Habanero-Scala: Async-Finish Programming in Scala

Scala Days

April 17, 2012

Shams Imam and Vivek Sarkar

Rice University

Introduction

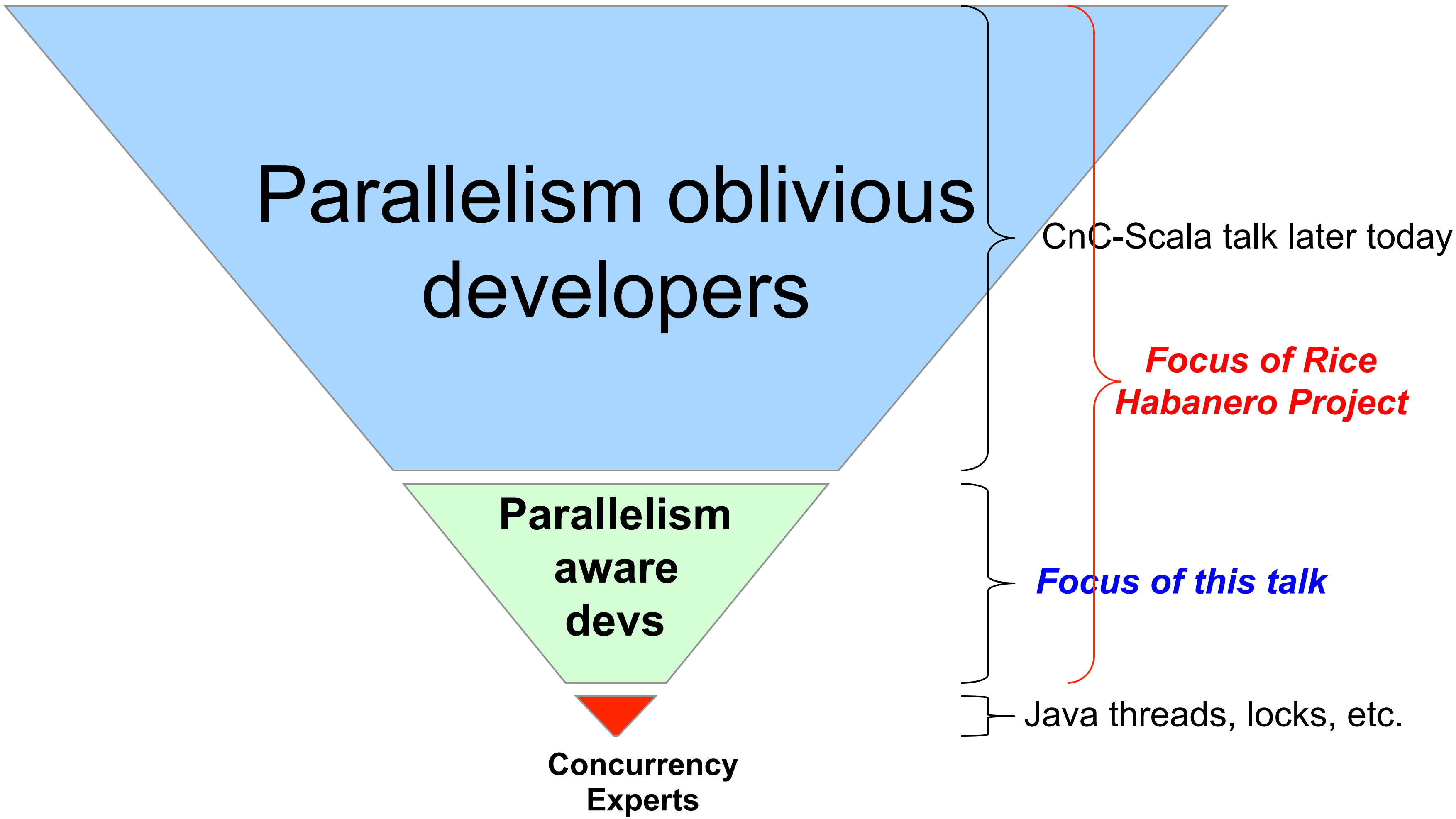
- Multi-core processors

  → Software Concurrency Revolution

  → renewed interest in parallel programming models

- Goal: increase productivity of parallel programming by both simplifying and generalizing current parallel programming models
  - Simplification → increase classes of developers who can write parallel programs
  - Generalization → increase classes of applications that can be supported by a common model

Inverted Pyramid of Parallel Programming Skills



http://habanero.rice.edu

Habanero-Scala

- Scala integration of Habanero-Java features
- Habanero-Java
  - developed at Rice University
  - derived from Java-based version of X10 language (v1.5) in 2007
  - targeted at parallelism-aware developers, not necessarily concurrency experts
  - used in sophomore-level undergraduate course on "Fundamentals of Parallel Programming" at Rice
    - https://wiki.rice.edu/confluence/display/PARPROG/COMP322
    - Or search for "comp322 wiki"

Goals for this talk

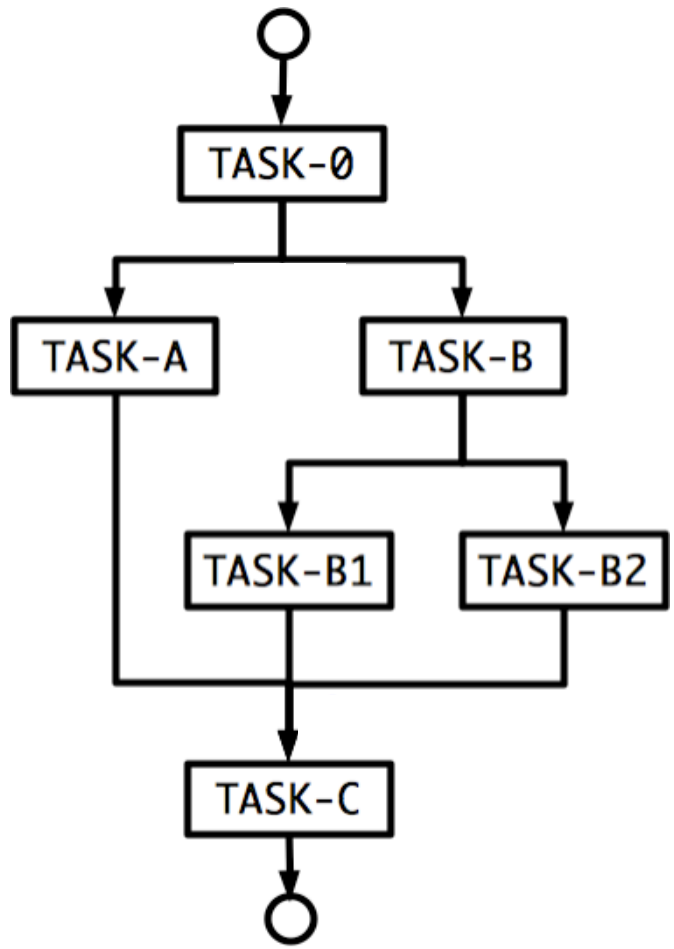
- Task parallelism
  1. Dynamic task creation & termination
     - async, finish, forall, foreach
  2. Mutual exclusion: isolated
  3. Coordination
     - futures, data-driven futures
  4. Collective and P2P synchronization:
     - phaser, next
  5. Locality control for tasks and data: places
- Actor extensions and unification with task parallelism

Async and Finish

- `async` { <stmt> }
  - creates a new child task that executes <stmt>
  - parent task proceeds to operation following the async
  - `asyncSeq(<cond>)` { S } ≡ if `(<cond>) S else async { S }`
- `finish` { <stmt> }
  - execute <stmt>, but wait until *all* (transitively) spawned asyncs in <stmt>'s scope have terminated
  - Implicit finish between start and end of main program
- Async-Finish programs **cannot** create a deadlock cycle

Async-Finish Example

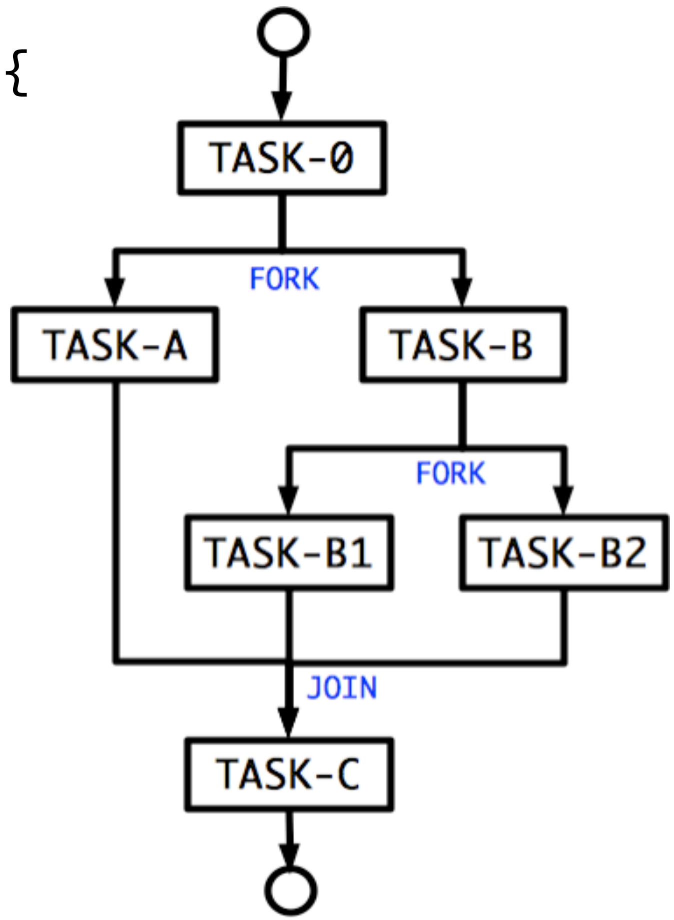
```
// imports
object SeqApp extends App {

  println("Task 0")


    println("Task A")

    println("Task B")
      println("Task B1")
      println("Task B2")

  println("Task C")

}
```



[image adapted from: http://www.coopsoft.com/ar/ForkJoinArticle.html]

Async-Finish Example (contd)

```
// imports
object ParApp extends HabaneroApp {

  println("Task 0")
  finish {
    async {
      println("Task A")
    }
    async {
      println("Task B")
      async { println("Task B1") }
      async { println("Task B2") }
    }
  }
  println("Task C")

}
```



[image adapted from: http://www.coopsoft.com/ar/ForkJoinArticle.html]

Forall and Foreach

- `forall(start, end)` { f(i) } ≡

  `finish` { `for(i <- start until end)` async { f(i) } }
- `foreach(start, end)` { f(i) } ≡

  `for(i <- start until end)` async { f(i) }
- `scala.collection.Iterables` support `asyncForall` and `asyncForeach` as extension methods
- E.g.

```
1 to 20 asyncForeach {
    i =>
      println(" i = " + i)
}
```

Synchronized access - isolated

- `isolated` { <stmt> }
  - Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
  - isolated statements can be nested (redundant)
  - support weak isolation, i.e. atomicity is guaranteed only with respect to other statements also executing inside isolated scopes

Parallel DFS - example

```
object DepthFirstSearchApp extends HabaneroApp {
  ...
  finish { root.parent = root; root.compute() }
}

class Node() {
  def tryLabelling(p: Node): Boolean = {
    isolated {
      if (parent eq null)
        parent = p
    }
    (parent eq p)
  }

  def compute(): Unit = {
    neighbors foreach { child =>
        if (child.tryLabelling(this)) async {
          child.compute()
} } } }
```

Futures – Tasks with Return Values

- `asyncFuture[T]` { <stmt> }
  - creates a new child task that executes <stmt>
  - parent task proceeds to operation following the async
  - return value of <stmt> must be of type T
  - `asyncFuture` expression returns a reference to a *container* of type `habanero.Future[T]`
  - `aFuture.get()` blocks if value is unavailable
    - `aFuture.get()` only waits for specified async
- Assignment of future references to final variables **guarantees** deadlock freedom with `get()` operations
- In addition, no data races are possible on future return values

Futures – example

```
def fib(n: Int): Int = {
   if (n < 2) {
     n
   } else {
     val x = asyncFuture {
       fib(n - 1)
     }
     val y = asyncFuture {
       fib(n - 2)
     }

     x.get() + y.get()
   }
 }
```

Data-Driven Futures (DDFs)

- separation of classical “futures” into data (DDF) and control (`asyncAwait`) parts
- Operations:
  - `ddf[T]()`: new instance using factory method
  - `put(someValue)`: only a single `put()` is allowed on the DDF
  - `asyncAwait()`: declare data/control dependency in an `async`
  - `get()`: returns the value associated with the DDF
- Accesses to values inside the DDF are guaranteed to be race-free and deterministic

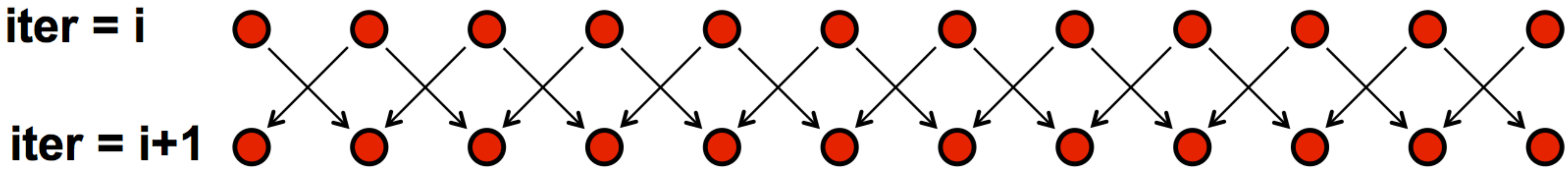
DDF – Fib example

```
finish {
  val res = ddf[Int]()
  async {
    fib(N, res)
  }
}
println("fib(" + N + ") = " + res.get())

def fib(n: Int, v: DataDrivenFuture[Int]): Unit = {
    if (n < 2) {
      v.put(n)
    } else {
      val (res1, res2) = (ddf[Int](), ddf[Int]())
      async {
        fib(n - 1, res1)
      }
      async {
        fib(n - 2, res2)
      }
      asyncAwait(res1, res2) {
        v.put(res1.get() + res2.get())
} } }
```

Phasers

- Support Collective and Point-to-Point synchronization
- Tasks can register in
  - *signal-only/wait-only mode* for producer/consumer synchronization
  - *signal-wait* mode for barrier synchronization
- *next* operation is guaranteed to be deadlock-free
- HJ programs with phasers, finish, async, async-await (but not isolated) are guaranteed to be deterministic if they are data-race-free

Phasers – Iterative Averaging example




```
finish {
  val myPhasers = Array.tabulate[Phaser](n + 2)(i => phaser())
  for (index <- 1 to n) {
    val (me, left, right) = (index, index - 1, index + 1)
    val leftPhaser = myPhasers(left).inMode(PhaserMode.WAIT)
    val selfPhaser = myPhasers(me).inMode(PhaserMode.SIG)
    val rightPhaser = myPhasers(right).inMode(PhaserMode.WAIT)

    asyncPhased(leftPhaser, selfPhaser, rightPhaser) {
      for (iter <- 0 until N) {
        val loopVal = 0.5 * (dataArray(left) + dataArray(right))
        // Allow others to proceed and modify dataArray
        next
        // update the 'owning' element
        dataArray(me) = loopVal
        // notify others that value has been updated
        next
} } } }
```

Phaser Accumulators

- A parallel reduction construct which separates reduction computations into the parts of
  - sending data,
  - performing the computation itself, and
  - retrieving the result
- Support two logical operations:
  - `send(value)`: to send a value for accumulation in the current phase
  - `result()`: to receive the accumulated value from the previous phase

Sum Reduction example

```
finish {
  val ph = phaser()
  val sumAccum = intAccumulator(Operator.SUM, ph)

  for (i <- 1 to 30) asyncPhased(ph.inMode(PhaserMode.SIG)) {
    sumAccum.send(i)
    sumAccum.send(i + 30)
    sumAccum.send(i + 60)
  }

  asyncPhased(ph.inMode(PhaserMode.WAIT)) {
    // wait for the tasks from for to complete
    next
    val resVal: Int = sumAccum.result()
    println("Sum(1..90) = " + resVal)
  }
}
```

Places

- Logical location where tasks are run
  - enables locality control and load balancing among worker threads
- `async`(<some-place>) { <stmt> } launches an `async` at the specified place
- Current place can be obtained by invoking `here()`
- Set of places are ordered and `aPlace.next()` and
- `aPlace.prev()` may be used to cycle through them
- System property, -Dhs.places p:w, allows the user to specify how many places (p) and workers per place (w) the runtime should be initialized with.

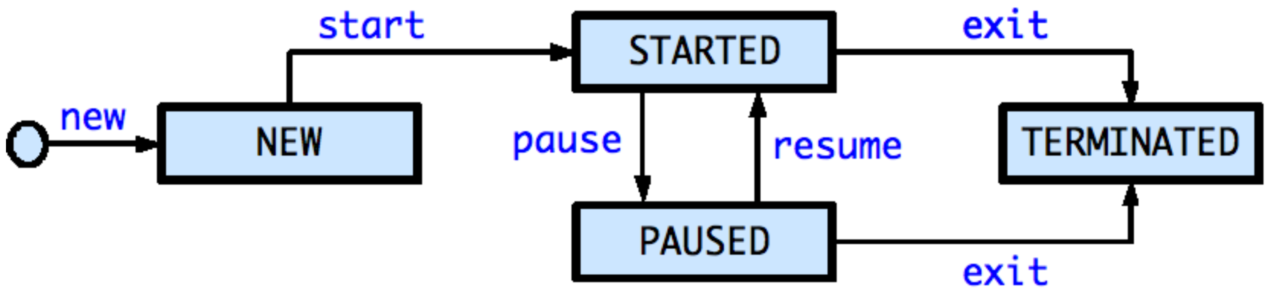
Actors and Async/Finish Tasks

- Actors interact seamlessly with async and finish compliant constructs in Habanero-Scala
- Simplifies termination detection
  - wrap actors in a finish scope
- Parallelize message processing inside actors
- Two actor implementations:
  - compliant with Standard Scala actors, extend from HabaneroActor instead of Actor
  - a more efficient implementation, that extends from the HabaneroReactor class

Example of detecting Actor Termination using finish

```
object LightActorApp extends HabaneroApp {
  finish {
    val pong = new PongActor().start()
    val ping = new PingActor(msgs, pong).start()
    ping ! StartMessage()
  }
  println("Both actors terminated")
}
// class PingActor not displayed
class PongActor extends HabaneroActor {
  var pongCount = 0
  def act() {
    loop { react {
        case PingMessage =>
          sender ! PongMessage
          pongCount = pongCount + 1
        case StopMessage =>
          exit('stop)
} } } }
```

Pause/Resume extension for Actors



- 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
- Pausing an actor prevents it from processing the next message until it is resumed

Non-blocking receives

- Simulates synchronous communication **without** blocking

```
class ActorPerformingReceive extends HabaneroReactor {
  override def behavior() = {
    case msg: SomeMessage =>
      ...
      val theDdf = ddf[ValueType]()
      anotherActor ! new Message(theDdf)
      pause() // delay processing next message
      asyncAwait(theDdf) {
        val responseVal = theDdf.get()
        // process the current message
        ...
        resume() // enable next message processing
      }
      // return in paused state
    ...
} }
```

Stateless Actors

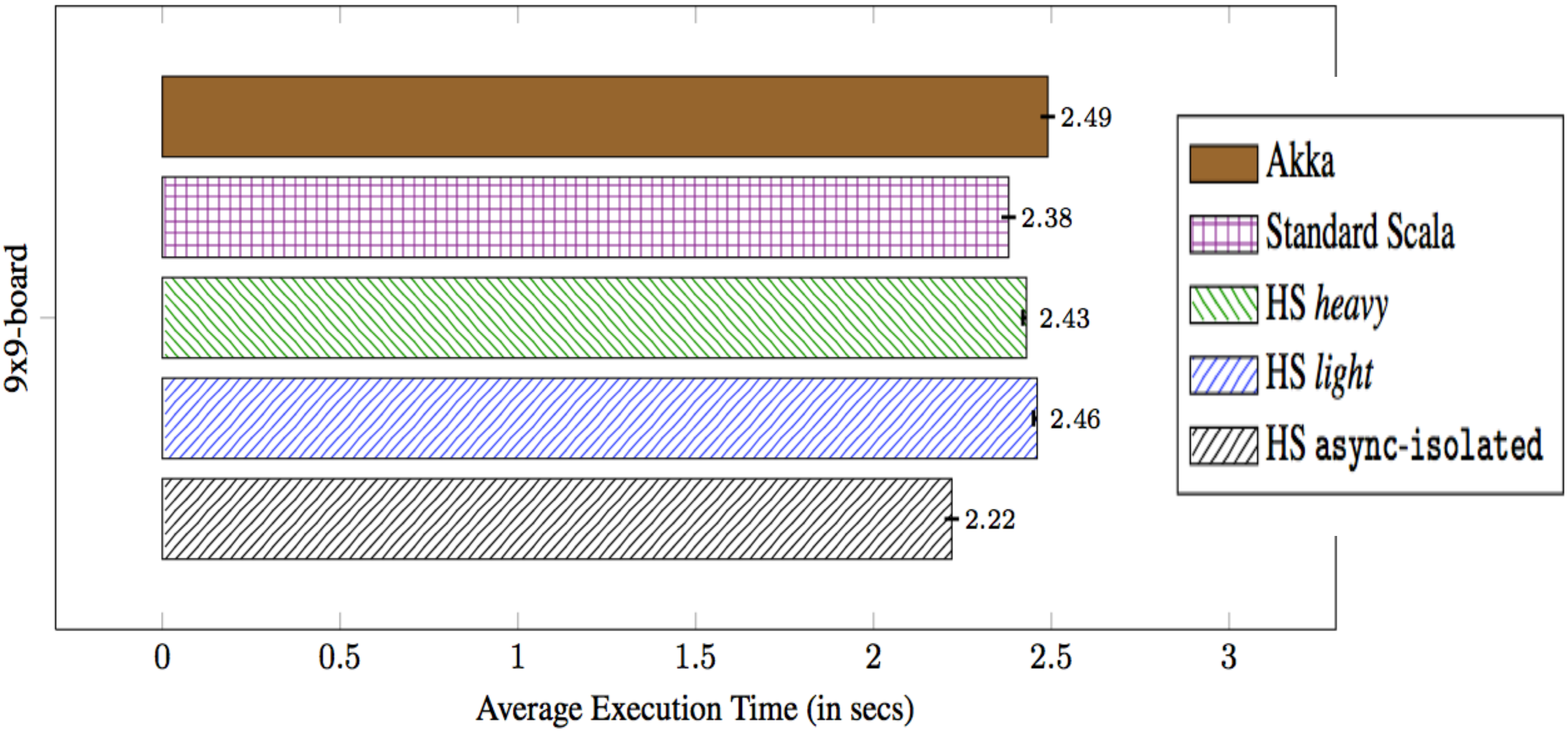
- Actors with no state, can actively process multiple messages without violating actor constraints

```
class StatelessActor() extends Habanero-Rea/A-ctor {
  ...
  override def behavior() = {
    case msg: SomeMessage =>
      async {
        processMessage(msg)
      }
      if (enoughMessagesProcessed) {
        exit()
      }
      // return immediately to process next message
} }
```

Experimental Setup

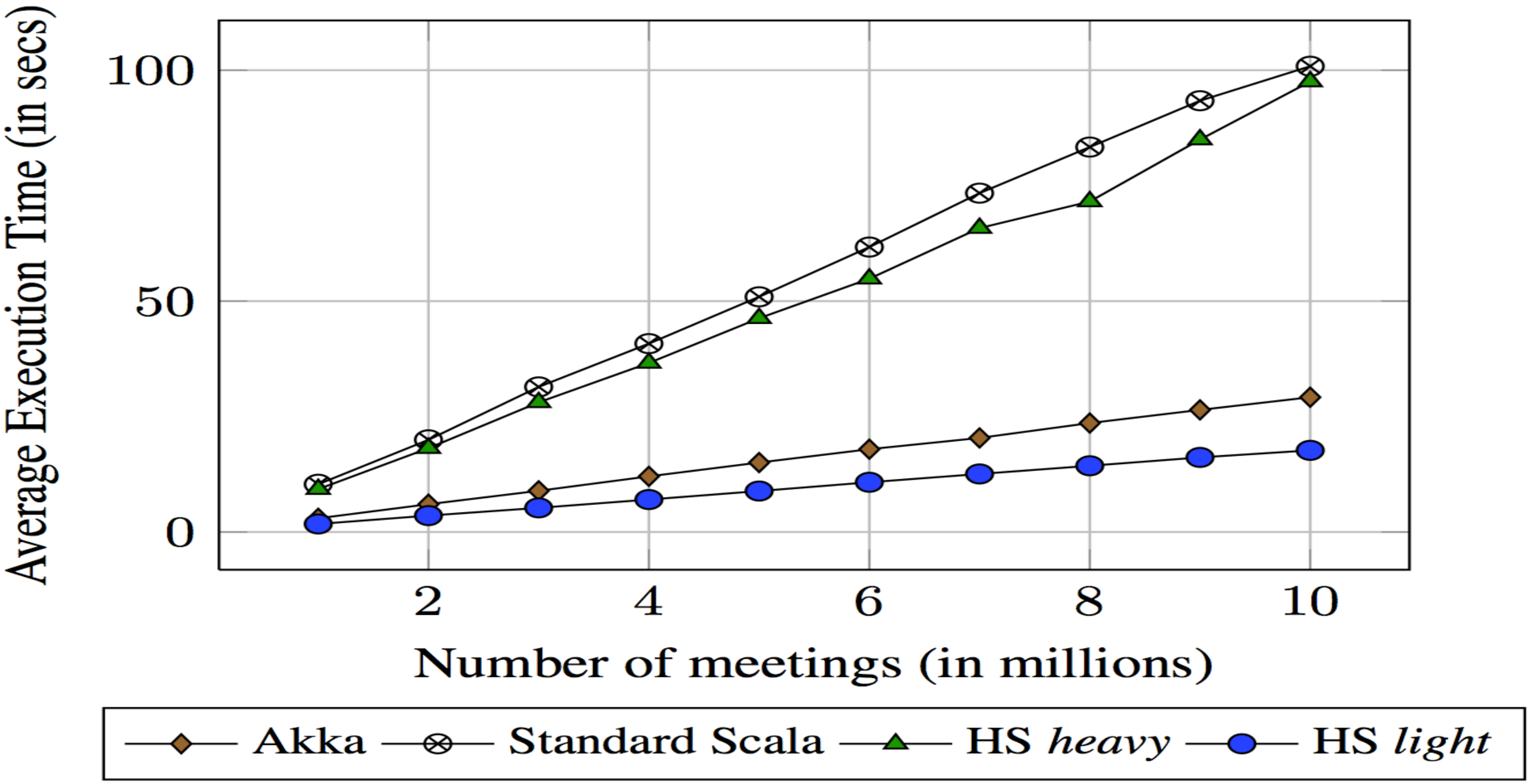
- 12-core (two hex-cores) 2.8 GHz Intel Westmere SMP
- 48 GB memory, running Red Hat Linux (RHEL 6.0)
- Hotspot JDK 1.7
- Scala version 2.9.1-1
- Habanero-Scala 0.1.3
- Arithmetic mean of last thirty iterations from hundred iterations on ten separate JVM invocations

Sudoku – Constraint Satisfaction



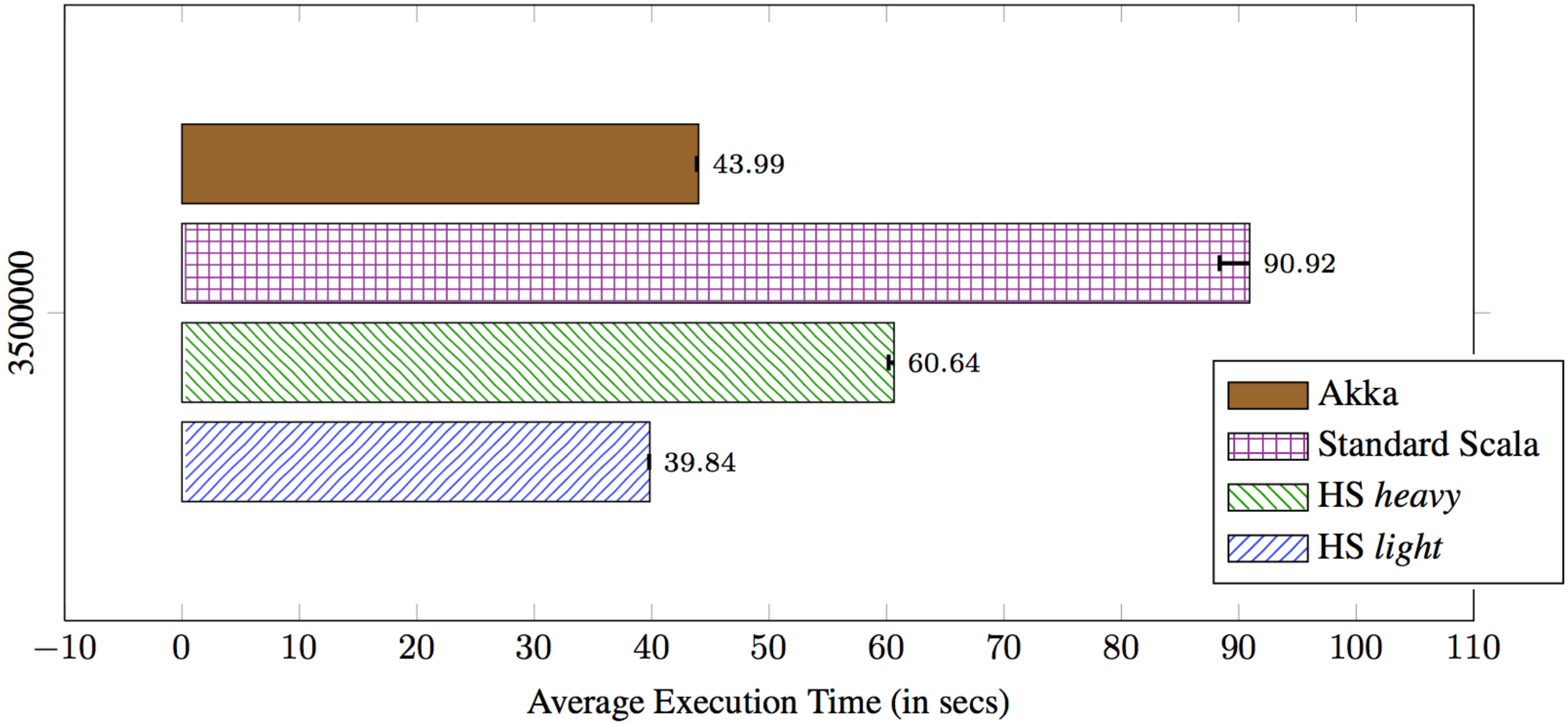
- Actors use Master-Worker style and perform similarly
- Async-Isolated version 7% faster than the actor solutions, about 10% faster than other HS solutions.

Chameneos Benchmark



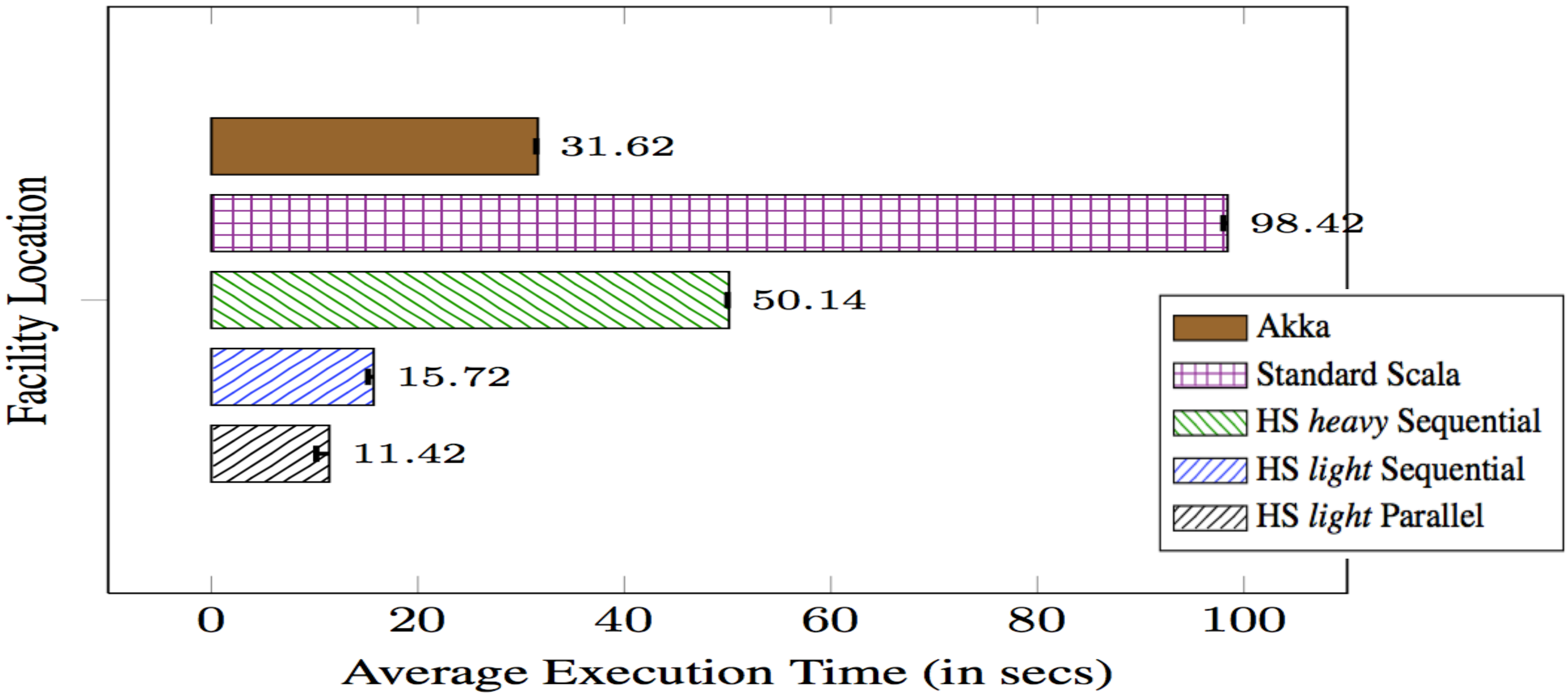
- Measures effects of contention – adding messages to mailbox
- HS *light* actors performs best

Prime Sieve Benchmark



- Example of a dynamic pipeline, good fit for actors.
- HS places and thread binding benefits

Hierarchical Facility Location



- Hybrid solution fastest, about 27% faster than any of the other Actor solutions

Summary

- HS is a safe and powerful mid-level parallel language
  - programmers with a basic knowledge of Scala to get started quickly with expressing a wide range of parallel patterns
  - Deadlock freedom for programs using finish, async, futures, phasers, isolated
  - Data-race freedom for values accessed through futures and data-driven futures
  - Simplifies writing actor programs
- Runs on standard JRE's and delivers good performance on multicore SMPs
- Available for download at:

  http://habanero-scala.rice.edu/

Acknowledgments

- Habanero Group
  - Vincent Cavé
  - Dragos Sbirlea
  - Sagnak Tasirlar

Thank you!

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