Habanero-Scala: A Hybrid Programming model integrating Fork-Join and Actor models

MS Thesis Defense
November 2011

Shams Imam
Rice University
Introduction

• Multi-core processors → renewed interest in programming concurrency models
• Goal is to reduce the burden of reasoning about and writing concurrent programs
• Some popular programming models:
  – Fork/Join
  – Actors
  – Synchronous Message Passing
  – Partitioned Global Address Space
  – Software Transactional Memory
A hybrid parallel programming model that integrates the Fork/Join Model and the Actor Model helps solve certain class of problems more productively and efficiently than either of the aforementioned models individually.
Outline

• The Fork/Join Model
• The Actor Model
• The Hybrid Model
• Applications of the Hybrid Model
• Implementation and Experimental Results
The Fork/Join Model (FJM)

• A special case of the Task Parallel Model
• Regained popularity due to Cilk from MIT
  – spawn/sync
• At Rice, we have Habanero-Java and Habanero-C
  – async/finish
  – soon a Habanero-Scala release
The Fork/Join Model (FJM)

- Parent tasks forks child tasks
- Synchronization when tasks join into another task

[Image source: http://www.coopsoft.com/ar/ForkJoinArticle.html]
FJM problems

• Difficult to achieve data locality
  – tasks are free to access arbitrary data

• Fork and Join are not expressive enough for general synchronization and coordination between tasks

• Additional synchronization/coordination constructs
  – Phasers
  – Data Driven constructs
Phasers

• Support Collective and Point-to-Point synchronization

• Pros:
  – Can guarantee deadlock freedom

• Cons:
  – Phaser registration limits synchronization between arbitrary tasks
  – Blocking calls do not scale in current implementations when there are more tasks than workers
Data-Driven Futures (DDFs)

- Arbitrary producer-consumer relationships
- Single assignment from producer

Pros:
- Creation of task independent of data consumed
- Accesses to values inside the DDF are guaranteed to be race-free and deterministic

Cons:
- Strict ordering enforced for tasks waiting on multiple DDFs
public static void quicksort(final int[] inArr,
                                final DataDrivenFuture result) {

    if (inArr.length == 1) {
        result.put(inArr);
    } else {
        final int pivotIndex = selectPivot(inArr);
        final int pivotValue = inArr[pivotIndex];
        final DataDrivenFuture left = new DataDrivenFuture();
        async {
            final int[] lessThanArr = getLessThan(inArr, pivotValue);
            quicksort(lessThanArr, left);
        }
        final DataDrivenFuture right = new DataDrivenFuture();
        async {
            final int[] moreThanArr = getMoreThan(inArr, pivotValue);
            quicksort(moreThanArr, right);
        }
        final int[] center = getSumsTo(inArr, pivotValue);
        async await(left, right) {
            final int[] sorted = merge(left.get(), center, right.get());
            result.put(sorted);
        }
    }
}
Outline

• The Fork/Join Model
• The Actor Model
• The Hybrid Model
• Applications of the Hybrid Model
• Implementation and Experimental Results
The Actor Model

• A message-based concurrency model
• First defined in 1973 by Carl Hewitt
  – Research for Artificial Intelligence on Distributed machines
• Key concepts
  – An Actor encapsulates mutable state
  – Actors coordinate using *asynchronous* messaging
  – Non-deterministic ordering of messages
• new: actor instance has been created
• started: actor can receive and process messages sent to it
• terminated: actor will no longer process messages sent to it
Actors

mailbox

local state
process one message at a time
Actor - Interactions

- send messages to other actors
- create new actors
object ScalaActorPingPong {
  def run(numMsgs: Int): Unit = {
    val latch = new CountDownLatch(2)

    val pong = new ScPing(verbose, latch)
    val ping = new ScPing(numMsgs, pong, latch)
    ping.start
    pong.start
    ping ! ScStart

    latch.await()
  }
}

class ScPing(count: Int, pong: Actor, latch: CountDownLatch) extends Actor {
  def act() {
    var pingsLeft = count
    loop {
      react {
        case ScStart =>
          pong ! ScPing
          pingsLeft = pingsLeft - 1
        case ScSendPing =>
          pong ! ScPing
          pingsLeft = pingsLeft - 1
        case ScPing =>
          if (pingsLeft > 0)
            self ! ScSendPing
          else {
            pong ! ScStop
            latch.countDown()
            exit('stop)
          }
      }
    }
  }
}

scala code
Actor – *pro et contra*

• **Pros**
  – No data races
  – Easier to achieve data locality
  – Allows arbitrary coordination between actors

• **Cons**
  – Harder to implement synchronous messaging
  – Requires support for pattern matching on messages in implementations
  – Hard to implement concurrent objects since actors serialize message processing
Outline

• The Fork/Join Model
• The Actor Model
• The Hybrid Model
• Applications of the Hybrid Model
• Implementation and Experimental Results
The Hybrid Model

• Uses the Async/Finish model (AFM)
  – AFM is a generalization of the FJM

• Actors mapped onto the AFM
  – Mapping needs to be seamless
  – No additional constraints on actors

• Benefits
  – extend actor capabilities in the hybrid model
  – allow arbitrary coordination patterns between tasks
Actors and Async/Finish Tasks

• Actor creation:
  – synchronous operation (i.e. trivial)

• Actor termination:
  – synchronous operation (i.e. trivial)
  – all future send requests can be ignored synchronously
• Starting an Actor:
  – will determine the finish scope for the actor
  – actor will start processing messages asynchronously in this finish scope
  – needs to keep the finish scope “alive” to process any messages sent to it in the future
  • 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
    - no more child tasks spawned
Message Processing invariant

- *lingering* task provides the finish scope
- still need to enforce invariant of actor processing only one message at a time
- one-to-one mapping between a message and a task that processes it
- use Data-Driven Controls (DDCs)
Data-Driven Control

• has two fields
  – a data
  – an execution body

• dynamic single assignment of both fields

• task is scheduled when both data and body available

```
class DataDrivenControl
  data Some-Message
  body Some-Runnable
```
• actor mailbox is a concurrent linked-list of DDCs
• DDC tasks inherit finish scope from the *lingering* task
Actors mapped to AFM

- Asynchronous messaging handled
- One message processed at a time invariant preserved
- Additional constructs used
  - lingering tasks
  - data-driven controls
- No extra constraints placed on the Actors
- Benefits:
  - easier termination detection
  - parallelize actors
Termination detection in actor programs

• Two existing techniques
  – users explicitly manage blocking constructs
  – detect quiescence

• AFM mapping makes it easy
  – wrap actors in a finish scope
  – finish scope is blocked under all async spawned inside it have not terminated
    • actor alive $\rightarrow$ lingering task pending
    • actor terminated $\rightarrow$ lingering task complete
Hybrid Actor – PingPong

```scala
object LightweightActorPingPong {
  def run(numMsgs: Int): Unit = {
    finish {
      val pong = new LwPingActor()
      val ping = new LwPingActor(numMsgs, pong)
      ping.start
      pong.start
      ping ! LwStart
    }
  }
}

class LwPingActor extends HabaneroReactor {
  private var pingsLeft = count

  override def behavior() = {
    case LwStart =>
      pong ! LwPing(self)
      pingsLeft = pingsLeft - 1
    case LwSendPing =>
      pong ! LwPing(self)
      pingsLeft = pingsLeft - 1
    case LwPong(sender) =>
      if (pingsLeft > 0)
        self ! LwSendPing
      else {
        pong ! LwStop
        exit()
      }
  }
}
```

habanero-scala code
• Traditionally actor message processing (MP) has been sequential

• Under the AFM, we can use one of two techniques to parallelize the MP
  – Use finish construct in MP body and spawn child tasks (asyncs)
  – allow *escaping* asyncs inside MP body

  • **WAIT!** What about the single message processing invariant?
  • use pause and resume
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
New constructs in Hybrid model

- Event-driven tasks in AFM
- Non-blocking receives for actors
- Stateless actors
Event-Driven Tasks

- Actors are AFM tasks with continuations
- Actors (tasks) can resume continuations when they receive messages
- Tasks can coordinate by messaging each other
class QuicksortActor(parent: QuicksortActor,
    positionRelativeToParent: Position) extends HabaneroReactor {

  private val selfActor = this
  var result: ListBuffer[Int] = null
  private var numFragments = 0

  def notifyParentAndTerminate() = {
    if (parent ne null) parent ! Result(result, positionRelativeToParent)
    exit()
  }

  override def behavior() = {
    case Sort(data) =>
      val dataLength: Int = data.length
      if (dataLength < QuickSortConfig.CUTOFF) {
        result = quicksortSeq(data)
        notifyParentAndTerminate()
      } else {
        val pivot = data(dataLength / 2)
        async {
          val leftUnsorted = filterLessThan(data, pivot)
          val leftActor = new QuicksortActor(selfActor, LEFT)
          leftActor.start(); leftActor ! Sort(leftUnsorted)
        }
        async { /* similar code for right fragment */ }
        result = filterEqualsTo(data, pivot)
        numFragments += 1
      }
    case Result(data, position) =>
      if (position eq LEFT) result = data ++ result
      else if (position eq RIGHT) result = result ++ data
      numFragments += 1
      if (numFragments == 3) notifyParentAndTerminate()
  }
}
Non-blocking receives

- Simulates synchronous communication **without** blocking

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

habanero-scala code
Stateless Actors

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

```scala
class StatelessActor() extends ParallelActor {
  override def behavior() = {
    case msg: SomeMessage ⇒
      async {
        // process the current message
      }
      if (enoughMessagesProcessed) {
        exit()
      }
      // return immediately to be ready to process the next message
  }
}
```
Hybrid Model – *pro et contra*

• **Pros**
  – easier to achieve data locality using places
  – provides new coordination construct (actors) for arbitrary computation DAGs in AFM style
  – actors seamlessly interact with any of the other AFM compliant constructs (DDF, Phaser, etc.)

• **Cons**
  – possible data-races inside actors
  – all started actors need to be explicitly terminated
Outline

• The Fork/Join Model
• The Actor Model
• The Hybrid Model
• Applications of the Hybrid Model
• Implementation and Experimental Results
Hybrid Model – Applications

• Multiple Producer-Consumer with Bounded Buffer
  – producers, consumers and buffer are all actors
  – producers and consumer bodies can be parallelized
  – no data-races in the buffer as only one message processed at a time
Hybrid Model – Applications...

- Pipelined Parallelism
  - natural fit with the AM since each stage can be represented as an actor
  - single message processing
  - stages however need to ensure ordering of messages while processing them
  - introduce parallelism within the stages to reduce effects of slowest stage of pipeline
  - e.g. Sieve of Eratosthenes
Hybrid Model – Applications...

• Speculative Parallelization
  – common while processing data structures such as trees and graphs
  – each node represented as an actor
  – nodes can coordinate with other nodes for dependences but execute in parallel when no dependences exist
  – hybrid model can be used to exploit the parallelism inside the actors
    – e.g. Online (Hierarchical) Facility Location
Outline

• The Fork/Join Model
• The Actor Model
• The Hybrid Model
• Applications of the Hybrid Model
• Implementation and Experimental Results
Habanero-Scala

• Reference implementation of the hybrid model
• Scala is the host language
  – DSL features mean no new compiler required
  – runs on the JVM like Habanero-Java (HJ)
    • use library approach to port HJ constructs
    • use Scala DSL to retain close to HJ syntax
      – pattern matching constructs allows elegant support for actors
• Supports finish, async, futures, DDF, Phasers,...
Habanero-Scala Actors

• heavy actors
  – standard Scala actors extended to fit the hybrid model
  – no support for pause/resume
  – heavy as standard Scala actors use exceptions for control flow

• light actors
  – support pause resume (thus non-blocking receives)
  – use DDCs for control flow
  – supports become/unbecome operations that allow actor to change behavior
Experimental Setup

• Intel Xeon 2.4GHz system
• 16-core (quad-socket, quad-core per socket)
• 32 GB memory, running Red Hat Linux (RHEL 5)
• Sun Hotspot JDK 1.6
• Scala version 2.9.1.final
• latest versions of Habanero-Java and Habanero-Scala from Rice subversion repository
• geometric mean of best eight out of ten runs in the same JVM instance reported
Ping-Pong Benchmark

- measures raw message throughput
- Jetlang fastest, provides a low-level messaging API
- HS Light actor faster than standard Scala actors: no exceptions
Chameneos Benchmark

- measures cost of synchronization
- Jetlang again fastest
- HS Light actor faster than standard Scala actors
• Hybrid solution fastest, up to 22% faster than pure Actor solution
• Hybrid faster than DDFs for larger arrays as evaluation from partial results gets more profitable
• Habanero Isolation based solution does not scale
Sieve of Eratosthenes

- Phaser solution fastest: tuned to not create more tasks than workers
- Hybrid solution up to 10% faster than Light actor solution
- Hybrid solution faster than Jetlang solution
- Pause-Resume faster than Finish version
- Heavy actor faster than Scala version: thread binding benefits
Hierarchical Facility Location

- Larger alpha → more customers to process while creating children → more benefits of parallelism from hybrid model
- Hybrid solution fastest, up to 11% faster than pure Actor solution
- Heavy actors faster than Scala actors: thread binding
Contributions

• A hybrid programming model that integrates the Fork/Join model and the Actor model
• An implementation: Habanero-Scala
  • the Actor model using data-driven constructs in the Async/Finish model
  • the hybrid programming model supporting async/finish/… and pause/resume
• A study of application characteristics that are amenable to being more efficiently solved using the hybrid model compared to the FJM or AM
Future work

• Batch message processing, as in Jetlang, to avoid extraneous creation of tasks
• Use the Hybrid model to port Async Finish model constructs to a distributed memory system
Acknowledgments

• Vivek Sarkar
• Rest of the Thesis Committee
  – Robert S. Cartwright Jr
  – Swarat Chaudhuri
• Habanero Group
  – Vincent Cavé
  – Dragoş Sbîrlea
  – Sağnak Taşırlar
Thank you!

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