A Java Library for Actor-Based Cooperative Scheduling

Anonymous Author(s)

Abstract

In this paper we introduce and evaluate a new actor-based library in Java with a scalable implementation of cooperative scheduling of messages within an actor. This allows for multiple entry points for suspending and resuming execution while processing a message. By means of a typical benchmark we evaluate our proposed solution and compare it to other thread-based implementations of cooperative scheduling.

Keywords Actors, Cooperative Scheduling, Functional Programming, Object-Orientation, Java Library,Scala

ACM Reference Format:
Anonymous Author(s). 2017. A Java Library for Actor-Based Cooperative Scheduling. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, New York, NY, USA, January 01–03, 2017 (PL'17),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Actor-based models of computation in general assume a run-to-completion mode of execution of the messages. The Abstract Behavioral Specification (ABS) Language [14] extends the Actor-based model with a high-level synchronization mechanism which allows actors to suspend the execution of the current message and schedule in a cooperative manner another queued message. This extension is a powerful means for the expression and analysis of fine-grained run-time dependencies between messages.

In ABS, asynchronous messages are modeled as methods of an actor. Sending an asynchronous message will schedule the execution of the corresponding method in the callee, returning immediately a future that, upon completion, will hold the result of the method execution, or in case of errors, the thrown exception. This "programming to interfaces" paradigm enables static type checking of message passing at compile time. This is in contrast to the typical approach of actors in Scala where messages are allowed to have any type and thus are only checked at run-time whether the receiver can handle them.

ABS further extends the Actor-based model with a high-level synchronization mechanism that allows actors to suspend in a cooperative manner the execution of the current message and start another queued message. The continuation of the suspended message, which is automatically put back in the message queue, may also be assigned an enabling condition. Cooperative scheduling is a powerful means for the expression and analysis of fine-grained run-time dependencies between messages. A typical use case is awaiting the completion of a future in the same method that has sent the corresponding message. This simplifies the programming logic by enabling the programmer to process the outcome of an asynchronous method call in the same place where it was sent (in a way similar to dealing with synchronous calls). It is important to observe that the continuation will be executed in the same actor thread thus (unlike the `onComplete` hooks in Scala that may run in a different thread) pose no threat to actor semantics.

The main challenge is the development of an efficient and scalable implementation of cooperative scheduling. The basic feature of our proposed solution is the implementation of messages as described in [15] by means of lambda expressions (as provided by Java 8), i.e., the method call specified in a message is translated into a corresponding lambda expression which is passed and stored as an object of type Callable or Runnable (depending on whether it returns a result). We show how this basic feature is integrated in a general Java run-time system for ABS which also caters for synchronous calls that give rise to the suspension of the entire call stack generated from a message.

We describe how we implemented ABS actors and cooperative scheduling using Java thread pools and executor services. As elaborated in Section 6 our benchmark, we took an approach with negligible performance penalties regarding message passing and cooperative scheduling. We describe step by step how to support up to millions of messages, actors, and suspended continuations.

We will also use the N-Queens problem provided by the Savina benchmark for programming with actors[13] to evaluate our proposed solution and compare it to other thread-based implementations in Java or Erlang(a language that uses lightweight threads) of cooperative scheduling, as well as Akka actor implementation provided by the test suite.

Related Work. In this paper we provide a Java implementation of the ABS language as a full-fledged Actor-based programming language which features cooperative scheduling and fully supports the "programming to interfaces" paradigm. In contrast, Java libraries for programming actors like

A note.
PL'17, January 01–03, 2017, New York, NY, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Akka [10] mainly provide pure asynchronous message passing which does not support the use of application programming interfaces (API) because, for example, in Akka(Java), a message is only typed as a Java Object, so there is no static typing of messages, nor are they part of the actor interface. A first approach for supporting programming to interfaces was proposed in Takka library [12]. Distinguishing features of the ABS comprise of high-level constructs for cooperative scheduling which allow the application of formal methods, e.g., formal analysis of deadlocks [8]. The Actor model in Scala [11] does provide a suspension mechanism, but its use is *not* recommended because it actually blocks the whole thread and causes degradation of performance. It is possible to register a *continuation* piece of code to run upon completion of a future, but that may run in a separate thread which however breaks the actor semantics and may cause race conditions inside the actor.

The ABS Language has been developed for modeling parallel and distributed systems based on the actor model and provides extensive support for formal analysis like functional analysis [6], resource analysis [1] and deadlock analysis [7].

There exist various implementation attempts for cooperative scheduling in ABS. The approach followed in the Java backend of ABS [14, 16] and in the Erlang backend [9] is process-oriented in the sense that sending a message is implemented by the generation of a corresponding process. We will compare these backends with our solution in Java which allows to store messages (as objects of type Callable or Runnable) before executing them (data-oriented). A different approach is in the Haskell backend for ABS [2], where the use of continuations allows to queue a message as a process and dequeue it for execution.

Implementations in other languages allow for different approaches: The C implementation of cooperative scheduling for the Encore language [5] uses low-level (e.g., machine code) operations for storing and retrieving call stacks from memory. In contrast, in the Actor-based modeling language Rebeca [17], for example, messages are queued and processed in a run-to-completion mode of execution.

The rest of this paper is organized as follows: In Section 2 we give an overview of the main features that the target actor-based model has. Section 3 presents the API for using the library with examples for passing messages and futures between actors. Section 4 presents how our solution transitions from a process-oriented approach to a data-oriented approach. Section 5 describes the implementation the runtime system for the actor-based model. In Section 6 we show the experimental evaluation of our solution followed by the conclusions drawn in Section 7.

2 Cooperative Scheduling in ABS

In this section we informally describe the main features of the flow of control underlying the semantics of cooperative scheduling in the ABS language. For a detailed description of the syntax of the ABS language we refer to [14].

In ABS a statement of the form `await f?` suspends the executing process which can only be rescheduled if the method invocation corresponding to the future `f` has computed the return value. Similarly, a statement of the form `await b`, where `b` denotes a boolean expression, suspends the executing process which can only be rescheduled if `b` evaluates to true. Finally a `suspend` statement invokes cooperative scheduling without a condition. In all cases the executing process can be suspended and another (enabled) message can be scheduled for execution. In contrast, a statement like `f.get` does not allow the scheduling of another process of the same actor. It thus blocks the entire actor. Listing 1 gives an example ABS model comparing cooperative scheduling to pure asynchronous communication as in pure actor model. It is important to observe the call by CooperativeMaster that is followed by an `await f?`. This allows the master to send computation requests to the workers and then wait for the results. This call suspends the current message execution of the master, but still allows it to process other messages. In contrast, PureMaster receives the results in another call (success) enforcing the programmer to divide the logic in two separate methods. In Section 6.2 we will discuss the performance of ABS actors in comparison with the Akka Actor model implementation.

Listing 1. Cooperative scheduling vs. pure async approach

```
class PureMaster (Int numWorkers) implements IMaster {
 Unit sendWorkRoundRobin(...) {
  IWorker worker = chooseWorker();
  worker ! work(...);
 }
 Unit success(Result result) { /* process results */ }
}
class CooperativeMaster (Int numWorkers) implements IMaster {
 Unit sendWorkRoundRobin(...) {
  IWorker worker = chooseWorker();
  Future<Result> fut = worker ! work(...); // start computation
  await fut?;
  // process results
}}
```

3 Library API

The core of ABS actors is written in Java and can be used in both Scala and Java, but due to the native functional approach in Scala, we describe the examples in Scala. The library can be obtained as a maven project and is available online [1], along with several N-Queens benchmark implementations. Defining an actor is simply done by extending the LocalActor class. Throughout this section we will show the use of the API through the implementation of such an actor (the Master) in the N-Queens benchmark in Listing 2. Thus by instantiating an instance of such actor classes, automatically the object will have the message queue and conceptually one thread of

[1]https://github.com/vlad-serbanescu/abs-api-cwi.git (get-spawn branch)

execution. The actor's reference can then be used and passed in between objects and actors normally. It can then receive messages asynchronously and react to them. The Actor API provides three methods for asynchronous communication and concurrency control.

Listing 2. Master Actor Class

```
1  class Master(var numWorkers : Int,var priorities : Int,
2   var solutionsLimit : Int,var threshold : Int,var size : Int)
3   extends LocalActor with IMaster {
4
5   private val workerSeq = for (_ <− 1 to numWorkers) yield {
6     new Worker(this, threshold, size)
7   }
8
9   private final val workers = Iterator.continually(workerSeq).flatten
10  private val t1 = System.currentTimeMillis()
11
12  def sendWork(list: Array[Int], depth: Int, priorities: Int):
13    ABSFuture[List[Array[Int]]] = {
14      val worker = workers.next()
15      worker.send(() =>
16        worker.nqueensKernelPar(list, depth, priorities))
17  }
18
19  def init: ABSFuture[Void] = {
20    val inArray: Array[Int] = new Array[Int](0)
21    val fut: ABSFuture[List[Array[Int]]] =
22    this.send(() => this.sendWork(inArray, 0, priorities))
23    getSpawn(fut, (result: List[Array[Int]]) => {
24      println("Found␣{result.size}␣solutions")
25      println("−−␣Program␣successfully␣completed!␣in␣"
26        + (System.currentTimeMillis() − t1))
27      ActorSystem.shutdown()
28      ABSFuture.done()
29    } )
30  }
31  {
32    this.send(() => this.init)
33 } }
```

Sending Asynchronous messages The send method has the following signature:

```
<V> ABSFuture<V> send(Callable<ABSFuture<V>> message);
```

Messages an actor can receive must return an ABSFuture. The method takes a *message* parameter that specifies the task to be run asynchronously by the actor. An important observation to make here is that we cannot enforce the *message* to be a method provided by one of the interfaces that the actor extends, although it is recommended to only use methods exposed by these interfaces. An example of sending an asynchronous message is shown in line 15-16 of Listing 2

Spawning an internal task To implement cooperative scheduling our library provides abstractions for guards that control suspension and release points. The are supported through the abstract class *Guard* and its subclasses *FutureGuard*, *PureExpressionGuard* and *ConjunctionGuard* that describe the conditions on which an actor's message can await: either a future, a particular valid expression or a group of these conditions respectively. The spawn method has the following signature:

```
<V> ABSFuture<V> spawn(Guard guard, Callable<ABSFuture<V>> message);
```

The method takes as parameters an enabling condition (guard) and task to run (message) once the guard is satisfied. The method returns a ABSFuturewhich is the return type of the message to be executed. The *spawn* method of the API together with the run-time system presented in Section 5 is used to emulate the *await* semantics and cooperative scheduling. The pattern is very simple, it works as a normal call to spawnif the condition is not satisfied, while the continuation that follows await is ran directly if the condition is satisfied straight away.

Blocking an Actor The getSpawn method has the following signatures:

```
<T,V> ABSFuture<T> getSpawn(ABSFuture<V> f, CallableGet<T, V> message);
<T,V> ABSFuture<T> getSpawn(ABSFuture<V> f, CallableGet<T, V> message,
                int priority, boolean strict);
```

This method has a similar behaviour to spawnbut may only be used together with a *FutureGuard* (converted from ABSFuture f) and also propagates the value of this future (inside CallableGet) to the *spawned* task once it is ready. This method must be used to read a future as reading it directly will cause the blocking of the whole thread if the value is not available. Future references can be passed between actors. Reading the future value using getSpawn, allows the actor to run other tasks in case the future is not completed without forcing the actor to block. Once the future is ready, the *message* parameter will contain the value of this future together with the task that is to be run. This limits the numbers of threads that live in the system, yet are blocked, without breaking the actor encapsulation. The usage of getSpawn is shown in an example in line 24-30 of Listing 2. If we want messages and continuations to execute in a particular order (such as preserving a synchronous call stack) we can set a higher or lower *priority* for a message. Furthermore, there may the case when we would like the actor to block and therefore the setting the boolean parameter *strict* will determine if the actor will block.

4 Implementation Schemes in Java

In this section we present how the library API described in Section 3 is used to provide a run-time system and backend for ABS in Java and further extended into Scala. We first look a a summary of the evolution of the scheme used to implement cooperative scheduling in the Java Runtime environment. We start from a very trivial approach to using several data structures and Java 8 features that the latest version of SDK provides. The first feature we will use is the Executor interface that facilitates thread programming. The objects that implement this interface provide a queue of tasks and an efficient way of running those tasks on multiple threads. Throughout this section we will use the terms executor and thread pool interchangeably to refer to this feature. The second notion discussed in our solution was introduced starting with the Java 8 version, and it is the construct for a lambda expression [2]. This allows us to model a method call as a Runnable or Callable and we will refer to this model as a **task**. When cooperative scheduling occurs, the executing task of an actor will be suspended and therefore still live in the system as a thread so the application's live threads will be equal to the number of "await" statements in the program.

4.1 Summary of the Implementation Evolution

The trivial straightforward approach for implementing cooperative scheduling in the library is for an asynchronous call to generate a new native thread with its own stack and context. We would then model each actor as an object with a lock for which threads compete. The main drawback of this approach is the large number of threads that are created, which restricts any application from having more method calls than the number of live native threads.

To reduce the number of live threads in an application, we can model each invocation as a task using lambda expressions and organize each actor as a thread pool. This gives the actor an implicit queue to which tasks are submitted. We obtain a small reduction in the number of threads corresponding to the number of tasks that have been submitted, but not started. The number of live threads can be restricted to the number of threads allowed by each Thread Pool, while the rest of the invocations remain in the thread pool queue as tasks.

To avoid unnecessary blocking on the lock, we introduce *one thread pool per system* or *the system's executor*, and instead of submitting the messages directly to the thread pool, we add an indirection by storing them into an explicit queue first and introduce as a second phase the submission to the thread pool. We assign a separate task for each actor to iterate through its associated queue and submit the next available message to the system thread pool. We will refer to this task from now on as the **Main Task** of an actor. This removes the requirement to store every message handler as a thread, after it starts and attempts to acquire a lock, saving a task as data in the heap instead. However it comes at the cost of having to manage message queues manually as we need an explicit queue for all the messages that have not yet been submitted (to the thread pool).

4.2 Fully asynchronous environment.

In the absence of synchronous calls, to eliminate the problem of having live threads when cooperative scheduling occurs, we can also use lambda expressions to turn the continuation of an await statement (which is determined by its lexical scope) into an internal method call and pass its current state as parameters to this method. Essentially what we do is allocate memory for the continuation on the heap, instead of holding a stack and context for it. We will benchmark this

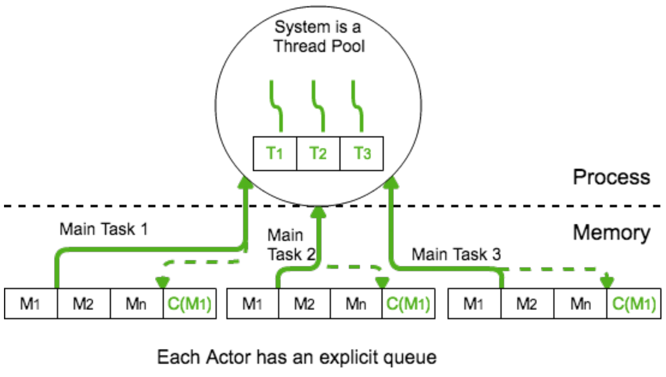

Figure 1. Full data-oriented approach

trade-off between the memory footprint of a native thread and the customized encoding of a thread state in memory. As there are no more suspended threads in the system to compete with the actor's Main Task, we can eliminate the lock per actor. The four features of the Actor are as follows:

- The actor is initialized as an object with an explicit queue and an implementation of the of the **Main Task**, but without a lock;
- Asynchronous calls are created as new tasks that will be run by the **Main Task** and will compete for the lock only with suspended tasks;
- Suspension of a call first saves the continuation as a lambda expression guarded by the release condition, and the **Main Task** continues to iterate through the queue;
- The process of saving the call stack of a synchronous call is detailed in the next paragraph.

Synchronous calls context. In the presence of synchronous calls, the problem that remains is how to save this call stack without degrading performance? To do this we can try to alter the bytecode to resume execution at runtime from a particular point, but we want our approach to be independent of the runtime and be extensible to other programming languages. Therefore we try to approach the problem differently; if we can turn a continuation, that does not originate from a stack of calls, into a task, is it possible to extend this to synchronous calls as well? We know that this issue arises when methods that contain an "await" statement are called synchronously. When the `await f?` is encountered, the continuation consists of both the lexical scope of the release statement that is followed by the a complete synchronous call chain that has been generated(i.e. $C(M_1)$ in Figure 1).

A fully asynchronous environment means a synchronous call `x = this.m();` must return a ABSFuturewhich will then be used to control the rest of the method execution. There are two constraints here. First, we still need to make sure that "m" must be the exact next method that will run.

[2]https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Second, when "m" finishes, the actor scheduler must immediately schedule the method that is waiting for its result. Both can be implemented by using getSpawn with the pattern in Listing 3:

Listing 3. Synchronous Call Pattern

```
ABSFuture<Int> f = this.m();
if(f.isDone()){
  //execute continuation
}
else{
  return getSpawn(f, (result: Int) => {
    //execute continuation
  }, this.HIGH_PRIORITY, false );
}
```

In this manner, “m" executes normally, and if it does not complete, the HIGH_PRIORITY parameter of getSpawn allows the continuation to execute first once the ABSFuture f completes and preserves the call stack.

5 Run-time Implementation

The implementation of the full data-oriented scheme is done through a library which contains a set of classes and interfaces that have a direct mapping to the ABS language concepts described in Section 2. The library provides an implementation of cooperative scheduling behavior, the suspension and release mechanisms while respecting the logic of synchronous calls. The library provides the solution and the optimizations discussed in Section 4.

5.1 Actor Implementation

The library currently supports an implementation of this interface called **LocalActor** for actors running on the same machine. An important advantage of having a task assigned to each actor and a manually processed queue is that we can start and stop the task depending on the queue state. To avoid keeping the Main Task alive, we make it part of the functionality of sending a message or completing a future to notify the Main Task. This is done by any other actor who sends an invocation to an empty queue and subsequently the Main Task stops when there are no more enabled messages in the queue.

Inside the **LocalActor** class there is a *messageQueue* defined which holds all the invocations (synchronous or asynchronous) that have been submitted to the actor as tasks. The implementation defines an inner class *MainTask* which is a Java Runnable that corresponds to the task responsible for taking messages from the queue and running them. When the queue is empty, we do not want to make this task busy-wait until a message arrives, and so, upon insertion of a new message into the queue, there is a check whether such a task exists already. This gives rise to some subtle synchronization points. For every actor, we keep a local atomic boolean flag *mainTaskIsRunning*. A first approach looks like in Listing 4:

Listing 4. Basic Synchronization Approach

```
 // inside the task
 class MainTask implements Runnable{
  public void run() ({
     // iterate through queue and take one message and run it
    mainTaskIsRunning.set(false);
 } }

 // when inserting a new message
 messageQueue.add(m);
 if (!mainTaskIsRunning.compareAndSet(false, true)) {
  ActorSystem.submit(new MainTask())
 }
```

Listing 5. Complete Synchronization Approach

```
private boolean takeOrDie() {
  synchronized (mainTaskIsRunning) {
    // iterate through queue and take one ready message
    // if it exists set the next message for the main task and then
    return true;

    //if the queue if empty or no message is able to run
    mainTaskIsRunning.set(false);
    return false;
} }
private boolean notRunningThenStart() {
  synchronized (mainTaskIsRunning) {
    return mainTaskIsRunning.compareAndSet(false, true);
} }

// inside the task
class MainTask implements Runnable{

  public void run() {
  while (takeOrDie())
    // proceed to take the next message message and run it
} }

// when inserting a new message
messageQueue.add(m);
if (notRunningThenStart()) {
  ActorSystem.submit(new MainTask());
}
```

The problem with the code in Listing 4 is that the check of the queue for emptiness and setting the flag to "false" is not atomic, and in between these two statements, a new message may be inserted into the queue without spawning a new Main Task. To remedy this, in Listing 5, we introduce a new method that can check the queue for emptiness and set the flag to "false" in a critical section, for example inside a "synchronized" block using the *messageQueue* or *mainTaskIsRunning* as the lock. Additionally, either the insertion into the *messageQueue* or *compareAndSet* of *mainTaskIsRunning* should also use the same lock. Another problem with this approach above is that, when there are more actors than available threads, there is no fairness policy when assigning a thread to an actor. To remedy this, we could change the while loop to an if statement like in Listing 6:

Listing 6. Fairness Between Actors

```
class MainTask implements Runnable{
  public void run() {
    if (!takeOrDie())
      return;
    // proceed to take the next message message and run it
    ActorSystem.submit(this);
}}
```

5.2 Actor System Implementation

The system executor uses a *newFixedThreadPool* singleton *ExecutorService* that automatically creates and allocates threads to handle the current load of the system. An actor may start a new MainTask (for processing messages in the queue) by calling the static method submit(**new** MainTask()). Inside the class there is also support to safely call shutdown() when all the actors in the application have completed execution. Further in this section we explain some details and optimizations implemented in management of actors and their corresponding tasks.

Eliminating busy waiting. Cooperative scheduling through release points may suspend the current message run by the actor based on either a particular inner state or future requiring completion on a different actor. We discussed how the Main Task can start or stop based on release points, but it is important to observe how exactly does an actor verify that a release point has completed. Having a task continuously iterate through all suspended messages (busy-waiting) is inefficient. We can separate the messages that are guarded by an incomplete future until that future completes such that the Main Task does not need to check them. Subsequently they can be put in the *messageQueue* once *FutureGuard* completes. The same procedure cannot be applied for a message whose enabling condition depends on the actor state, as its state can change again, so it always has to be verified before a message is scheduled. We assign this verification to the Main Task that iterates through the queue, and simply stop the task if no message is available after one iteration. If the task does stop, it means that the actor is in a state in which it is unable to execute any of its suspended messages. Therefore, it requires another actor to either send a new invocation that will change its state or the system to send a notification about a future that may release some of its messages and re-enable the Main Task. Therefore we maintain a global hash table, mapping every future to the set of actors that are awaiting on its completion. When a Future completes it can call the method notifyDependant() (provided at the system level) to notify actors that contain messages suspended on that particular future.

Using JVM Garbage Collection Using the approach explained so far in this section, the only extra references we need for the actors are the ones inside the global hash-table required for the notification mechanism for futures. Once the future is completed and notifications are sent, the key is deleted and the actor references become unreachable. Therefore we can leave the entire garbage collection process to the Java Runtime Environment as no other bookkeeping mechanisms are required. This way we do not keep a registry of the actors like the `context` in Scala and Akka.

6 Benchmarking Results

In this section we want to evaluate Scala library in terms of the impact that the cooperative scheduling feature has on performance. We will first use a benchmark that involves heavy cooperative scheduling to compare the data-oriented approach to the process-oriented approach when translating from ABS to Java. Furthermore we will use a benchmark tailored towards CPU-intensive to evaluate the potential of this Scala library to support ABS as a software-development language. We will compare our solution with both the Erlang backend of ABS which benefits from lightweight threads as well as the Akka Actor library for Scala.

Listing 7. Benchmark Example

```
trait AInterface extends Actor {
  def recursive_m( i : Int,  id : Int): ABSFuture[Int];
}

class A extends LocalActor with AInterface {

  var result: Int = 0;

  def recursive_m(i: Int, id: Int): ABSFuture[Int] = {
    if (i > 0) {
      val f: ABSFuture[Int] = this.recursive_m(i − 1, id);
      if (f.isDone) {
        return ABSFuture.of(1);
      } else {
        return getSpawn(f, (res: Int) => {
          return ABSFuture.of(1);
        }, Actor.HIGH_PRIORITY, false);
    }}
    else {
      var cf: ABSFuture[Int] = this.send(() => this.compute())
      return spawn(Guard.convert(cf),()=>{ABSFuture.of(1)})
  }}

  def compute(): ABSFuture[Int] = {
    this.result = this.result + 1;
    ABSFuture.of(result);
}}

object Main extends LocalActor {

  def main(args: Array[String]): Unit = {
    var i: Int = 0;
    val master: AInterface = new A();
    val futures: util.LinkedList[ABSFuture[Int]] = new util.LinkedList[ABSFuture[Int]]()
    while (i < 500) {
      val f: ABSFuture[Int] = master.send(() => master.recursive_m(5, i))
      futures.add(f);
      i = i + 1;
    }
    while (!futures.isEmpty) {
      //wait until all actors finish
}}}
```

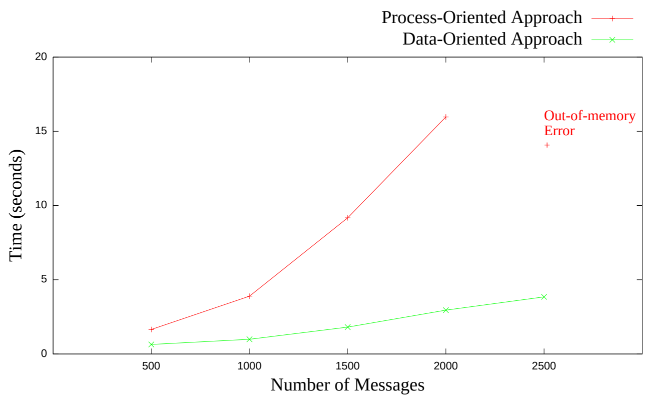

Figure 2. Performance figures for pure Cooperative Scheduling

6.1 Cooperative Scheduling Benchmark

The main problem that we encountered when implementing cooperative scheduling was saving the context of an execution and resuming from that context. To do this in Java using threads and context switches heavily limits the application to the number of native threads that can be created. To measure the improvement provided by our Java library features using a simple example that creates a recursive stack of synchronous calls. A sketch of the ABS model is presented in Listing 7.

The model creates an Actor of type "A" and sends a large number of messages to it to execute a method recursive_m(5,i. This method creates a call chain of size 5 before sending an asynchronous message to itself to execute a basic method compute() and awaits on its result. Although simple, this example allows us to benchmark the pure overhead that arises from having a runtime system with cooperative scheduling support, both in a data-oriented approach and a process-oriented approach. The results are shown in Figure 2. The performance figures presented are for one actor that is running 500-2500 method invocations. It is important to observe that each invocation generates 2 messages in the actor's queue, so as the number of calls increases the number of messages doubles. The figures show that the trade-off for storing continuations into memory instead of saving them in native threads removes limitations on the application and significantly reduces overhead.

6.2 N-Queens Benchmark

While our solution is catered towards a widely-used software development language, we would like to also compare with other languages that implement ABS language concepts efficiently using threads and without these limitations. In Section 5 we listed several optimizations that were inferred from our implementation solution. What we want to do is compare this optimized solution to an ABS backend implemented in Erlang that uses the same process-oriented

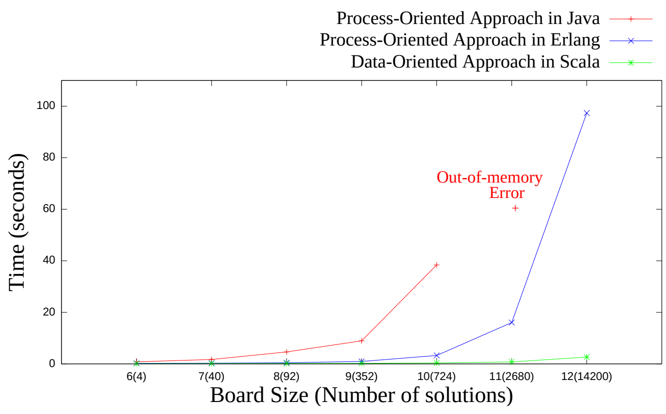

Figure 3. Results for the N-Queens problem in 3 ABS backends

approach but does not suffer from any limitation of native threads. For this comparison, we use the Savina benchmark for programming with actors [13]. We chose the problem of arranging N queens on a NxN chessboard as it provides a master-slave model that can be implemented using actors. The benchmark divides the task of finding all the valid solutions to the N queens problem to a fixed number of workers that at each step have to find an intermediary solution of placing a queen K on the board before relaying the message back to the master which then assigns the next job of placing queen $K + 1$ to another worker. As the search space becomes smaller, w impose a threshold where the worker has to find the complete solution up to N queens before sending a message back to the master.

We ran the benchmark with a board size varying from 6 to 12 with a fixed number of 4 workers on a core i5 machine which supports hyper-threading. The results are shown in Figure 3. It is important to observe that as the board size increases, the number of solutions grows from 40 to 2680. The results show that our approach is better once the board size reaches 12 and the number of solutions to be found is 14200. This result also strengthens ABS purpose to provide a programming language for real applications.

Furthermore, we would like to observe the performance impact that support for cooperative scheduling adds to an application. This feature is a powerful programming abstraction, but we would like to see the cost of developing a benchmark in comparison to an existing actor implementation like the Akka Actor library. Both the Pure Asynchronous model that uses the API and the Akka implementation both require the expected number of solutions for a particular board size, while the Cooperative Scheduling just has to wait for all the possible solutions to be computed. It is also important to observe that the Scala model for this benchmark is 116 lines of code, while the benchmark written directly using the Akka library is 351 lines of code. This reduction in the code is due cooperative scheduling and the API not requiring a

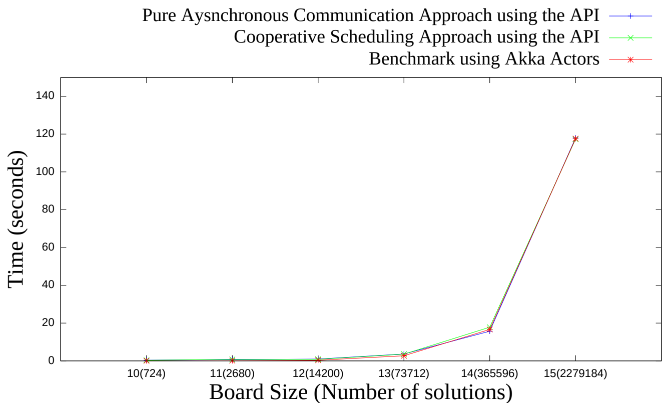

Figure 4. Results for the N-Queens problem implementations in Akka and Java Library

context setup for actors and explicit build-up of messages and subsequent pattern matching on them. We compared the implementation offered by the Savina benchmark with both our approaches for a board size from 10 to 15 on the same machine as before. The results in Figure4 show that the cooperative scheduling feature has the same performance results compared to the Akka Actor library while offering a seamless programming experience. The library currently uses a Fixed Thread Pool for the actors but this data structure can be fine-tuned depending on the application that is run to improve performance.

7 Conclusions

In this paper we proposed a Java library for efficiently implementing the cooperative scheduling behavior of ABS which provides a powerful programming abstraction. To provide support for the functional data types required by ABS we embedded the Java run-time system into the Scala programming language. Having a portable JVM library gives us a basis for industrial adoption of the ABS language and provides ABS as a powerful extension of Scala with support for formal verification, resource analysis and deadlock detection as well as extensions that support real-time programming [4], in a software development context.

For future work, the ABS extension of Scala provide can be integrated with the distributed ABS implementation that exists in the Haskell backend [3] to provide a distributed programming model for Actor-based applications.We also plan to extend the library to statically type-check the message submitted via the *send* method in order to prevent the user from running unwanted code on the actors. To this end we plan to provide a syntactic sugar for an asynchronous invocation that can be used directly to send a message to an Actor.

References

[1] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. 2014. SACO: Static Analyzer for Concurrent Objects.. In *TACAS*, Vol. 14. 562–567.

[2] Elvira Albert, Nikolaos Bezirgiannis, Frank de Boer, and Enrique Martin-Martin. 2016. A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. *arXiv preprint arXiv:1608.02896* (2016).

[3] Nikolaos Bezirgiannis and Frank de Boer. 2016. ABS: a high-level modeling language for cloud-aware programming. In *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 433–444.

[4] Joakim Bjørk, Frank S de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S Lizeth Tapia Tarifa. 2013. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering* 9, 1 (2013), 29–43.

[5] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, S Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel objects for multicores: A glimpse at the parallel language Encore. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 1–56.

[6] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. 2015. KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In *International Conference on Automated Deduction*. Springer, 517–526.

[7] Antonio E Flores-Montoya, Elvira Albert, and Samir Genaim. 2013. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Formal Techniques for Distributed Systems*. Springer, 273–288.

[8] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. 2016. A framework for deadlock detection in core ABS. *Software & Systems Modeling* 15, 4 (2016), 1013–1048.

[9] Georg Göri, Einar Broch Johnsen, Rudolf Schlatte, and Volker Stolz. 2014. Erlang-style error recovery for concurrent objects with cooperative scheduling. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 5–21.

[10] Munish Gupta. 2012. *Akka essentials*. Packt Publishing Ltd.

[11] Philipp Haller and Martin Odersky. 2009. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410, 2 (2009), 202–220.

[12] Jiansen He, Philip Wadler, and Philip Trinder. 2014. Typecasting actors: from Akka to TAkka. In *Proceedings of the Fifth Annual Scala Workshop*. ACM, 23–33.

[13] Shams M Imam and Vivek Sarkar. 2014. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*. ACM, 67–80.

[14] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2012. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*. Springer, 142–164.

[15] Behrooz Nobakht and Frank S de Boer. 2014. Programming with actors in Java 8. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 37–53.

[16] Jan Schäfer. 2011. *A programming model and language for concurrent and distributed object-oriented systems*. University of Kaiserslautern.

[17] M Sirjani and A Movaghar. 2001. An actor-based model for formal modelling of reactive systems: Rebeca. *Technical Report* (2001).