Integrating Task Parallelism with Actors

Shams Imam
Rice University
shams@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

Abstract

This paper introduces a unified concurrent programming model combining the previously developed Actor Model (AM) and the task-parallel Async-Finish Model (AFM). With the advent of multi-core computers, there is a renewed interest in programming models that can support a wide range of parallel programming patterns. The proposed unified model shows how the divide-and-conquer approach of the AFM and the no-shared mutable state and event-driven philosophy of the AM can be combined to solve certain classes of problems more efficiently and productively than either of the aforementioned models individually. The unified model adds actor creation and coordination to the AFM, while also enabling parallelization within actors. This paper describes two implementations of the unified model as extensions of Habanero-Java and Habanero-Scala. The unified model adds to the foundations of parallel programs, and to the tools available for the programmer to aid in productivity and performance while developing parallel software.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Design, Languages, Performance

Keywords Parallel Programming, Actor Model, Fork-Join Model, Async-Finish Model, Habanero-Java, Habanero-Scala

1. Introduction

Current mainstream programming languages provide limited support for expressing parallelism. Programmers need parallel programming models and constructs that can productively support a wide range of parallel programming patterns. This has led to a renewed interest in parallel programming models in the research community. Programs exhibit varying degrees of task, data, and pipeline parallelism [9] and extensions thereof e.g., event-driven parallelism as an extension of pipeline parallelism. In this paper, we focus on two such models:

- The Async-Finish Model (AFM), as exemplified by the `async` and `finish` constructs [4, 5] and the data-driven future extension [28], which is well-suited to exploit task parallelism in divide-and-conquer style and loop-style programs.
- The Actor Model (AM) which promotes the *no-shared mutable state* and an event-driven philosophy.

We introduce a unified parallel programming model that integrates the previously developed Async-Finish Model [5] and Actor Model [1, 12]. This integration not only adds actors as a new coordination construct in the AFM, but also enables parallelization of message-processing within actors[1]. It also simplifies detecting termination and managing synchronous operations in actors. We developed two reference implementations of this unified model by extending Habanero-Java (HJ) [4] and Habanero-Scala (HS) [14][2]. Both HJ and HS include an implementation of *unified* actors using data-driven controls (explained in Section 6.2.1) which we call *light* actors. Habanero-Scala also includes a *heavy* actor implementation that extends the standard Scala actor library [10] which uses exceptions for control flow. The proposed unified model shows how the AFM and the AM can be combined to solve certain classes of problems more productively than either of the aforementioned models individually. In our performance evaluation, we include a summary of application characteristics that can be more efficiently solved using the unified model compared to the AFM or AM and show that benchmarks exhibiting such characteristics can execute up to 30% faster using constructs from the unified model in our implementations relative to the AFM or AM.

The paper is organized as follows: in Section 2 we give a brief description of the AFM and the AM and some limita-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10. . . $15.00

[1] This subsumes the parallelization (while processing different messages) offered by the `become` primitive [1] and enables parallelization while processing a single message in actors.

[2] Previous versions of HJ and HS supported only the `async-finish` style computations without support for actors.

tions of the two models. Section 3 summarizes the syntax of some of the parallel constructs from these models used in the rest of the paper. We introduce the proposed unified model in Section 4, which defines how actors and `async-finish` tasks can be integrated. Section 5 presents some of the new capabilities in the unified model including parallelization of message processing. In Section 6, we describe our reference implementations and compare them with implementations of other JVM based actor frameworks in Section 7. Section 8 discusses related work and we summarize our conclusions and future work in Section 9.

2. Background

2.1 The Async-Finish Model (AFM)

The AFM is a task parallel model and a variant of the Fork-Join Model. The central features of any AFM implementation on multicore architectures include the abilities to create lightweight tasks and to efficiently manage the synchronization constraints among tasks. In the AFM, a parent task can fork (`async`) multiple child tasks which can execute in parallel. In addition, these child tasks can recursively fork even more tasks. A parent/ancestor task can selectively join (`finish`) on a subset of child/descendent tasks. The task executing the join has to wait for all tasks created in the `finish` scope to terminate before it can proceed. This is the primary form of synchronization among tasks in the AFM. The child tasks are said to execute in the `finish` scope represented by the aforementioned join. In the AFM, each task is guaranteed to have a unique dynamic Immediately Enclosing Finish (IEF) which may be the implicit `finish` construct for the entire program. An example AFM program is discussed later in Figure 2.

2.1.1 Desirable Properties

Async-finish style computations are guaranteed to be deadlock free [5]. In addition, in the absence of data races, these programs also have the extremely desirable property that they are deterministic [21]. Two well-known manifestations of the AFM can be found in the X10 [5] and Habanero-Java [4] languages. The new task-parallel constructs in OpenMP 3.0 [19] also represent a variant of the AFM.

2.2 The Actor Model

The Actor Model (AM) was first defined in 1973 by Carl Hewitt et al. during their research on Artificial Intelligent (AI) agents [12]. It was designed to address the problems that arise while writing distributed applications. Further work by Henry Baker [13], Gul Agha [1], and others added to the theoretical development of the AM. The AM is different from task parallelism in that it is primarily an asynchronous message-based concurrency model. An actor is the central entity in the AM that defines how computation proceeds. The key idea is to encapsulate mutable state and use asynchronous messaging to coordinate activities among actors.

An actor is defined as an object that has the capability to process incoming messages. It has a well-defined life cycle and restrictions on the actions it performs in the different states. During its life cycle an actor is in one of the following three states:

- *new*: An instance of the actor has been created; however, the actor is not yet ready to receive or process messages.
- *started*: An actor moves to this state from the *new* state when it has been started using the `start` operation. It can now receive asynchronous messages and process them one at a time. While processing a message, the actor should continually receive any messages sent to it without blocking the sender.
- *terminated*: The actor moves to this state from the *started* state when it has been terminated and will not process any messages in its mailbox or new messages sent to it. An actor signals termination by using the `exit` operation on itself while processing some message.

Typically, the actor has a *mailbox* to store its incoming messages. Other actors act as producers for messages that go into the mailbox. An actor also maintains local state which is initialized during creation. After creation, the actor is only allowed to update its local state using data from the messages it receives and from the intermediate results it computes while processing the message. The actor is restricted to process at most one message at a time. There is no restriction on the order in which the actor decides to process incoming messages, thereby leading to non-determinism in actor systems. As an actor processes a message, it is allowed to change its state and behavior affecting how it processes the subsequent messages. While processing a message, an actor may perform a finite combination of the following steps:

1. Asynchronously send a message to another actor whose address is known;
2. Create a new actor providing all the parameters required for initialization;
3. Become another actor, which specifies the replacement behavior to use while processing the subsequent messages [20].

2.2.1 Desirable Properties

The only way an actor conveys its internal state to other actors is explicitly via messages and responses to messages. This property obtains benefits similar to encapsulation in object-oriented programming and encourages modularity. The encapsulation of the local state also helps prevent data races because only the actor can modify its local state. Due to the asynchronous mode of communication, the lack of restriction on the order of processing messages sent from different actors, and the absence of synchronization via encapsulation of local data, actors expose inherent concurrency and can work in parallel with other actors.

2.3 Limitations of the AFM and the AM

The AFM is well-suited to exploit parallelism from deterministic interaction patterns; however many algorithms and applications involve interaction patterns that are non-deterministic. For example, in producer-consumer applications the production or consumption of an individual item may exhibit deterministic parallelism; however interactions between multiple producers and consumers are non-deterministic in general. Another example is Quicksort which exhibits both deterministic (creating the left and right fragments around the partition element) and nondeterministic (availability of sorted left and right fragments) forms of task parallelism. General AFM implementations (e.g. CnC [3], DDFs [28]) cannot exploit the inherent non-determinism in the arrival of results from the subtasks (see Figure 3 for an example). The AM can exploit this nondeterminism in the arrival of results from subtasks as well as guarantee synchronized access while computing partial results. The unified model can be used to exploit both forms of parallelism fairly simply by creating relatively inexpensive `async`s when required and using actors to manage the non-determinism in the availability of partial results.

In the AM, simulating *non-blocking* synchronous replies requires some amount of effort mainly due to the lack of a guarantee of when a given message will be processed and the need to temporarily disable processing of other messages at the actor waiting for the reply. Similarly, achieving global consensus among a group of actors is a non-trivial task; for example, one approach requires implementing *coordinator* actors (which can become a sequential bottleneck) for tracking the flow of messages in the *coordinated* actors. Such coordination patterns are simple in the AFM, for example by using `finish` to wrap `async`s or by using `phaser`s [26] as communication barriers. Additionally, these AFM constructs are implemented in a scalable manner to avoid bottlenecks.

As another example, the pipeline pattern is a natural fit with the AM since each stage can be represented as an actor. The single-message-processing rule ensures that each stage/actor processes one message at a time before handing it off to the next actor in the pipeline. However, the amount of concurrency (parallelism) in a full pipeline is limited by the number of stages. One way to increase the available parallelism, apart from creating more stages, is to introduce parallelism within the stage, achieved in the unified model by spawning `async`s. The slowest stage is usually the bottleneck in a pipeline, increasing the parallelism can help speed up the slowest stage in the pipeline and improve performance. An example for such a pipeline is the Filterbank benchmark (from StreamIt [30]) in which the finite impulse response (FIR) stage is the slowest stage and is discussed further in Section 3.4.

3. Overview of Parallel Constructs

In this section, we briefly summarize the syntax (in Habanero-Java and Habanero-Scala) of the parallel constructs we use in the following sections to explain various features of the unified model. The three main constructs are:

- `async` and `finish`,
- data-driven futures, and
- actors.

In the unified model, each of these constructs has first-class status and can be arbitrarily composed with the others. We end this section with a motivating example displaying a composition of these constructs.

3.1 `async` and `finish`

In the unified model (and the AFM), tasks are created at fork points using the `async` keyword. The statement `async` ⟨*stmt*⟩ causes the parent task to create a new child task to execute ⟨*stmt*⟩ (logically) in parallel with the parent task [4]. The scheduling of tasks created by `async`s on actual threads is done by the runtime and is transparent to the user and to the tasks in the program.

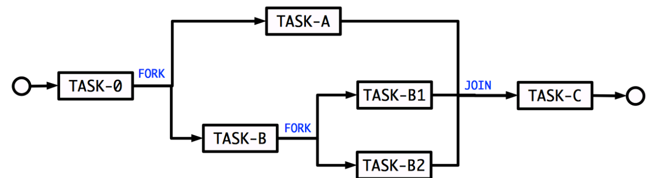


Figure 1: Fork-Join Parallelism achieved by forking new tasks and joining before proceeding. Note that until all forked tasks (Task A, Task B, Task B1, and Task B2) reach the join point, Task C cannot be executed. [source=http://www.coopsoft.com/ar/ForkJoinArticle.html].

```
1 /*** Habanero-Java code ***/
2 public class ForkJoinPrimer {
3   // An implicit global finish wraps main() which
4   // must wait for all nested tasks to terminate
5   public static void main(String args[]) {
6     System.out.println("Task O"); // Task-O
7     finish {
8       async { // Task-A
9         System.out.println("Task A");
10      }
11      async { // Task-B
12        System.out.println("Task B");
13        async { // Task-B1 created by Task-B
14          System.out.println("Task B1");
15        }
16        async { // Task-B2 created by Task-B
17          System.out.println("Task B2");
18    } } }// Wait for tasks A, B, B1 and B2 to finish
19    System.out.println("Task C"); // Task-C
20 } }
```

Figure 2: HJ version of the Fork-Join program from Figure 1

The `finish` keyword is used to represent a join operation. The task executing `finish` ⟨*stmt*⟩ has to wait for all child tasks created inside ⟨*stmt*⟩ to terminate before it can proceed. A program is allowed to terminate when all tasks nested inside the global finish terminate. Figure 2 shows an example HJ program using the `async` and `finish` constructs to represent the fork and join constraints of tasks in Figure 1.

3.2 Data-Driven Futures (DDFs)

DDFs are an extension to futures to support the dataflow model [28]. They support a single assignment property in which each DDF must have at most one producer and any `async` can register on a DDF as a consumer causing the execution of the `async` to be delayed until a value becomes available in the DDF. There are three main operations allowed on a DDF:

- `put(some-value)`: associates a value with the DDF. Only a single `put()` is allowed on the DDF during the execution of the program.
- `await()`: used by `async`s to delay their execution until some other task has `put()` a value into the DDF.
- `get()`: used to retrieve the value stored in the DDF. It can legally be invoked by a task that was previously `await`ing on the DDF. This guarantees that if such a task is now executing, there was already a `put()` and the DDF is now associated with a value.

The exact syntax for an `async` waiting on DDFs is as follows: `asyncAwait(`*`ddf1, ..., ddfN`*`)` ⟨*stmt*⟩. Figure 3 shows the implementation of Quicksort using `async`s and DDFs.

3.3 Actors

In the unified model, actors are defined by extending an actor base class. Concrete sub-classes are required to implement the method used to process messages[3]. Actors are like other objects and can be created by a `new` operation on concrete classes. An actor is activated by the `start()` method, after which the runtime ensures that the actor's message processing method is called for each message sent to the actor's mailbox. The actor can terminate itself by calling the `exit()` method while processing a message. Messages can be sent to actors from actor code or non-actor code by invoking the actor's `send()` method using a call as follows, `someActor.send(aMessage)`. A `send()` operation is non-blocking and the recipient actor processes the message asynchronously. As in the AM, there are no guarantees on the order of message delivery in the unified model. However, in our implementations (HJ and HS) the runtime preserves the order of messages with the same sender task and receiver actor, but messages from different senders may be interleaved

[3] This method is named `process()` in HJ *light* actors while it is named `act()` and `behavior()` in HS *heavy* and *light* actors, respectively.

```
1 /*** Habanero-Scala code ***/
2 object QuicksortApp extends HabaneroApp {
3   val input: ListBuffer[Int] = ...
4   val resDdf = ddf[ListBuffer[Int]]()
5   finish {
6     asyncAwait(resultDdf) {
7       val sortedList = resDdf.get()
8       ...
9     }
10    quicksort(input, resDdf)
11  }

13  private def quicksort(data, resultDdf) = {
14    if (data.length < 1) {
15      resultDdf.put(data)
16    } else {
17      val pivot = ...
18      val (ddfL, ddfR) = (ddf(), ddf())
19      async { // asynchronously sort left fragment
20        quicksort(filter(<, data, pivot), ddfL)
21      }
22      async { // asynchronously sort right fragment
23        quicksort(filter(>, data, pivot), ddfR)
24      }
25      val eqs = filter(==, data, pivot)
26      asyncAwait(ddfL, ddfR) {
27        // wait for both left and right to complete
28        val res = ddfL.get() ++ eqs ++ ddfR.get()
29        resultDdf.put(res)
30      } } } }
```

Figure 3: HS version of Quicksort using DDFs. The `async` at line 26 is triggered only after a value is `put` into both the DDFs (at line 15 or line 29 by the recursive calls) it is awaiting on. As mentioned in Section 2.3, we cannot nondeterministically precompute partial results depending on whether the left or right fragments are available early.

in an arbitrary order. This is similar to the message ordering provided by ABCL [35]. Figure 4 shows a `HelloWorld` example using HJ actors.

```
1 /*** Habanero-Java code ***/
2 public class HelloWorld {
3   public static void main(final String[] args) {
4    final UnifiedActor printActor = new PrintActor();
5    finish {
6      printActor.start();
7      printActor.send("Hello");
8      printActor.send("World");
9      actor.send(PrintActor.STOP_MSG);
10   } // wait until actor terminates
11   System.out.println("PrintActor has terminated");
12 } }

14 class PrintActor extends UnifiedActor {
15   static final Object STOP_MSG = new Object();
16   protected void process(final Object msg) {
17     if (STOP_MSG.equals(msg)) {
18       exit();
19     } else {
20       System.out.println(msg);
21 } } }
```

Figure 4: `HelloWorld` using HJ actors. We are guaranteed ordered sends, i.e. though `Hello` and `World` will be processed asynchronously, they will be processed in that order.

```
/*** Habanero-Scala code ***/
object FilterBankApp extends HabaneroApp {
  finish {
    ...
    val sampler = ...
    val fir = new FirFilter(..., sampler).start()
    ...
} }
class FirFilter(..., nextStage: UnifiedActor)
    extends UnifiedActor {
  ...
  def behavior() = {
    case FirItemMessage(value, coeffs) =>
      ...
      val numHelpers = ... // number of helper tasks
      // allocate the DDFs to use
      val stores = Array.tabulate(numHelpers) {
        index => ddf[Double]() }
      finish {
        // compute the sum using divide-and-conquer
        (0 until numHelpers) foreach { helperId =>
          val myDdf = stores(helperId)
          async {
            val (start, end) = ...
            var sum: Double = 0.0
            start until end foreach { index =>
              sum += buffer(index) * coeffs(index)
            }
            myDdf.put(sum)
        } }
        ...
        // wait for the partial results
        asyncAwait(stores) {
          // propagate the sum down the pipeline
          val sum = stores.foldLeft(0.0) {
            (acc, loopDdf) => acc + loopDdf.get()
          }
          nextStage.send(DataItemMessage(sum))
      } }
    case ... => ...
} }
```

Figure 5: HS version of the FIR stage in the Filter Bank pipeline. In this example, the computation of the dot product between the coefficients and a local buffer has been parallelized to speedup this stage in the application.

3.4 Composing the constructs

We now discuss an example application in which these constructs can be composed in ways that cannot easily be achieved with current programming models. This example, shown in Figure 5, is the FIR stage of the Filter Bank application. The application represents a pipeline and each of the stages can be represented using actors. The FIR stage is the slowest stage in the pipeline and limits the pipeline rate, the performance can be improved by speeding up this stage by parallelizing it. We do so by partitioning the computation of the dot product using `async`s (line 23 in the example). Each `async` computes the dot product of a partition before writing back the result into its assigned DDF (line 29), avoiding the possibility of a data races. The `async` at line 33 `await`s on the results in the DDFs to be available before computing the final result and propagating the value to the next stage in pipeline. All the spawned `async`s between line 22 and line 39 join to their IEF, the `finish` at line 19. This ensures the FIR stage does not start processing the next message until it has completed processing the current message and has propagated values to the next stage in the pipeline. While such parallelism in the FIR stage could be simulated in the AM, it requires distributing the logic among multiple actors and significantly complicates the code as the actor representing the FIR stage needs to maintain additional state to track the arrival of partial results and maintain the order of values it passes along the pipeline (the AM does not guarantee the order in which messages will be serviced). In addition, there will be overhead associated with the data copying required to send the data fragments to the helper actors. Comparatively, the use of `async`s and `finish` avoids such drawbacks making the code easier to maintain and helping with productivity.

```
/*** 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 ... => ...
} } } }
```

Figure 6: HS version of the FIR stage in the Filter Bank pipeline. In this example, the computation of the dot product between the coefficients and a local buffer has been parallelized to speedup this stage in the application.

4. The Unified Model

Although both the AFM and AM have existed as independent parallel programming models for a while now, we are unaware of previous efforts to systematically combine these two models. We integrate the AFM and the AM so as to get the benefits of actor coordination construct in the AFM and also of parallelizing message-processing within actors. In this section, we describe how actor message processing and `async-finish` tasks can be integrated in the unified model and the benefits of this integration.

4.1 Coordination of Actors in the Unified Model

Integrating actors and tasks requires understanding how the actor life cycle interacts with task creation and termination events. The creation of an actor is a simple operation and can be performed synchronously inside the task executing the action. Similarly, terminating the actor is a synchronous

operation that can be effected by an actor on itself while it is processing a message. Once an actor enters the terminated state, it avoids processing any messages sent to it without blocking the sender (such messages are effectively no-ops and do not need to be placed in the mailbox). Since tasks always execute inside an enclosing finish scope, both these operations are easily mapped to the AFM. The more interesting case is handling the actions of the actor while it is active in the *started* state and processing messages.

Starting an actor activates it and allows it to continuously receive messages and to process these messages one at a time. This operation can hence be represented as an asynchronous task whose body is a long-running loop which keeps processing one message at a time from its mailbox until it is terminated. This newly spawned asynchronous task inherits the IEF (as per normal `async-finish` semantics) of the task which started the actor. The long-running loop in the task enforces the IEF to block until all actors started inside it terminate. In Section 6.1 we present the *lingering* task technique which avoids having to explicitly use the long-running loop mentioned above (with its accompanying overheads when the mailbox is empty).

```
1 /*** Habanero-Java code ***/
2 public class HelloWorld2 {
3   public static void main(final String[] args) {
4     final UnifiedActor printActor = new PrintActor()↩
        ;
5     async {
6       finish { // F1, IEF for printActor
7         ...
8         printActor.start(); // similar to an async
9         ...
10      }
11      System.out.println("PrintActor terminated");
12    }
13    async {
14      finish { // F2
15        ...
16        // task T2
17        printActor.send("Hello");
18        printActor.send("World");
19        printActor.send(PrintActor.STOP_MSG);
20        ...
21      }
22      System.out.println("Done sending messages");
23 } } }
```

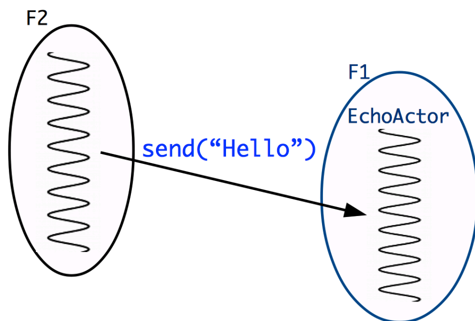


Figure 7: `HelloWorld` example with `printActor`, executing in `finish` scope `F1`, receiving messages from a different `finish` scope, `F2`.

We now discuss the actions to be performed after the actor has started and is receiving and processing messages. By definition, actors process the messages they receive asynchronously. This translates to the creation of a new task that processes the message and runs in parallel with the task that initiated the send of the message. Under normal `async-finish` semantics, this means that both these tasks share the same IEF. Now, consider the case where an actor is receiving messages from a task/actor executing in a different IEF, as shown in Figure 7. Under normal `async-finish` semantics, when `T2` sends a message to `printActor` it places the message in `printActor`'s mailbox and creates a new task, say `T3`, to process this message. This causes `F2` to unnecessarily (and incorrectly) block until `T3` completes. Since the message will end up in `printActor`'s mailbox, the processing of the message is done by `printActor` and semantically `T3` should have `F1` as its IEF as opposed to `F2`. Hence, when `T2` sends a message to `printActor`, the new asynchronous task must be spawned in the `finish` scope of `printActor`. In the unified model, this generalizes to all asynchronous tasks spawned to process a message inheriting the IEF of the recipient actor. Note that this ability to attach a different `finish` scope while spawning a task is a feature of the unified model which is unavailable in the general AFM. The use of newly spawned tasks to send messages is also facilitated by the fact that no message-ordering restrictions apply in the AM and these spawned tasks can thus be executed in any order. In addition, since the new task inherits the `finish` scope of the recipient actor, it allows the sender to be any arbitrary task executing under the unified model.

Mapping the entire life cycle of actors into the AFM provides a clean and transparent mechanism to detect the termination of actors. Some actor implementations on the Java VM (JVM) (e.g., Scala Actors library [10], Kilim [27], Jetlang [23]) require the user to write explicit code to detect whether an actor has terminated before proceeding with the rest of the code in the control flow. A common pattern is to explicitly use countdown latches and wait on the latch until the count reaches zero. In programs written using the AFM, a similar effect is achieved by joining tasks inside their `finish` scope without the programmer having to worry about low-level synchronization constructs such as latches. Consequently, mapping actors to a `finish` scope provides a transparent mechanism to detect actor termination and relieves the user from writing boiler plate code.

Figure 8 shows a simple `PingPong` example using the *unified* actors and the `finish` construct to detect termination easily. The Scala version (8a) needs to maintain a `latch` and pass it around to the different actors, while the main thread waits on the latch. In addition, actors need additional logic to decrement the count on the latch. The use of such shared latches breaks the pure actor model by not encapsulating state. On the other hand, the unified model example (8b) benefits from the `finish` construct. Terminating the actor

```
/*** Scala code ***/
object ScalaActorApp extends App {
  val latch = new CountDownLatch(2)
  val pong = new PongActor(latch).start()
  val ping = new PingActor(msgs, pong, latch).start()
  ping ! StartMsg
  latch.await()
  println("Both actors terminated")
}
// class PingActor not displayed
class PongActor(latch: CountDownLatch) extends Actor {
  var pongCount = 0
  def act() {
    loop { react {
        case PingMessage =>
          sender ! PongMessage
          pongCount = pongCount + 1
        case StopMessage =>
          latch.countDown()
          exit('stop)
} } } }
```

(a) Actor Model (Scala)

```
/*** Habanero-Scala code ***/
object LightActorApp extends HabaneroApp {
  finish {
    val pong = new PongActor().start()
    val ping = new PingActor(msgs, pong).start()
    ping.send(StartMessage())
  }
  println("Both actors terminated")
}
// class PingActor not displayed
class PongActor extends UnifiedActor {
  var pongCount = 0
  override def behavior() = {
    case PingMessage(sender) =>
      sender.send(PongMessage())
      pongCount = pongCount + 1
    case StopMessage =>
      exit()
} }
```

(b) Unified Model (Habanero-Scala)

Figure 8: Implicit actor termination detection using `finish` in the unified model (Figure 8b). Note the elegant syntax for Habanero-Scala with pattern matching as opposed to the use of `instanceof` which are required in Habanero-Java.

using the call to `exit` notifies the IEF that the actor has terminated and the statements following the `finish` are free to proceed (when all other spawned tasks inside the `finish` scope have also completed). The actor no longer worries about the cross-cutting concern of invoking methods on a latch.

4.2 Desirable Properties

Actors in the unified model continue to encapsulate their local state and process one message at a time. Thus the benefits of modularity are still preserved. Similarly, the data locality properties of the AM continue to hold. Actors also introduce a means of a new coordination construct in the AFM in addition to the existing constructs such as futures, DDFs, and phasers. With actors inside the AFM, it is now possible to create arbitrary computation DAGs impossible in the pure AFM. Since actors have been integrated into the AFM, actors can co-exist with any of the other constructs in the AFM, and they can be arbitrarily nested. The implementation of the `receive` operation using DDFs (mentioned in Section 5.2) is an example of this.

5. New Capabilities in the Unified Model

With the unified model in place, there are a number of constructs that can now be supported in the AFM. The key to each of these constructs is the ability to reason about the enclosing `finish` under which the actors execute. Some of these constructs are presented below.

5.1 Parallelization inside Actors

The requirement that the actor must process at most one message at a time is often misunderstood to mean that the processing must be done via sequential execution. In fact, there can be parallelism exposed even while processing messages as long as the invariant of processing at most one message at a time is maintained. One advantage of integrating the AFM and the AM is that it allows us to use `async-finish` constructs inside the message-processing code to expose this parallelism. There are two main ways in which this is achieved, discussed below:

- Using `finish` constructs during message processing and
- Allowing escaping `async` tasks.

5.1.1 Using `finish` during message processing

The traditional actor model already ensures that the actor processes one message at a time. Since no additional restrictions are placed on the message-processing body (MPB), we can achieve parallelism by creating new `async-finish` constructs inside the MPB. In this approach, we can spawn new tasks to achieve the parallelism at the cost of blocking the original message-processing task at the new `finish`. Since the main message-processing task only returns after all spawned tasks inside the `finish` have completed, the invariant that only one message is processed at a time is maintained. Figure 9 shows an example code snippet that achieves this. Note that there is no restriction on the constructs used inside the newly constructed `finish`. As such, all the `async-finish` compliant coordination constructs can also be used.

5.1.2 Allowing escaping `async`s during message processing

Requiring all spawned `async`s to be contained in a single MPB instance is too restrictive. This constraint can be relaxed based on the observation that the *at most one message-processing rule* is required to ensure there are no internal state changes of an actor being affected by two or more message-processing tasks of the same actor. As long as this rule is obeyed, *escaping* `async`s (tasks) can be allowed inside the MPB. We can achieve this invariant by introducing a *paused* state in the actor life cycle and by adding two new

```
/*** Habanero-Scala code ***/
class ParallelizedActor() extends UnifiedActor {
  override def behavior() = {
    case msg: SomeMessage =>
      // optional preprocessing of the message
      finish { // ensures spawned tasks complete
        async { ... /* processing in parallel */ }
        async { ... /* more parallel processing */ }
      }
      // optional post processing after finish
    ...
} }
```

Figure 9: An actor exploiting the `async-finish` parallelism inside actors message-processing body. The nested `finish` ensures no spawned tasks escape, thereby ensuring that an actor does not process multiple messages at a time.

operations: `pause` and `resume`. In the *paused* state, the actor is not processing any messages from its mailbox. The actor is simply idle as in the *new* state; however, the actor can continue receiving messages from other actors. The actor will resume processing its messages, at most one at a time, when it returns to the *started* state. The `pause` operation takes the actor from a *started* state to a *paused* state while the `resume` operation achieves the reverse. The actor is also allowed to terminate from the *paused* state using the `exit` operation. The `pause` and `resume` operations are similar to the `wait` and `notify` operations in Java threads for coordination. Similar to the restriction that thread coordination operations can only be executed by the thread owning the monitor, `pause` and `resume` operations on an actor can only be executed in tasks spawned within an actor either explicitly by the user or implicitly by the runtime to process messages (e.g., the MPB task). However, unlike the thread coordination operations neither the `pause` nor the `resume` operations are blocking, they only affect the internal state of the actor that coordinates when messages are processed from the actor's mailbox.

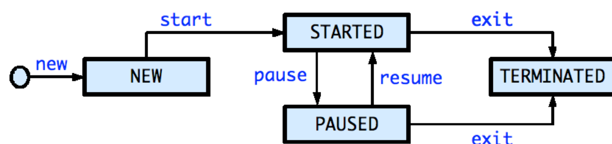


Figure 10: Actor life cycle extended with a *paused* state. The actor can now continually switch between the *started* and *paused* states using the `pause` and `resume` operations.

With the two new operations, we can now allow spawned tasks to escape the MPB task. These spawned tasks are safe to run in parallel with the next message-processing task of the same actor as long as they are not concurrently affecting the internal state of the actor. The actor can be suspended in a *paused* state while these spawned tasks are executing and can be signaled to resume processing messages once the spawned tasks determine they will no longer be modifying the internal state of the actor and hence not violating the one message-processing rule. Figure 11 shows an example in which the `pause` and `resume` operations are used to achieve parallelism inside the MPB while delaying the processing of the next message in the actor's mailbox.

```
/*** 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
      } ...
} }
```

Figure 11: An actor exploiting parallelism via `async`s while avoiding an enclosing `finish`. The `async`s escape the MPB, but the `pause` and `resume` operations control processing of subsequent messages by the actor.

Unfortunately, the ability to spawn new tasks inside the actor's MPB creates the potential to introduce data races, since multiple tasks can be working on the actor's local data. In fact, data races are also possible in AM implementations which do not guarantee data isolation. We plan on extending the Scalable Parallel Dynamic Datarace Detection (SPD3) algorithm [22] for the AFM to the unified model for data race detection. Introducing the `pause` and `resume` operations also increases the possibility of reaching deadlocks. If an actor is never resumed after it has been paused, the actor will never terminate and hence the IEF will block indefinitely. Like the AM, under the unified model it is required that every actor terminate, e.g., by a call to `exit`. Terminating actors explicitly is required so that the IEF for an actor does not block indefinitely.

5.2 Non-blocking `receive` operations

Implementing the *synchronous* `receive` operation[4] often involves blocking and can limit scalability in virtual machines that do not allow explicit call stack management and continuations. For example, the implementation of `receive` in the Scala actor library involves blocking the currently executing thread and degrades performance. The alternate approach requires use of exceptions to unwind the stack and maintain control flow, as in Scala's `react` construct, and is also relatively expensive.

With the support for `pause` and `resume`, the `receive` operation can now be implemented in the unified model without blocking threads or using exceptions. This requires support of the DDF coordination construct. DDFs allow the execution of the `async` to be delayed until a value is available in the DDF. A DDF can be passed along to the actor

[4] Synchronous receives are called now type messages in ABCL [35].

```
1 /*** Habanero-Scala code ***/
2 class ActorPerformingReceive extends UnifiedActor {
3   override def behavior() = {
4     case msg: SomeMessage =>
5       ...
6       val theDdf = ddf[ValueType]()
7       anotherActor ! new Message(theDdf)
8       pause() // delay processing next message
9       asyncAwait(theDdf) {
10        val responseVal = theDdf.get()
11        // process the current message
12        ...
13        resume() // enable next message processing
14      }
15      // return in paused state
16 } }
```

Figure 12: An actor in the unified model that uses Data-Driven Futures (DDFs) to perform the `receive` operation without blocking. The actor that processes the message needs to perform a `put` of a value on the DDF to trigger the waiting `async` (in `asyncAwait`). When the `async` is triggered, the actor processes the value in the DDF and performs the `resume` operation to continue processing subsequent messages.

which fills the result on the DDF when it is ready. Meanwhile the actor that sent the DDF can `pause` and create an `async` which waits for the DDF to be filled with a value and can `resume` itself. Figure 12 shows an example of a non-blocking `receive` implementation. This presents an instance of the sender and the recipient actors coordinating with each other without explicit message-passing and thus violates the pure AM. Non-blocking `receive`s present an excellent case in which constructs from the two different models, AFM and AM, can work together to ease the implementation of other nontrivial constructs. Figure 17 introduces syntactic sugar that can be used to implement the synchronous receive operation.

5.3 Stateless Actors

Stateless actors can process multiple messages simultaneously since they maintain no mutable internal state. It is easy to create such actors in the unified model using escaping `async`s as shown in Figure 13. There is no need to use the `pause` operation, and the escaping `async` tasks can process multiple messages to the same actor in parallel. Stateless actors can be used to implement concurrent reads in a data structure which would not be possible in most actor implementations since the message-processing would be completely serialized.

6. Implementation

We developed two implementations of the unified model described in Section 4, namely Habanero-Java (HJ) and Habanero-Scala (HS). We extended both HJ and HS to include the *unified* actor coordination construct. In our implementations, any of the AFM compliant constructs can be arbitrarily nested with these *unified* actors and vice versa. Both the actor implementations rely on the use of *lingering* tasks

```
1 /*** Habanero-Scala code ***/
2 class StatelessActor() extends ParallelActor {
3   override def behavior() = {
4     case msg: SomeMessage =>
5       async { processMessage(msg) }
6       if (enoughMessagesProcessed) { exit() }
7       // return immediately to process next message
8 } }
```

Figure 13: A simple stateless actor created using the unified model. The message processing body spawns a new task to process the current message and returns immediately to process the next message. Because the `async` tasks are allowed to escape, the actor may be processing multiple messages simultaneously.

to integrate actors into the AFM. We explain the *lingering* tasks technique before introducing the two implementations.

6.1 *Lingering* Tasks

Section 4.1 explained how to map actors to tasks. There starting an actor was likened to a long-running asynchronous task processing one message at a time. However, such a long-running task would waste resources as it would be involved in some sort of busy waiting mode until a message arrives. The purpose of this long-running task is to attach the actor's MPB to an IEF; a more efficient technique is to use a *lingering* task.

A *lingering* task is a task with an empty body that attaches itself to an IEF like a normal asynchronous task spawned inside a finish scope. Thus, the finish scope is aware of the existence of this task and will block until the task is scheduled and executed. However, the *lingering* task does not make itself available for scheduling immediately (unlike normal asynchronous tasks) and thus forces the IEF to block under the constraints of the AFM[5]. At some later point in time, the *lingering* task will be scheduled and executed, allowing the finish scope to complete execution and move ahead.

The *lingering* task provides a hook into its finish scope that may be used to spawn more tasks. All these spawned tasks execute under the same IEF as the *lingering* task. When a *unified* actor is started, a *lingering* task is created by the runtime and stored in the actor. This allows the actor to continue spawning subsequent tasks under the same IEF when it asynchronously processes messages sent to it. When the actor terminates, the runtime schedules the *lingering* task for execution. Once the *lingering* task has been scheduled, the actor stops creating any further asynchronous tasks realizing that the IEF may no longer be available to spawn tasks. This is consistent with the notion that termination of an actor is a stable property, and the actor is not allowed to process messages once it terminates. Any messages sent to a terminated actor are ignored and no task is created to process the message.

[5] A finish scope can only complete after all its transitively spawned tasks have completed.

6.2 Habanero-Java

Habanero-Java (HJ) already provides AFM constructs, we extended HJ with an actor coordination construct which we refer to as *light* actors. *Light* actors support all the features in the unified model discussed in Section 5. *Light* actors are started using a call to `start()` and the MPB is triggered only on the messages they receive. *Light* actors do not use exceptions to manage the control flow and require the user to implement a `process()` method to determine the steps to execute while processing a message. If any uncaught exception is thrown by `process()`, then the actor terminates itself by calling `exit()` and the exception is collected by its enclosing `finish` scope which then throws a *MultiException* [5]. An actor can explicitly terminate itself by calling `exit()` while processing a message.

The implementation of *light* actors relies on the use of *data-driven controls* to implement the mailbox. The mailbox supports a *push*-based implementation where `asyncs` are created without the runtime having to poll (i.e. *pull*-based) the actor's mailbox to decide when to launch an `async` to process messages.

6.2.1 Data-Driven Controls (DDCs)

A Data-Driven Control (DDC) lazily binds a value and a block of code called the execution body (EB). When both these are available a task that executes the EB using the value is scheduled. Both the value and the EB follow the dynamic single assignment property ensuring data-race freedom. Until both fields are available, the scheduler is unaware of the existence of the task. Figure 14 shows a simplified implementation of a DDC excluding synchronization constructs. The DDC may be implemented using an asynchronous or a synchronous scheduler. *Light* actors use both forms of DDCs: asynchronous execution of the task by involving the Habanero scheduler and synchronous execution of the EB.

DDCs differ from Taşırlar's DDFs [28] in that only a single task may be associated with a value at a time. DDFs apply the dynamic single assignment property only to the value and allow multiple tasks to be waiting for the value. In addition, the scheduler is aware of the existence of these data-driven tasks (DDTs) and causes the `finish` scopes of the tasks to block until the DDTs are scheduled. In contrast, with DDCs the scheduler is unaware of the existence of the task until it is scheduled, and only then will it schedule and execute the task. This may lead to issues with the `finish` scope of the activity in the asynchronous scheduler, but we will see below that coupling the DDC with the lifespan of the *lingering* task avoids the potential problem.

6.2.2 The Mailbox: Linked List of DDCs

The mailbox for the *light* actors is implemented as a linked list of DDCs (Figure 15). As messages are sent to the actor, the chain of DDCs are built with each message populating the value field of a DDC. The linked list is concurrent and

```
1 /*** Habanero-Java code ***/
2 class DataDrivenControl {
3   private ValueType value = null;
4   private ExecBody execBody = null;

6   void addValue(ValueType theValue) {
7     if (!valueAvailable()) {
8       value = theValue;
9       // resume awaiting task
10      if (execBody != null) {
11        execBody.scheduleWith(value);
12  } } }

14  void addResumable(ExecBody theBody) {
15    if (valueAvailable()) {
16      // value is available, execute immediately
17      theBody.scheduleWith(value);
18    } else {
19      // need to wait for the value
20      execBody = theBody;
21 } } }
```

Figure 14: Simplified implementation of a DDC not including synchronization constructs or validations. Both the value and the execution body can be lazily attached. The execution body determines whether scheduling happens synchronously or asynchronously.

multiple messages can be sent to an actor safely. *Light* actors guarantee that the order of the messages sent from the same actor will be preserved in the mailbox. No guarantee is provided for the order of messages in the mailbox for messages sent from different actors.

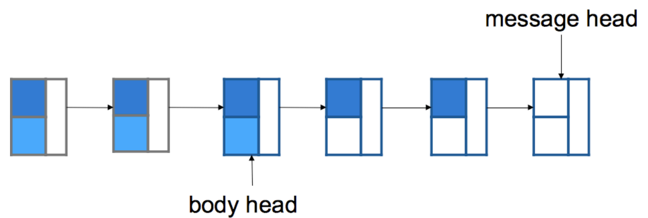


Figure 15: The actor mailbox is represented as a linked list of DDCs. The message head determines where the next message is stored, while the body head determines which message is being processed currently.

Once the actor has started (via the call to `start()`), it proceeds to traverse the messages in the mailbox, one at a time, lazily attaching some execution logic as the EB of the DDC. As each EB executes it attaches new execution logic to the next DDC in the list. Since at any time only one DDC is actively executed, the guarantee that only one message is processed at a time is provided. Attempts are made to synchronously execute the EB when messages are available in the DDC. If a message is unavailable at the DDC pointed by the body head (Figure 15), the EB is set up to execute asynchronously in a task when a message arrives. The *lingering* activity gains access to the `finish` scope to which this asynchronous task needs to join. When the asynchronous task is ultimately scheduled and executed, the body head of the mailbox is moved ahead and the next DDC processed. If the actor was terminated via a call to `exit()`

in the EB of the DDC the actor stops processing messages from its mailbox, no more asynchronous tasks are scheduled by the actor, the *lingering* task is scheduled and subsequent messages sent to the actor are ignored.

6.2.3 Supporting `pause` and `resume` with DDCs

Light actors support the `pause` and `resume` operations explained in Section 5.1.2 using synchronous DDCs. A call to `pause()` changes the state of the actor to paused. Before processing the next message in the mailbox, we check whether the actor is in a paused state. If so, the next message from the mailbox is not processed. Instead, a block of code to process the next message from the mailbox is created and set as the EB of a *pause-resume* DDC. When `resume()` is called, the state of the actor is reset, and the *pause-resume* DDC is provided a value to synchronously trigger the execution of the EB.

6.3 Habanero-Scala

Habanero-Scala (HS) is an extension of the Scala language [18] with AFM compliant constructs. In HS, the AFM constructs were added as a library and an existing actor implementation (standard Scala actors) extended to support the unified model. We refer to these *unified* actors as *heavy* actors. *Heavy* actors provide support the operations presented in the unified model excluding the `pause` and `resume` operations. In addition, we have also ported the *light* actor implementation into HS.

An important reason to choose Scala is its support for powerful abstractions to express various programming constructs. One such construct, pattern-matching, is an elegant way to write actor code since the MPB needs to pattern-match on the messages received by the actor. Scala also has a relatively lenient constraint on the naming of methods, which coupled with its expressiveness makes it extremely easy to create domain-specific languages. This allows for easy transition of HJ constructs into Scala without the need to build a front-end compiler. Most of the Habanero work-sharing runtime can be reused in HS since both HJ and Scala run on the Java Virtual Machine.

6.3.1 *Heavy* Actors

Heavy actors are an extension of the standard Scala actors. These are called *heavy* actors since their implementation involves more overhead than the *light* actors presented in Section 6.2. To support operations like `receive` (called `react` for event-based Scala actors) and to avoid blocking, Scala actors throw exceptions to roll back the call stack and to allow the underlying thread to process messages of other actors. The need to throw and then ultimately catch these exceptions, even without the overhead of building the stack trace, is relatively expensive compared to an implementation that does not rely on the use of exceptions for control flow.

The *heavy* actor in HS is implemented as a trait that extends the standard Scala Actor trait (Figure 16). HS *heavy* actors do not support the `pause` and `resume` operations explained in Section 5.1.2. However, they support all the other AFM compliant constructs inside the MPB including `finish`, `async`, `futures`, etc. HS *heavy* actors still need to rely on exceptions for control flow and explicit management of the actor continuations, both implemented in the standard actors, and are thus more expensive to operate than the corresponding *light* actors.

```
1 /*** Habanero-Scala code ***/
2 package edu.rice.habanero
3 ...
4 trait HabaneroActor extends Actor {

6   // custom scheduler to create asynchronous
7   // tasks under IEF of lingering activity
8   var lingeringActivity: HabaneroActivity = null
9   val habaneroExecutor = ...
10  override def scheduler = habaneroExecutor
11  ...
12  override def start() = {
13    // activity causes the IEF to wait on this actor
14    lingeringActivity = ...
15    // delegate to the parent implementation
16    super.start()
17  }
18  override def exit(): Nothing = {
19    // schedule activity allowing IEF to terminate
20    resumeWaitingActivity(lingeringActivity)
21    // delegate to the parent implementation
22    super.exit()
23 } }
```

Figure 16: *Heavy* actors in HS extend the standard Scala `Actor` trait. The `start` and `exit` events are used to maintain some bookkeeping for the *heavy* actors and to interact with the Habanero runtime to schedule and execute tasks. The *lingering* activity is explained in Section 6.1.

6.3.2 *Light* Actors

HS also includes an implementation of *light* actors. They extend the features of HJ *light* actors by using Scala's pattern matching construct to represent the message processing body. While the pattern matching construct is more elegant it also entails a performance penalty compared to using simple `instanceof` checks used in the HJ actor implementation. Pattern matching also allows us to abstract away the synchronous reply operation (Section 5.2) so that the user does not have to manually manage the calls to `pause()` and `resume()` as shown in Figure 17.

HS *light* actors also support the `become` and `unbecome` operations. The `become` primitive specifies the behavior that will be used by the actor to process the next message allowing the actor to dynamically change its behavior at runtime. If no replacement behavior is specified, the current behavior will be used to process the next message. In the pure AM, actors are functional and the become operation provides the ability for the actor to maintain local state by creating a new actor and becoming this new actor. In Scala, the same effect can be achieved by having dynamic pattern matching

```
/*** Habanero-Scala code ***/
abstract class HabaneroReactor extends Actor[Any] {
  // updated as each reply message is processed
  private var replyDdf: DataDrivenFuture[Any] = null
  def reply(msg: Any): Unit = {
    if (replyDdf ne null) {
      replyDdf.put(msg)
    } else {
      // report error ...
  } }
  def awaitReply(receiver: HabaneroReactor, msg:Any,
      handler: PartialFunction[Any, Unit]): Unit = {
    // create DDF and message to send to the actor
    val replyMsg = new ReplyMessage(msg, ddf[Any]())
    // disable processing messages from the mailbox
    receiver.send(replyMsg); pause()
    // await reply from the receiver actor
    asyncAwait(replyMsg.replyDdf) {
      // process the response message
      handler(replyDdf.get())
      // continue processing further messages
      resume()
  } } }
```

Figure 17: *Light* actors in HS abstract away the synchronous reply operation, end-users use the `awaitReply()` and `reply()` method invocations in their actor code.

constructs which work in conjunction with mutable member variables.

HS *light* actors support the `become` and `unbecome` operations to allow the actor to change its behavior as it processes messages. In addition, the *light* actor is required to define the `behavior()` operation that provides a default behavior to use while processing messages. All these behaviors are presented as partial functions which Scala provides native support for. The behavior history is maintained in a stack and the old behavior can be retrieved by an `unbecome` operation. The support for `become` and `unbecome` is an improvement over the standard Scala actors in which the user has to rely on manipulation of local state or explicit management of behaviors to simulate the same operations. If at any point, the current behavior cannot process a message (i.e. the partial function is not defined for the message), that actor terminates and throws an exception by default; users can customize this and avoid throwing exceptions and terminating.

7. Experimental Results

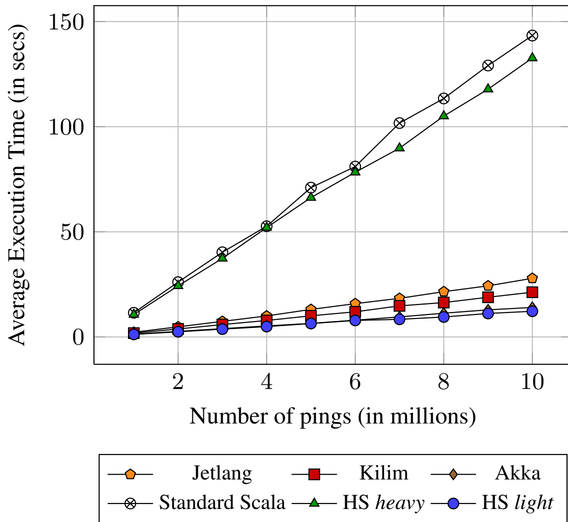
The actor frameworks used for comparison with our implementations all run on the JVM and include Jetlang [23], Kilim [27], Scala actors [10], and Akka [31]. Jetlang provides a low-level messaging API in Java that can be used to build actors with the onus of ensuring the single message processing rule delegated to the user. The use of batching while processing actor messages instead of creating a new asynchronous task to process each message in our implementation of *light* actors is inspired by Jetlang. Kilim is an actor implementation that ensures data isolation as required in the AM. Our actor implementations, however, do not support data isolation in messages. Scala includes an actor library that provides event-based actors which allow multiple actors to run on a thread. Our actor API is inspired from Scala's event-based actors, however we do not use exceptions to maintain control flow and use a push-based implementation using DDCs for light actors. Akka is a framework for building event-driven applications on the JVM and has support for highly performant lightweight actors. We chose not to include Erlang [34] since it does not run on the JVM, but has already been shown to have performance competitive to Kilim and Jetlang [15].

7.1 Experimental Setup

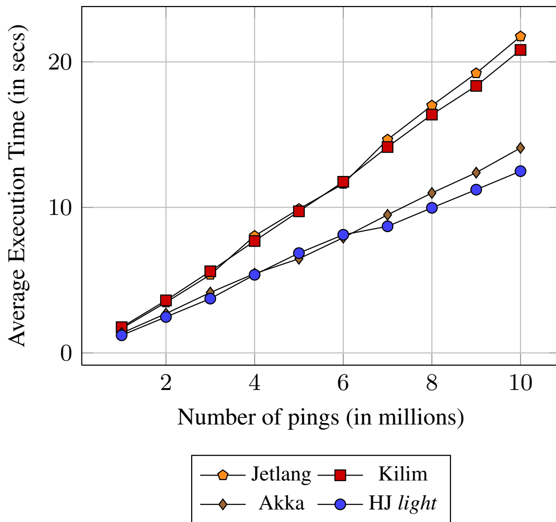
The benchmarks were run on a 12-core (two hex-cores) 2.8 GHz Intel Westmere SMP node with 48 GB of RAM per node (4 GB per core), running Red Hat Linux (RHEL 6.0). Each core had a 32 kB L1 cache and a 256 kB L2 cache. The software stack includes a Java Hotspot JDK 1.7, Habanero-Java 1.3.1, Habanero-Scala 0.1.3, and Scala 2.9.1-1. Each benchmark used the same JVM configuration flags (`-Xmx8192m -XX:MaxPermSize=256m -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:-UseGCOverheadLimit`) and was run for ten iterations in ten separate JVM invocations, the arithmetic mean of thirty execution times (last three from each invocation) are reported. This method is inspired from [8] and the last three execution times are used to approximate the steady state behavior. In the bar charts, the error bars represent one standard deviation. All actor implementations of a benchmark use the same algorithm and mostly involved renaming the parent class of the actors (in the Scala and Habanero-Scala versions) to switch from one implementation to the other.

7.2 Microbenchmarks comparing Actor frameworks

The first benchmark (Figure 18) is the `PingPong` benchmark in which two processes send each other messages back and forth. The benchmark was configured to run using two workers since there are two concurrent actors. This benchmark tests the overheads in the message delivery implementation for actors. The original version of the code was obtained from [29] and ported to use each of the different actor frameworks. Scala actors and HS *heavy* actors have the same underlying messaging implementation but use different schedulers. The HS *heavy* actors benefit from the thread binding support in the Habanero runtime. HS *light* actors perform better than Scala and HS *heavy* actors because it avoids the use of exceptions to maintain control flow (as discussed in Section 6.3.1). Both the Scala and Java versions of Kilim, Jetlang, Akka and *light* actors benefit from avoiding generating exceptions to maintain control flow. The Java versions follow the same pattern with Akka and *light* actors performing the best. In general, the Akka and *light* actor versions benefit from the use of fork-join schedulers as opposed to threadpool schedulers available in standard implementations of Kilim and Jetlang actors. Jetlang's Scala version is much slower than the Java version as the Scala implementa-



(a) Scala versions which use pattern matching.

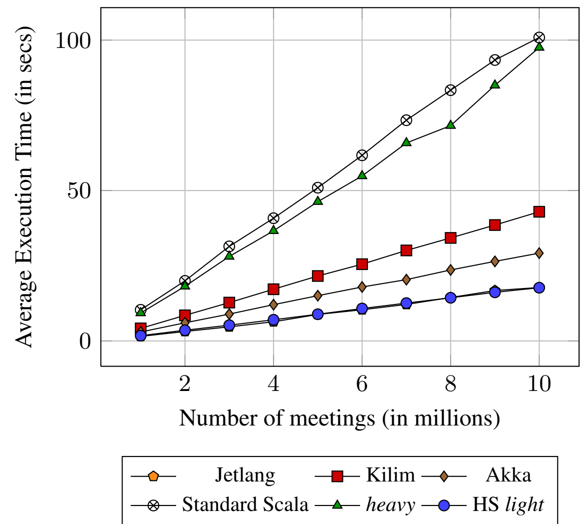


(b) Java versions which use `instanceof` operator.

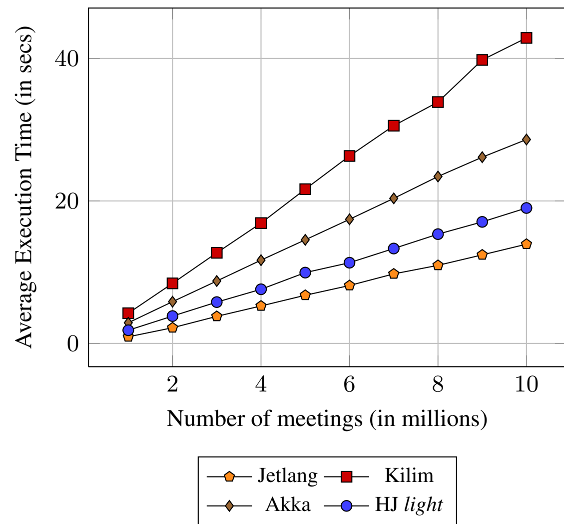
Figure 18: The `PingPong` benchmark exposes the throughput and latency while delivering messages. There is no parallelism to be exploited in the application.

tion pays the overhead for pattern matching twice as opposed to once in Kilim, Akka and *light* actors.

The `Chameneos` benchmark, shown in Figure 19, tests the effects of contention on shared resources (the mailbox implementation) while processing messages. The Scala implementation was obtained from the public Scala SVN repository [11]. The other actor versions were obtained in a manner similar to the `PingPong` benchmark. The benchmark was run with 500 chameneos (actors) constantly *arriving* at a mall (another actor) and it was configured to run using twelve workers. The mailbox implementation of the mall serves as a point for contention. In this benchmark, the benefits of thread binding are neutralized since the contention on the mailbox is the dominating factor and since both the Scala

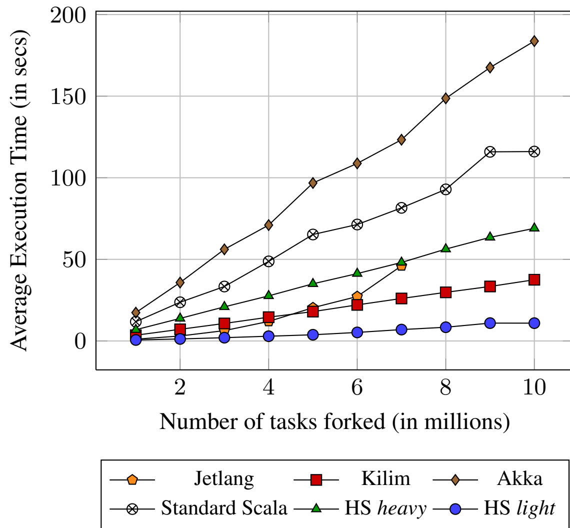


(a) Scala versions which use pattern matching.

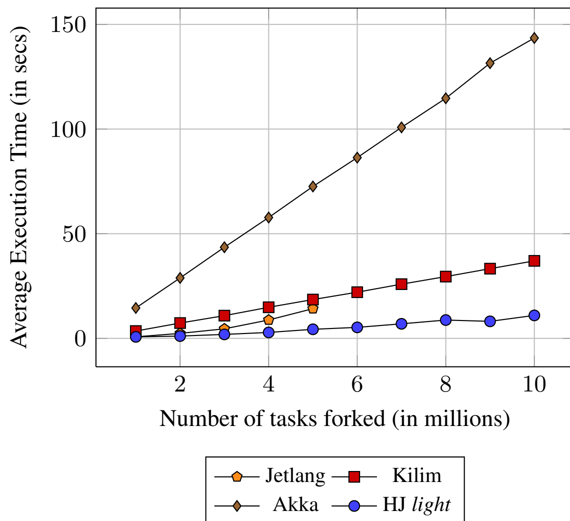


(b) Java versions which use `instanceof` operator.

Figure 19: The `Chameneos` benchmark exposes the effects of contention on shared resources. The `Chameneos` benchmark involves all *chameneos* constantly sending messages to a mall actor that coordinates which two *chameneos* get to meet. Adding messages into the mall actor's mailbox serves as a contention point.



(a) Scala versions which use pattern matching.



(b) Java versions which use `instanceof` operator.

Figure 20: The Java Grande Forum `Fork-Join` benchmark ported for actors. Individual invocations were configured to run using twelve workers. Both Jetlang versions run out of memory on larger problem sizes.

and HS *heavy* actors share the same implementation they show similar performance. Both the Scala and Java versions of Kilim, Jetlang, Akka and *light* actors benefit from batch-processing messages inside tasks and from avoiding generating exceptions to maintain control flow. The *light* actor implementations that uses DDCs (Section 6.2.1) outperforms the linked list implementation in actors. Jetlang, which uses iterative batch-processing of messages sent to the mall, is in general faster than the *light* actor implementation which uses recursive batch processing of messages.

The Java Grande Forum `Fork-Join` benchmark [7], shown in Figure 20, measures the time taken to create and destroy actor instances. Each actor does a minimal amount of work processing one message before it terminates. The Akka implementation is noticeably slower while the Jetlang implementation quickly runs out of memory as it uses an `ArrayList` to maintain the work queue. The *heavy* actor implementation again benefits from thread binding support compared to standard Scala actors. The *light* actor implementation which uses lightweight `async` tasks to implement actors performs best.

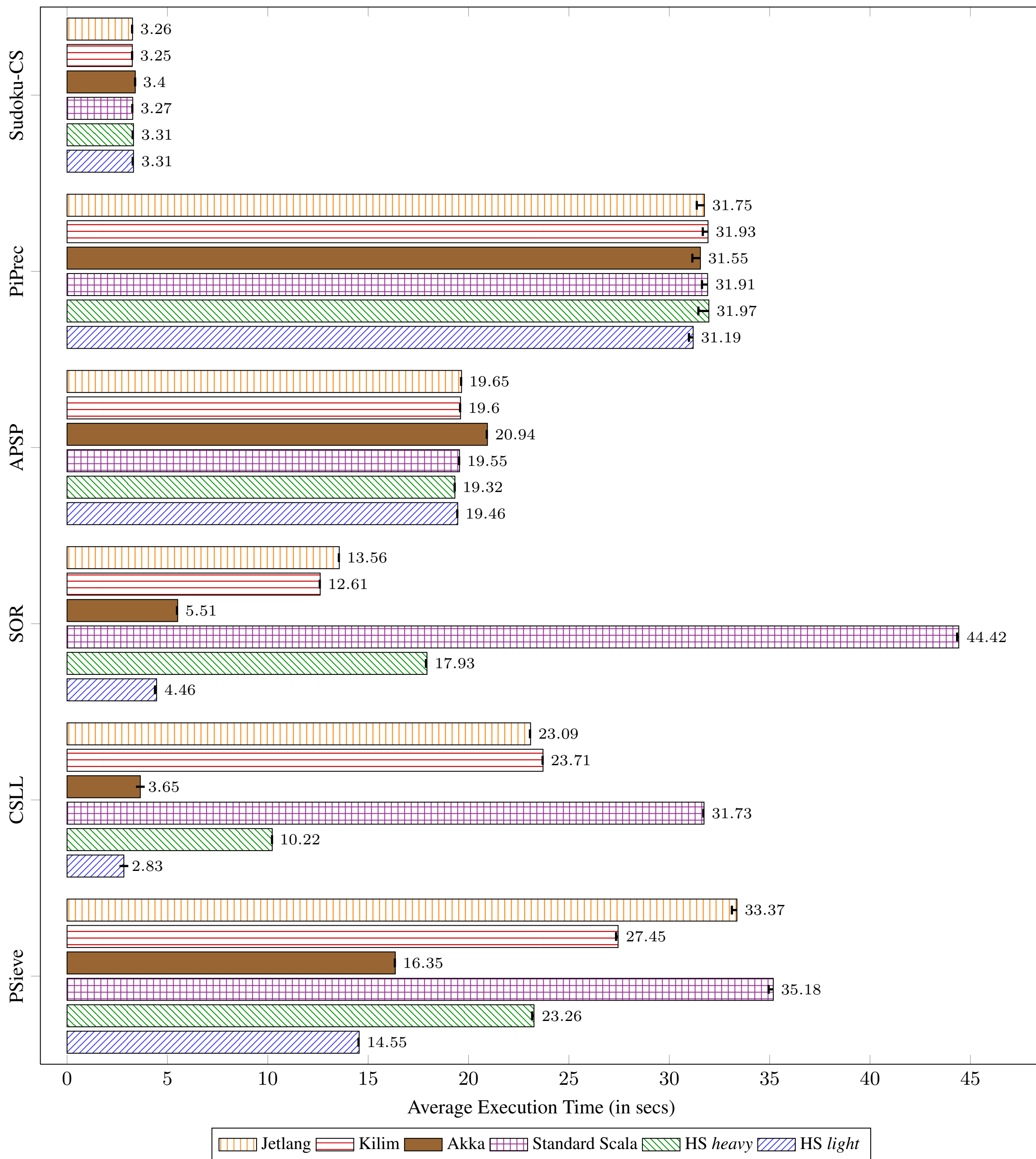
7.3 Application Benchmarks

In this section, we compare the performance of the actor frameworks on applications displaying different parallel patterns. We also analyze the benefits of parallelizing the actor message processing in the unified model in some applications. Each application benchmark was run with the schedulers set up to use 12 worker threads.

7.3.1 General Applications Compared

Figure 21 displays results of running different applications using the different actor frameworks. The first two applications, Sudoku Constraint Satisfaction (Sudoku-CS) and Pi Precision (PiPrec), represent master-worker style actor programs where the master incrementally discovers work to be done and allocates work fragments to the workers. Workers only have at most one message pending in their mailbox and there is no scope for batch processing messages. The master is the central bottleneck in such applications and all frameworks perform similarly. The next application, All-Pairs Shortest Path (APSP), represents a phased computation where all actor effectively join on a barrier in each iteration of the outermost loop in Floyd-Warshall's algorithm before proceeding to the next iteration. In each iteration the slowest actor dominates the computation and as a result we see similar execution times for all the frameworks.

The next three applications have relatively larger memory footprints and we see the benefits of thread binding as well as efficient implementation for throughput. HS *heavy* is faster than standard Scala actors. Similarly the *light* and Akka actors outperform the other actor frameworks. The actor implementation of Successive Over-Relaxation (SOR) represents a 4-point stencil computation and was ported from SOTER [32]. The next two applications, Concurrent Sorted Linked-List (CSLL) and Prime Sieve (PSieve), use a pipeline pattern to expose some parallelism. CSLL measures the performance of adding elements, removing elements, and performing collective operations on a linked-list. The implementation maintains a list of helper actors with each actor responsible for handling request for a given value range for individual element operations. Collective operations, such as `length` or `sum`, are implemented using a pipeline starting from the head of the list of the helper actors and only the tail actor returning a response to the requester. There are multiple request actors requesting various operations on the linked-list and non-conflicting requests are



- Sudoku-CS: Sudoku Constraint Satisfaction
- PiPrec: Pi Precision
- APSP: All-Pairs Shortest Path (Floyd-Warshall)
- SOR: Successive Over Relaxation
- CSLL: Concurrent Sorted Linked List
- PSieve: Prime Sieve

Figure 21: Comparison of implementations of some applications using different JVM actor frameworks (Scala version).

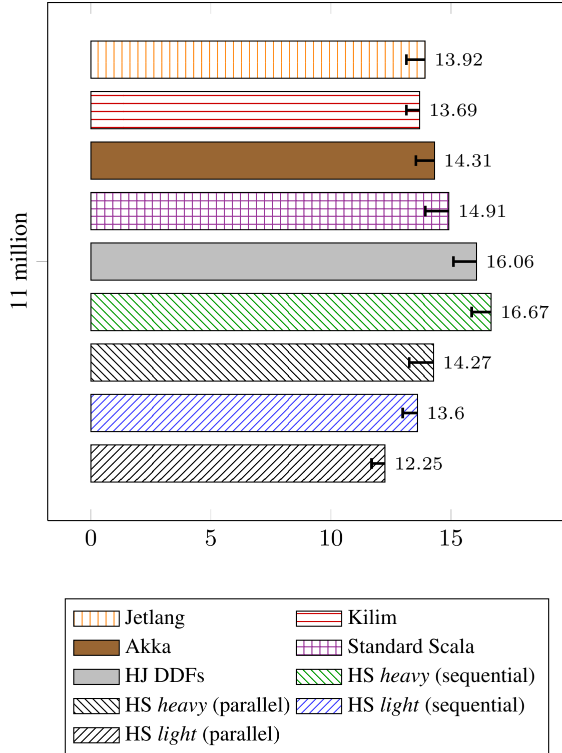


Figure 22: Results of the Quicksort benchmark on input of length 11 million.

processed in parallel. The PSieve application represents a dynamic pipeline in which a fixed number of local primes are buffered in each stage. Every time the buffer overflows, a new stage is created and linked to the pipeline, thus growing the pipeline dynamically. There is overhead in filling and draining items in the pipeline for each stage and thus a buffered solution with multiple primes per stage performs better.

In summary, the geometric means of the execution times in seconds for the different actor frameworks in sorted order are as follows: HS *light* (8.47), Akka (9.51), HS *heavy* (14.35), Kilim (15.99), Jetlang (16.64), and standard Scala (21.59). The HS *light* is more than 10% faster than Akka and more than 33% faster than the other actor frameworks while using sequential message processing in actors.

7.3.2 Quicksort

Quicksort lends itself to divide-and-conquer strategy and is a good fit for the AFM, however as mentioned in Section 2.3 it exposes some amount of non-determinism in availability of partial results which cannot entirely be captured by the AFM. Figure 22 compares the unified actor implementations in HJ with previously existing `async-finish` extensions such as `isolated` and DDFs. Pure actor implementations in HJ involve sequential message-processing. The *light* actor implementation is faster than the DDF-based implementation as it can make progress computing the partial result from

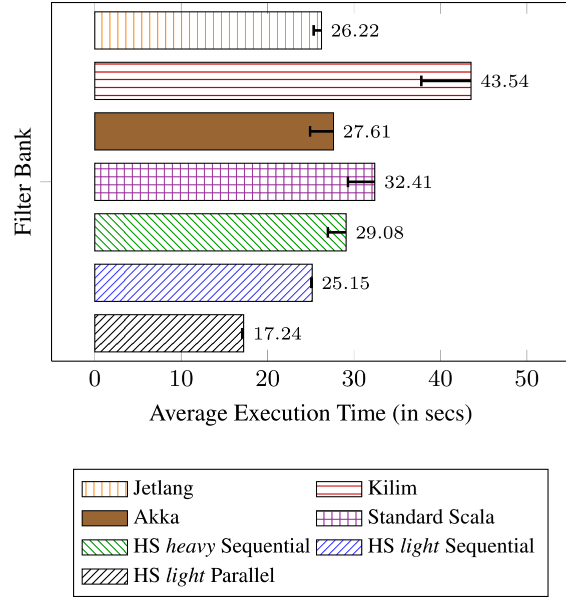


Figure 23: Filter Bank benchmark results configured to use three branches.

fragments. In the unified model, parallelization inside the actor is achieved by performing the left and right splits around the partition in parallel for arrays with sizes larger than a configured threshold. The parallelized unified actor implementations perform better than the implementation that use sequential message processing by around 10% and 14% for *light* and *heavy* actors, respectively. The HS *light* (parallel) actor is the best-performing and is around 10% faster than other actor implementations and more than 23% faster than DDF implementation.

7.3.3 Filter Bank for multirate signal processing

Filter Bank has been ported from the StreamIt [30] set of benchmarks. It is used to perform multirate signal processing and consists of multiple pipeline branches. On each branch the pipeline involves multiple stages including multiple delay stages, multiple FIR filter stages, and sampling. Since Filter Bank represents a pipeline, it can easily be implemented using actors. The FIR filter stage is stateful, appears early in the pipeline, and is a bottleneck in the pipeline. Parallelizing the computation of the weighted sum to pass down the pipeline in this FIR stage shortens the critical length of the pipeline and helps speed up the application. Figure 23 compares the performance of the actor implementations of the Filter Bank benchmark with a unified implementation which parallelizes the FIR stage. The HS *light* parallel version is at least 30% faster than the other actor implementations.

7.3.4 Online Hierarchical Facility Location

Facility Location algorithms are used to decide when and where to open facilities in order to minimize the cost of

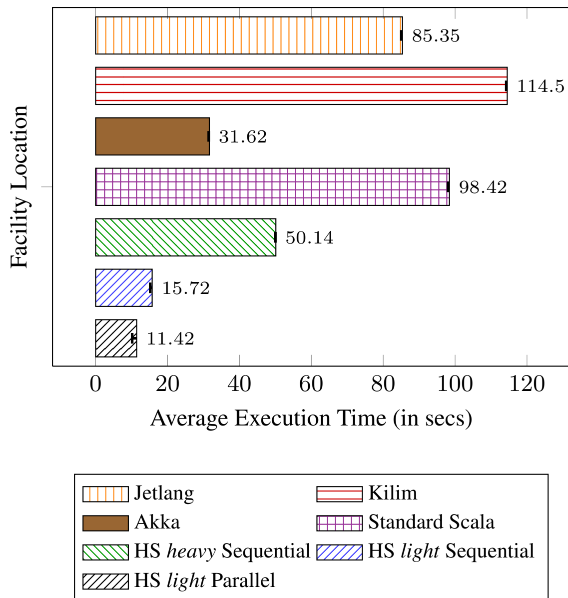


Figure 24: Online Hierarchical Facility Location benchmark results. Results displayed for 6 million customers and an alpha value of 5.

opening a facility and the cost of servicing customers. In the online version, the locations of customers are not known beforehand and the algorithm needs to make these decisions *on-the-fly*. One algorithm for this problem is the Online Hierarchical Facility Location [2]. The algorithm exposes a hierarchical tree structure (quadrants in the algorithm) while performing the computation. Information in the form of customer locations initially flows down the tree. In the algorithm, each node maintains a list of customers it plans to service and this list is partitioned at decision points to form new child nodes. In addition, the decision to create child nodes needs to be propagated up the tree and to selected siblings. A speculatively parallel version of this algorithm can be mapped to use actors. Each actor represents a quadrant and are arranged in a tree structure. Each quadrant maintains a cost value as it receives customers. When the threshold is exceeded, the customers are partitioned and transferred to newly formed child quadrants. In the unified model `async-finish` additional parallelism is achieved while partitioning the customers to prepare the child nodes. The pure AM variants do not support such parallelism while processing a message and need to split the child nodes sequentially.

Figure 24 compares the performance of the actor implementations of the Facility Location benchmark with a unified implementation. In Online Hierarchical Facility Location, parallelism from the unified model is used when a quadrant (actor) splits and creates its four children. The split happens based on a threshold determined by the value of alpha, which is an input to the program. A smaller value of alpha means there are larger number of splits and the tree is deeper. The performance of the HS *light* with parallelized splits is better than the HS *light* actor implementation by about 27% and is comfortably better than Jetlang, Kilim, Akka, and Scala.

8. Related Work

Schäfer et al. have proposed the notion of Parallel Actor Monitors (PAM) [25] to extend the actor model with support for intra-actor parallelization. In PAM, the end user specifies schedulers, separate from the actor's message processing body (MPB), that control when an actor is able to process multiple messages safely. For example, a scheduler in PAM might allow an actor to process multiple messages for read requests in parallel but only allow a single message for a write request to be in flight by an actor. Similarly, it is trivial to express stateless actors in PAM by writing a scheduler that allows all messages to be processed in parallel. In the actor model, only one message for an actor would be in flight at a time. Our approach allows us to specify similar intra-actor parallelism constraints by allowing escaping `asyncs` but currently requires modification of the actor's MPB. Additionally, our model allows expressing parallelization inside the actor's MPB, for example, exploiting data parallelism while processing a message as seen in the Filterbank example in Figure 5. Such parallelism cannot be expressed with PAM.

The CoBox model [24] proposed by Schafer et al. is inspired by the actor model and exposes parallelism among asynchronously communicating objects. Objects are partitioned into separate concurrently executing CoBoxes and allocated dynamically but never leave their host CoBox. A task performs synchronous operations on co-located objects. CoBoxes can have multiple ready tasks, but actively execute a single task at a time thus ensuring data race freedom. An active task can cooperatively decide to suspend itself and activate another task in the same CoBox when it discovers some condition which prevents its progress. This notion of isolating data into different partitions and cooperative execution has also been used by Lublinerman et al. in the Chorus programming model [16]. Chorus is used for applications with irregular data parallelism where the partitions, called Object Assemblies, can merge or split dynamically when new data isolation constraints are discovered in synchronously communicating objects. Our approach differs from these models in that we can expose parallelism inside the MPB which would be equivalent to multiple tasks executing simultaneously in a CoBox/Object Assembly, but at the cost of possible data races. We are planning on extending the SPD3 algorithm [22] for HJ's finish, async and isolated constructs to also detect races in our combination of actor and task parallelism.

Non-blocking receive operations between actors are available in the E language [17] under the form of promises to futures. These are created every time a message is asynchronously sent to an actor. Actions can be registered to the promise using the `when` clause; these actions are triggered

when the promise resolves, i.e. when the message sent to the actor is processed. This is similar to how DDFs work with the `asyncAwait` clause. The difference is that in our model DDFs need to resolved explicitly by putting values into the DDF, though this resolution could be done automatically by a runtime system in response to certain events such as when an actor processes a message. AmbientTalk [6], inspired from E, also supports creation of futures on message sends and requires explicit resolution of future values. In addition to the use of DDFs, the `pause` and `resume` operations in our model allow us to implement non-blocking receive operations while preventing the next message in an actor's mailbox from being processed.

SALSA [33] also supports non-blocking receive operations using the notion of tokens (similar to implicit futures) whose resolved value gets passed automatically to registered actions; the tokens themselves cannot be passed in messages to other actors. As mentioned previously, in our model DDFs are resolved explicitly and can be passed around in messages to other actors. SALSA also support join tokens which can register an action to execute only after all messages sent inside the join token has been processed. In our model we can achieve this by registering on multiple DDFs. The join block is also similar to the `finish` construct in our model, the difference being that a statement following a `finish` may not execute until all nested `async` tasks have completed execution. In SALSA, the join token only delays the execution of the action registered on the token.

9. Conclusions and Future Work

This paper focuses on a unified model that integrates the Async-Finish model (AFM) and the Actor model (AM). To the best of our knowledge, this is the first effort to systematically combine these two models. The unified model allows for parallelism inside actors while also making termination detection easier in actor programs. It also allows arbitrary coordination patterns among tasks in the AFM, in an arguably more productive manner than other extensions, such as phasers and DDFs. The unified model allows for easier implementation of certain constructs: for example, the normally blocking `receive` can be implemented in a non-blocking manner in the unified model. The paper also studies properties of applications that can benefit from the unified model.

We also present two implementations of this unified model in Habanero-Java (HJ) and Habanero-Scala (HS). HJ is a mature AFM implementation which we extend with support for the unified actors. On the other hand, HS is an extension of Scala in which we ported AFM constructs and modified the existing actor implementation to work under the unified model. In addition, HS provides a faster actor implementation than the standard Scala actor library. These implementations served as tools to run experiments that corroborate the claim that unified solutions to certain problems are more efficient than solutions that exclusively use the AFM or the AM.

The unified model suffers from the possibility of data races when the message processing inside actors is parallelized. In fact, data races can also exist in many actor implementations on the JVM as they do not enforce data isolation. Data race detection in the *unified* actors is an interesting area for future research and we plan to extend the SPD3 algorithm [22] for data race detection in the unified model.

Availability

Public distributions of Habanero-Java and Habanero-Scala, including code examples, are available for download at `http://habanero.rice.edu/hj.html` and `http://habanero-scala.rice.edu/`, respectively.

Acknowledgments

We are grateful to Vincent Cavé, Dragoş Sbîrlea and Sağnak Taşırlar for discussions on the Habanero Java runtime system, phasers and DDFs, respectively. We thank Carlos Varela and Travis Desell for feedback on an earlier draft of this paper, and for general discussions on designing and implementing actor languages and runtimes as well as specific details on the SALSA language. We also thank Philipp Haller for his feedback on an earlier draft of this paper. Finally, we are grateful to Jill Delsigne at Rice University for her assistance with proof-reading an earlier draft of this paper. This work was supported in part by the U.S. National Science Foundation through awards 0926127 and 0964520.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[2] A. Anagnostopoulos, R. Bent, E. Upfal, and P. V. Hentenryck. A simple and deterministic competitive algorithm for online facility location. *Inf. Comput.*, 194:175–202, November 2004. ISSN 0890-5401.

[3] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar. Concurrent Collections. *Sci. Program.*, 18: 203–217, August 2010. ISSN 1058-9244.

[4] V. Cavé, J. Zhao, Y. Guo, and V. Sarkar. Habanero-Java: the New Adventures of Old X10. *9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, August 2011.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40:519–538, Oct. 2005. ISSN 0362-1340. doi: 10.1145/1094811.1094852.

[6] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-Oriented Programming in AmbientTalk. In *Proceedings of the 20th European Conference*

on Object-Oriented Programming, ECOOP'06, pages 230–254, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35726-2, 978-3-540-35726-1. doi: 10.1007/11785477_16. URL http://dx.doi.org/10.1007/11785477_16.

[7] EPCC. The Java Grande Forum Multi-threaded Benchmarks. URL http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/s1contents.html.

[8] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5.

[9] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGOPS Oper. Syst. Rev.*, 40:151–162, October 2006. ISSN 0163-5980.

[10] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009. ISSN 0304–3975. doi: 10.1016/j.tcs.2008.09.019. URL http://www.sciencedirect.com/science/article/pii/S0304397508006695. Distributed Computing Techniques.

[11] Haller, Philipp. chameneos-redux.scala — FishEye: browsing scala-svn, 2011. URL https://codereview.scala-lang.org/fisheye/browse/scala-svn/scala/branches/translucent/docs/examples/actors/chameneos-redux.scala?hb=true.

[12] C. Hewitt, P. Bishop, and R. Steiger. Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence. Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Stanford, CA, August 1973.

[13] Hewitt, Carl and Baker, Henry G. Actors and Continuous Functionals. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, February 1978.

[14] Imam, Shams and Sarkar, Vivek. Habanero-Scala: Async-Finish Programming in Scala. In *The Third Scala Workshop (Scala Days 2012)*, April 2012.

[15] R. K. Karmani, A. Shali, and G. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-598-7. doi: 10.1145/1596655.1596658. URL http://doi.acm.org/10.1145/1596655.1596658.

[16] R. Lublinerman, S. Chaudhuri, and P. Cerny. Parallel Programming with Object Assemblies. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 61–80, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640095. URL http://doi.acm.org/10.1145/1640089.1640095.

[17] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency Among Strangers: Programming in E as Plan Coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing*, TGC'05, pages 195–229, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-30007-4, 978-3-540-30007-6. URL http://dl.acm.org/citation.cfm?id=1986262.1986274.

[18] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. Mcdirmid, S. Micheloud, N. Mihaylov, M. Schinz, and et al. An Overview of the Scala Programming Language Second Edition. *System*, (Section 2):15–30, 2006.

[19] OpenMP Architecture Review Board. OpenMP Application Program Interface - Version 3.0 May 2008. URL www.openmp.org/mp-documents/spec30.pdf.

[20] N. Raja and R. K. Shyamasundar. Actors as a Coordinating Model of Computation. In *Proceedings of the 2nd International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 191–202. Springer-Verlag, 2004. ISBN 3-540-62064-8.

[21] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient Data Race Detection for Async-Finish Parallelism. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 368–383, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16611-3, 978-3-642-16611-2.

[22] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and Precise Dynamic Data Race Detection for Structured Parallelism. In *PLDI*, 2012.

[23] Rettig, Mike. jetlang: Message based concurrency for Java. URL http://code.google.com/p/jetlang/.

[24] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. URL http://dl.acm.org/citation.cfm?id=1883978.1883996.

[25] C. Scholliers, E. Tanter, and W. D. Meuter. Parallel Actor Monitors. In *14th Brazilian Symposium on Programming Languages*, 2010.

[26] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-Point Synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3.

[27] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java (A Million Actors, Safe Zero-Copy Communication). *European Conference on Object Oriented Programming ECOOP 2008*, 5142/2008:104–128, 2008.

[28] S. Taşırlar and V. Sarkar. Data-Driven Tasks and their Implementation. In *Proceedings of the International Conference on Parallel Processing (ICPP) 2011*, September 2011.

[29] The Scala Programming Language. pingpong.scala. URL http://www.scala-lang.org/node/54.

[30] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Computational Complexity*, pages 179–196, 2002.

[31] Typesafe Inc. Akka. URL http://akka.io/.

[32] UIUC. SOTER project. URL `http://osl.cs.uiuc.edu/soter/`.

[33] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, Dec. 2001. ISSN 0362-1340. doi: 10.1145/583960.583964. URL `http://doi.acm.org/10.1145/583960.583964`.

[34] R. Virding, C. Wikström, M. Williams, and J. Armstrong. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. ISBN 0-13-508301-X.

[35] A. Yonezawa, J. Briot, and E. Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 258–268, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28722. URL `http://doi.acm.org/10.1145/28697.28722`.