RICE UNIVERSITY

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

by

Shams Mahmood Imam

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Vivek Sarkar, Chair
Professor of Computer Science
E.D. Butcher Chair in Engineering

Robert S. Cartwright Jr
Professor of Computer Science

Swarat Chaudhuri
Assistant Professor of Computer Science

Houston, Texas

December, 2011

ABSTRACT

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

by

Shams Mahmood Imam

This study presents a hybrid concurrent programming model combining the previously developed Fork-Join model (FJM) and Actor model (AM). With the advent of multi-core computers, there is a renewed interest in programming models that reduce the burden of reasoning about and writing efficient concurrent programs. The proposed hybrid model shows how the divide-and-conquer approach of the FJM 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 hybrid model adds actor creation and coordination to into the FJM, while also enabling parallelization within actors. This study uses the Habanero-Java and Scala programming languages as the base for the FJM and AM respectively, and provides an implementation of the hybrid model as an extension of the Scala language called Habanero-Scala. The hybrid 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.

Acknowledgments

In the name of Allah, the Most Gracious and the Most Merciful.

I would like to express my sincere gratitude to my advisor, Vivek Sarkar, for his constant support, enthusiasm, and immense knowledge. His guidance and encouragement helped me in completing this thesis. I cannot imagine having a better advisor and mentor.

I am also grateful to other members of my thesis committee, Robert S. Cartwright Jr and Swarat Chaudhuri, for their patience, encouragement, and constructive feedback.

I would like to thank my colleagues in the Habanero Multicore research group whose work provided a base I could build upon. In particular, I am grateful to Vincent Cavè, Dragoş Sbîrlea and Sağnak Taşırlar for discussions on the Habanero runtime, phasers and DDFs, respectively. I would also like to thank Jill Delsigne for providing comments on early drafts of this thesis.

I thank my friends for all the support, camaraderie, and enjoyable times they provided.

Finally, and most importantly, I would like to thank my family. My parents, Khondker Ali Mahmood and Shamsun Nahar, have instilled an appreciation for education in me. It is thanks to my sister, Asma Imam, that I became interested in Computer Science. They have helped me get through the difficult times and provided constant emotional support. To them I dedicate this thesis.

Contents

Illustrations

Chapter 1

Introduction

1.1 Motivation

Until recently, increases in processor clock speed have provided steady performance improvements for programs without requiring any rewrites. This increase in processor clock speed has reached fundamental limits with current silicon technology due to the limitations in thermal management of the heat that needs to be dissipated away [1]. The processor industry has responded to this challenge by developing multi-core processors. These multi-core processors continue to provide increased computational ability but, the onus is shifted on to the software to utilize the parallelism available on the hardware. There are no more *free lunches* for the software developers and this has led to the Software Concurrency Revolution [2].

Current mainstream programming languages provide limited support for expressing parallelism in software [3]. Programmers, hence, need new parallel programming models to extract performance from the hardware with ease, and reduce on themselves the burden of reasoning about and writing parallel programs. This has led to a renewed interest of parallel programming models in the academic community. Programs typically exhibit varying degrees of task, data and pipeline parallelism [4]. A handful of various programming models have been developed to handle task and pipeline parallelism. In this thesis, we focus on two such models:

- The Fork-Join Model (FJM) which is well suited to exploit task parallelism

in divide-and-conquer style and loop-style programs usually written in shared memory models,

- The Actor Model (AM) which promotes the *no-shared mutable state* and event-driven philosophy.

1.2 Thesis Statement

The thesis statement is as follows:

A hybrid parallel programming model that integrates Fork-Join Model and Actor Model helps solve certain class of problems more efficiently and productively than either of the aforementioned models individually.

1.3 Contributions

This thesis makes the following contributions:

- A hybrid programming model that unifies the Fork-Join and Actor models.
- An implementation of the hybrid programming model in an extension of the Scala language called Habanero-Scala.
- An efficient implementation of the Actor model using data-driven constructs instead of using exceptions for control flow.
- A study of application characteristics that are amenable to being more efficiently solved using the hybrid model compared to the FJM or AM.
- Experimental evaluation of performance benefits of using the hybrid model in such applications.

1.4 Organization

This thesis is organized as follows:

- Chapter 2 provides some background and introduces related work in the FJM and the AM.
- Chapter 3 describes the FJM and a variant called the Async-Finish Model (AFM). It also analyzes the advantages and limitations of currently existing coordination constructs in the AFM.
- Chapter 4 describes the AM and its advantages and disadvantages.
- Chapter 5 introduces the Hybrid Model and describes how the AFM and AM can be integrated. It also covers new constructs in the hybrid model.
- Chapter 6 describes our implementation of the hybrid model in an an extension of Scala called Habanero-Scala.
- Chapter 7 analyzes properties of applications which can benefit from the hybrid model and also gives concrete examples.
- Chapter 8 presents the experimental results.
- Chapter 9 wraps up by summarizing the thesis and areas of future work.

Chapter 2

Background

In 1965, Gordon Moore predicted that the number of transistors on an integrated circuit will double every eighteen months to two years based on his observations of the trend between 1958 to 1965 [5]. Technology advancements meant transistors were getting smaller in size thus allowing a larger number of them to be placed on the circuits. The prediction popularly became known as *Moore's Law*. Since smaller transistors were accompanied by corresponding increase in processor clock frequencies, Moore's Law is often associated with a trend of exponential increases in the computing performance of processors.

However, as with all exponential models this improvement in technology could not be sustained and a limit was reached around early 2005. Mendelson [1] showed that power constraints forced on the system implied that the frequency trend extrapolated from Moore's Law could not continue much longer. The power consumed by a processor using current technologies is proportional to the cube of the clock frequency. In addition, the close packing of the transistors meant that there was more operational heat being generated per unit area than could be dissipated often causing the chips to overheat and malfunction. This phenomenon came to be known as the Power Wall and led to a change in processor architectures. The Power Wall resulted in the plateauing of processor clock frequencies and the development of multi-core processors. Multi-core processors allow Moore's Law to continue to hold with respect to the number of transistors, but the associated expectation of doubling in clock frequency

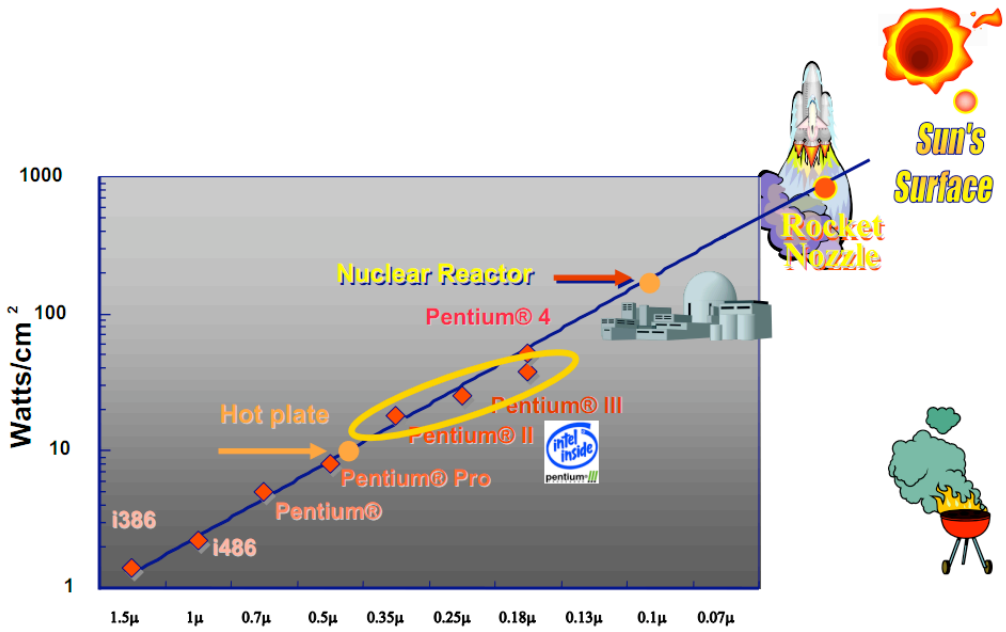


Figure 2.1 : Power-density of modern processors [1]. The figure displays that the exponential trend cannot continue as there will be extreme power generated in the processors which cannot realistically be dissipated. This poses limitations on the clock frequencies and transistor density on the chips.

no longer holds.

Herb Sutter, in 2005, noted that major processor manufacturers had moved to the development of multi-core architectures and claimed this to be a fundamental turning point for software [2]. Software applications written in the past could no longer rely on clock frequency improvements in hardware for continued performance improvement. Instead, software now has to change to utilize the many cores available in processors. The need to utilize these cores concurrently has been named as the Software Concurrency Revolution [6]. However, mainstream programming models were inherently sequential and were not equipped to efficiently utilize multi-core processors since the

available parallel programming models built on heavyweight threads offered neither performance nor productivity. There was a need to develop new parallel programming models for multi-core processors.

The concurrency revolution led to renewed interest in research in parallel programming models along with the related changes in compilers, runtime systems, and programming languages. The initial hope was to rely on implicit parallelism (also known as automatic parallelization) allowing the compiler to exploit parallelism opportunities in sequential code and improve programmer productivity by allowing the programmer to continue knowing old programming models. Proponents were encouraged by the success compilers had in exploiting instruction-level parallelism. However, the success of implicit parallelism has been limited to data-parallel languages [7, 8] and compiler techniques for parallelizing loops after performing dependence analysis on the loop bodies and determining that a parallel transformation of the loop body would be safe [9].

As Steffan and Mowry point out, complex access patterns in general programs makes detecting data dependences in solution fragments a difficult problem to solve and implicit parallelism by the compiler an unrealistic goal [10]. Mary Hall and et al., in [11], provide a similar tone when they state that current compilers have success stories in few areas, such as databases and computational science programs that deal with structured data. The general case in which programs deal with unstructured data is still in need of higher level abstractions to allow parallel programming to become mainstream and more easily accessible to programmers.

There has also been an effort to create programming models that encourage implicit parallelism by offering language constructs in specific domains. Examples include Hadoop [12] to solve problems amenable to the map-reduce paradigm, StreamIt [13]

for streaming applications, Matlab Parallel Computing Toolbox [14] for vector and matrix operations, and NESL [7] for nested data parallelism. While these languages have been successful in their targeted domains, there is a need for more generic programming models.

With limited success attained by implicit parallelism approaches, recent research has focused on explicit parallelism approaches where it is the programmer responsibility to identify and demarcate opportunities for parallelism in code. This burdens the programmer to worry about exploiting concurrency opportunities in her solution in addition to the core task of solving the problem.

A host of such parallel programming models have been developed which include the two-sided message passing model as in MPI [15], the partitioned global address space model [16] as in Co-array Fortran [17], Chapel [18], UPC [19], and X10 [20], the general purpose graphical processing unit [21] in CUDA [22]. This thesis focuses on two such parallel programming models the Fork-Join Model (FJM) and the Actor Model (AM). These models provide concurrency/parallelism abstractions to ease some of the low-level parallelism burdens from the shoulders of programmers. Both the FJM and AM are variants of the Task Parallel Model, which is discussed next.

2.1 The Task Parallel Model (TPM)

In the TPM, the problem to be solved is broken down into a number of *lightweight* tasks. A relatively smaller number of workers (typically one per core or hardware context) distribute these tasks among themselves and execute independent tasks in parallel to completely solve the problem. When two tasks depend on each other due to data dependences or shared resources, they cannot be run in parallel. The tasks must coordinate and synchronize among themselves and execute in an agreed upon

sequence that satisfies all semantic dependences.

In multi-core architectures, the heart of the TPM lies in the fact that the creation and management of individual OS-level threads to execute each task is not profitable since tasks are usually short-lived [23]. Instead, the workers are OS-level threads and the number of workers/threads is usually a small multiple of the number of cores available. Progress, under normal scenarios, can continually be made towards executing the program as workers are always expected to find some task they can execute, while other tasks wait on dependences. The TPM is one of the more promising parallel programming models as it is both high-level and generic [24]. This model allows solving applications that work on both regular and irregular data since tasks can be continually generated as the computation unfolds.

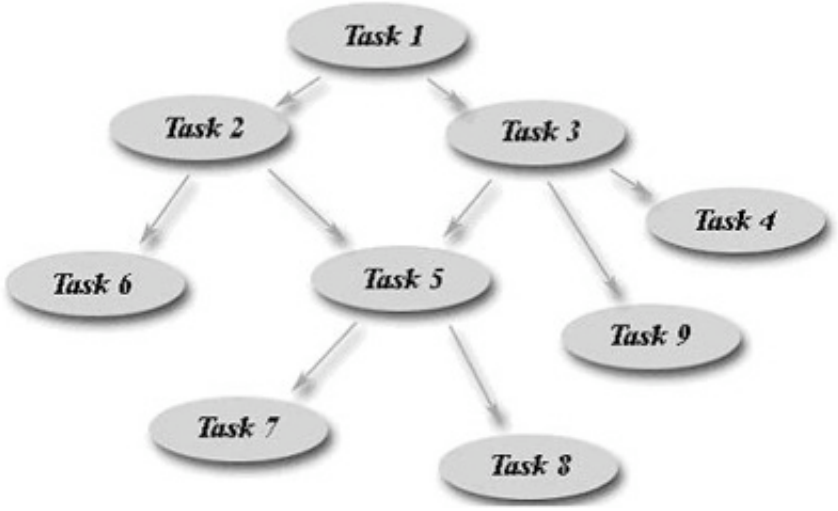


Figure 2.2 : Task Parallelism achieved by breaking down entire problem into many sub-tasks [source: http://nurkiewicz.blogspot.com/2011/01/activiti-processes-and-executions.html] .

2.1.1 The Fork-Join Model (FJM)

The FJM is a special case of the TPM. A recent popular implementation of the FJM in a programming language was presented in Cilk developed at MIT by Blumofe, Leiserson, and et al. [25]. In Cilk, computations are represented as directed acyclic graphs and proceed in a *fully-strict* manner. 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 TPM: Cilk++ [26], X10 [20], Thread Building Blocks [27], Java Fork-Join Framework [23], OpenMP 3.0 [28] etc. At Rice University, we have our own programming languages built on the FJM called Habanero-Java (HJ) [29] and Habanero-C [30].

The Habanero Multicore Software Research Group at Rice University has developed the HJ language which builds on past work with the X10 project at IBM [31]. As the presence of the term Java in the name suggests, HJ is an extension of the Java language and runs on standard Java Virtual Machines (JVMs). HJ implements a general version of the FJM called the Async-Finish Model to support lightweight dynamic task creation and termination [32]. Additional constructs such as Locality Control with the *place* construct [33, 34], Mutual Exclusion and isolation among tasks using the `isolated` construct [35, 36] and Collective and Point-to-Point synchronization using the *phasers* construct [37, 38] are also supported in HJ.

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 [39]. It was designed to address the problems that arise while writing distributed applications. Further work by Henry Baker [40], Gul Agha [41], and others added to the theoretical development of the AM.

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. With the emergence of multicore computers, nearly three decades later, the AM has gained renewed interest. The programming language Erlang, developed at Ericsson, opted to implement the AM as their preferred model of concurrency [42]. Erlang reported high scalability and an availability of 99.999%, i.e. a downtime of only 31ms in a year, in its telecom switch application at Ericsson [43].

Coupled with the success of Erlang in production settings, the AM was catapulted into the mainstream and there has been a proliferation of the development of Actor frameworks in popular sequential languages like C/C++ (Act++ [44]), Smalltalk (Actalk [45]), Python (Stackless Python [46], Stage [47]), Ruby (Stage [48]), .NET (Microsoft's Asynchronous Agents Library [49], Retlang [50]) and JVM-based languages (Scala Actors library [51], Kilim [52], jetlang [53], ActorFoundry [54], GPars [55]). In this thesis we focus on a modern implementation of the AM implemented in the Scala programming language [56] which runs on the JVM.

2.3 The Proposed Hybrid (Fork-Join + Actor) Model

Although both the FJM and AM have existed as parallel programming models for a while now, no systematic study has previously been undertaken to combine these two models. In this thesis, we present a hybrid parallel programming model that combines the divide-and-conquer approach of the FJM and the no-shared state and event-driven philosophy of the AM to collectively avoid synchronization issues and efficiently solve compute-intensive problems.

Chapter 3

The Fork-Join Model

In this chapter, we discuss various synchronization and coordination constructs currently supported in the Fork-Join model (FJM). As mentioned in Section 2.1.1, the FJM is a variant of the Task Parallel model. In the FJM, a parent task can *fork* 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* on a subset of child/descendent tasks. The task executing the join has to wait for all tasks created in the join scope to terminate before it can proceed. This is the primary form of synchronization among tasks in the FJM.

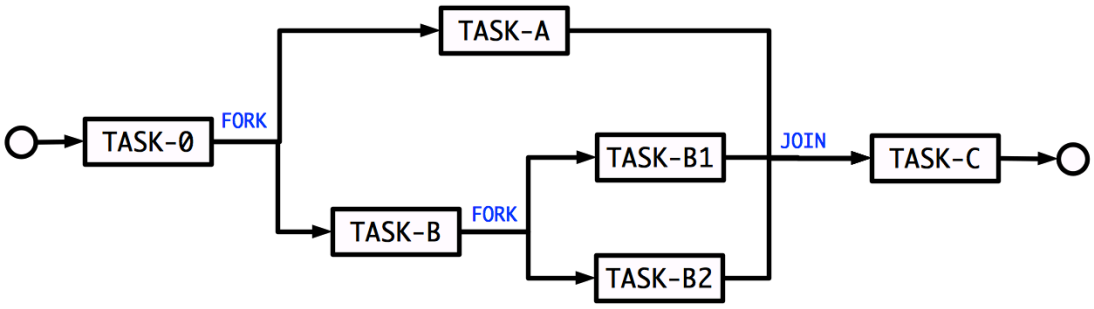


Figure 3.1 : Fork-Join Parallelism achieved by forking new tasks and joining before proceeding. Note that until all forked tasks (Task A, Task B, Task B1, and Task B2) reach the join point, Task C cannot be executed. [source: http://www.coopsoft.com/ar/ForkJoinArticle.html].

3.0.1 Fork-Join Parallelism in Habanero-Java (HJ)

Habanero-Java (HJ) is a parallel programming language developed by the Habanero Multicore Software Research Group at Rice University [29]. HJ supports Async-Finish Model (AFM) of programming which is a variant of the FJM. HJ also supports a handful of additional synchronization and coordination constructs between tasks that serve as extensions to the AFM. In the rest of this chapter, we use HJ to explain the various features of the such constructs in the context of the FJM.

3.1 Lightweight Tasks, Async-Finish synchronization

The central feature of any FJM implementation on multicore architectures is the ability to create and manage lightweight tasks. Tasks are created at *fork* points. HJ 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 [29]. 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.

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. The global finish rule ensures any executing HJ task has a unique *Immediately Enclosing Finish* (IEF). Besides termination detection, the `finish` statement plays an important role with regard to exception semantics. The IEF throws a *MultiException* [31] formed from the collection of all exceptions thrown by all `async`s in the IEF.

```
1  public class ForkJoinPrimer {
2    public static void main(String args[]) {
3      /* finish { implicit global finish wrapping main() in a HJ program */
4        System.out.println("Task O"); // Task-O
5        finish {
6          async { // Task-A
7            System.out.println("Task A");
8          }
9          async { // Task-B
10           System.out.println("Task B");
11           async { // Task-B1 created by Task-B
12             System.out.println("Task B1");
13           }
14           async { // Task-B2 created by Task-B
15             System.out.println("Task B2");
16           }
17         }
18       } // Wait for tasks A, B, B1 and B2 to finish
19       System.out.println("Task C"); // Task-C
20     /* } end of implicit global finish  */
21     // the global finish must wait for all nested tasks to terminate
22     // program terminates when the global finish terminates
23   }
24 }
```

Figure 3.2 : HJ version of the Fork-Join program from Figure 3.1

3.2 HJ Properties

3.2.1 Deadlock-Freedom and Determinism

Async-finish style computations usually represent directed-acyclic graphs, more specifically trees, and hence have property that they are deadlock free. In addition, in the absence of data races, these programs also have the extremely desirable property that they are deterministic [57].

3.2.2 Data Locality in HJ

Despite its attractiveness, one of the shortcomings in the popular implementations of the FJM is the issue of *data locality*. For non-trivial algorithms that are non-recursive,

encoding data locality implicitly by rewriting recursive versions of these algorithms is a challenging prospect for the best of programmers [24]. There is usually no way to pass around information to the task scheduler to influence the scheduler to schedule tasks based on data locality. This can lead to inefficient schedules for the tasks and hamper the performance of the application.

The concept of hierarchical *places* [33, 34] helps ease this data locality problem to some extent. However, the current HJ release does not include an implementation of hierarchical places. Part of the difficulty stems from the fact that an HJ task is free to access all visible variable references and the runtime or hardware often has to resort to implicit copying and syncing of data when accessed by tasks from different places. In fact, the management of the data layout and syncing of data is an area of active research in the PGAS programming community A programmatic construct that ensures data locality and avoids referencing remote data would be ideal. One approach is to use a place-based type system as in X10 [33]. Another is to use the Actor Model (discussed in Section 4.2).

3.2.3 Data Races and Synchronized Access

Another concern with the FJM, common to most shared memory models, is the issue of data races and the need to synchronize the accesses to shared resources/variables between tasks. Data races are notoriously difficult to get right even by experienced programmers. In addition to ordered synchronization constructs such as `finish`, HJ also provides an `isolated` $\langle stmt \rangle$ construct to support weak isolation, i.e. atomicity is guaranteed only with respect to other statements also executing inside isolated scope.

In HJ, isolated statements are implemented by using a single lock causing all iso-

```
1  public class IsolatedPrimer {
2    public static void main(String args[]) {
3      /* finish { implicit global finish wrapping main() in a HJ program */
4      final int[4] counter = new int[4];
5      finish {
6        for (int n = 0; n < 4; n++) {
7          for (int i = 0; i < 100; i++) async {
8            isolated {
9              // counter is modified in isolation, no data race
10             counter[n] = counter[n] + 1;
11           }
12         }
13       }
14     }
15     // the statement below would introduce a data-race as
16     // it is outside an isolated scope
17     // counter[0] = counter[0] + 1;

19     for (int n = 0; n < 4; n++) {
20       assertEquals(100, counter[n], "No data-race detected");
21     }
22     /* } end of implicit global finish  */
23   }
24 }
```

Figure 3.3 : HJ 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 [36] can exploit parallelism even when `isolated` is used extensively.

lated statements to be serialized. This can be a serious performance bottleneck in applications with moderate contention [29]. There is an alternate prototype implementation of HJ isolated statements using a technique called delegated isolation [36] which doesn't serialize non-interfering isolated statements and results in better performance and scalability.

Since HJ follows the shared memory model, it is easy for programmers to write programs in which multiple tasks access and mutate shared variables. To avoid data races, programmers need to resort to using synchronized accesses with `finish` or using `isolated` fragments or using atomic variables provided by the Java standard library

to avoid data races. Excessive use of these constructs introduces overhead into the runtime and can limit scalability of an application. It is desirable to have constructs provided by the language that can ensure data-race freedom while accessing variables. One approach, available in Deterministic Parallel Java [58], is to provide data-race freedom using an effect system to partition the heap and ensure, at compile time, that concurrent tasks are not involved in possible data-races. In HJ, Westbrook and et al. are exploring the idea of *Permission Regions* [59] to detect data-races. *Permission Regions* track read or write permissions on regions of code at runtime and report errors when tasks executing in parallel have conflicting permissions. Yet another approach, as present in the Actor model (discussed in Section 4.2), is to provide a mechanism to avoid data-races by ensuring only one task is executing on data at any point in time.

3.2.4 Coordination between tasks

While independent tasks can run in parallel, there are often dependencies between 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 alone. HJ augments the AFM with a handful of coordination constructs: Futures, Data Driven Futures and Phasers. These are described in the following sections.

Futures

A future represents the result of an asynchronous computation and extends HJ's `async` statements to `async` expressions. The statement:

```
1    final future<T> f = async<T> expression;
```

creates a new child task to evaluate `expression` 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 `expression` becomes available. One advantage of using futures is that there can never be a data race on accesses to a future's return value.

```
1  public class FibFuturePrimer {
2    public static Integer fib(final int n) {
3      if ( n < 2) {
4        return n;
5      } else {
6        final future<Integer> f1 = async<Integer> fib(n-1);
7        final future<Integer> f2 = async<Integer> fib(n-2);
8        return f1.get().intValue() + f2.get().intValue();
9      }
10   }

12   public static void main(String args[]) {
13     final int n = java.lang.Integer.parseInt(args[0]);
14     final Integer result = fib(n);
15     System.out.println("fib(" + n + ") = " + result);
16   }
17 }
```

Figure 3.4 : HJ Fib using futures. A relatively large value of n will cause the program to run out of memory due to excessive creation of threads by the HJ work-sharing runtime.

While futures are very simple to use, their injudicious use limits the performance and scalability of HJ programs. This is because calls to the `get()` on the future object blocks the current worker thread. In order to maintain the parallelism the HJ 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 [60]. Creating too many threads in one JVM can cause the system to run out of memory or thrash due to excessive memory consumption.

Data-Driven Futures (DDFs)

DDFs are an extension to futures to support the dataflow model [61]. 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 is available in the DDF. There are three main operations allowed on a DDF:

- `put(some-value)`: this non-blocking operation associates a value with the DDF. DDFs support the single assignment property which means only a single `put()` is allowed on the DDF during the execution of the program.
- `await()`: this is a blocking operation used by asyncs to delay their execution until some other task has `put()` a value into the DDF.

*Like other work-stealing runtimes, HJ's work-stealing runtime currently does not support blocking operations such as futures

- `get()`: this is a non-blocking operation 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.

DDFs allow the programmer to create arbitrary task graphs as advocated in libraries and frameworks that support directed acyclic graph parallelism. Traditionally, the FJM 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 race-free and deterministic [61].

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 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. 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 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. For example, in the `quicksort` example presented in Figure 3.6 the `async await(left, right) {...}` requires the task to wait until both the left and right values are available. An alternate construction of the waiting `async` as in Figure 3.7 mitigates the problem but still cannot handle the

```
1  public class FibDdfPrimer {
2    public static void fib(final int n, final DataDrivenFuture result) {
3      if ( n < 2) {
4        result.put(Integer.valueOf(n));
5      } else {
6        final DataDrivenFuture f1 = new DataDrivenFuture();
7        async fib(n-1, f1);
8
9        final DataDrivenFuture f2 = new DataDrivenFuture();
10       async fib(n-2, f2);
11
12       async await(f1, f2) {
13         final Integer v1 = (Integer) f1.get();
14         final Integer v2 = (Integer) f2.get();
15         final int resInt = v1.intValue() + v2.intValue();
16         result.put(Integer.valueOf(resInt));
17       }
18     }
19   }
20
21   public static void main(String args[]) {
22     final int n = java.lang.Integer.parseInt(args[0]);
23     finish {
24       final DataDrivenFuture result = new DataDrivenFuture();
25       fib(n, result);
26       async await(result) {
27         System.out.println("fib(" + n + ") = " + result.get());
28       }
29     }
30   }
31 }
```

Figure 3.5 : HJ 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 `result` DDF. This version is more scalable compared to futures version in Figure 3.4. It requires the programmer to change the natural flow of the program to think in terms of continuations and the DDFs.

case where `right` is available before `left`. In addition, the problem gets harder to manage when there are multiple DDF dependences.

```
1  public class QuicksortDdfPrimer {
2    public static void quicksort(final int[] inArr, final DataDrivenFuture result↩
         ) {
3      if (inArr.length == 1) {
4        result.put(inArr);
5      } else {
6        final int pivotIndex = selectPivot(inArr);
7        final int pivotValue = inArr[pivotIndex];

9        final int[] lessThanArr = getLessThan(inArr, pivotValue);
10       final DataDrivenFuture left = new DataDrivenFuture();
11       async quicksort(lessThanArr, left);

13       final int[] moreThanArr = getMoreThan(inArr, pivotValue);
14       final DataDrivenFuture right = new DataDrivenFuture();
15       async quicksort(moreThanArr, right);

17       final int[] center = getEqualsTo(inArr, pivotValue);
18       async await(left, right) {
19         final int[] sorted = merge(left.get(), center, right.get());
20         result.put(sorted);
21       }
22     }
23   }

25   public static void main(String args[]) {
26     final int[] input = generateInput();
27     final DataDrivenFuture result = new DataDrivenFuture();
28     quicksort(input, result);
29     async await(result) {
30     System.out.println("quicksort(" + toString(input) + ") = " + toString(↩
           result.get()));
31     }
32   }
33 }
```

Figure 3.6 : HJ Quicksort using DDFs. The async needs to wait on both `left` and `right` before it can make progress. In some cases it would be better to allow partial execution of the waiting `async` based on which result is available.

Phasers

Phasers are one of the more mature coordination constructs in HJ. They unify collective and point-to-point synchronization for phased computations. Details of phasers and their corresponding HJ syntax can be found in [29]. Phasers provide the ex-

```
1  public class QuicksortDdfPrimer {
2    public void quicksort(final int[] inArr, final DataDrivenFuture result) {
3      ...
4      async await(left) {
5        final int[] partial = merge(left.get(), center);
6        async await(right) {
7          final int[] sorted = merge(partial, right.get());
8          result.put(sorted);
9        }
10     }
11     ...
12   }

14 }
```

Figure 3.7 : HJ Quicksort allowing partial progress using DDFs. Partial progress can be made when value from left is available, however the still cannot handle the case when the value from right would be present before left.

tremely desirable property of deadlock freedom [37] 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 signals/waits on multiple phasers are required. Figure 3.8 shows an example of using phasers where the programmer has to explicitly manage the calls to `doWait()` and `signal()` in the `ThreadRing` benchmark. 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. Multiple producer-consumer coordination pattern is one scenario where careful management of phasers is required. Writing such code can prove to be quite cumbersome, especially if the producers join dynamically and produce arbitrary number of items.

In HJ, there is also an implementation limitation of phasers. There is a performance penalty when the number of tasks being generated in the program are larger than the number of available workers. This is because calls to `phaser.doWait()`

```
1  public class PhaserBasedThreadRingApp {

3    public static void main(String args[]) {
4      final int totalNumTasks = Integer.parseInt(args[0]);
5      final int hopsPerTask = Integer.parseInt(args[1]);

7      finish {

9        final phaser[] phasers = new phaser[totalNumTasks];
10       for (int i = 0; i < totalNumTasks; i++) {
11         phasers[i] = new phaser(phaserMode.SIG_WAIT);
12       }

14       for (int id = 0; id < totalNumTasks; id++) {
15         final int taskId = id;

17         final phaser selfPhaser = phasers[id];
18         final phaser nextPhaser = phasers[(id + 1) % totalNumTasks];
19         async phased(
20             selfPhaser<phaserMode.WAIT>,
21             nextPhaser<phaserMode.SIG>) {
22           for (int hop = 0; hop < hopsPerTask; hop++) {
23             if (taskId != 0 || hop != 0) {
24               selfPhaser.doWait();
25             }

27             // current async now has the token
28             System.out.println("Task-" + taskId + " has the token.");

30             if (hop + 1 != hopsPerTask || taskId + 1 != totalNumTasks) {
31               nextPhaser.signal();
32             }
33           }
34         }
35       }
36     }
37   }
38 }
```

Figure 3.8 : HJ ThreadRing using phasers. Tasks are connected in a ring and phasers make it simple to coordinate the passing of the token around the ring. The programmer has to carefully reason about the conditional clauses and the placement of the calls to `doWait()` and `signal()`.

(either explicit or implicit via the user of a `next` statement in a task registered on a phaser in wait mode) are blocking. As with futures, the HJ work-sharing runtime

has to compensate blocked threads by creating new threads and runs into similar limitations. Support for phasers in the HJ work-stealing runtime is still a subject for future work.

In summary, the coordination constructs provided in HJ are extremely powerful but suffer from certain limitations. A coordination construct that can mitigate/solve some of these limitations will be a welcome addition to the HJ language. In Chapter 4 the Actor model is introduced and its applicability into some of these scenarios is assessed. In Chapter 5 the hybrid model and its semantics are explained.

Chapter 4

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 [39]. It was designed to address the problems of distributed programs. Further work from Henry Baker [40] and Gul Agha [41] resulted in completing the theoretical development of the AM. The AM is primarily a message-based concurrency model. The key mantra is to encapsulate mutable state and use asynchronous messaging to coordinate activities between actors. An actor is the central entity in the AM that defines how computation proceeds.

4.1 Actors

An actor is defined as an object that has the capability to process incoming messages. Usually the actor has a mailbox, as shown in Figure 4.1, 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. Henceforth, the actor is only allowed to update its local state using data (usually immutable) from the messages it receives and intermediate results it computes while processing the message. The actor is restricted to process at most one message at a time. This allows actors to avoid data races and to avoid the need for synchronization as there is no other actor contending for access to its local data. There is no restriction on the order in which the actor decides to process incoming messages. As an actor processes a message, it is

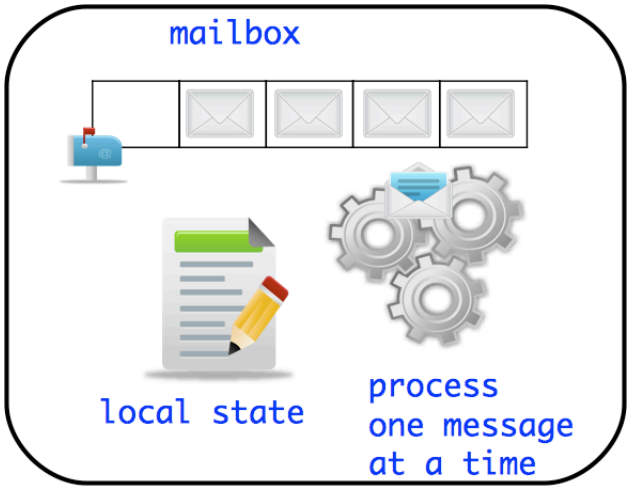


Figure 4.1 : Actors store incoming messages in a mailbox, maintain a local state which is not directly exposed to other actors, and process at most one message at a time.

allowed to change its behavior that affects how it processes the subsequent messages.

An actor interacts with other actors in two ways as shown in Figure 4.2. Firstly, it can send and receive messages to and from other actors. The sending and receiving of messages is done asynchronously, i.e. the sending actor can deliver a message without waiting for the receiving actor to be ready to process the message. An actor learns about the existence of other actors by either receiving their addresses in incoming messages or during creation. This brings us to the second manner of actor interaction: an actor can create new actors. This new actor can have its local state initialized with information from the *parent* actor. It is important to note that the network of actors an actor knows about can grow dynamically thus allowing formation of arbitrary connection graphs among actors and a wide range of communication and coordination patterns between them. In summary, while processing a message an actor may perform a finite combination of the following steps:

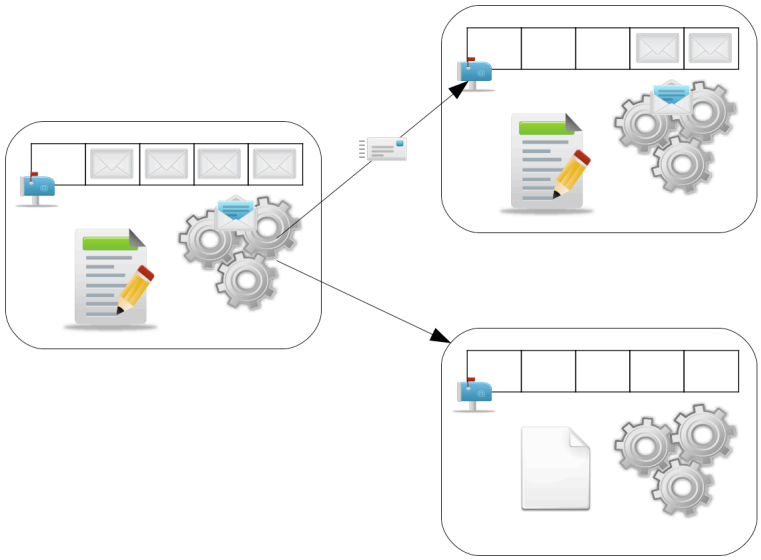


Figure 4.2 : During the processing of a message, actor interactions include exchanging messages with other actors and creating new actors.

1. Send a message to another actor whose address is known;
2. Create a new actor providing all the parameters required for initialization;
3. Become an actor, which specifies the replacement behavior to use while processing the subsequent messages [62].

Actors in the AM are required to have the following properties [63]:

1. State Encapsulation: An actor cannot directly access the internal state of another actor. An actor may affect the state of another actor only by sending the second actor a message. There is no shared state between actors;
2. Fair Scheduling: The Actor model assumes a notion of fairness: a message is eventually delivered to and processed by its destination actor, unless the

destination actor is permanently *disabled*. Another notion of fairness states that no actor can be permanently starved;

3. Location Transparency: The actors an actor knows could be on the same core, on the same CPU, or on a different node in a network. Because one actor does not know the address space of another actor, a desirable consequence of location transparency is state encapsulation;

4. Mobility: It is the ability of a computation performed by an actor to move across different nodes. Because actors provide modularity of control and encapsulation, mobility is quite natural to the Actor model. At the system level, mobility is important for load balancing, fault-tolerance and reconfiguration [63].

4.2 Desirable Properties

As mentioned earlier, message passing between actors is performed asynchronously and the data inside the message is preferred to be immutable. When the data in the message is mutable, a copy of the message is made at the receiver's mailbox. This ensures there is no data sharing and modification made to the data by the receiver does not introduce side-effects or data races that affect the sender.

The encapsulation of local data also means that other actors cannot directly modify the state of an actor. The only way an actor conveys its internal state to other actors is explicitly via responses to request messages, this is the behavior of the actor visible to other actors externally. Thus, benefits similar to encapsulation in object-oriented programming is obtained and modularity is encouraged. The encapsulation of local state also helps in preventing data races, because only the actor can modify its local state.

Since the same actor works only on local data, the AM does not suffer from the kind of data locality issues present in the FJM. However, location of the actual actors can still have an impact on performance; for example, it will be beneficial to have frequently communicating actors reside close to each other.

Due to the asynchronous mode of communication, lack of restriction on the order of processing messages, and absence of synchronization via encapsulation of local data, actors expose inherent concurrency and can work in parallel with other actors. The ability to reduce conflicts over shared data access by encapsulating local data is promising. The absence of data races encourages scalability. In addition, the lack of synchronization constructs in actors makes the user's code easier to reason about, maintain and refactor.

4.3 Disadvantages and Drawbacks

The AM cannot guarantee deadlock freedom. Two actors can deadlock, each simply waiting on a message from the other. In addition, since the order of processing messages is inherently non-deterministic it can be hard to reproduce such deadlock scenarios. In practice, it can also be hard to ensure the fair scheduling property since an actor can take an arbitrarily long time to process a message. This scenario also presents itself in the FJM, but it is not an issue since fairness is not a required property. However, implementations of the AM often choose not to guarantee fairness in scheduling.

The notion of synchronous replies, also known as the `receive` operations, where an actor sends another actor a message and stalls further processing of messages until it receives a reply to its message is relatively hard to implement. To avoid complications in processing of existing messages, often this behavior is implemented

using some notion of blocking and can limit scalability. Another option is to use pattern matching on the set of pending messages to implement `receive` and this can be expensive to implement due to increase in time while searching for the next message to process from the mailbox.

The AM is not a silver bullet, not all programming problems are best solved by the actor model. Actors perform poorly when there is a need for synchronous messaging. An actor will block if it is waiting on a reply from another actor. In the FJM, a parent task can avoid blocking by *handing-off* control to the new helper task to continue doing its work. Additionally, since message processing is serialized in actors, they cannot be used to simulate concurrent data structures. For example, actors cannot be used to support concurrent reads since the semantics of the AM require processing of one message at a time. As such, all read requests on a concurrent data structure represented via actors will be serialized.

4.4 Actors in Scala

Scala has been developed, since 2001, in the programming methods laboratory at EPFL [56]. Scala unifies object-oriented and functional programming paradigms into a statically typed programming language. Like HJ, Scala code compiles into Java bytecode that can then be run on the JVM. Two key features present in Scala are the support for functions as first-class values and pattern-matching over algebraic data types (see Figure 4.3). These features allow Scala to support an AM implementation as a library.

Actors are the primary concurrency model supported by Scala. The Scala Actor library is based on Erlang's actor concurrency model. Erlang actors rely on creation of lightweight processes and assign each process to an Actor. Since the JVM isn't

```
1  object PatternMatchingApp {
2    def main(args: Array[String]): Unit = {
3      val x: Any = 2.0
4      x match {
5        case i: Int => println("Found int: " + i)
6        case d: Double => println("Found double: " + d)
7        case s: String => println("Found String: " + s)
8      }
9      // prints "Found double: 2.0"
10   }
11 }
```

Figure 4.3 : An example of pattern matching in Scala. `x` is defined to be of type `Any` which is the root of the Scala class hierarchy. Then `x` is pattern matched for various cases to perform specific actions. Pattern matching can be considered as a generalization of if-else or switch statements in Java. Pattern matching is implemented using partial functions in Scala where each case statement is converted into a partial function. Each partial function is queried for a match before the body of the function is applied on the argument, in this case `x`

stackless and the creation of individual threads per actor is expensive, the Scala actor library does not follow the exact Erlang style of having one process per actor.

Instead, Scala actors come in two major flavors: thread-based and event-based [64, 51]. The thread-based actor mimics Erlang's actor style by assigning a thread to each actor. Java threads are substantially heavier than a lightweight Erlang process limiting the number of thread-based actors one can create in Scala as compared to Erlang. Event-based actors are comparatively lightweight and allow multiple actors to run on a thread. For apparent reasons, the event-based actors are more scalable. Event-based actors preserve the continuation while invoking blocking procedures such as synchronous send-and-receive. This is achieved by throwing *control* exceptions that are caught and handled by the actor runtime. The actor runtime then resumes the continuation when an event triggering the completion of the blocking operation is received. This is usually much cheaper than suspending a thread and hence more

scalable. The exact Scala construct used to implement event-based actors is `react` which accepts a partial function as input to determine how to process messages. Since messages represent to a static type, the pattern matching feature from Scala is used to simplify the syntax of event-processing.

Figure 4.4 shows an example of quicksort implemented using the Scala actor library. Since actors process messages individually and there is no predetermined order of message receipts, actors can be used to avoid the waiting problem DDFs face as explained in Figure 3.6. While using DDFs, the continuation task has to wait for both the left and right sub-computations to complete. There is no way to optimally handle results from one of the sub-computations that completes earlier. Since actors naturally promote continuation based programming, this case can easily be handled and partial results continually evaluated and stored.

There are a few of drawbacks of the Scala actor library. First, there is the problem of isolation. Messages in Scala are allowed to contain reference variables and they are not actually copied as part of an optimization technique. This means that messages can refer to shared variables and actors can end up introducing data races. It is left to the programmer to write code responsibly that ensures such data sharing is not introduced while passing messages. Scala provides *case classes* to help in this regard. Case classes make excellent messages since they are immutable and provide compiler-generated support to help with pattern matching. Despite the opportunity presented by the pattern matching construct, the Scala Actors library does not directly support the `become` operation. To support the become operation, the user can explicitly write code manipulating pattern expressions and local state.

Scala Actors allow executing code independent of receiving messages by placing the code in the `act` method. This is a violation of the AM, since actors in the AM

```
1  class QuicksortActor(parent: Actor,
2      positionRelativeToParent: Position) extends Actor {

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

8    def notifyParentAndTerminate() {
9      parent ! Result(result, positionRelativeToParent)
10     exit()
11   }

13   override def act() = {
14     loop { react {
15         case Sort(data) =>
16           val dataLength: Int = data.length
17           if (dataLength < QuicksortConfig.CUTOFF) {
18             result = quicksortSeq(data)
19             notifyParentAndTerminate()
20           } else {
21             val pivot = data(dataLength / 2)

23             val leftActor = new QuicksortActor(selfActor, PositionLeft).start()
24             leftActor ! Sort(data.filter(pivot >))

26             val rightActor = new QuicksortActor(selfActor, PositionRight).start↩
                   ()
27             rightActor ! Sort(data.filter(pivot <))

29             result = data.filter(pivot ==)
30             numFragments += 1
31           }
32         case Result(data, position) =>
33           if (position eq PositionLeft) { result = data ++ result }
34           else if (position eq PositionRight) { result = result ++ data }
35           numFragments += 1
36           if (numFragments == 3) { notifyParentAndTerminate() }
37     } } } }
```

Figure 4.4 : Quicksort using Scala event-based actors. The `loop` and `react` constructs ensure the actor can repeatedly process messages sent to the actor. The user has to explicitly invoke `exit()` to notify the runtime that the actor has completed processing and is ready to terminate. `react` accepts a partial function as an argument that defines the body of the actor and how to process each message. Note the absence of any synchronization constructs in the actor's member variables and execution body inside `react`.

are passive entities and can only execute code in response to receiving messages.

The next couple of drawbacks relate to performance penalties introduced by the implementation. Each case statement is interpreted by Scala as a partial function. Partial functions can be relatively slow to compute and to execute when compared to direct dynamic casts. More importantly, event-based actors use exceptions to maintain control flow and manage the execution of continuations. This again is slower than a corresponding runtime that avoids the use of exceptions in the management of the actor behavior.

Chapter 5

The Hybrid 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. In this thesis, 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. Integrating actors and tasks requires understanding how the actor life cycle interacts with task creation and termination events. In addition, the integration should be seamless and not enforce additional restrictions on either model.

5.1 Actors and Async-Finish Tasks

Integrating actors and tasks requires understanding how the actor life cycle interacts with task creation and termination events. It is helpful to understand the actor life cycle and the actions the actor performs in these states. The transitions between the actor states are shown in Figure 5.1. During its life cycle an actor is in one of the following states:

- *new* - An instance of the actor has been created, however the actor is not yet ready to receive or process messages. An actor instance is created by a `new` operation.

- *started* - The actor 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 has been terminated and will not process any messages in its mailbox or new messages sent to it henceforth. Termination is signaled by the actor itself while the processing of some message using the `exit` operation.

Figure 5.1 : Actors have a simple life cycle. The most interesting state is *started* which is where the actor is receiving and processing messages.

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 which can be effected by an actor on itself while it is processing a message. Assuming the message processing is happening in the current task, this can be done synchronously too. Once an actor enters the terminated state, it avoids processing any messages sent to it without blocking the sender (since such messages are effectively no-ops). Since tasks always execute inside an enclosing finish scope (as mentioned in Section 3.1), 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.

We now discuss the actions to be performed after the actor has started and is receiving and processing messages. By definition, actors are required to send and

```
1  object HelloWorldApp extends HabaneroApp {
2      val echoActor = new EchoActor()
3      async {
4        finish { // F1, IEF for echoActor
5          ...
6          echoActor.start(); // similar to an async
7          ...
8        }
9        println("EchoActor terminated");
10     }
11     async {
12       finish { // F2
13         ...
14         // task T1
15         echoActor ! "Hello"
16         echoActor ! "World"
17         echoActor ! EchoActor.STOP_MSG
18         ...
19       }
20       println("Done sending messages")
21 } } }
```

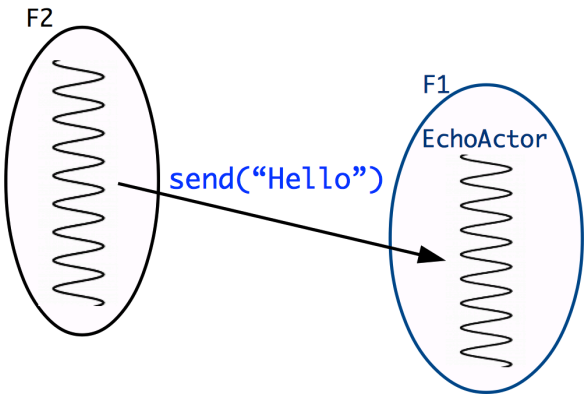


Figure 5.2 : `HelloWorld` example with `echoActor`, executing in `finish` scope `F1`, receiving messages from a different `finish` scope, `F2`.

receive messages asynchronously. This translates to 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 5.2. Under nor-

mal `async-finish` semantics, when `T1` sends a message to `echoActor` it creates a new task, say `T2`. This causes `F2` to unnecessarily (and incorrectly) block until `T2` completes. Since the message will end up in `echoActor`'s mailbox, the processing of the message is done by `echoActor` and semantically `T2` should have `F1` as its IEF as opposed to `F2`. Hence, when `T1` sends a message to `echoActor`, the new asynchronous task must be spawned in the `finish` scope of `echoActor`. In the hybrid model, this generalizes to all asynchronous tasks spawned to send and 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 hybrid 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 hybrid model.

5.1.1 Termination detection

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 JVM (e.g., Scala Actors library [51], Kilim [52], jetlang [53]) 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.

```
1  object ScalaActorPingPong {
2    def run(numMsgs: Int, verbose: Boolean): Unit = {
4      val latch = new CountDownLatch(1)
6      val pong = new Pong(verbose)
7      val ping = new Ping(numMsgs, pong, verbose, latch)
8      ping.start
9      pong.start
10     ping ! Start
12     latch.await()
13     println("ping has terminated, print handle results")
14   }
15 }
16 class Ping(count: Int, pong: Actor, verbose: Boolean, latch: CountDownLatch) ↩
       extends Actor {
17   def act() {
18     loop {
19       react {
20         case Start =>
21           // handle start message
22         case SendPing =>
23           // handle ping message
24         case Pong =>
25           if (pingsLeft > 0)
26             self ! SendPing
27           else {
28             pong ! Stop
29             latch.countDown()
30             exit('stop)
31 } } } } }
```

Figure 5.3 : Explicit actor termination detection using latches. Note the explicit management of the count while creating the latch as well as the need to call both `latch.countDown` and `actor.exit` while terminating the actor. In addition, the latch has to be explicitly passed around to the relevant actors.

For example, the Scala version of actors (Figure 5.3) 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. In contrast,

```
1  object HabaneroActorPingPong {
2    def run(numMsgs: Int, verbose: Boolean): Unit = {

4      val pong = new Pong(verbose)
5      pong.start

7      val ping = new Ping(numMsgs, pong, verbose)

9      finish {
10       ping.start
11       ping ! Start
12     }
13     println("ping has terminated, print handle results")
14 } }
15 class Ping(count: Int, pong: Actor, verbose: Boolean) extends Actor {
16   override def behavior() = {
17     case Start =>
18       // handle start message
19     case SendPing =>
20       // handle ping message
21     case Pong =>
22       if (pingsLeft > 0)
23         self ! SendPing
24       else {
25         pong ! Stop
26         exit('stop)
27       }
28 } }
```

Figure 5.4 : Implicit actor termination detection using `finish`. 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 for example.

actors in the hybrid model (Figure 5.4) benefit from the `finish` construct. Figure 5.4 shows a simple `PingPong` example using the hybrid actors and the `finish` construct to detect termination easily. 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.

5.2 New constructs under the hybrid model

With the hybrid 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 being able to reason about the enclosing `finish` under which the actors execute. Some of these constructs are presented below.

5.2.1 Parallelization inside Actors

There is internal concurrency in an actor in that it can be processing a message, receiving messages from other actors and sending messages to other actors at the same time. However, 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 during message processing as long as the invariant of processing at most one message at a time is maintained. A major contribution of this thesis is that integrating the AFM and the AM 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
- Allowing escaping `async` tasks

Both these techniques are discussed in the following subsections:

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 message-processing body (MPB), we can achieve parallelism by creating new `async-finish` constructs inside the MPB. We 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 have completed, the invariant that only one message is processed at a time is maintained. Figure 5.5 shows an example code snippet that achieves this. Note that there is no restriction on the AFM compliant constructs used inside the newly constructed `finish`. As such all the coordination constructs explained in Section 3.2.4 can also be used.

```
1  class ParallelizedProcessingActor() extends HybridActor {
2    override def behavior() = {
3      case msg: SomeMessage =>
4        // preprocess the message
5        finish { // finish to ensure all spawned tasks complete
6          async {
7            // do some processing in parallel
8          }
9          async {
10           // do some more processing in parallel
11         }
12       }
13       // postprocess the message after spawned tasks inside finish complete
14     ...
15 } }
```

Figure 5.5 : An actor exploiting the `async-finish` parallelism inside actors message processing body. The nested `finish` ensures no spawned tasks escape away causing the actor to process multiple messages at a time.

Allowing escaping `async` tasks

Requiring all spawned asyncs inside the MPB are captured 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 effected by two or more message processing tasks of the same actor. As long as this rule is obeyed, escaping asyncs (tasks) can be allowed inside the MPB.

We can achieve this invariant by introducing a *paused* state in the actor life cycle (Figure 5.6) and by adding two new 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.

With the two new operations, we can now allow spawned tasks to escape the main message processing 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

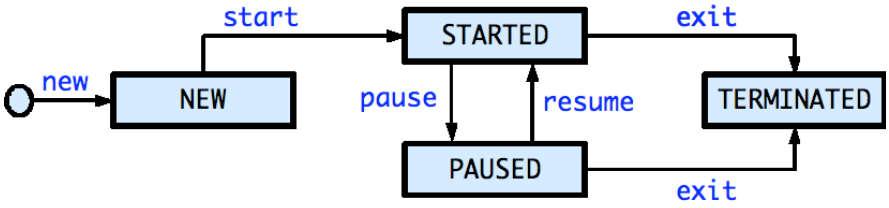


Figure 5.6 : Actor life cycle from Figure 5.1 extended with a *paused* state. The actor can now continually switch between a *started* and *paused* state.

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 5.7 shows an example where the `pause` and `resume` operations are used to achieve parallelism inside the MPB.

5.2.2 Non-blocking `receive` operations

As mentioned in Section 4.3, 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. 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 im-

```
1  class ParallelizedWithEscapingAsyncsActor() extends HybridActor {
2    override def behavior() = {
3      case msg: SomeMessage =>
4        // preprocess the message
5        async {
6          // do some processing in parallel
7        }
8        pause // to prevent the actor from processing the next message
9        // note that pause/resume is not blocking
10       async {
11         // do some more processing in parallel
12         // it is now safe for the actor to resume processing other messages
13         resume
14         // some more processing
15       }
16     ...
17 } }
```

Figure 5.7 : An actor exploiting parallelism via `asyncs` while avoiding an enclosing `finish`. The asyncs escape the message processing body, but the `pause` and `resume` operations are used to control processing of subsequent messages by the actor.

```
1  class ActorSimulatingReceive() extends HybridActor {
2    override def behavior() = {
3      case msg: SomeMessage =>
4        ...
5        val theDdf = ddf[ValueType]()
6        anotherActor ! new Message(theDdf)
7        pause() // temporarily disable further message processing
8        asyncAwait(theDdf) {
9          val responseVal = theDdf.get()
10         // process the current message
11         ...
12         resume() // enable further message processing
13       }
14       // return in paused state
15 } }
```

Figure 5.8 : An actor in the hybrid model that uses DDFs to simulate 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.

plemented in the unified model without blocking threads or using exceptions. This requires support of the DDF coordination construct presented in Section 3.2.4. 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 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 5.8 shows an example of a non-blocking `receive` implementation. This presents an instance of 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.

5.2.3 Stateless Actors

```
1  class StatelessActor() extends HybridActor {
2    override def behavior() = {
3      case msg: SomeMessage =>
4        async {
5          // process the current message
6        }
7        if (enoughMessagesProcessed) {
8          exit()
9        }
10       // return immediately to be ready to process the next message
11   }
12 }
```

Figure 5.9 : A simple stateless actor created using the hybrid 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.

The GPars [55] project, implemented in Groovy, provides an actor library inspired

by the Scala implementation. GPars also has a notion of stateless actors which do not keep track of what messages have arrived previously. In effect, these actors are allowed to process multiple messages simultaneously since they maintain no internal state. As mentioned in Section 5.2.1 it is easy to create such actors in the hybrid model. There is no need to use the `pause` operation and the escaping `async` tasks can process multiple messages to the same actor in parallel.

5.3 Desirable Properties

Actors in the hybrid 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 5.2.2) is an example of this.

5.4 Disadvantages or Drawbacks

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 Dynamic Program Structure Tree (DPST) based data race detection algorithm [65] 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.

Chapter 6

Implementation - Habanero-Scala

Habanero-Scala (HS) [66] is an extension of the Scala language [56] with AFM compliant constructs present including (but not limited to) `async`, `finish`, `future`s, DDFs, and `phaser`s. In HS, AFM constructs were added as a library and an existing actor implementation (standard Scala actors) extended to support the hybrid model. We refer to these hybrid actors as *heavy* actors. *Heavy* actors provide support for a subset of the operations presented in the hybrid model, excluding the `pause` and `resume` operations. In addition, HS provides its own implementation of hybrid actors, called *light* actors, that supports the full complement of operations including the `pause` and `resume` operations.

6.1 Choice of Scala

Scala, developed since 2001 in the programming methods laboratory at EPFL [56], unifies object-oriented and functional programming paradigms into a statically typed programming language. Scala as a language provides powerful abstractions to express various programming constructs. It has a relatively lenient constraint on the naming of methods, which coupled with its expressiveness make it extremely easy to create domain-specific languages (DSLs). This allows for easy transition of Habanero-Java (HJ) constructs into Scala without the need to build a front-end compiler. Most of the HJ work-sharing runtime can be reused in HS since both HJ and Scala run

on the Java Virtual Machine. However, the most important reason to choose Scala is its support for pattern matching. Pattern matching is an elegant way for writing actor code since the message processing body needs to pattern match on the messages received by the actor.

6.2 Previous Async-Finish compliant constructs

HS provides a variety of constructs also present in HJ which have already been discussed in Chapter 3. The following subsections introduce the syntax for relevant constructs (`async-finish`, `future`s, and data-driven futures) via code snippets in HS. A comprehensive explanation for all supported constructs along with examples is available at the HS homepage [67].

6.2.1 `async-finish`

```
1  class ForkJoinPrimer extends HabaneroApp {
2    println("Task O"); // Task-O
3    finish {
4      async { // Task-A
5        println("Task A");
6      }
7      async { // Task-B
8        println("Task B");
9        async { // Task-B1 created by Task-B
10         println("Task B1");
11       }
12       async { // Task-B2 created by Task-B
13         println("Task B2");
14       }
15     }
16   } // Wait for tasks A, B, B1 and B2 to finish
17   println("Task C"); // Task-C
18 }
```

6.2.2 `future`

```
1  object FibFutureApp extends HabaneroApp {
```

Figure 6.1 : Habanero-Scala version of the Fork-Join program from Figure 3.1. Note the similarity with the corresponding HJ program from Figure 3.2. The syntax of the code is unchanged due to the DSL support in Scala.

```
3    println("fib(10) = " + fib(10))

5    def fib(n: Int): Int = {
6      if (n < 2) {
7        return n
8      } else {
9        val x: HjFuture[Int] = asyncFuture {
10         fib(n - 1);
11       };
12       val y: HjFuture[Int] = asyncFuture {
13         fib(n - 2);
14       };

16       return x.get() + y.get();
17     }
18   }
19 }
```

Figure 6.2 : Habanero-Scala `future` example. A `future` represents the result of an asynchronous computation and extends HS's `async` statements to `async` expressions. Calls to `get()` are blocking operations if the task that computes the value of the future has not executed yet.

6.2.3 Data-driven futures

```
1  object FibDdfApp extends HabaneroApp {

3    val res = ddf[Int]()
4    fib(N, res)
5    asyncAwait(res) {
6      val fibResult = res.get()
7      println("fib(" + N + ") = " + fibResult)
8    }

10   def fib(n: Int, v: HjDataDrivenFuture[Int]): Unit = {
11     if (n <= CUTOFF) {
12       v.put(seqFib(n))
```

```
13      } else {
14        val res1 = ddf[Int]();
15        async {
16          fib(n - 1, res1);
17        };
18        val res2 = ddf[Int]();
19        async {
20          fib(n - 2, res2);
21        };

23        asyncAwait(res1, res2) {
24          v.put(res1.get() + res2.get())
25        }
26      }
27    }
28  }
```

Figure 6.3 : Habanero-Scala DDF and `asyncAwait` example. DDF are a generalization of `futures` and avoid blocking calls to `get()` since an `asyncAwait` only executes when data is available (i.e. `put()` has been called on the DDFs.

6.3 Hybrid Actors

Scala actors allow execution of logic independent of message processing similar to an `async` in HS. This is a violation of the pure AM since actors are supposed to trigger executions only when they receive messages. In HS, with the support for `async` such use of actors is redundant. HS supports two implementations of actors, both these implementations support the hybrid model. The two implementations are referred to as the *heavy* and *light* actors. Both these actor implementations rely on the use of *lingering* tasks to fit the actors into the AFM.

6.3.1 *Lingering* Tasks

Section 5.1 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 message-processing body (MPB) to an immediately enclosing finish (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*. 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 hybrid 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.

---

*A finish scope can only complete after all its transitively spawned tasks have completed.

6.3.2 *Heavy* Actors

The first form of hybrid actors supported by HS is 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 6.3.3. The standard Scala actor model is inspired from the actor implementation in Erlang and supports two types of actors: *thread-based* actors (TBAs) and *event-based* actors (EBAs) [51]. However, to support operations like `receive` (called `react` for EBAs) and avoid blocking, EBAs throw exceptions to roll back the call stack and allow the underlying thread to service other EBAs. 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. Hence support for the standard scala actors in HS is called *heavy* as compared to the *light* actors which do not rely on exceptions for control flow.

Actors are defined as a trait in the standard Scala library. In Scala, a trait is a set of method and field definitions that can be mixed into other classes. Traits are often introduced as Java interfaces with the ability to support default implementations. The *heavy* actor in HS is implemented as a trait that extends the Actor trait (see Figure 6.4). In fact, migrating to these hybrid actors in a Scala program that uses actors from the standard library is easy and it involves two steps:

- changing the `import` statements in the user code,
- renaming references to `Actor` with `HabaneroActor`

HS *heavy* actors do not support the `pause` and `resume` operations explained in Section 5.2.1. They however, support all the other AFM compliant constructs inside the message processing body including `finish`, `async`, `futures`, etc. HS *heavy* ac-

```
1  trait HabaneroActor extends Actor {

3    private val habaneroExecutorService = ...
4    override def scheduler = habaneroExecutorService

6    private var lingeringActivity: HabaneroActivity = null

8    override def start() = {
9      // register a pseudo activity to cause the IEF to wait on this actor
10     lingeringActivity = ...
11     // delegate to the parent implementation
12     super.start()
13   }

15   override def exit(): Nothing = {
16     // schedule this activity allowing the IEF to terminate
17     scheduleLingeringActivity(lingeringActivity)
18     // delegate to the parent implementation
19     super.exit()
20   }
21   ...
22 }
```

Figure 6.4 : *Heavy* actors in HS extending the standard Scala actors trait. The `start` and `exit` events are used to maintain some book-keeping for the heavy actors and interact with the Habanero runtime to schedule and execute tasks. The *lingering* activity is explained in Section 6.3.1.

tors still need to rely on exceptions for control flow and explicit management of the actor continuations, both implemented in the standard actors, and are hence more expensive to operate than the corresponding *light* actors.

6.3.3 *Light* Actors

Light actors are a complete implementation of actors in the hybrid model. With the support for `async`s in HS, there is no need to allow execution of logic independent of message processing that is available in standard scala and *heavy* actors. *Light* actors

are started using a call to `start()` and the MPB triggered only on the messages they receive. *Light* actors do not need to use exceptions to manage the control flow and 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 supports a *push*-based implementation where `async`s 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. The implementation of *light* actors relies on the use of *data-driven controls* to implement the mailbox.

6.3.4 Data-driven controls (DDCs)

A Data-Driven Control (DDC) lazily binds a value and a block of code called the execution body (EB) (Figure 6.5). 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. Figures 6.6 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 data-driven futures (DDFs) [61] 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 made aware of the existence of these data-driven tasks (DDTs) and causes the finish scopes of the tasks to block until the DDTs

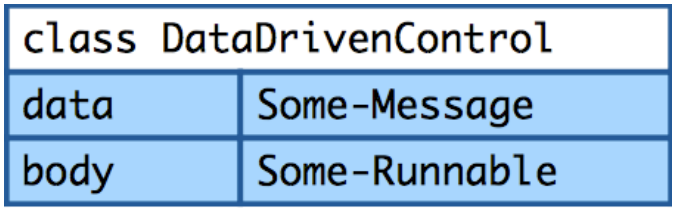


Figure 6.5 : Data-Driven Control has two fields. The fields are assigned only once, once both fields are assigned the body is scheduled.

are scheduled. In contrast, with DDCs the scheduler is unaware of the existence of the task until it is scheduled at which point, it will go ahead to 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 this.

6.3.5 The Mailbox: Linked List of DDCs

The *mailbox* for the *light* actors is implemented as a linked list of DDCs (Figure 6.7). As messages are sent to the actor, the chain of DDCs are built. The linked list is concurrent and multiple messages can be sent to an actor simultaneously. *Light* actors support both ordered and unordered adding of messages into the mailbox. When the ordered mode is used, it guarantees that 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 6.8 highlights the implementation of the `send()` operation used to send messages to a *light* actor in HS.

Once the actor has started (via the call to `start()`), it proceeds to traverse the head of the mailbox one at a time lazily attaching the task to execute with the value

```
1  private abstract class AbstractDataDrivenControl {

3    private var linkedValue: Any = null
4    private var linkedActivity: Runnable = null

6    def getValue(): Any = {
7      if (linkedValue == null) {
8        throw new IllegalStateException("value is null")
9      }
10     return linkedValue
11   }

13   def putValue(value: Any): Unit = {
14     if (value == null) {
15       throw new IllegalOperationException("attempted put of null value")
16     }
17     synchronized {
18       if (linkedValue == null) {
19         linkedValue = value
20         if (linkedActivity != null) {
21           scheduleActivity(linkedActivity)
22         }
23       } else { /* handle error scenario */ }
24     }
25   }

27   def registerActivity(activity: Runnable): Unit = {
28     synchronized {
29       linkedActivity = activity
30       if (linkedValue != null) {
31         scheduleActivity(linkedActivity)
32       } else { /* handle error scenario */ }
33     }
34   }

36   def scheduleActivity(activity: Runnable): Unit
37 }
```

Figure 6.6 : Data-driven control implemented in Habanero-Scala. The `synchronized` blocks ensure the single assignment of the value and the activity and avoids data races while trying to schedule the activity. The `scheduleActivity()` method is abstract to allow different implementations of scheduling the activity to be implemented.

of the DDC, i.e. the message. The DDCs support asynchronous scheduling of the tasks using the Habanero scheduler. The *lingering* activity is used to gain access to the finish scope under which the DDC task needs to be scheduled. When the DDC

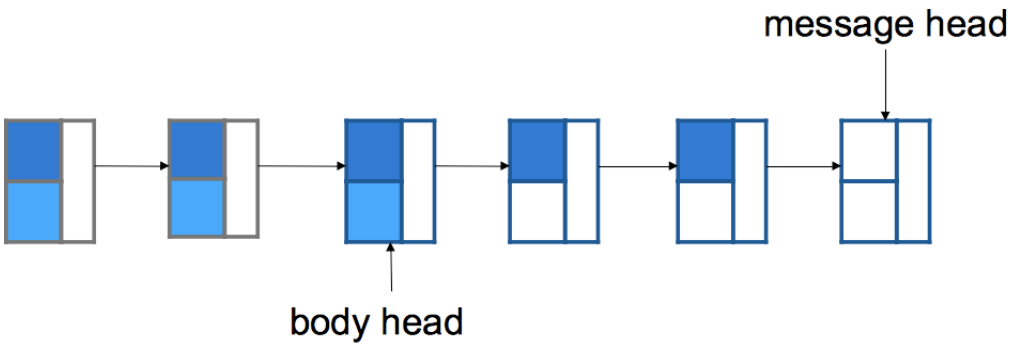


Figure 6.7 : The Actor mailbox is represented as a linked-list of DDCs. The message head determines where the next message is stored while the body head determines which message is being processed currently.

task is ultimately scheduled and executed, the head of the mailbox is moved ahead and the next DDC processed. If the actor was terminated via a call to `exit()` in the DDC task, the actor stops processing messages. No more DDC tasks are scheduled and the *lingering* task is resumed and the *hook* on the enclosing finish scope from the *lingering* task released. Since at any time, only one DDC is actively executed the guarantee that only one message is processed at a time is provided. There is no need for an explicit flag or state to track and schedule the next message processing task.

6.3.6 Supporting `pause` and `resume` with DDCs

Light actors support the `pause` and `resume` operations explained in Section 5.2.1 using synchronous DDCs. A call to `pause()` creates a new DDC. The message processing body (`actOn()` in Figure 6.9) 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 trigger the execution of the

```
1  trait HabaneroReactor extends OutputChannel[Any] {

3    private var currentDdc = new LinkedDdcWrapper()
4    private var hasExited = false

6    def useOrderedSend = ...

8    def send(msg: Any): Unit = {
9      if (hasExited) {
10       return // actor has exited, synchronously ignore messages
11     }
12     if (useOrderedSend) {
13       processMessage(msg)
14     } else {
15       asyncSend(msg)
16     }
17   }

19   def asyncSend(msg: Any): Unit = {
20     val runnableBlock: Runnable = new Runnable {
21       def run = { processMessage(msg) }
22     }
23     val newActivity = createHabaneroActivity(lingeringActivity, runnableBlock)
24     HabaneroReactor.executor.execute(newActivity)
25   }

27   protected def processMessage(msg: Any): Unit = {
28     if (hasExited) {
29       return // actor has exited, ignore messages
30     }
31     synchronized {
32       val oldDdc = currentDdc // update the current ddc
33       val newDdc = oldDdc.nextNode
34       currentDdc = newDdc
35       oldDdc.value.putValue(msg) // put the message into the ddc
36     }
37   }
38   ...
39 }
```

Figure 6.8 : DDC used as the mailbox in HS *Light* actors.

continuation.

```
1  trait HabaneroReactor extends OutputChannel[Any] {

3    private var resumeDdc: SynchronousDataDrivenControl = null
4    ...
5    private def actOn(msgDdcWrapper: LinkedDdcWrapper): Unit = {
6      val msgDdc: HabaneroDataDrivenControl = msgDdcWrapper.value
```

```
7       ...
8           // continue further processing if actor has not exited
9           if (!hasExited) {
10            val tempResumeDdc = resumeDdc
11            if (tempResumeDdc == null) {
12              // actor is active, asynchronously act on the next message
13              actOn(msgDdcWrapper.nextNode)
14            } else {
15              // actor is paused, wait for it to resume before processing the ↩
                    next message
16              prepareForResume(tempResumeDdc, msgDdcWrapper.nextNode)
17            }
18          }
19      ...
20    }

22    private def prepareForResume(resumeOnDdc: SynchronousDataDrivenControl,
23                                 actOnDdc: LinkedDdcWrapper): Unit = {
24      val runnableBlock: Runnable = new Runnable {
25        def run = { actOn(actOnDdc) }
26      }
27      // delay execution of actOn() until actor is resumed
28      resumeOnDdc.registerActivity(runnableBlock)
29    }

31    /**
32     * This method is not thread safe, should only be called in thread-safe ↩
           manner to avoid data races
33     */
34    protected def pause(): Unit = {
35      if (resumeDdc == null) {
36        resumeDdc = new SynchronousDataDrivenControl()
37      }
38    }

40    /**
41     * resume a paused actor.
42     */
43    protected def resume(): Unit = {
44      if (resumeDdc != null) {
45        val tempDdc = resumeDdc
46        resumeDdc = null
47        tempDdc.putValue(true) // dummy true value to trigger the activity
48      } else { /* error: called resume with nothing to resume */ }
49    }
50    ...
51  }
```

Figure 6.9 : Support for `pause` and `resume` in HS *Light* actors using synchronous DDCs.

6.3.7 Supporting `become` and `unbecome` with DDCs

As mentioned in Section 4.1, 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 constructs which work in conjunction with mutable member variables.

Light actors support the `become` and `unbecome` operation 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. 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 the Habanero runtime and associated with an instance of `MultipleExceptions` thrown from the finish scope.

```
1  trait HabaneroReactor extends OutputChannel[Any] {
2    private val behaviorHistory = new Stack[PartialFunction[Any, Unit]]
3
4    def become(newBehavior: PartialFunction[Any, Unit]): Unit = {
5      behaviorHistory.push(newBehavior)
6    }
7
8    def unbecome(): Unit = {
9      if (behaviorHistory.isEmpty) {
10       throw new RuntimeException("Actor behavior history is empty!")
11     }
12     behaviorHistory.pop()
13   }
14
15   /**
16    * abstract method which must be defined to allow the
17    * actor to have custom behaviors. User will usually implement
18    * this with a partial function. This supports dynamic behavior changes.
19    */
20   def behavior(): PartialFunction[Any, Unit]
21
22   private def actOn(msgDdcWrapper: LinkedDdcWrapper): Unit = {
23     val msgDdc: HabaneroDataDrivenControl = msgDdcWrapper.value
24     val runnableBlock: Runnable = new Runnable {
25       def run: Unit = {
26         // act on the message
27         val theMsg = msgDdc.getValue()
28
29           ...
30           // allow the actor to dynamically change its behavior
31           val curBehavior = if (behaviorHistory.isEmpty) {
32             behavior()
33           } else {
34             behaviorHistory.head
35           }
36           ...
37       }
38     }
39     // create awaiting activity that will be scheduled when a message
40     // is put on the current ddc
41     val newActivity: Runnable = createActivity(lingeringActivity, runnableBlock↩
           )
42     msgDdc.registerActivity(newActivity)
43   }
44   ...
45 }
```

Figure 6.10 : Support for `become` and `unbecome` in HS *Light* actors. A stack is used to store the partial functions that represent the history of behaviors used by the actor.

6.3.8 Using *Light* actors

Light actors are Scala objects that are created by instantiating subclasses of the `edu.rice.habanero.actor.HabaneroReactor` trait. Subclasses need to provide an implementation of the `behavior` method which defines the default message processing behavior of the actor. Inside this behavior, the actor can make calls to `exit()`, `pause()`, `resume()`, `become(...)`, and `unbecome()` to trigger the various state changes in the hybrid model. In addition, the behavior can include calls to other Habanero parallel constructs like `finish`, `async`, `future`, data-driven futures, etc. Figure 6.11 shows a simple example of using *light* actors to solve quicksort in the hybrid model.

```
1  object HybridActorQuicksortApp {
2    def run(input: ListBuffer[Int]): ListBuffer[Int] = {
3      val rootActor = new QuicksortActor(null, PositionInitial)
4      finish {
5        rootActor.start()
6        rootActor ! Sort(input)
7      }
8      rootActor.result
9    }
10 }
11 class QuicksortActor(parent: QuicksortActor, positionRelativeToParent: Position↩
       ) extends HabaneroReactor {

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

17   def notifyParentAndTerminate() = {
18     if (parent ne null) {
19       parent ! Result(result, positionRelativeToParent)
20     }
21     exit()
22   }
```

```
24    override def behavior() = {
25      case Sort(data) =>
26        val dataLength: Int = data.length
27        if (dataLength < QuicksortConfig.CUTOFF) {
28          result = quicksortSeq(data)
29          notifyParentAndTerminate()
30        } else {
31          val pivot = data(dataLength / 2)
32          async {
33            val leftUnsorted = filterLessThan(data, pivot)
34            if (!leftUnsorted.isEmpty) {
35              val leftActor = new QuicksortActor(selfActor, PositionLeft)
36              leftActor.start()
37              leftActor ! Sort(leftUnsorted)
38            } else {
39              selfActor ! Result(leftUnsorted, PositionLeft)
40            }
41          }
42          async {
43            val rightUnsorted = filterGreaterThan(data, pivot)
44            if (!rightUnsorted.isEmpty) {
45              val rightActor = new QuicksortActor(selfActor, PositionRight)
46              rightActor.start()
47              rightActor ! Sort(rightUnsorted)
48            } else {
49              selfActor ! Result(rightUnsorted, PositionRight)
50            }
51          }
52          result = filterEqualsTo(data, pivot)
53          numFragments += 1
54        }
55      case Result(data, position) =>
56        if (!data.isEmpty) {
57          if (position eq PositionLeft) {
58            result = data ++ result
59          } else if (position eq PositionRight) {
60            result = result ++ data
61          }
62        }
63        numFragments += 1
64        if (numFragments == 3) {
65          notifyParentAndTerminate()
66        }
67    }
68  }
```

Figure 6.11 : Quicksort using *light* actors in HS. This version introduces parallelism inside the actor message processing body by creating *escaping* `asyncs`. Note the use of `finish` construct to detect actor termination.

6.3.9 *Light* and *Heavy* actors compared

Heavy actors are instances of the `edu.rice.habanero.actor.HabaneroActor` trait and inherit all the abilities of standard Scala actors which include support for nested `receive` and `react`. However, *heavy* actors do not support `pause` and `resume`. *Light* actors do not directly support nesting of `receive` and `react`, but can simulate the behavior using `pause`, `resume` and DDTs. Such an implementation in *light* actors has the added benefit that there is no blocking of threads. Another feature left out of *light* actors is the ability to *link* actors which cause a group of actors to be notified when an actor terminates. This is available in the standard Scala actors as a convenience and an influence from the Erlang actors. The same behavior is achieved by passing *exit* messages in the pure AM and hybrid model and has been left out of the *light* actor implementation. The hybrid actors can be enclosed inside finish scopes and there is no need to maintain explicit latches to detect termination. Finally, the two actor implementations in HS can seamlessly interact with each other since they are scheduled and run under the same runtime and scheduler.

Chapter 7

Applications

The hybrid model allows problems to be solved not only using either approach individually but also mixing both models. The Async-Finish Model (AFM) is very useful in decomposing a problem into independent sub-tasks whose results are then combined to produce the end result. However, if coordination is required between these subtasks extensions are required to the AFM. The Actor Model (AM) is useful to implement asynchronous event-based problems where individual actors coordinate with each other using messages as events. In the AM, simulating non-blocking synchronous replies requires some amount of effort mainly due to lack of a guarantee of when a given message will be processed. Similarly, achieving global consensus among a group of actors is a non-trivial task. Such patterns are simpler to realize in the AFM, for example by using `finish` to wrap `async`s or by using `phaser`s [37] as communication barriers. Some applications that exhibit patterns that prove to be a good fit for the hybrid model are presented in the following sections.

7.1 Multiple Producer-Consumer with Bounded Buffer

One of the most common synchronization problems is multiple producers and consumers with a bounded buffer [68]. In this problem, producer tasks produce items that are stored into the buffer, while consumer tasks removes these items from the buffer and process them. Synchronization is needed to guarantee that

- producer tasks do not insert items into the buffer when the buffer is full,
- consumers do not try to remove items from an empty buffer, and
- each item is consumed and processed by exactly one consumer.

Often, it is possible to parallelize the production or consumption of an individual item. The AFM is often a good fit for this. However, coordination and synchronization is required when either the producer or consumer interacts with the buffer. Traditionally, a host of synchronization constructs like semaphores, monitors or locks are used to handle the synchronization. The issue with implementations of these primitives is that they are (thread) blocking and can lead to performance bottlenecks on most parallel runtime implementations.

In contrast, in the AM, the producer, consumer and the buffer can be modeled as actors. The producer and consumers send messages to the buffer actor which coordinates the transfer of items between the producers and consumers. Producers notify the buffer when they are ready to produce items. When the buffer has enough space it signals (via messages) the producer to produce an item. The producer sends the newly produced item to the buffer. If the buffer is full, the buffer actor waits until a consumer consumes an item from the buffer before signaling the producer to produce the next item. Consumers notify the buffer when they are ready to consume an item. If there is an item in the buffer, the item is handed off to the consumer, else the buffer caches the consumer and waits for an item to be produced by a producer. The messages sent to the buffer (and the other actors involved, i.e. the producers and consumers) are processed one at a time and there are no data races. In addition, none of the operations mentioned above involve actively blocking threads as actors store their own continuations and can resume whenever they receive messages. In

addition, the processing logic of actually producing or consuming the items can be parallelized using the AFM. Hence, the entire problem can be solved effectively using the hybrid model.

7.2 Pipelined Parallelism

Pipelining is used for repetitive tasks where each task can be broken down into independent sub-tasks (also called stages) which must be performed sequentially, one after the other [69]. Each stage partially processes data and then forwards the partially processed result to the next stage in the pipeline for further processing. This pattern works best if the operations performed by the various stages of the pipeline are balanced, i.e. take comparable time. If the stages in the pipeline vary widely in computational effort, the slowest stage creates a bottleneck for the aggregate throughput.

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 off to the next actor in the pipeline. The stages however need to ensure ordering of messages while processing them. In Habanero-Scala, this ordering support is provided by default for messages from the same actor to another *light* actor. However, the amount of concurrency (hence 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 stages. This can be achieved by using the hybrid model. Increasing the parallelism may also help in speeding up the slowest stage in the pipeline.

7.2.1 Filterbank

Filter Bank has been ported from the StreamIt [13] 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. The FIR stage limits the pipeline rate, the performance can be improved by speeding up this stage by parallelizing it. We can do so in the hybrid model by partitioning the computation of the dot product using `async`s (line 24 in the example). Each `async` computes the dot product of a partition before writing back the result into its assigned DDF (line 30). The `async` at line 33 `await`s on the results 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 38 join to their IEF, the `finish` at line 21. 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.

```
1 object FilterBankApp extends HabaneroApp {
```

```
2     finish {
3       ...
4       val sampler = ...
5       val fir = new FirFilter(..., sampler).start()
6       ...
7   } }
8   class FirFilter(..., nextStage: HybridActor[Any])
9       extends HybridActor[FirMessage] {
10    ...
11    def behavior() = {
12      case FirItemMessage(value, coeffs) =>
13        buffer(dataIndex) = value
14        dataIndex = (dataIndex + 1) % bufferSize
15        val numHelpers = ... // number of helper tasks
16        // allocate the DDFs to use
17        val stores = Array.tabulate(numHelpers) {
18          index => ddf[Double]()
19        }

21        finish {
22          // compute the sum using divide-and-conquer
23          (0 until numHelpers) foreach { helperId =>
24            async {
25              val (start, end) = ...
26              var sum: Double = 0.0
27              start until end foreach { index =>
28                sum += buffer(index) * coeffs(index)
29              }
30              stores(helperId).put(sum)
31          } }
32          // wait for the partial results
33          asyncAwait(stores) {
34            // propagate the sum down the pipeline
35            val sum = stores.foldLeft(0.0) {
36              (acc, loopDdf) => acc + loopDdf.get()
37            }
38            nextStage.send(DataItemMessage(sum))
39          }
40        }
41      case ... => ...
42  } }
```

Figure 7.1 : 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.

7.2.2 Sieve of Eratosthenes

One algorithm that can be solved elegantly using a dynamic pipeline is the Sieve of Eratosthenes [70]. The algorithm incrementally builds knowledge of primes. Each new candidate is sequentially tested against the known *local* primes. If none of these local primes divide the candidate, the candidate is deemed to be prime and added to the list of local primes. The next candidate is then tested using the same policy. Using this approach, it is easy to build a pipelined version where 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. A degenerate case would be to store just one prime in the buffer and create as many stages as known primes. However, 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. 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.

7.3 Speculative Parallelization

Speculative parallelization is a technique used to extract parallelism from sequential programs. The idea is to optimistically execute some fragments of code in parallel assuming that no dependences exist. The parallel tasks synchronize when necessary to ensure that no sequential semantics are violated [71]. Speculative parallelization is particularly common while processing data structures such as trees and graphs, where opportunities for deterministic parallelism are highly limited. The AM can be used to parallelize such applications by having each node represented as an actor. The actors

can coordinate with their *parent* and *sibling* nodes for dependences but execute in parallel when no dependences exist. With actors in place, the hybrid model can be used to exploit the parallelism inside the actors.

7.3.1 Online (Hierarchical) Facility Location

The goal of a Facility Location algorithm is to decide when and where to open facilities in order to minimize the associated cost of opening a facility and the (transportation) 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 solution to this problem is the Online Hierarchical Facility Location [72]. The algorithm exposes a hierarchical tree structure (quadrants in the algorithm) while performing the computation. The location of the customer initially flows down the tree, but then need to flow back up the tree at certain decision points. When these values flow back up the tree, they may change how the subsequent values flow down the tree. In the algorithm, each node maintains a list of customers it plans to service. At decision points, it needs to process this list and partition them to form new child nodes. In addition, the decision to create child nodes needs to be propagated up the tree and to selected siblings. Thus the tree grows dynamically and new communication patterns may develop between nodes. A parallel version of this algorithm can be mapped to the hybrid model where each node is treated as an actor and the `async-finish` parallelism can be used to exploit the data parallelism while partitioning the customers to prepare the child nodes.

Chapter 8

Results and Discussion

8.1 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-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 [73] 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.

8.2 Microbenchmarks comparing Actor frameworks

The first benchmark (Figure 8.1) 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

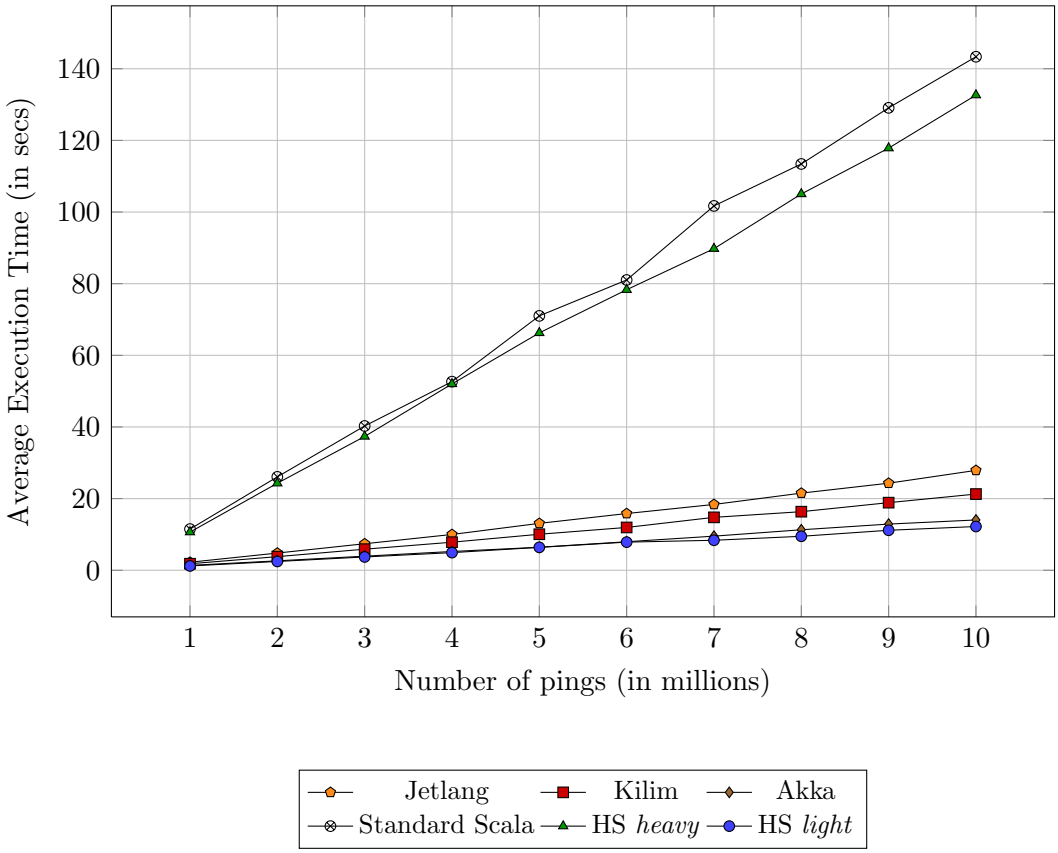


Figure 8.1 : The `PingPong` benchmark exposes the throughput and latency while delivering messages. There is no parallelism to be exploited in the application.

overheads in the message delivery implementation for actors. The original version of the code was obtained from [74] 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 6.3.2). Kilim, Jetlang, Akka and *light* actors benefit from avoiding generating exceptions to maintain control flow. In general,

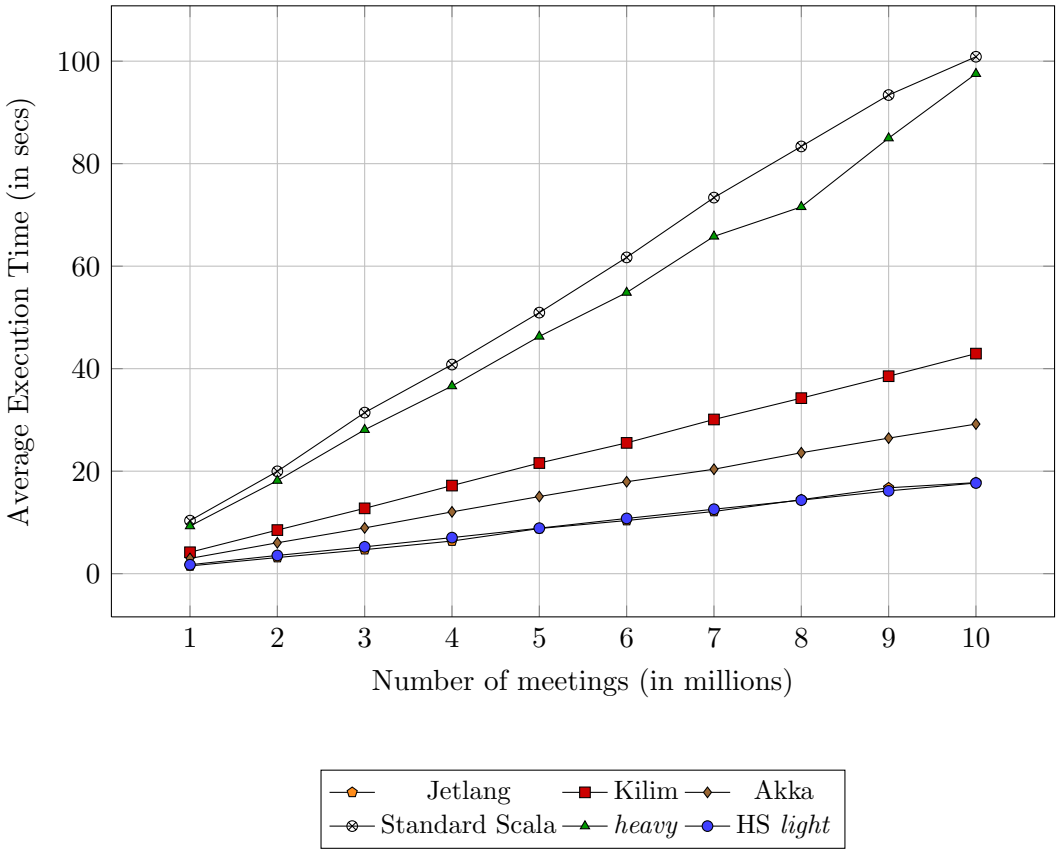


Figure 8.2 : The `Chameneos` benchmark exposes the effects of contention on shared resources. The `Chameneos` benchmark involves all *chameneos* constantly sending messages to a mall actor that coordinates which two *chameneos* get to meet. Adding messages into the mall actor's mailbox serves as a contention point.

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 version is much slower as the Scala implementation pays the overhead for pattern matching twice as opposed to once in Kilim, Akka and *light* actors.

The `Chameneos` benchmark, shown in Figure 8.2, 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 [75]. 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 and HS *heavy* actors share the same implementation they show similar performance. 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 6.3.4) outperforms the simple linked list implementation in other actor implementations. Jetlang, which uses iterative batch-processing of messages sent to the mall, is in general slightly faster than the *light* actor implementation which uses recursive batch processing of messages.

The Java Grande Forum `Fork-Join` benchmark [76], shown in Figure 8.3, 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.

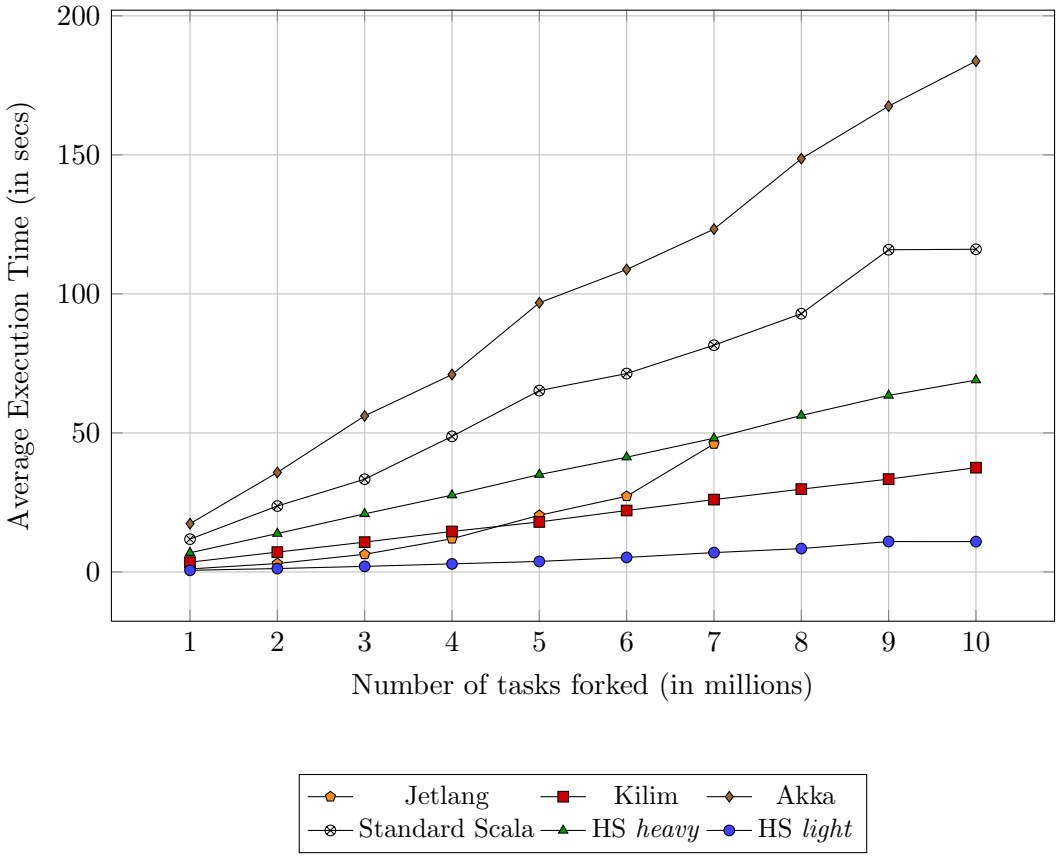


Figure 8.3 : The Java Grande Forum `Fork-Join` benchmark ported for actors. Individual invocations were configured to run using twelve workers. Both Jetlang versions run out of memory on larger problem sizes.

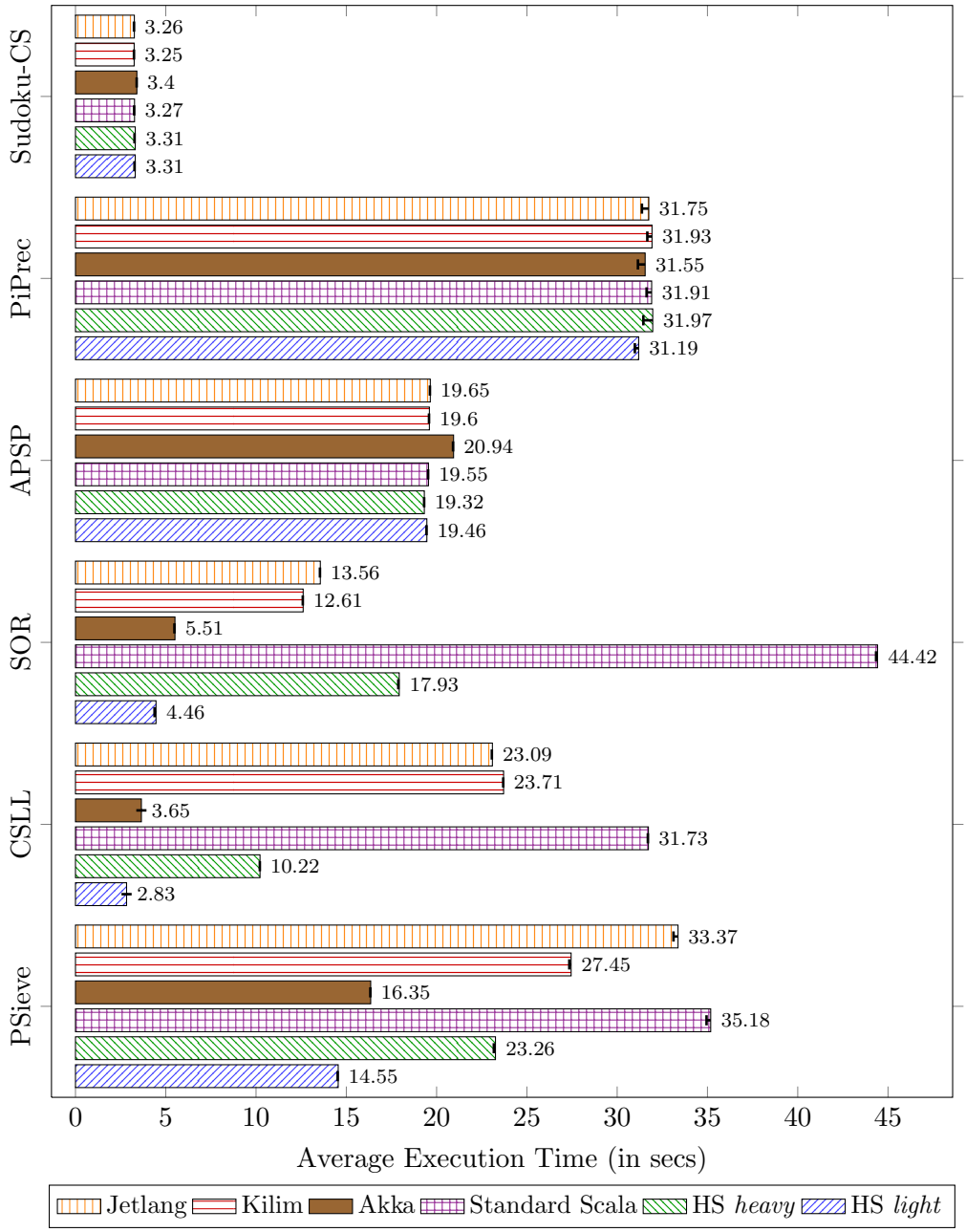
8.3 Application Benchmarks

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 hybrid model in some applications. Each application benchmark was run with the schedulers set up to use 12 worker threads.

8.3.1 General Applications Compared

Figure 8.4 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 [77]. The next two applications, Concurrent Sorted Linked-List (CSLL) and Prime Sieve (PSieve), use a pipeline pattern to expose some parallelism. CSLL



- 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 8.4 : Comparison of some applications using different JVM actor frameworks.

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 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.

8.3.2 Quicksort

Quicksort lends itself to divide-and-conquer strategy and is a good fit for the AFM, however it exposes some amount of non-determinism in availability of partial results which cannot entirely be captured by the AFM. Figure 8.5 compares the hybrid actor implementations in HS with previously existing `async-finish` extensions such as DDFs. Pure actor implementations in HS involve sequential message-processing. The *light* actor implementation is faster than the DDF-based implementation as it can

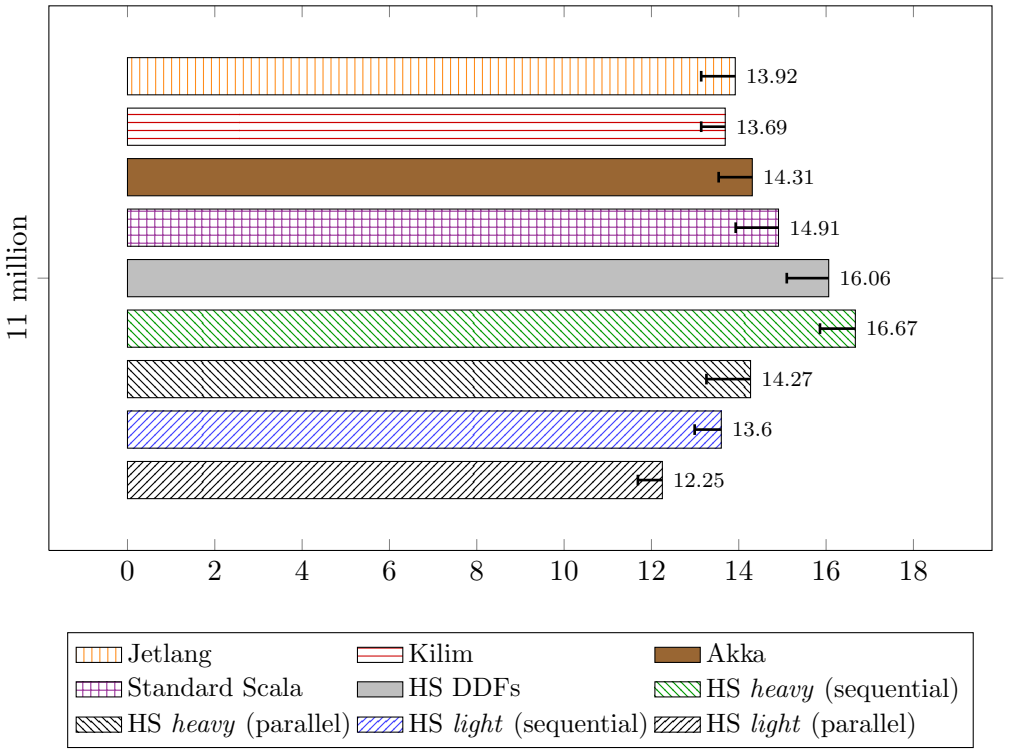


Figure 8.5 : Results of the Quicksort benchmark on input of length 11 million.

make progress computing the partial result from fragments. In the hybrid 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 hybrid 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.

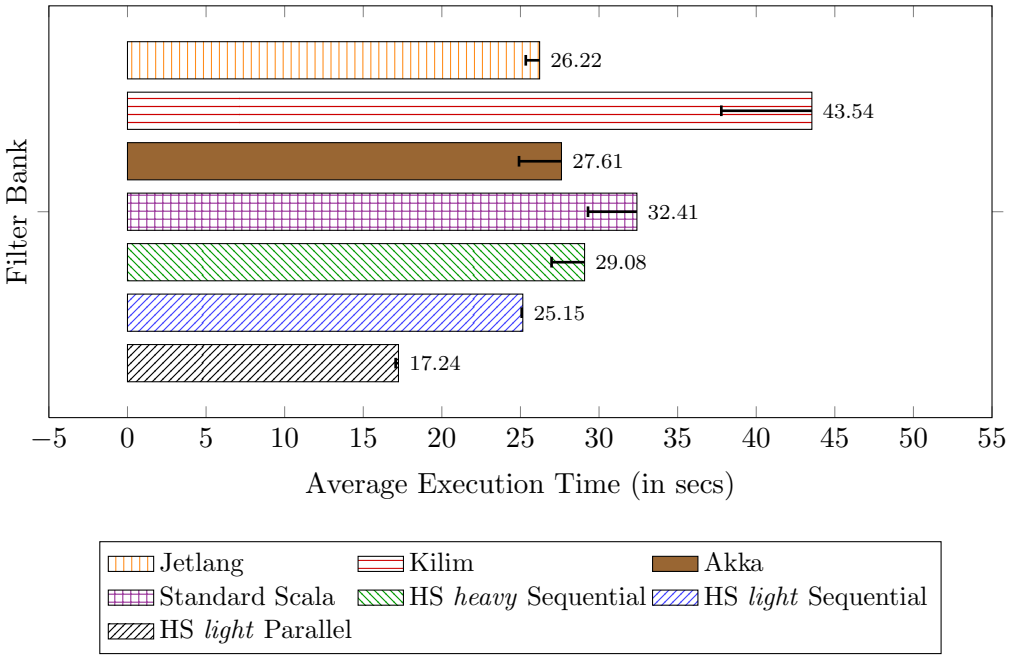


Figure 8.6 : Filter Bank benchmark results configured to use three branches.

8.3.3 Filter Bank for multirate signal processing

Filter Bank has been ported from the StreamIt [13] set of benchmarks and has been described in Section 7.2.1. 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 8.6 compares the performance of the actor implementations of the Filter Bank benchmark with a hybrid implementation which parallelizes the FIR stage. The HS *light* parallel version is at least 30% faster than the other actor implementations which use sequential message processing.

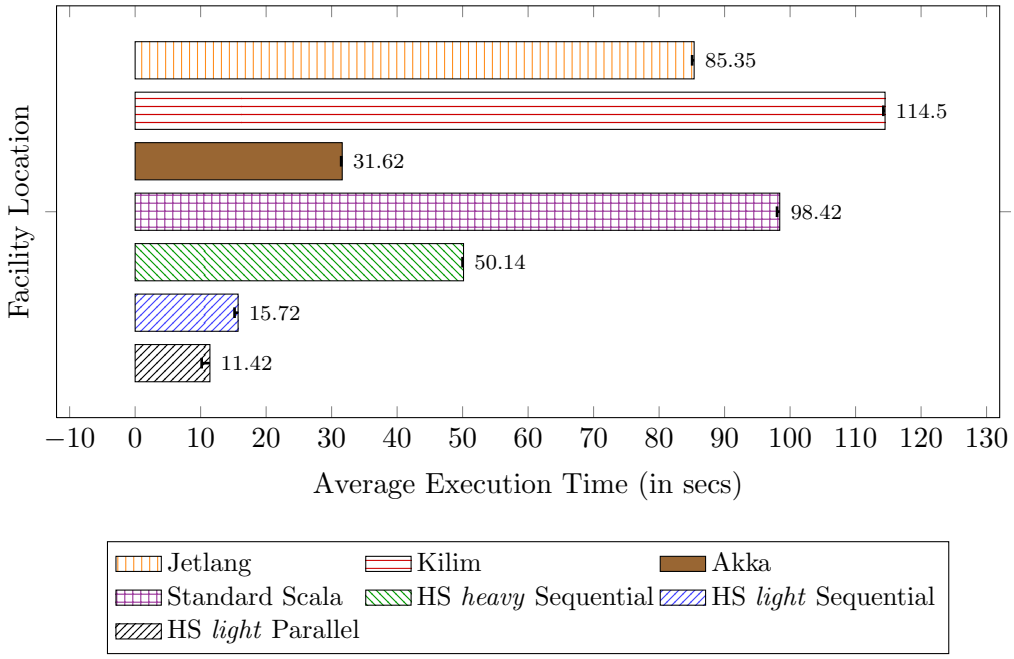


Figure 8.7 : Online Hierarchical Facility Location benchmark results. Results displayed for 6 million customers and an alpha value of 5.

8.3.4 Online Hierarchical Facility Location

Figure 8.7 compares the performance of the actor implementations of the Facility Location benchmark (described in Section 7.3.1) with a hybrid implementation. In Online Hierarchical Facility Location, parallelism from the hybrid 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.

Chapter 9

Conclusions & Future Work

9.1 Conclusions

With the advent of the multi-core era there is a renewed interest in developing new programming models for parallelism. This thesis focuses on a hybrid model that combines two such programming models: the Async-Finish model (AFM) and the Actor model (AM). To the best of our knowledge, this is the first such study to systematically combine these two models.

In this thesis, we presented the case for integrating actors in the AFM as the hybrid model. The hybrid model allows for parallelism inside actors while providing a useful coordination construct between tasks in the AFM. The hybrid model makes termination detection easier in actor programs. It also allows arbitrary coordination patterns, in arguably a more productive manner than other extensions such as phasers and data-driven futures, among tasks in the AFM. The hybrid model allows for easier implementation of certain constructs, for example the normally blocking `receive` can be implemented in a non-blocking manner in the hybrid model.

The thesis also presents an implementation of this hybrid model called Habanero-Scala (HS), which is an extension of the Scala programming language. HS provides a faster actor implementation than the the standard Scala actor library and many other actor implementations on the JVM. HS also served as a tool to run experiments to verify the claims of the hybrid model. The thesis also presented properties

of applications that can benefit from the hybrid model and experimental results corroborate the claim that hybrid solutions to certain problems are more efficient than exclusively using the AFM or AM. The hybrid model thus adds to the tools available for the programmer to aid in productivity and performance while developing parallel software.

9.2 Future Work

The hybrid 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 hybrid actors is an interesting area for future research and we plan to extend the DPST-based data race detection algorithm [65] for data race detection in the unified model.

Bibliography

[1] A. Mendelson, "How many cores is too many cores?." 3rd HiPEAC Industrial Workshop, Haifa, Israel, April 2007.

[2] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, March 2005.

[3] J. L. Manferdelli, "The Many-Core Inflection Point for Mass Market Computer Systems." `http://www.ctwatch.org/quarterly/articles/2007/02/the-many-core-inflection-point-for-mass-market-computer-systems/`.

[4] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 151–162, October 2006.

[5] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, January 1998.

[6] H. Sutter and J. Larus, "Software and the Concurrency Revolution," *Queue*, vol. 3, pp. 54–62, September 2005.

[7] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a Portable Nested Data-Parallel Language," *Journal of Parallel and Distributed Computing*, vol. 21, pp. 102–111, 1994.

[8] Ghuloum, Anwar and Sharp, Amanda and Clemons, Noah and Du Toit, Stefanus and Malladi, Rama and Gangadhar, Mukesh and McCool, Michael and Pabst, Hans, "Array Building Blocks: A Flexible Parallel Programming Model for Multicore and Many-Core Architectures." `http://drdobbs.com/cpp/227300084`.

[9] Wikipedia, The Free Encyclopedia, "Automatic Parallelization." `http://en.wikipedia.org/wiki/Automatic_parallelization`, 2011. [Online; accessed 27-March-2011].

[10] J. G. Steffan and T. C. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, (Las Vegas, Nevada), February 1998.

[11] M. Hall, D. Padua, and K. Pingali, "Compiler research: The Next 50 Years," *Communications of the ACM*, vol. 52, pp. 60–67, February 2009.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008.

[13] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Computational Complexity*, pp. 179–196, 2002.

[14] Mathworks, "Matlab Parallel Computing Toolbox." `http://www.mathworks.com/products/parallel-computing/`.

[15] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," 1994.

[16] Wikipedia, The Free Encyclopedia, "Partitioned Global Address Space." http://en.wikipedia.org/wiki/Partitioned_global_address_space, 2010. [Online; accessed 2-August-2010].

[17] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, pp. 1–31, August 1998.

[18] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[19] UPC Consortium, "UPC Language Specification (V 1.2)." http://upc.gwu.edu/docs/upc_specs_1.2.pdf.

[20] K. Ebcioglu, V. Saraswat, and V. Sarkar, "X10: An Experimental Language for High Productivity Programming of Scalable Systems," *In Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)*, January 2005.

[21] Wikipedia, The Free Encyclopedia, "GPGPU: General-purpose computing on graphics processing units." http://en.wikipedia.org/wiki/GPGPU, 2010. [Online; accessed 2-August-2010].

[22] Corporation, Nvidia, "NVIDIA CUDA Programming Guide 2.0." http://www.nvidia.com/cuda.

[23] D. Lea, "A Java Fork/Join Framework," in *Java Grande*, pp. 36–43, 2000.

[24] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine, "PFunc: Modern Task Parallelism For Modern High Performance Computing," in *Pro-*

ceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, (New York, NY, USA), pp. 43:1–43:11, ACM, 2009.

[25] 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.*, vol. 30, pp. 207–216, August 1995.

[26] C. E. Leiserson, "The Cilk++ Concurrency Platform," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, (New York, NY, USA), pp. 522–527, ACM, 2009.

[27] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O'Reilly, 2007.

[28] OpenMP Architecture Review Board, "OpenMP Application Program Interface," *Review Literature And Arts Of The Americas*, vol. 1, no. May, pp. 1997–2008, 2008.

[29] 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.

[30] Budimlič, Zoran and Yan, Yonghong and Cavè, Vincent and Sarkar, Vivek, "Habanero-C Project." `https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C`.

[31] 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.*, vol. 40, pp. 519–538, Oct. 2005.

[32] R. Barik, Z. Budimlič, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar, "The Habanero Multicore Software Research Project," in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, (New York, NY, USA), pp. 735–736, ACM, 2009.

[33] S. Chandra, V. Saraswat, V. Sarkar, and R. Bodik, "Type Inference for Locality Analysis of Distributed Data Structures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, (New York, NY, USA), pp. 11–22, ACM, 2008.

[34] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical Place Trees : A Portable Abstraction for Task Parallelism and Data Movement," *22nd International Workshop on Languages and Compilers for Parallel Computing*, 2009.

[35] R. Barik and V. Sarkar, "Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 41–52, IEEE Computer Society, 2009.

[36] R. Lublinerman, J. Zhao, Z. Budimlič, S. Chaudhuri, and V. Sarkar, "Delegated Isolation," in *Proceedings of OOPSLA 2011*, October 2011.

[37] 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, (New York, NY, USA), pp. 277–288, ACM, 2008.

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

[39] 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.

[40] Hewitt, Carl and Baker, Henry G., "ACTORS AND CONTINUOUS FUNCTIONALS," tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, February 1978.

[41] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[42] R. Virding, C. Wikström, M. Williams, and J. Armstrong, *Concurrent programming in ERLANG (2nd ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[43] J. Armstrong, "Concurrency Oriented Programming in Erlang," *Challenge*, November 2002.

[44] D. Kafura, "ACT++: Building a Concurrent C++ with Actors," *J. Object Oriented Program*, vol. 3, pp. 25–37, April 1990.

[45] J.-P. Briot, *Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment*, pp. 109–129. Cambridge University Press, 1989.

[46] C. Tismer, "Continuations and Stackless Python," in *Proceedings of the 8th International Python Conference*, 2000.

[47] J. Ayres and S. Eisenbach, "Stage: Python with Actors," in *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, (Washington, DC, USA), pp. 25–32, IEEE Computer Society, 2009.

[48] J. Sillito, "Stage: Exploring Erlang Style Concurrency in Ruby," in *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, (New York, NY, USA), pp. 33–40, ACM, 2008.

[49] Microsoft Corporation, "Asynchronous Agents Library." `http://msdn.microsoft.com/en-us/library/dd492627.aspx`.

[50] Rettig, Mike, "retlang: Message based concurrency in .NET." `http://code.google.com/p/retlang/`.

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

[52] 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*, vol. 5142/2008, pp. 104–128, 2008.

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

[54] M. Astley, "The Actor Foundry: A Java-based Actor Programming Environment," 1998.

[55] The GPars team, "The GPars Project - Reference Documentation." `http://www.gpars.org/guide/guide/`. [Online; accessed 22-October-2011].

[56] 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*, no. Section 2, pp. 15–30, 2006.

[57] 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, (Berlin, Heidelberg), pp. 368–383, Springer-Verlag, 2010.

[58] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir, "Parallel Programming Must Be Deterministic by Default," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2009.

[59] E. Westbrook, J. Zhao, Z. Budimlič, and V. Sarkar, "Permission Regions for Race-Free Parallelism," in *Proceedings of the 2nd International Conference on Runtime Verification (RV)*, 2011.

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

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

[62] N. Raja and R. K. Shyamasundar, "Actors as a Coordinating Model of Computation," in *Proceedings of the Second International Andrei Ershov Memorial*

Conference on Perspectives of System Informatics, (London, UK), pp. 191–202, Springer-Verlag, 1996.

[63] 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, (New York, NY, USA), pp. 11–20, ACM, 2009.

[64] P. Haller and M. Odersky, "Event-Based Programming Without Inversion of Control," in *In Proc. Joint Modular Languages Conference (2006), Springer LNCS*, vol. 4228 of *Lecture Notes in Computer Science*, pp. 4–22, Springer Berlin / Heidelberg, 2006.

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

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

[67] Habanero Research Group, "Habanero-Scala." `http://habanero-scala.rice.edu/`.

[68] Wikipedia, The Free Encyclopedia, "Producer-consumer problem." `http://en.wikipedia.org/wiki/Producer-consumer_problem`.

[69] Sanders, Beverly A. and Massingill, Berna L. and Mattson, Timothy G., "The Pipeline Pattern." `http://www.informit.com/articles/article.aspx?p=366887&seqNum=8`.

[70] Wikipedia, The Free Encyclopedia, "Sieve of Eratosthenes." http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

[71] Llanos, Diego R., "Speculative parallelization of sequential algorithms." http://www.infor.uva.es/~diego/speculative.html.

[72] A. Anagnostopoulos, R. Bent, E. Upfal, and P. V. Hentenryck, "A simple and deterministic competitive algorithm for online facility location," *Inf. Comput.*, vol. 194, pp. 175–202, November 2004.

[73] 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, (New York, NY, USA), pp. 57–76, ACM, 2007.

[74] The Scala Programming Language, "pingpong.scala." http://www.scala-lang.org/node/54.

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

[76] EPCC, "The Java Grande Forum Multi-threaded Benchmarks."

[77] UIUC, "SOTER project."