Integrating Task Parallelism with Actors

OOPSLA, October 25, 2012

Shams Imam, Vivek Sarkar
Rice University

Introduction

- Advent of multi-core processors
- Writing programs exploiting parallelism is hard!
- Renewed interest in parallel and concurrent programming models
- Reduce the burden of reasoning about and writing concurrent programs

Goal

Integrate

- Actor Concurrency (Message Passing)
- Async-Finish (Task Parallelism)

to

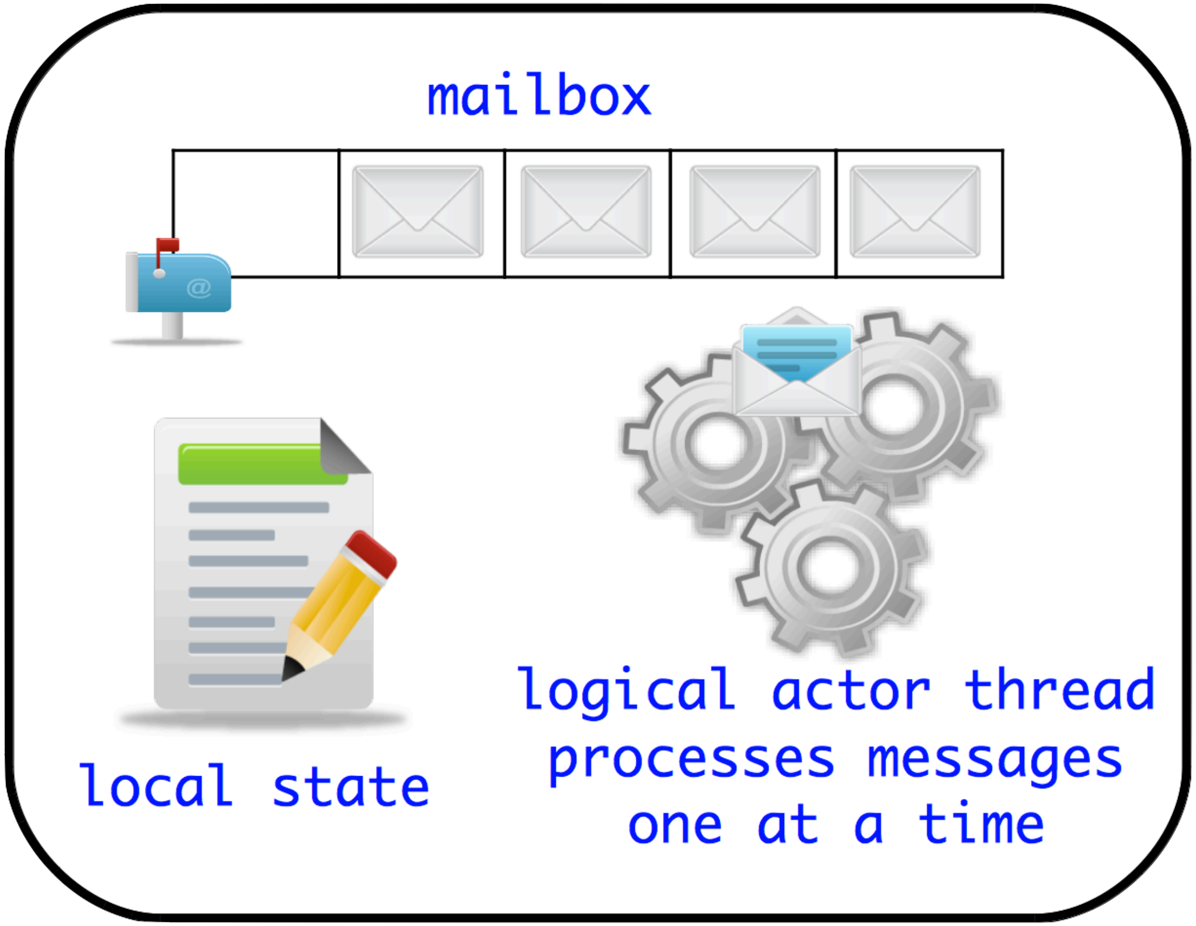
- Exploit nondeterministic communication patterns
- Enable easier expression of potential parallelism
- Achieve better performance

Outline

- Introduction
- The Actor and Async/Finish Models
- The Unified Model
- Intra-Actor Parallelization
- Experimental Results

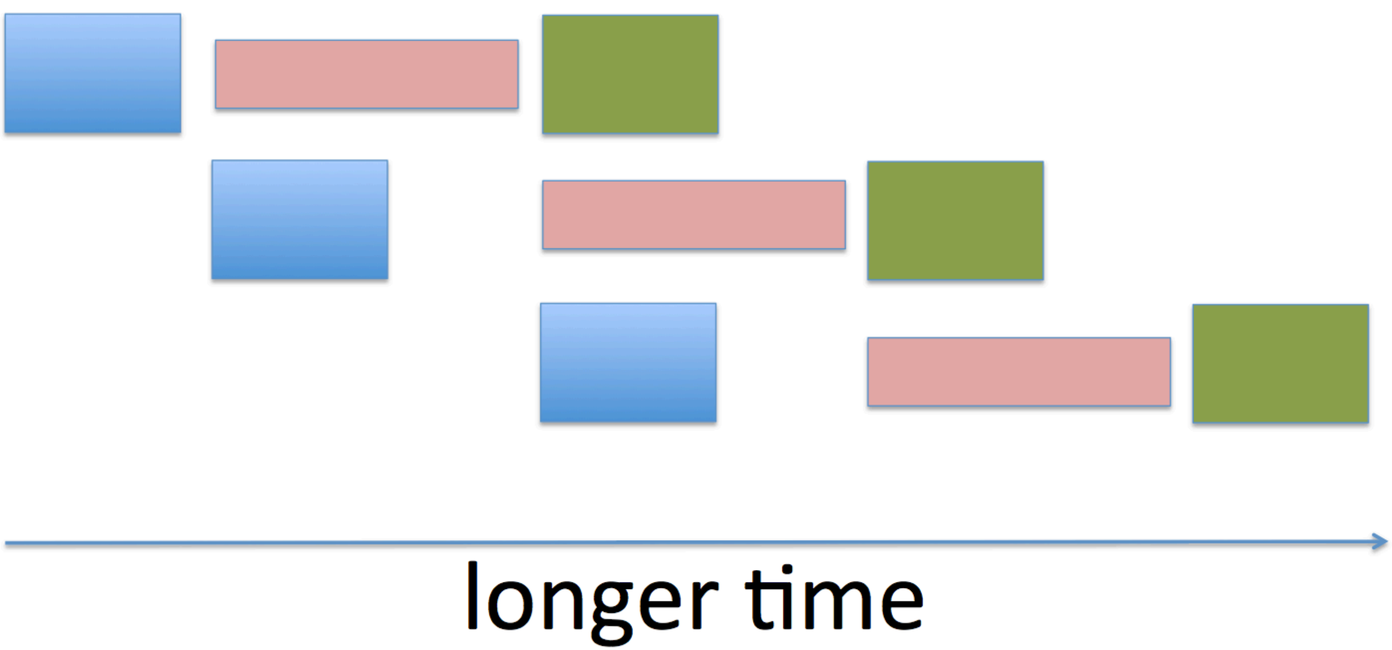
The Actor Model

- A message-based concurrency model
- An Actor encapsulates mutable state



Example Scenario

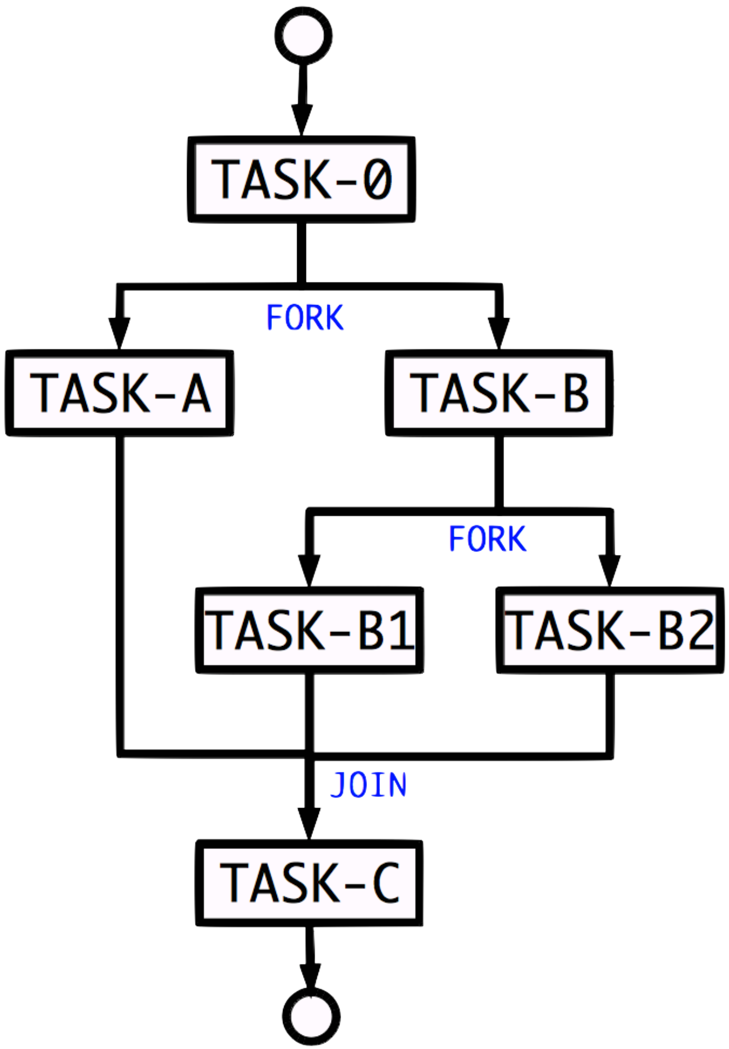
- Pipelined Parallelism
  - each stage can be represented as an actor
  - stages however need to ensure ordering of messages while processing them
  - slowest stage is a throughput bottleneck



The Async/Finish Model (AFM)

- A special case of the Task Parallel Model
- Parent tasks forks child tasks
- Synchronization when tasks join into another task

AFM Example




```
/*** Habanero-Scala code ***/
object AFMPrimer extends HabaneroApp {
  println("Task O"); // Task-O
  finish {
    async { // Task-A
      println("Task A");
    }
    async { // Task-B
      println("Task B");
      async { // Task-B1
        println("Task B1");
      }
      async { // Task-B2
        println("Task B2");
  } } }// Wait for A, B, B1 and B2
  println("Task C"); // Task-C
} }
```

Outline

- Introduction
- The Actor and Async/Finish Models
- The Unified Model
- Intra-Actor Parallelization
- Experimental Results

The Unified Model

- Actors integrated with the AFM
  - Integration needs to be seamless
  - No additional constraints on actors
- Benefits
  - Extend actor capabilities with task parallelism in the unified model
  - Extend task capabilities with more general forms of actor coordination

Actors and Async/Finish Tasks

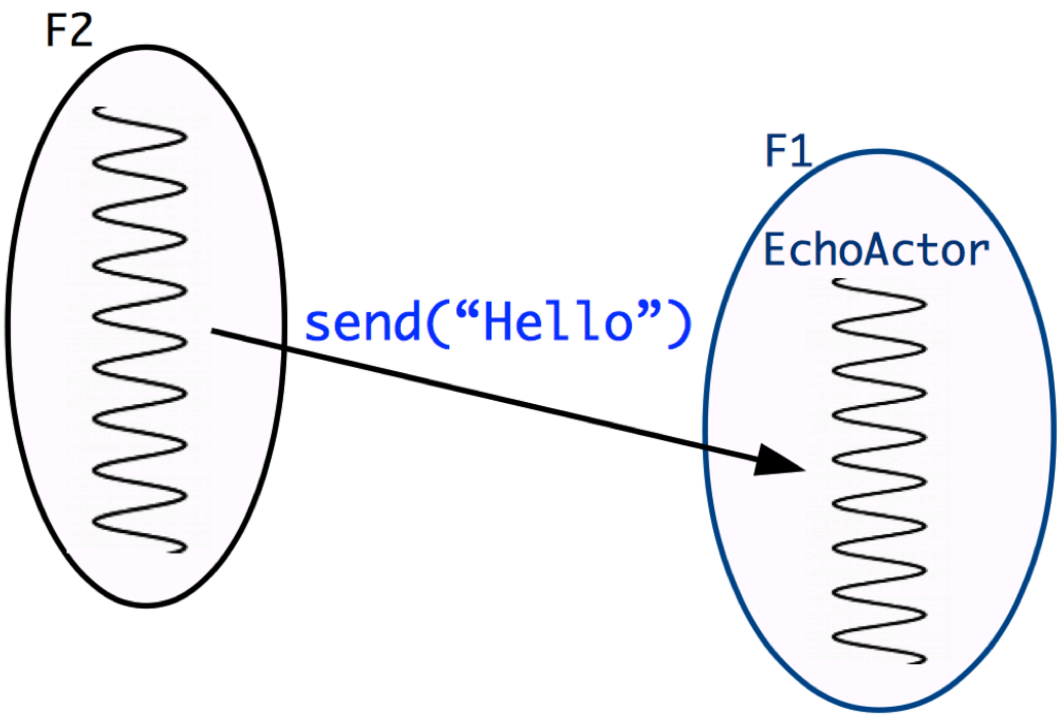
- Actor creation:
  - instantiate (but do not start) actor instance
  - synchronous operation for creator task (i.e. trivial)
- Actor termination:
  - actor will no longer process messages sent to it
  - synchronous operation by actor (i.e. trivial)
  - all future send requests can be ignored synchronously

Actors and Async/Finish Tasks...

- Starting an Actor:
  - finish scope for actor = finish scope for call to start()
  - actor will start processing messages asynchronously in this finish scope
    - performed by actor task created by runtime
  - actor needs to keep the finish scope “alive” until actor is terminated (even if mailbox is empty)
    - use *lingering* task technique (in a couple of slides)

Actors and Async/Finish Tasks...

- Sending messages:



  - possible via *lingering* task technique

Lingering Tasks

- Provide a hook into some finish scope
- Use the *lingering* task to spawn new send and message processing tasks
- One *lingering* task per actor
  - created when the actor is started
  - *lingering* task completes execution only when the actor terminates

Easier Termination Detection in Unified Model

```
/*** Scala code ***/
object Terminator extends App {
  val latch = new CountDownLatch(1)
  val printActor = new PrintActor(latch)
  printActor.start()
  printActor.send("Hello World")
  printActor.send(StopMsg())
  latch.await()
  println("Actor terminated")
}

class PrintActor(latch: CountDownLatch)
    extends Actor {
  def act() {
    loop { react {
    case msg: String =>
      println(msg)
    case msg: StopMsg =>
      latch.countDown()
      exit()
} } } }
```

```
/*** Habanero-Scala code ***/
object Terminator extends HabaneroApp {
  val printActor = new PrintActor()
  finish {
    printActor.start()
    printActor.send("Hello World")
    printActor.send(StopMsg())
  } // wait until actor terminates
  println("Actor terminated")
}

class PrintActor extends UnifiedActor {
  def behavior() = {
    case msg: String =>
      println(msg)
    case msg: StopMsg =>
      exit()
} }
```

Outline

- Introduction
- The Actor and Async/Finish Model
- The Unified Model
- Intra-Actor Parallelization
- Experimental Results

Parallelizing Actors

- Traditionally actor message processing (MP) has been sequential
- Under the AFM, we can use two techniques to parallelize the MP…

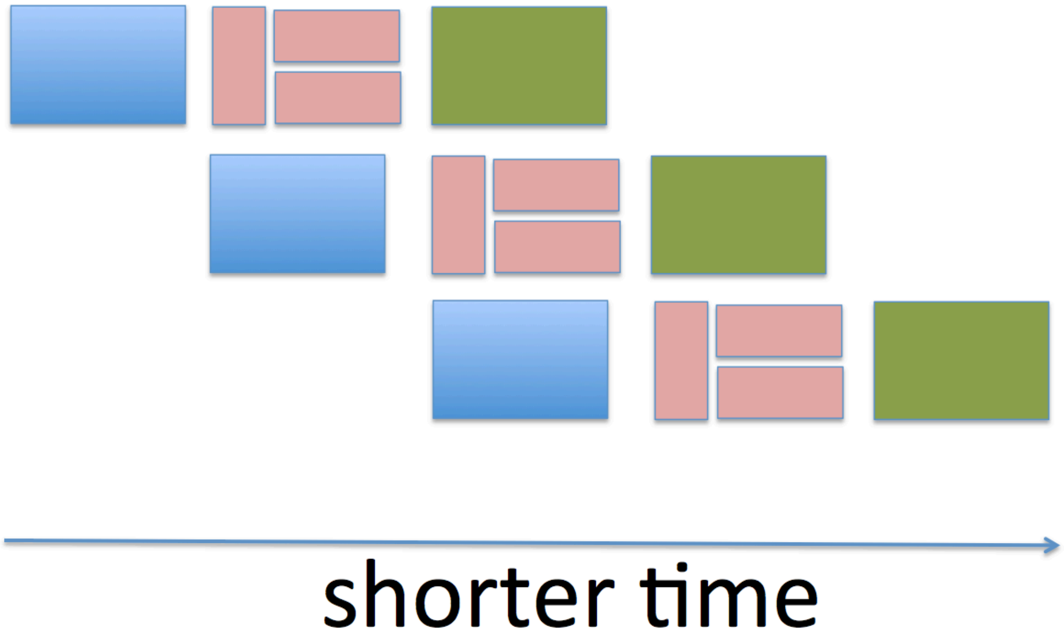
Parallelizing Actors (contd)

1. Use finish construct in MP body and spawn child tasks

```
/*** Habanero-Scala code ***/
class FirFilter(..., nextStage: UnifiedActor)
    extends UnifiedActor {
  ...
  def behavior() = {
    case FirItemMessage(value, coeffs) =>
      ...
      val helpers = ... // number of helper tasks
      val stores = Array.ofDim[Double](helpers)
      finish {
        // compute the sum using divide-and-conquer
        (0 until helpers) foreach { helperId =>
          async {
            val (start, end) = ...
            var sum: Double = 0.0
            start until end foreach { index =>
              sum += buffer(index) * coeffs(index)
            }
            stores(helperId) = sum
      } } }
      // propagate the sum down the pipeline
      val globalSum = stores.foldLeft(0.0) {
        (acc, loopVal) => acc + loopVal
      }
      nextStage.send(DataItemMessage(globalSum))
    case ... => ...
} }
```

Example Scenario

- Pipelined Parallelism
  - reduce effects of slowest stage by introducing task parallelism
  - increases the throughput



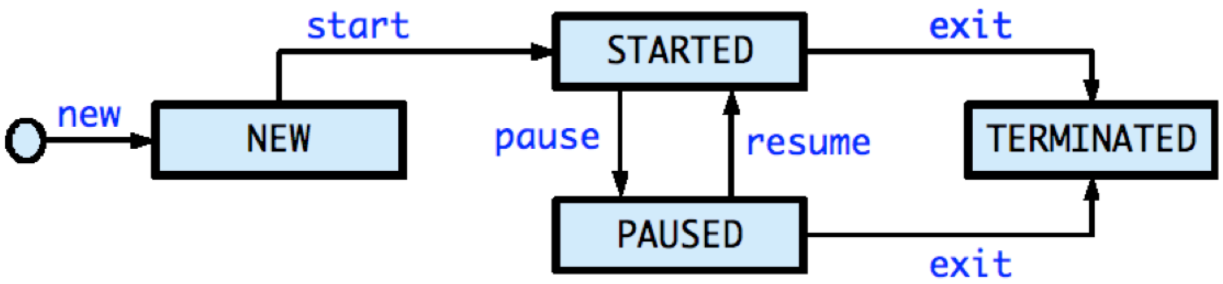
Parallelizing Actors (contd)

2. Allow *escaping* asyncs inside MP body

```
/*** Habanero-Scala code ***/
class ParallelizedActor() extends UnifiedActor {
  override def behavior() = {
    case msg: SomeMessage =>
      finish {
        async { ... /* processing in parallel */ }
        // some more processing
      }
      async { ... /* escaping async */ }
    ...
} }
```

- WAIT! What about the single message processing invariant?

Pause and Resume an Actor



- paused state
  - actor will no longer process messages sent to it
- new operations:
  - pause(): move from started to paused state
  - resume(): move from paused to started state
- pause actor before returning from MP body
- resume actor when safe to process next message

Parallelizing Actors (contd)

```
/*** Habanero-Scala code ***/
class EscapingAsyncsActor() extends UnifiedActor {
  override def behavior() = {
    case msg: SomeMessage =>
      async { /* do some processing in parallel */ }
      // preprocess the message
      pause() // delay processing the next message
      // pause/resume is not thread blocking
      async {
        // do some more processing in parallel
        // safe to resume processing other messages
        resume()
        // some more processing
      } ...
} }
```

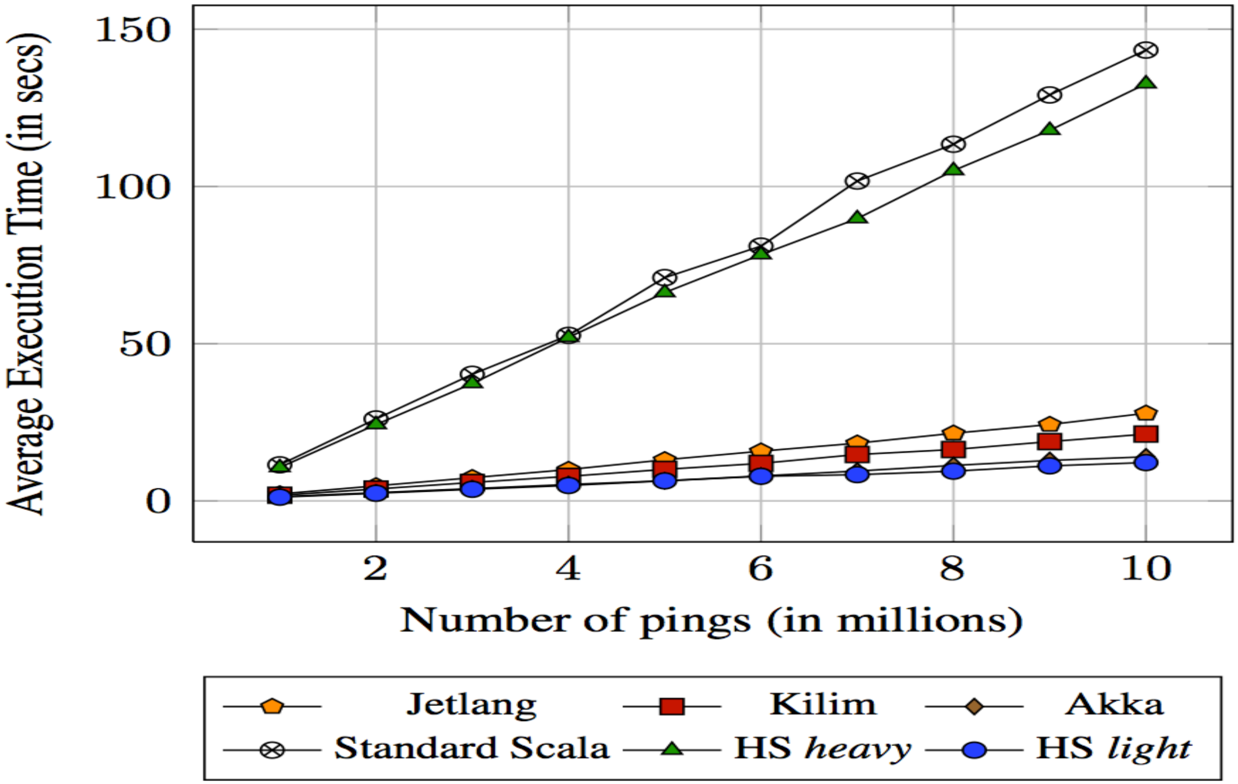
Outline

- Introduction
- The Actor and Async/Finish Model
- The Unified Model
- Intra-Actor Parallelization
- Experimental Results

Experimental Setup

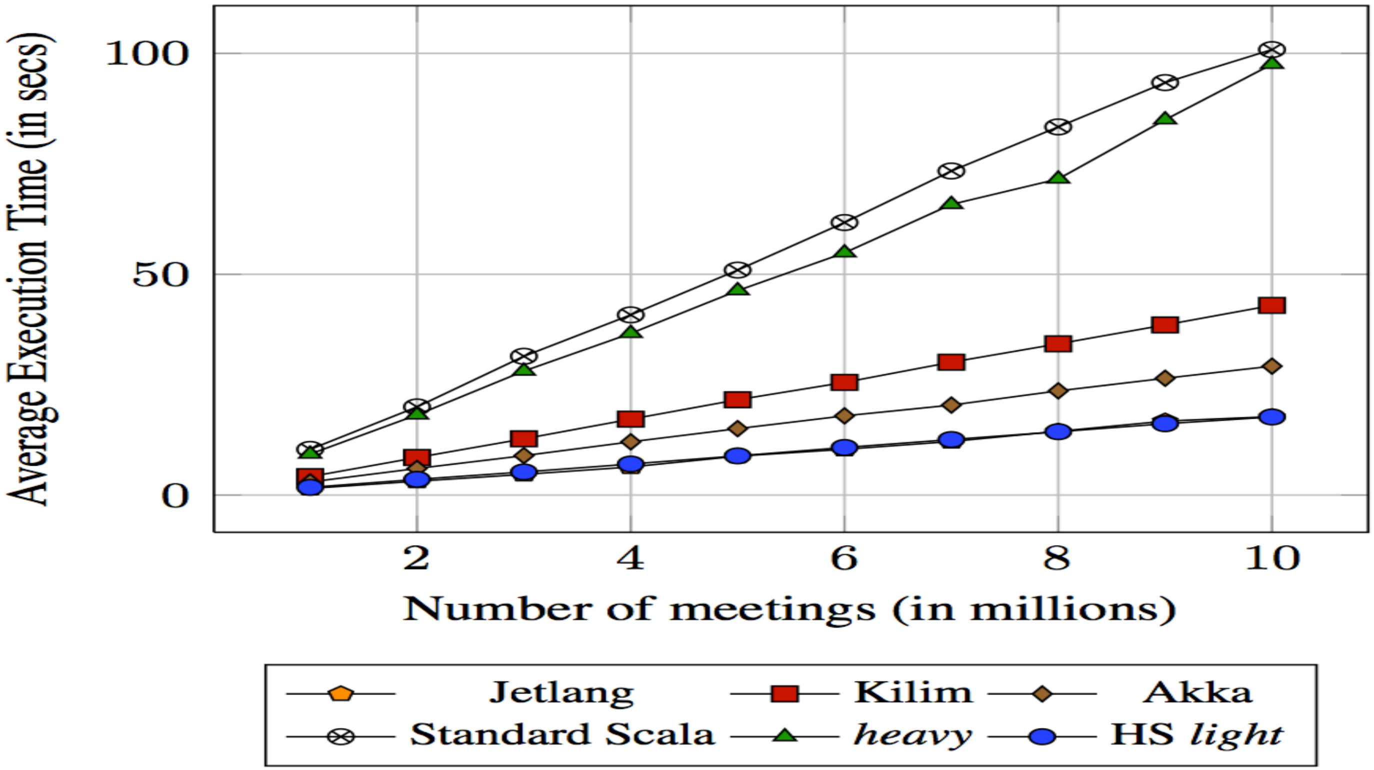
- Configuration
  - 2.8 GHz Intel Westmere, 12-core SMP node
  - 48 GB of RAM per node (4 GB per core)
  - Red Hat Linux (RHEL 6.0)
  - Sun Hotspot JDK 1.7
  - Scala 2.9.1-1
  - Habanero-Scala 0.1.3 (http://habanero-scala.rice.edu/)
- Execution time reported using variant of "Statistically Rigorous Java Performance Evaluation" by A. Georges et al.

Ping-Pong Benchmark



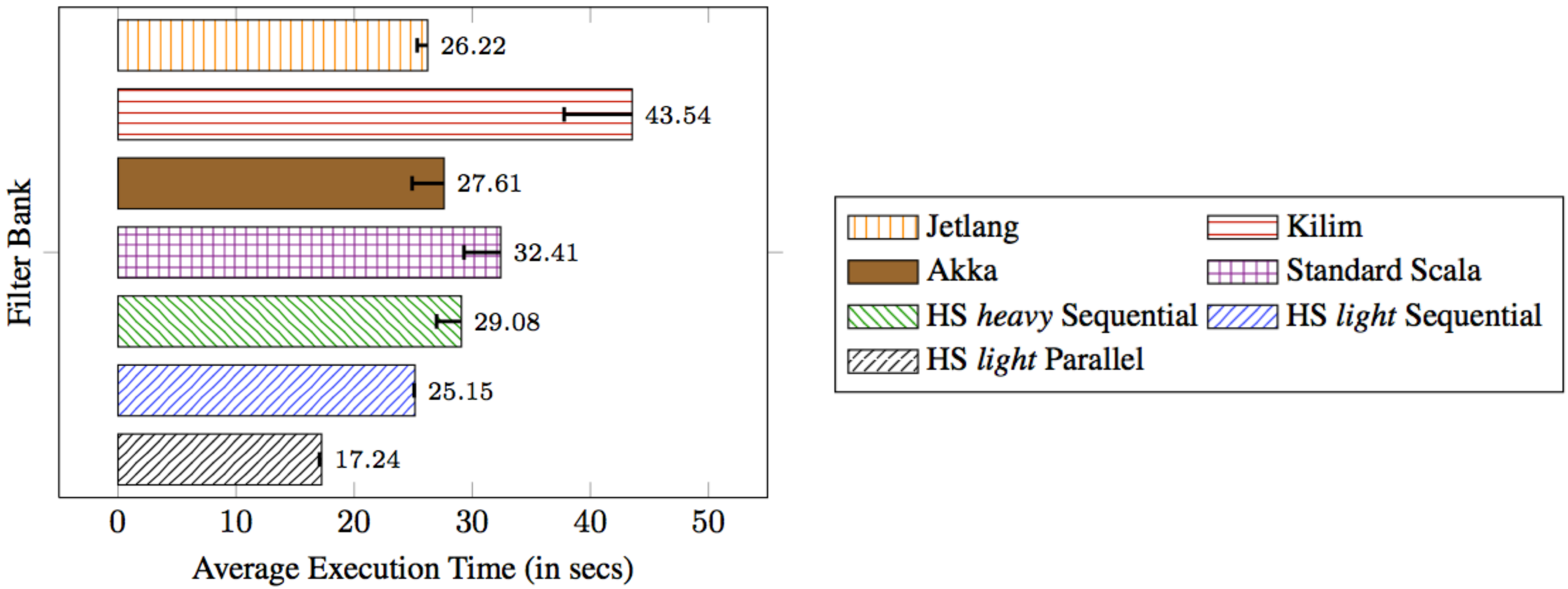
- measures raw message throughput
- HS Light and Akka actors fastest
  - no exceptions
  - Fork-join scheduler

Chameneos Benchmark



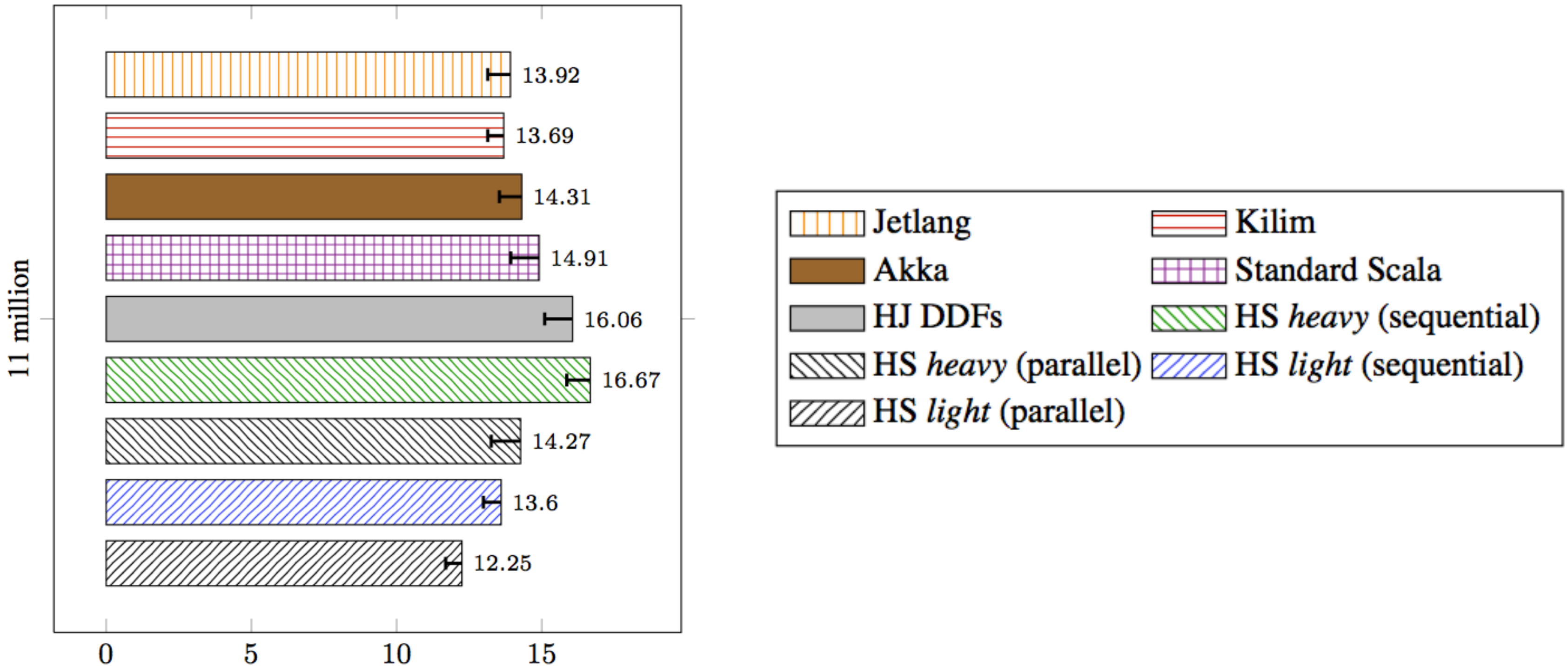
- measures cost of synchronization in the mailbox
- HS Light actor and Jetlang fastest
  - both uses batch processing of messages
  - Light actors use DDCs

Filter Bank (Pipeline)



- Unified solution fastest
  - up to 30% faster than pure Actor solutions
  - stage parallelization shortens critical length of the pipeline

Quicksort Benchmark



- Unified solution fastest
  - up to 10% faster than pure Actor solutions
  - around 23% faster than DDF solution

Related Work

- Parallel Actor Monitors by Scholliers et al.
  - Parallelism by modifying message processing scheduler
  - Does not allow parallelism inside message processing body
- CoBox Model by Schäfer et al.
  - Communicating CoBoxes host multiple objects
  - Each CoBox contains multiple tasks but cooperatively executes only one at a time
  - Our model is equivalent to allowing a CoBox to execute multiple tasks at a time
- Other related work discussed in the paper

Summary

- A unified programming model that integrates
  - the Async/Finish model
  - the Actor model
- Application characteristics that benefit from the unified model
- Future work
  - Extend past work on data race detection for the AFM to the unified model

Acknowledgments

- U.S. NSF awards 0926127 and 0964520
- Habanero Group
  - Vincent Cavé
  - Dragoș Sbîrlea
  - Sağnak Taşırlar
- External Feedback
  - Anonymous reviewers of OOPSLA 2012 submission
  - Carlos Varela
  - Travis Dessell

Thank you!

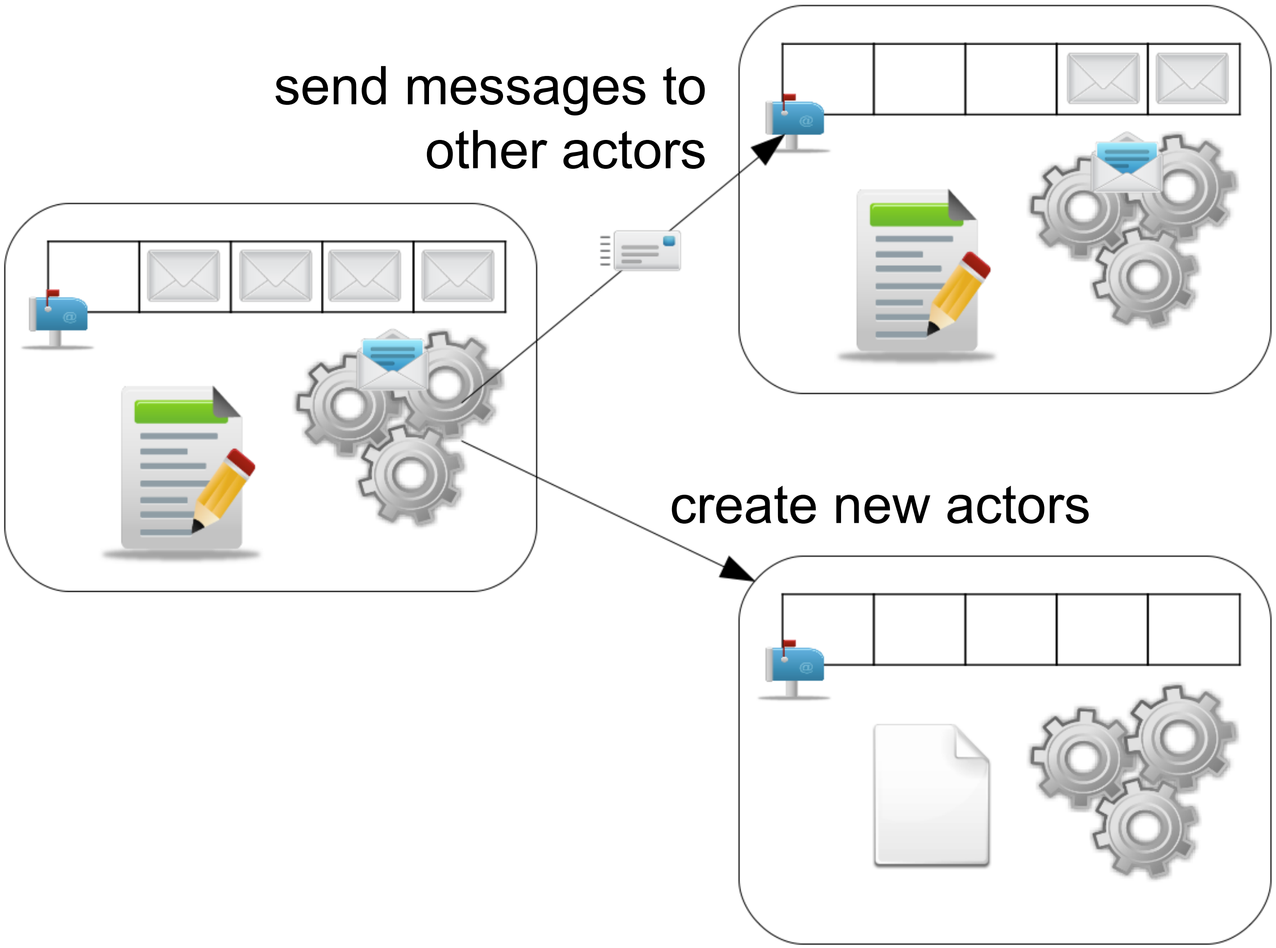
```
import audience.questions.*
```

[image source: http://www.jerryzeinfeld.com/tag/question-of-the-day/]

BACKUP SLIDES START HERE

Actor - Interactions

- Actors coordinate using *asynchronous* messaging
- Non-deterministic ordering of messages



Actor – Motivating Example

```
/*** Scala code ***/
object FilterBankApp extends App {
  ...
  val sampler = ...
  val fir = new FirFilter(..., sampler).start()
  ...
  latch.await()
}
class FirFilter(..., nextStage: Actor)
      extends Actor {
  ...
  def act() = {
    loop { react {
    case FirItemMessage(value, coeffs) =>
        ...
        // compute the sum
        var sum = 0.0
        0 until coeffs.length foreach { index =>
          sum += buffer(index) * coeffs(index)
        }
        ...
        nextStage.send(DataItemMessage(sum))
    case ... => ...
} } } }
```

Actors mapped to AFM

- Asynchronous messaging handled
- One message processed at a time invariant preserved
- Additional constructs used
    - *lingering* tasks
    - data-driven controls [for mailbox, see paper for details]
- No extra constraints placed on the Actors