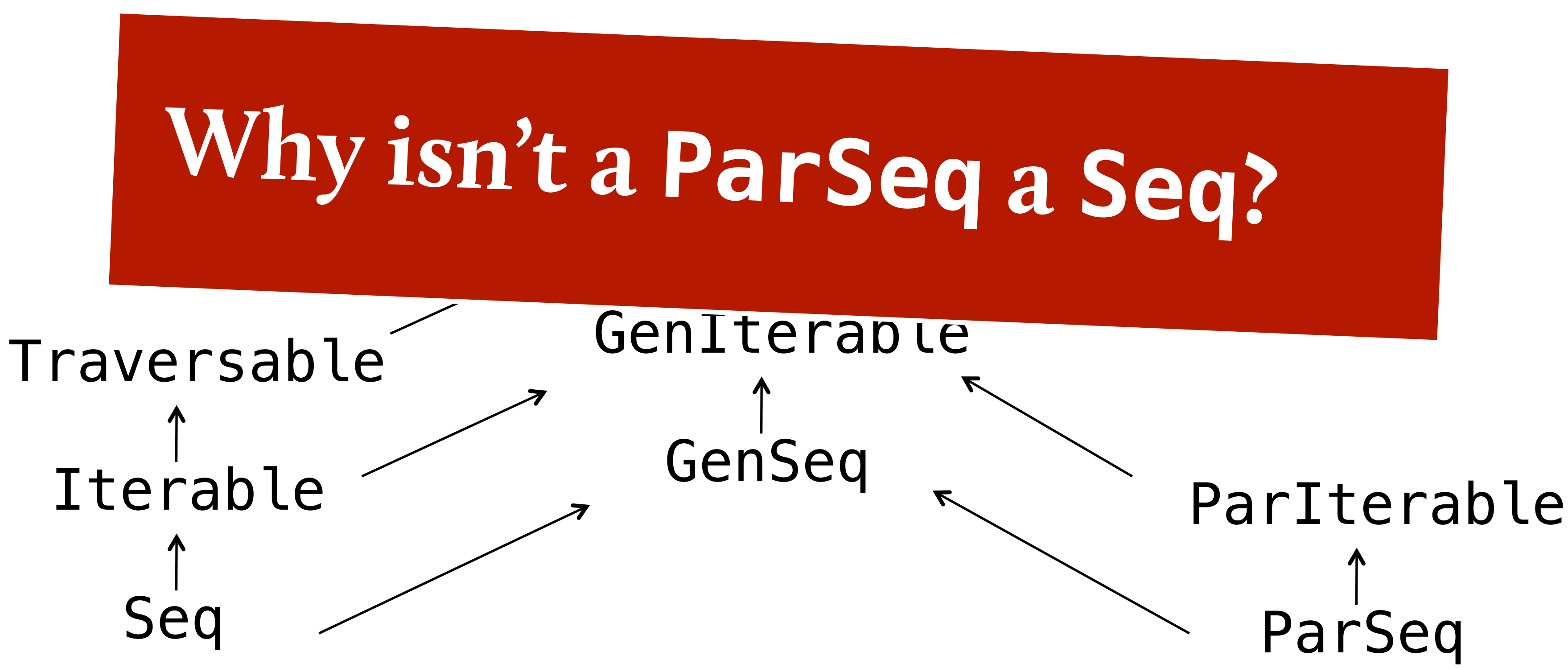
`ParArray`
`ParHashMap`



The Collections Hierarchy.



The Collections Hierarchy.



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.

Splitters and Combiners.

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

```
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:

```
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

```
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?