Habanero-Scala: Async-Finish Programming in Scala

Shams Imam   Vivek Sarkar
Rice University
{shams, vsarkar}@rice.edu

Abstract

With the advent of the multicore era, it is clear that improvements in application performance will require parallelism. Programming models that utilize multiple cores offer promising directions for the future where core counts are expected to increase. There is, hence, a renewed interest in programming models that simplify the reasoning and writing of efficient parallel programs. In this paper, we present Habanero-Scala (HS) which implements a generic task parallel programming model that can be used to parallelize both regular and irregular applications. HS is a library extension of Scala and supports a hybrid programming model combining the previously developed Async/Finish Model (AFM) and Actor Model (AM). HS extends Scala's actor-based concurrency model with creation of lightweight tasks embodied in `async`, future, and `foreach` constructs; termination detection using the `finish` construct; locality in the form of `places`; weak isolation using `isolated` blocks; and task coordination patterns using `phasers`, data-driven futures. HS offers a simpler parallel programming model compared to writing programs using threads and adds 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

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

1. Introduction

With the advent of the multicore era, it is clear that improvements in application performance will primarily come from increased parallelism. Programming models that utilize multiple cores offer a scalable solution for the future where core counts are expected to increase. Current mainstream programming languages provide limited support for expressing parallelism in software. Programmers, hence, need new software concurrency/parallel programming models to extract performance from multi-core hardware with ease and to reduce the burden of reasoning about and writing parallel programs. This has led to a renewed interest in parallel programming models in the academic community. In this paper, we introduce Habanero-Scala (HS) which extends Scala with a generic task parallel programming model that can be used to parallelize both regular and irregular applications.

Programs typically exhibit varying degrees of task, data, and pipeline parallelism [7]. A handful of various programming models have been developed to handle task and pipeline parallelism. HS provides support for two such models:

- The Async/Finish Model (AFM) 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.

HS is an implementation of a hybrid model integrating the AFM and the AM. It is based on Habanero-Java (HJ), a parallel programming language developed by the Habanero Multicore Software Research Group at Rice University that implements the AFM [3]. Scala as a language provides powerful abstractions to express various programming constructs. Scala has a relatively lenient constraint on the naming of methods, which coupled with its expressiveness makes it extremely easy to create domain-specific languages (DSLs). This allows for easy transition of HJ constructs into Scala without the need to modify the front-end compiler. Most of HJ's work-sharing runtime can be reused in HS since both HJ and Scala run on the Java Virtual Machine. As a result, HS provides parallel constructs as library extensions of Scala unlike HJ which is a language-based approach with its own compiler. HS supports the creation of lightweight tasks embodied in `async`, `future`, and `foreach` constructs; termination detection using the `finish` construct; locality in the form of `places`; weak isolation using `isolated` blocks; and task coordination and synchronization constructs using `phasers`, data-driven futures and actors.

2. The Async/Finish Constructs

The Async/Finish Model (AFM) is a variant of the Fork/Join Model. In the AFM, a task can *fork* a group of child tasks. These child tasks can recursively fork even more tasks. Each of these tasks can potentially run in parallel with each other. Further, a parent/ancestor task can selectively *join* on a subset of child/descendent tasks. This is the primary form of synchronization between tasks in the AFM.

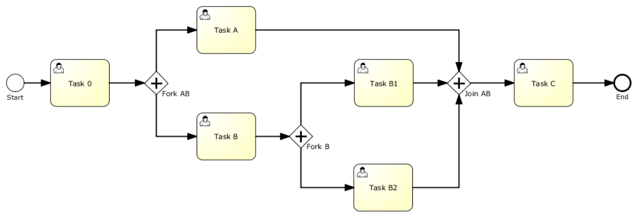

Figure 1. Fork/Join Parallelism achieved by forking new tasks and joining before proceeding. Note that until all forked tasks (Task A, Task B, Task B1, and Task B2) reach the join point, Task C cannot be executed. [source=http://www.coopsoft.com/ar/ForkJoinArticle.html].

The central feature of any AFM implementation on multicore architectures is the ability to create and manage lightweight tasks. Tasks are created at *fork* points and HS provides the `async` keyword to create a task. The statement `async` ⟨*stmt*⟩ causes the parent task to create a new child task to execute ⟨*stmt*⟩ (logically) in parallel with the parent task. The HS runtime is responsible for the scheduling of tasks created by `async`s. The management of actual threads and related thread pools is done by the runtime and is transparent to the tasks in the program. The HS implementation treats the body of an `async` as a closure. As such, local variables are private to each task, whereas static and instance fields may be shared among tasks. An inner `async` is allowed to read and modify a local variable declared in an outer scope.

HS also supports a `asyncSeq` variant for an `async` statement with the following syntax and semantics:
`asyncSeq(cond)` ⟨*stmt*⟩ ≡
  `if (cond)` ⟨*stmt*⟩ `else async` ⟨*stmt*⟩
This variant is useful syntactic sugar while using thresholds to create parallel/serial tasks. However, the main benefit is that it removes the burden on the programmer to duplicate ⟨*stmt*⟩ and thereby making maintenance easier.

The `finish` keyword in HS 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. This is the primary form of synchronization between tasks in `async-finish` style programs. All HS programs execute inside a global finish scope for the main program: the program is allowed to terminate when all tasks nested inside the global finish terminate. The global finish rule ensures that each executing HS task has a unique *Immediately Enclosing Finish* (IEF). Besides termination de-

```
object AsyncFinishPrimer extends HabaneroApp {
  /* finish { implicit global finish wraps main() */
    println("Task O") // Task-O
    finish {
      async { // Task-A
        println("Task A")
      }
      async { // Task-B
        println("Task B")
        async { // Task-B1 created by Task-B
          println("Task B1")
        }
        async { // Task-B2 created by Task-B
          println("Task B2")
      } }
    } // Wait for tasks A, B, B1 and B2 to finish
    println("Task C") // Task-C
  /* } end of implicit global finish  */
  // the global finish must wait for all nested tasks
  // to terminate before the program terminates
} }
```

Figure 2. HS version of the Fork/Join program from Figure 1 using `async` and `finish` constructs.

tection, the `finish` statement plays an important role with regard to exception semantics. If any `async` throws an exception, then it is collected by its IEF. The IEF then throws a *MultiException* [4] formed from the collection of all exceptions thrown by all `async`s in the IEF. Figure 2 shows a sample HS program which uses `async` and `finish` constructs to preserve the task dependences of Figure 1.

The scopes of `async` and `finish` can span method boundaries. As a result, parallelizing sequential programs using `async-finish` is fairly easy. `async`s are inserted to wrap statements which can be executed in parallel and then these `async`s are wrapped inside `finish` blocks to ensure the parallel version produces the same result as the sequential version. `async-finish` style computations are guaranteed to be deadlock free [4]. In addition, in the absence of data races, these programs also have the extremely desirable property that they are deterministic [19].

2.1 `forall` and `foreach` constructs

HS supports a variant of the for loop over rectangular regions using the `forall` and `foreach` constructs to iteratively spawn parallel tasks inside the loop, one per iteration. HS also provides an implicit conversion for `Iterable`s to remove the restriction of rectangular regions and to simplify use of `asyncForall` and `asyncForeach` methods on Scala collections. The one-dimensional version of `forall` and `foreach` has the following syntax and semantics:
`forall(start, end)` f`(i)` ≡
  `finish for(i <- start to end) async` f`(i)`[1]
`foreach(start, end)` f`(i)` ≡
  `for(i <- start to end) async` f`(i)`
A `foreach` statement does not have an implicit `finish`, but its termination can be ensured by enclosing it within a `finish` at an appropriate outer level. Any exceptions thrown

[1] The implicit `phaser` not displayed for simplicity

by the spawned iterations are propagated to its IEF instance. While `foreach` can be used for achieve data parallelism, they are mainly provided as a syntax sugar to simplify creating lightweight tasks in loops. Simple data parallelism in HS is best achieved by using the standard Parallel Collection framework [18]. The `forall` includes an implicit `finish` and waits for the spawned iterations (`asyncs`) to terminate. Each task spawned by the `forall` construct has a pre-allocated `phaser` registered in *signal-wait* mode; `phasers` are addressed in Section 3.2.3. The `forall` is designed to easily express parallel loops with individual iterations participating in phased computations.

3. Synchronization and Coordination constructs

3.1 Synchronized access using `isolated`

A concern common in most shared memory models is the issue of data races and the need to synchronize the accesses to shared resources/variables between tasks. In addition to ordered synchronization constructs such as `finish`, HS provides an `isolated` ⟨*stmt*⟩ construct to support weak isolation, i.e. atomicity is guaranteed only with respect to other statements also executing inside `isolated` scopes. No guarantees are provided on interactions with non-isolated statements; accesses to shared variables by parallel tasks outside `isolated` blocks may participate in data races. In HS, we do not allow other parallel constructs to be nested inside `isolated` blocks and the runtime reports an error in such scenarios.

```
object IsolatedPrimer extends HabaneroApp {
  val counter = Array.ofDim[Int](4)
  finish {
    forall(1, 399) { i =>
      isolated {
        // counter modified in isolated, no data race
        val n = i % 4
        counter(n) = counter(n) + 1
    } }
    // the statement below would introduce a
    // data-race as it is outside an isolated scope
    /* counter(0) = counter(0) + 1 */
  }

  for (0 to 3) { n =>
    assert(100, counter(n), "No data-race detected")
} }
```

Figure 3. HS isolated statements at work. Each isolated block executes sequentially and there are no data-races. Excessive use of isolated results in loss of parallelism, though optimistic concurrency implementations such as Delegated Isolation [13] can exploit parallelism even when `isolated` is used extensively.

In HS, the default implementation of `isolated` statements uses a single lock causing all `isolated` statements to be serialized. This can be a serious performance bottleneck in applications with moderate contention [3]. We are planning to add support for finer-grained support to `isolated` blocks of the form `isolated(`*variable11, ..., variableN*`)` ⟨*stmt*⟩ which will provide better performance than the current implementation. There is an alternate prototype implementation of HJ isolated statements using a technique called Delegated Isolation [13] which doesn't serialize non-interfering isolated statements and results in better performance and scalability.

3.2 Coordination constructs

While independent tasks can run in parallel, there are often dependences among tasks. In such scenarios, coordination between tasks is required to determine when dependent tasks can be executed. Coordination of parallel tasks is one of the major sources of complexity in parallel programs and runtimes. In addition, this often involves some sort of communication between the tasks and is a source of overhead in the program. The basic coordination mechanism between tasks in the AFM is that between a task created via an `async` and its IEF. However, there may be dependences between sibling tasks which cannot be realized by the AFM. HS augments the AFM with a handful of coordination constructs: futures, data-driven futures, phasers, and actors.

3.2.1 Futures

A `future` represents the result of an asynchronous computation and extends HS `async` statements to `async` expressions. The statement `val f = asyncFuture[T]` ⟨*expr*⟩ creates a new child task to evaluate *expr* that is ready to execute immediately. In this case, `f` contains a *future handle* to the newly created task and the operation `f.get()` can be performed to obtain the result of the `future` task. If the `future` task has not completed as yet, the task performing the `f.get()` operation blocks until the `future` task completes and the result of *expr* becomes available. One advantage of using `futures` is that there can never be a data race on accesses to a `future`'s return value. In addition, if all `futures` are stored in immutable variables, it ensures that no deadlock cycle can be created with `future` tasks.

While `futures` are simple to use, their injudicious use limits the performance and scalability of HS programs. This is because calls to the `get()` on the `future` object currently blocks the worker thread. In order to maintain parallelism, the HS work-sharing runtime responds by creating more worker threads. Threads are heavyweight resources and the management of their life cycle is expensive and this eventually hurts the program's performance. In addition to consuming resources such as memory, each thread requires two execution call stacks, which can be large [6]. Creating too many threads in one JVM can cause the system to run out of memory or thrash due to excessive memory consumption. Currently we are working on using delimited continuations [5] to avoid blocking of threads in the Habanero runtime in such instances.

```
object FibFuturePrimer extends HabaneroApp {
  def fib(n: Int): Int = {
    if (n < 2) {
      return n
    } else {
      val x = asyncFuture[Int] { fib(n − 1) }
      val y = asyncFuture[Int] { fib(n − 2) }

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

  val n = Integer.parseInt(args(0))
  val result = fib(n)
  println("fib(" + n + ") = " + result)
}
```

Figure 4. HS Fib using `futures`. A relatively large value of n will cause the program to run out of memory due to excessive creation of threads in the current HS runtime. However, the standard Async/Finish parallelism does not have this limitation.

3.2.2 Data-Driven Futures (DDFs)

DDFs are an extension to futures to support the dataflow model [25]. DDFs 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. The exact syntax for an `async` waiting on a DDF is as follows: `asyncAwait(`*ddf1, ..., ddfN*`)` ⟨*stmt*⟩

There are three main operations allowed on a DDF:

- `put(some-value)`: this non-blocking operation associates a value with the DDF. Due to the single assignment property, only a single `put()` is allowed on the DDF during the execution of the program.
- `await()`: this is a blocking operation used by `async`s to delay their execution until some other task has `put()` a value into the DDF.
- `get()`: this is a non-blocking operation is used to retrieve the value stored in the DDF. It can legally be invoked by a task that was previously waiting 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.

Traditionally, the AFM requires the parent of a task to also ensure the data consumed by the child task is available when the child is being created. With DDFs, the creation of a task can be independent of when the data consumed by the task is produced. Another advantage is that accesses to values passed inside DDFs are guaranteed to be data race free and deterministic [25].

DDFs are an important generalization over futures, since in addition to allowing arbitrary data dependences they also allow the compiler to avoid blocking operations while tasks wait on the results of a computation. This is possible because of the explicit declaration of a data dependence in a DDF by an `async` in the `await` clause. However, there are two features currently lacking in the HS implementation of DDFs. Firstly, it is not possible to cancel a task which is waiting on a DDF. This translates to ensuring there is always a `put()` on a DDF if there is an `async` waiting on the DDF. Cancellations need to be handled inside the waiting `async` by checking the value inside the DDF and having different control paths for different values. Secondly, and more importantly, an `async` waiting on a chain of DDFs can only begin executing after a `put()` has been invoked on all the DDFs. This can limit the available parallelism in some applications. In such scenarios, it is advisable to use the actors construct provided in HS (Section 4).

```
object FibDdfPrimer extends HabaneroApp {
  def fib(n: Int, v: DataDrivenFuture[Int]): Unit = {
    if (n < 2) {
      v.put(n)
    } else {
      val res1 = ddf[Int]()
      val res2 = ddf[Int]()
      async { fib(n − 1, res1) }
      async { fib(n − 2, res2) }

      asyncAwait(res1, res2) {
        v.put(res1.get() + res2.get())
  } } }

  val N = Integer.parseInt(args(0))
  finish {
    val res = ddf[Int]()
    async { fib(N, res) }
    asyncAwait(res) {
      val fibResult = res.get()
      println("fib(" + N + ") = " + fibResult)
} } }
```

Figure 5. HS Fib using DDFs. Each call to `fib()` produces an `async` task that waits on values to be produced by its children before it computes the local result and stores it in the `v` (result) DDF. This version is more scalable compared to the futures version in Figure 4. It requires the programmer to change the natural flow of the program to think in terms of continuations and the DDFs.

3.2.3 Phasers and Accumulators

The `phaser` construct [22] unifies collective and point-to-point synchronization for phased computations. Each task has the option of registering with a `phaser` in *signal-only/wait-only mode* for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. These properties, along with the generality of *dynamic parallelism*, *phase-ordering* and *deadlock-freedom* safety properties, distinguish `phasers` from synchronization constructs in past work including barriers and X10s clocks [4]. The latest release of j.u.c in Java 7 includes Phaser synchronizer objects, which are derived in part [15] from the `phaser` construct in the Habanero runtime. (The j.u.c. Phaser class only supports a subset of the functionality available in HJ `phasers`). In general, a task may be registered on multiple `phasers`, and a `phaser` may have multiple tasks registered on it. Three key `phaser` operations are:

- *new*: When a task T performs a new `phaser()` operation, it results in the creation of a new `phaser` ph such that T is registered with ph in the *signal-wait* mode (by default).
- *registration*: The statement:
`asyncPhased (ph1.inMode(mode1), ... )` ⟨*stmt*⟩, creates a child task that is registered on `phaser ph1` with `mode1`, etc. The registrations of a child task must be a subset of the parent task's registrations. `asyncPhased` ⟨*stmt*⟩ simply propagates all of the parent's `phaser` registrations to the child.
- *next*: The `next` operation has the effect of advancing each `phaser` on which the invoking task T is registered to its next phase, thereby synchronizing all tasks registered on the same `phaser`. In addition, a `next` statement for `phasers` can optionally include a *single* statement, `next` ⟨*stmt*⟩. This guarantees that the statement is executed exactly once during the phase transition.

```
object IterativeAveragingPhaserApp {
  def run(data: Array[Double]): Unit = {
    val n = data.length − 2
    finish {
      val allPhasers = Array.tabulate(n + 2) {
        i => phaser()
      }
      for (1 to n) { ctr =>
        val (me, left, right) = (ctr, ctr − 1, ctr + 1)

        val leftPh = allPhasers(left).inMode(WAIT)
        val selfPh = allPhasers(me).inMode(SIG)
        val rightPh = allPhasers(right).inMode(WAIT)

        asyncPhased(leftPh, selfPh, rightPh) {
          //arbitrary limit instead of convergence test
          for (0 until (10 ∗ n)) { loop =>
            // first compute the value
            val newVal = (data(left) + data(right)) / 2
            // Allow others to proceed and modify data
            next
            // update the local value
            data(me) = newVal
            // notify others of value update
            next
} } } } } }
```

Figure 6. Iterative averaging using `phasers` in HS. Each task owns a data location and waits on data to be produced at the left and right locations in each iteration (phase). Note the use of `next` to make progress in each phase of the computation.

`phasers` ensure deadlock freedom when programmers use only the `next` statements in their programs. In programs where tasks are involved with multiple point-to-point coordinations, explicit use of `doWait()` and `doSignal()` on multiple `phasers` might be required. In these situations, some effort is required on the part of the programmer to carefully reason about the sequence of such calls to ensure correctness and deadlock freedom.

HS also has support for phaser accumulators [23], a parallel reduction construct that meshes seamlessly with phasers. Accumulators overlap communication and computation by separating reduction computations into the parts of sending data, performing the computation itself, and retrieving the result. Accumulators support two logical operations:

- `send(value)`: to send a value for accumulation in the current phase, and
- `result()`: to receive the accumulated value from the previous phase.

Each `send()` operation on an accumulator in the same phase is treated as a separate contribution to the reduction. Since the `result()` operation returns the accumulated value from the previous phase, it does not encounter any race conditions with `send()` operations in the current phase. In the current implementation of HS, only tasks registered to their respective phasers in *signal-wait* or *signal-wait-next* modes are supported with accumulators.

4. Coordination using Actors

An actor is defined as an object that has the capability to process incoming messages produced by other actors. Typically, the actor has a mailbox to store its incoming messages. An actor also maintains local state which is initialized during creation. Henceforth, only the actor is allowed to update its local state using data (usually immutable) 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. As an actor processes a message, it is allowed to change its behavior affecting how it processes the subsequent messages.

HS combines the AFM and the AM to create a hybrid model with Actors. The actor implementation in HS uses data-driven constructs instead of using exceptions for control flow as available in the standard Scala actor library [8, 9]. From the AFM perspective, actors in HS are treated as long running `asyncs` and hence can nest any of the Async/Finish compliant constructs in their message-processing body (MPB). This simplifies termination detection and enables exposing parallelism inside the actor while processing messages.

Under the hybrid model, the life cycle of the actor includes a new *paused* state and two new operations: `pause` and `resume` as described in Figure 7. Neither of these operations are blocking, they only affect the internal state of the actor that controls when messages are processed from the mailbox. 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. As we will see in Section 4.2, the *paused* state simplifies parallelization in actors.

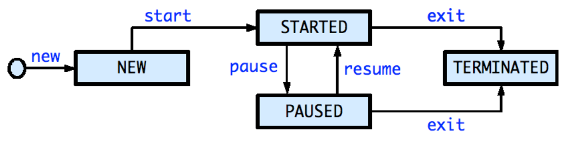

Figure 7. Actor life cycle extended with a *paused* state. The actor can now continually switch between the *started* and *paused* states using the `pause` and `resume` operations.

4.1 Habanero-Scala Actors

HS supports two implementations of actors that support the hybrid model: support to run the actors from the standard Scala actor library under the hybrid model that provide support for a subset of the operations presented in the hybrid model, and secondly, HS provides its own implementation of actors that supports the full complement of operations including the `pause` and `resume` operations. We designate the two actor implementations as the *heavy* and *light* actors, respectively.

4.1.1 *Heavy* Actors

Heavy actors are an extension of standard Scala actors and are so called since their implementation involves more overhead than *light* actors presented below. To support operations like `receive` (called `react` for EBAs) 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 is implemented as a trait that extends the standard Scala Actor trait. HS *heavy* actors do not support the `pause` and `resume` operations. 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.

4.1.2 *Light* Actors

Light actors are a complete implementation of actors in the hybrid model. *Light* actors are started using a call to `start()` and the MPB is triggered only on the messages they receive. *Light* actors do not need to use exceptions to manage the control flow, and they also execute more efficiently compared to the corresponding *heavy* actors. The continuations are stored via the state of member variables and an explicit partial function that defines the behavior of the actor (i.e., the steps to execute while processing a message). The mailbox of the actor is maintained using a concurrent linked list of *data-driven controls* (DDCs)[2]. *Light* actors support both ordered and unordered insertion of messages into the mailbox.

Supporting `pause` and `resume` *Light* actors support the `pause` and `resume` operations using synchronous DDCs. A call to `pause()` creates a new DDC. The MPB checks for the presence of the DDC to determine if the actor is in a *paused* state. If so, it creates a continuation to process subsequent messages and registers the activity with the DDC. When `resume()` is called, the state of the actor is reset, and the DDC is provided a value to synchronously trigger the execution of the continuation.

Supporting `become` and `unbecome` *Light* actors explicitly 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 defined as partial functions. The behavior history is maintained in a stack and the old behavior can be retrieved by an `unbecome` operation. If at any point, the current behavior cannot process a message (i.e. the partial function is not defined for the message), the actor terminates and throws an exception. This is unlike the standard Scala actor behavior where messages are retained in the hope that they will be processed later. The thrown exception is caught by IEF of the actor instance and handled as explained in Section 2.

4.2 New Capabilities with HS Actors

With the hybrid model in place, there are a number of constructs that can now be supported in HS actors. 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.

4.2.1 Termination Detection

Mapping actors into the AFM provides a clean and transparent mechanism to detect the termination of actors. Some actor implementations on the JVM (e.g., Scala Actors library [9], Kilim [24], jetlang [21]) 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 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 hybrid actors and the `finish` construct to detect termination easily.

[2] A DDC lazily binds a value and an execution body (EB), when both these are available a task that executes the EB using the value is scheduled

```
object HSPingPong extends HabaneroApp {
  finish {
    val pong = new HsPong()
    val ping = new HsPing(numMsgs, pong)
    ping.start
    pong.start
    ping ! HsStart
  }
  println("Both ping and pong terminated")
}
object ScalaPingPong extends App {
  val latch = new CountDownLatch(2)

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

  latch.await()
  println("Both ping and pong terminated")
}
// Code for Ping and Pong actors not displayed
```

Figure 8. Implicit actor termination detection using `finish` in HS actors. Terminating the actor 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` as used in the Scala actor example.

4.2.2 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 while processing messages as long as *at most one message-processing rule* is maintained. Retrofitting actors to run inside a AFM 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:

Using `finish` constructs 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 MPB, we can achieve parallelism by creating new `async-finish` constructs inside the MPB. We spawn off new tasks to achieve the parallelism at the cost of blocking the original message-processing task at the wrapping `finish`. Since the main message-processing task only returns after all spawned tasks 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.

Allowing escaping `async` tasks Requiring all spawned `async`s to be contained in a single MPB instance is too strict. This restriction 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

```
class ParallelizedActor() extends HabaneroReactor {
  override def behavior() = {
    case msg: SomeMessage =>
      // preprocess the message
      finish {
        async {... /*do some processing in parallel*/ }
        async {... /*do more processing in parallel*/ }
      } // finish to ensure all spawned tasks complete
    ...
} }
```

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

the same actor. We can achieve this invariant by using the *paused* state introduced earlier and 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 10 shows an example in which the `pause` and `resume` operations are used to achieve parallelism inside the MPB.

```
class ParallelWithEscapingAsyncsActor() extends ↩
    HabaneroReactor {
  override def behavior() = {
    case msg: SomeMessage =>
      pause() // prevent actor from proc. next message
      async { /*do some processing in parallel*/ }
      async {
        // do more processing in parallel.
        resume() // when safe to resume processing ↩
             other messages
        // some more processing
      }
    ...
  }
}
```

Figure 10. 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.

4.2.3 Non-blocking `receive` operations

Implementing the *synchronous* `receive` operation often involves blocking and can limit scalability in virtual machines that do not allow explicit call stack management and continuations. Indeed the implementation of `receive` in the Scala actor library involves blocking the currently executing thread and degrades performance. An 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 be implemented in the hybrid model without blocking threads or using exceptions. This, however, requires support of the DDF coordination construct. A DDF can be passed along to the actor 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 11 shows an example of a non-blocking `receive` implementation.

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

Figure 11. A HS actor that uses DDFs to simulate the synchronous `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 this `async` is triggered, the actor processes the value in the DDF and performs the `resume` operation to continue processing subsequent messages.

4.2.4 Stateless Actors

```
class StatelessActor() extends HabaneroReactor {
  ...
  override def behavior() = {
    case msg: SomeMessage =>
      async {
        processMessage(msg)
        if (enoughMessagesProcessed) { exit() }
} } }
```

Figure 12. A simple stateless actor. The MPB 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.

Stateless actors are allowed to process multiple messages simultaneously since they maintain no mutable internal state. As shown in Figure 12, it is easy to create such actors in HS using escaping `async`s. 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.

5. Locality Control using `places`

HS supports a concept called `places` which is a logical location where tasks are run. The number of `places` is fixed at the time a HS program is launched, the command line option, `-places p:w`, allows the user to specify how many `places` (`p`) and workers per `place` (`w`) the runtime should be initialized with. The Habanero runtime supports thread-binding capabilities to deliver on the programmer's expectation that workers in the same `place` should have greater affinity with each other than workers in different `places`. Places are used promote data locality while tasks are scheduled for execution by the runtime. A reference to the current `place` can be obtained by invoking `here()`. The set of `places` are ordered and the methods `place.next()` and `place.prev()` may be used to cycle through them. A Habanero task executes at a single `place`, but it may spawn activities in other `places`. Figure 13 shows a simple program using `places`. In addition to promoting data locality, `places` also allow the user to provide a hint to the runtime to achieve load balance of tasks among threads. As such, `places` are useful while using actors to control where the MPB of an actor is executed and helps reduce contention in the task queue and improves throughput. Programmers can configure the `place` for the message processing task for both *light* and *heavy* actors when the actors are started.

```
object PlaceApp extends HabaneroApp {
  finish {
    async {
      println("A. At " + here)
      async(here.prev()) {
        println("B. At " + here)
      }
      async(here.next()) {
        println("C. At " + here)
} } } }
```

Figure 13. HS program using `places` to control where task are executed

6. Experimental Results

6.1 Experimental Setup

The benchmarks were run on a 16-core (quad-socket, quad-core per socket) Intel Xeon 2.4 GHz system with 32 GB memory, running Red Hat Linux (RHEL 5). Each core had a 32 kB L1 cache and a 3 MB L2 cache. It also included a Sun Hotspot JDK 1.6, the latest versions of HJ and HS, and version 2.9.1.final of Scala. The JVM memory was set to 1 GB. Each of the benchmarks was run ten times in the same JVM instance, and the geometric mean of the middle eight execution times (sorted by time) are reported. We attempted to perform a full-heap garbage collection (GC) before each iteration to avoid non-determinism across multiple runs from GC. This process of reporting execution times is similar to the one used in [14]. All actor implementations of a program use the same algorithm and mostly involved renaming the parent class of the actors to switch from one implementation to the other. While evaluating the application benchmarks, the schedulers were set up to use 16 worker threads. All the implementations of the different actor frameworks use their corresponding Scala api and hence experience similar pattern matching overheads.

6.2 Micro-Benchmarks

6.2.1 Overhead of Habanero-Java vs Habanero-Scala

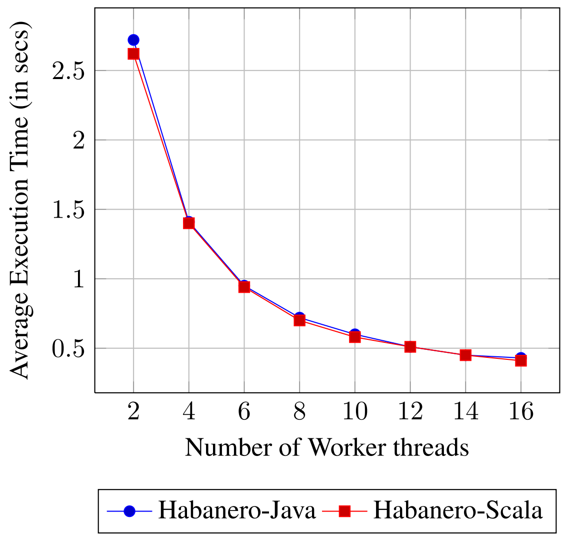

Figure 14. `daxpy` benchmark Habanero-Scala and Habanero-Java comparison.

In Figure 14 the performance of the same implementation of the `daxpy` benchmark written in HS and HJ are compared. In `daxpy` many tasks are spawned under a single finish scope and it is *embarrassingly parallel*. This benchmark stress tests the ability of the runtime to handle and schedule large number of tasks. Both HJ and HS share the same underlying Habanero runtime and show similar execution times. In HS, the library needs to wrap the closures for the tasks inside additional `Runnable` or `Callable` instances before handing off to the runtime, whereas in HJ the compiler translates the code and inserts direct calls to the runtime.

6.2.2 Overhead of Actors in Scala vs. Habanero-Scala

The first benchmark (Figure 15) 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 Scala version obtained from [27] is translated to all the other libraries. 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

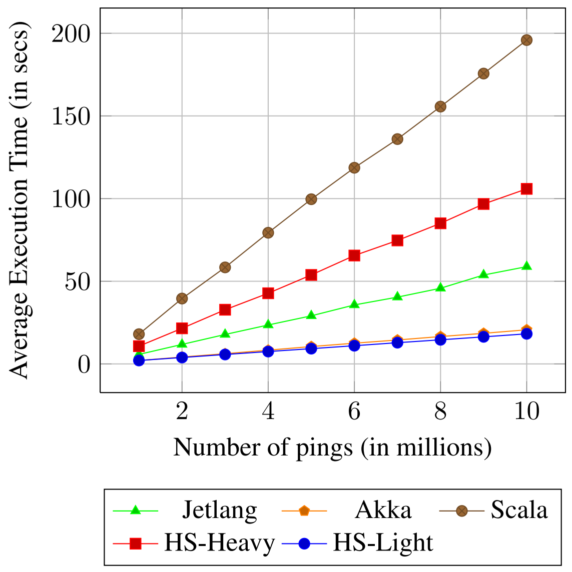

Figure 15. The `PingPong` benchmark exposes the throughput and latency while delivering messages. There is no parallelism to be exploited in the application.

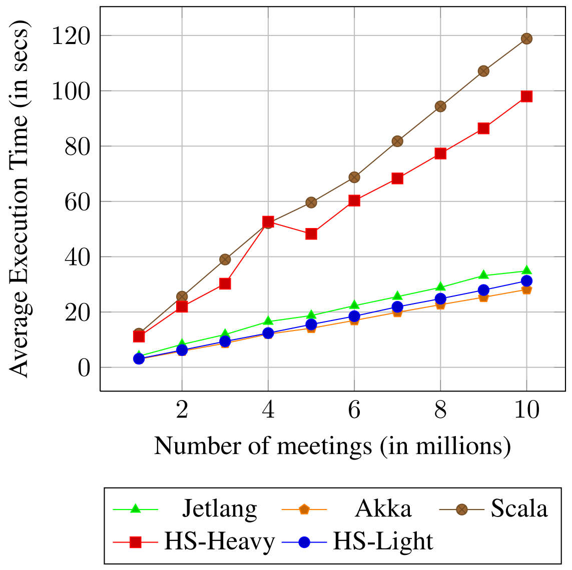

Figure 16. The `Chameneos` benchmark exposes the effects of contention on shared resources. The `Chameneos` benchmark involves all *chameneos* constantly sending messages to a coordinator actor that coordinates which two *chameneos* get to meet.

mentioned in Section 4.1.1). Akka and HS *light* actors perform best and display similar running times.

The `Chameneos` benchmark, shown in Figure 16, tests the effects of contention on shared resources (the mailbox implementation) while processing messages. The Scala implementation was obtained from the public Scala subversion repository [10]. The other implementations 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 two 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 and HS *heavy* actors share the same implementation they show similar performance. The HS *light* actor implementation that uses DDCs outperforms the immutable linked list implementation in Scala. Again, the HS *light* actor implementation benefits from avoiding exceptions in the control flow. Jetlang, Akka, and HS *light* actors benefit from batch-processing messages inside tasks and have similar running times.

6.3 Application Benchmarks

6.3.1 Quicksort

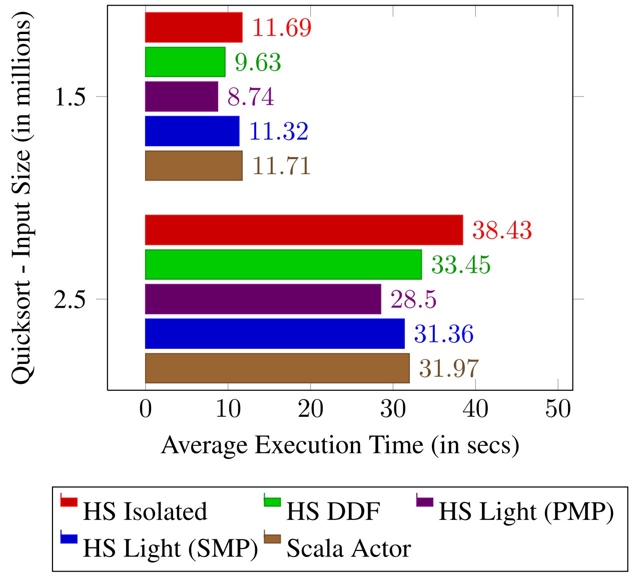

Figure 17. Quicksort results.

The Quicksort benchmark is an application that 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. Figure 17 compares a parallel actor message-processing (PMP) implementation in HS with pure actor implementations in HS and Scala and Async/Finish style implementations using isolated and DDFs in HS. Pure actor implementations in HS involve sequential message-processing (SMP). The HS *light* actor implementation in HS is faster than the Scala actor implementation and the DDF implementation. An isolated version that attempts to duplicate the hybrid implementation is initially competitive with the actor implementations but slows down as more parallel tasks are available (in the larger problem). It does not compete with the hybrid implementation due to the serialization of all isolated blocks by the Habanero runtime, though more sophisticated implementations of isolated (e.g. [13]) may mitigate this issue. The parallelized actor implementation exposes the most parallelism in the application with the lowest overhead and is easily the best-performing implementation, around 9%, 11% and 15% better than the HS *light* actor, Scala actor, and DDF implementations, respectively.

6.3.2 Sieve of Eratosthenes

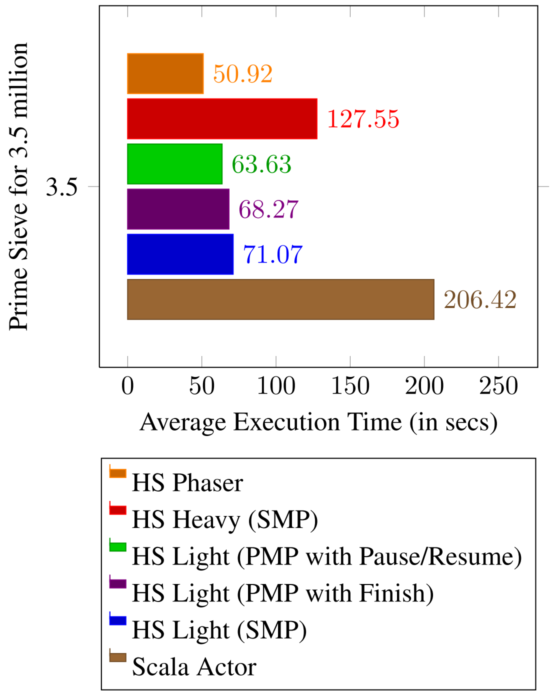

Figure 18. Sieve of Eratosthenes: Each individual actor maintains a local buffer of 2000 primes, creating a new actor when the buffer becomes full. In the parallelized actor implementations, a candidate is checked for local *primeness* in parallel. The HS actor implementations also utilize data locality by binding actors to places. There is a cut-off beyond a length of ten actors where parallelization of the prime checks are disabled.

Next, the Sieve of Eratosthenes [29] application which exhibits pipelined parallelism is benchmarked (Figure 18). The algorithm incrementally builds knowledge of primes. It is easy to build a pipelined version 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. It is possible to further expose the parallelism in the algorithm by performing the local prime checks in parallel using parallelization inside the stages, thus bringing the hybrid model into the picture. The phaser implementation, which uses mainly AFM constructs, is easily the fastest. However, the implementation requires careful tuning of the configuration to ensure that the number of tasks created are not greater than the number of available worker threads. In contrast, the different actor implementations can run on any problem size with configurations independent of the problem size. The actor implementations create and schedule tasks for each candidate flowing through the pipeline. As such, at least 3.5 million tasks are created as opposed to the 16 tasks created in the phaser implementation. Of the hybrid implementations, both the finish-based and the `pause-resume`-based implementations are faster than other actor implementations by at least 4% and 10% respectively. The `finish`-based implementation involves thread blocking operations around the `finish` in the habanero work-sharing runtime as opposed to the `pause-resume`-based implementation and is hence slower of the two. The HS *heavy* actor comfortably outperforms the Scala actor version. In general, the thread binding support (which binds worker threads to hardware processors) in the Habanero runtime helps the HS actor implementations outperform the Scala actor implementation.

7. Related Work

The different parallel constructs in HS, apart from actors, are based on the work-sharing implementation of Habanero-Java (HJ) [3]. HJ implements a variant of the Fork/Join Model (FJM) called the Async/Finish Model (AFM) to support lightweight dynamic task creation and termination. The advantages of using the constructs in HJ, and by extension that in HS, over the corresponding constructs provided in the `java.util.concurrent` package have previously been addressed in [2]. A popular implementation of the FJM in a programming language was presented in Cilk developed at MIT by Blumofe, et al. [1]. A key innovation in Cilk was to present an efficient work-stealing scheduler for these computations which tries to execute tasks in a depth-first manner on each worker. The success of Cilk gained much attention in the research community and led to the development of further programming languages supporting variants of the FJM: Cilk++ [12], X10 [4], Thread Building Blocks [20], Java Fork/Join Framework [11], OpenMP 3.0 [17] etc. Funicular [16] is a Scala library inspired by X10 which provides support for `async-finish` constructs. However, unlike HS, it has limited support for phaser registration modes and does not support DDFs, accumulators or integration of `async-finish` constructs with actors.

The AM was developed due to Hewitt's anticipation that a parallel combination of computing machines was needed to solve the problems posed by AI researchers. The programming language Erlang, developed at Ericsson, opted to implement the AM as their preferred model of concurrency [28]. The success of Erlang in production settings has catapulted the AM into the mainstream prompting a proliferation of the development of actor frameworks in popular sequential languages, such as C/C++, Smalltalk, Python, Ruby, and .NET. There are a handful of actor implementations for JVM languages. Jetlang [21] and Kilim [24] provide a low-level messaging API that can be used to build actors. Kilim also ensures data isolation as required in the AM. Scala includes an actor library [9] that provides thread-based and event-based actors. GPars [26], implemented in Groovy, provides a similar actor library. Both Scala and GPars actors are inspired from Erlang and simulate continuations by using exceptions to manage the control flow.

We are unaware of any previous work that attempts to integrate the AFM and AM, and, as such, HS provides a unique implementation of async-finish tasks and actors. HS *light* actors provide similar features to the Scala and GPars actors while avoiding exceptions for control flow. HS actors, however, do not support data isolation in messages as provided by Kilim. HS actors also implement the hybrid model that can use other `async-finish` compliant constructs to simulate more effectively certain actor constructs, for example, synchronous receive without blocking threads.

8. Conclusions and Future Work

This paper introduced Habanero-Scala (HS), an implementation that combines two programming models: the Async/Finish Model (AFM) and the Actor Model (AM). HS supports creation of lightweight tasks embodied in `async`, future, and `foreach` constructs; termination detection using the `finish` construct; locality in the form of `places`; weak isolation using `isolated` blocks; and task coordination patterns using `phasers`, data-driven futures and actors. These constructs make writing parallel programs much easier and improve programmer productivity. HS provides a competitive runtime to write parallel programs and programs written using HS constructs are easily faster than ones written using the standard Scala actor library.

We are currently exploring the option of using different continuation based solutions to avoid thread blocking operations on end of `finish` scopes, future `get()` and `phasers`. The `async-finish` compliant constructs such as `finish`, `async`, DDFs, and `phasers` supported by the Habanero runtime currently only work in shared memory systems. The AM was initially created for distributed memory systems. With the mapping of actors onto the AFM, it should be possible to port these constructs onto a distributed system. This is another interesting direction for extending HS and an area for future research.

Availability

A public distribution of Habanero-Scala is available for download at:
`http://habanero-scala.rice.edu/`

Acknowledgments

We thank the reviewers for their comments and suggestions. We are also grateful to Vincent Cavè, Dragoş Sbîrlea and

Sağnak Taşırlar for discussions on the Habanero runtime, phasers and DDFs, respectively.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30:207–216, August 1995. ISSN 0362-1340.

[2] V. Cavé, Z. Budimlić, and V. Sarkar. Comparing the Usability of Library vs. Language Approaches to Task Parallelism. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 9:1–9:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0547-1.

[3] V. Cavè, 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.

[4] 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.

[5] O. Danvy and A. Filinski. Abstracting Control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 151–160, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. doi: http://doi.acm.org/10.1145/91556.91622. URL `http://doi.acm.org/10.1145/91556.91622`.

[6] B. Goetz. Thread pools and work queues, July 2002. URL `http://www.ibm.com/developerworks/library/j-jtp0730/index.html`.

[7] 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.

[8] P. Haller and M. Odersky. Event-Based Programming Without Inversion of Control. In *In Proc. Joint Modular Languages Conference (2006), Springer LNCS*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-40927-4.

[9] P. Haller and M. Odersky. Actors That Unify Threads and Events. *In 9th International Conference on Coordination Models and Languages*, 4467:171–190, 2007.

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

[11] D. Lea. A Java Fork/Join Framework. In *Java Grande*, pages 36–43, 2000.

[12] C. E. Leiserson. The Cilk++ Concurrency Platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-497-3.

[13] R. Lublinerman, J. Zhao, Z. Budimliċ, S. Chaudhuri, and V. Sarkar. Delegated Isolation. In *Proceedings of OOPSLA 2011*, October 2011.

[14] P. McGachey and A. L. Hosking. Reducing Generational Copy Reserve Overhead with Fallback Compaction. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 17–28, New York, NY, USA, 2006. ACM. ISBN 1-59593-221-6.

[15] Miller, Alex. Set your Java 7 Phasers to stun. URL `http://tech.puredanger.com/2008/07/08/java7-phasers/`.

[16] Nystrom, Nathaniel. Funicular concurrency library for Scala, 2010. URL `http://ranger.uta.edu/~nystrom/funicular/`.

[17] OpenMP Architecture Review Board. OpenMP Application Program Interface. *Review Literature And Arts Of The Americas*, 1(May):1997–2008, 2008.

[18] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A Generic Parallel Collection Framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 136–147, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23396-8.

[19] 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.

[20] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O'Reilly, 2007. ISBN 978-0-596-51480-8.

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

[22] 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.

[23] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser Accumulators: a New Reduction Construct for Dynamic Parallelism. *Computer*, 2009.

[24] 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.

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

[26] The GPars team. The GPars Project - Reference Documentation. URL `http://www.gpars.org/guide/guide/`.

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

[28] 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.

[29] Wikipedia, The Free Encyclopedia. Sieve of Eratosthenes. URL `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.