Cover Page

The handle http://hdl.handle.net/1887/97598 holds various files of this Leiden University dissertation.

Author: Serbânescu, V.
Title: Software development by abstract behavioural specification
Issue Date: 2020-06-10

Chapter 8

Case Studies and Benchmarks

This chapter covers all the use cases of the integration of formal methods into software development using ABS, Java 8 and Scala. The first part covers several typical benchmarks for actor-based programming that validate the usage of ABS together with the Scala Backend compared to the existing state-of-the art Erlang backend that is the community's most stable and supports all of ABS's features. At the same time these benchmarks validate the Java Runtime System and ASCOOP as a standalone library compared to other state of the art Actor libraries like Scala and Akka. In the second part, a case study that uses Map Reduce is presented to underline the effectiveness of the runtime system on a single machine. It is compared to the ProActive backend which has stable distributed support, yet uses the thread-abstraction model. This case study makes of a comparative analysis of both backends, with advantages and disadvantages of each one. The third part of this chapter shows a very useful example of having support for ABS together with resource modeling support for simulating a cache coherence memory model. Again this case study is compared to the Erlang backend in terms of how well the program runs with high memory and CPU loads. The last part of this chapter presents a case study in which ABS together with symbolic time support in order to visualize the progression of a railway model. This case study highlights the importance of discrete time simulation extension of ABS.

8.1 Cooperative Scheduling Benchmarks

This section shows the comparison of having coroutine support available in Java through either thread-abstraction or spawning tasks. The comparison is first made through an example that relies heavily on coroutines, such that we can measure the overhead that programming with coroutines has on a program. The second example is selected from the Savina benchmark for programming with actors [IS14]. All the benchmarks are ran a core i5 machine which supports hyper-threading and 8GB of RAM on a single JVM. In the library repository[1] we provide implementations of several examples in the benchmark suite directly using the library, while in the compiler repository [2] we have several ABS models of these benchmarks.

8.1.1 Coroutine “Heavy" Benchmark

First the library is evaluated in terms of the impact that programming with coroutines has on performance. The first benchmark involves a large number of suspension and release points in an actor's life cycle in order to compare the spawning approach to the thread-abstraction approach when translating from

[1] `https://github.com/JaacRepo/JAAC`
[2] `https://github.com/JaacRepo/absCompiler`

ABS to Java. 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 we use a simple example that creates a recursive stack of synchronous calls. A sketch of the ABS model is presented in Listing 8.1.

Listing 8.1: Benchmark Example

```
1  interface Ainterface {
2    Int recursive_m(Int i, Int id);
3  }
4
5  class A( ) implements Ainterface{
6
7    Int result=0;
8
9    Int recursive_m(Int i, Int id){
10     if (i>0){
11       this.recursive_m(i − 1,id);
12     }else{
13       Fut<Int> f = this ! compute( );
14       await f ?;
15     }
16     return 1;
17   }
18
19   Int compute( ){
20     return result + 1; //no significant computation
21   }
22 }
23
24 { // Main block:
25   Int i = 0;
26   Ainterface master = new A ( );
27   List<Fut<Int>> futures = Nil;
28
29   while( i < 500){
30     Fut<Int> f = master ! recursive_m (5, i);
31     futures = Cons( f, futures );
32     i = i + 1 ;
33   }
34   while ( futures != Nil ){
35     Fut<Int> f1 = head(futures);
36     futures = tail(futures);
37     Int r = f1.get;
38   }
39 }
```

The model creates an Actor of type “A" and sends a large number of messages to it to execute a method `recursive_m(5,id)`. This method creates a call chain of size 5 before sending an asynchronous message to itself to execute 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 coroutine support,

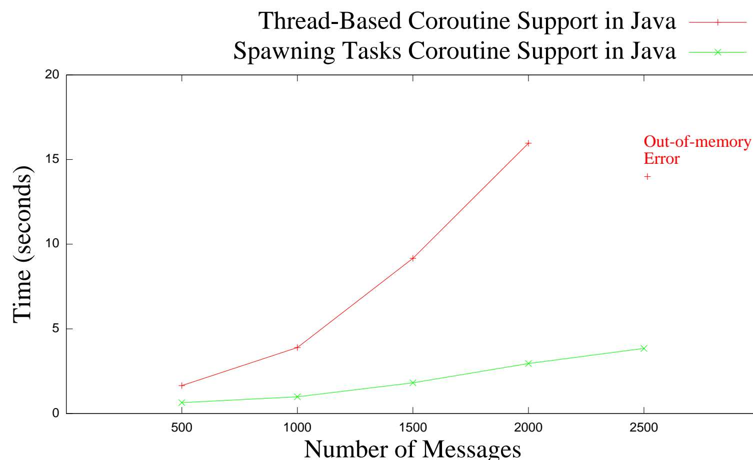

Figure 8.1: Performance figures for Coroutine Overhead

both in a thread-based approach and through spawning of tasks. The results are shown in Figure 8.1. 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 tasks in the actor's queue, so as the number of calls increases, the number of tasks doubles. The figures show that the trade-off for storing continuations and context as tasks into heap memory instead of saving them in native threads removes limitations on the application and significantly reduces overhead.

8.1.2 NQueens Benchmark

From the Savina test suite, we selected the NQueens problem as it is a typical problem with both memory operations and CPU-intensive tasks. Listings 8.2 and 8.3 describe in ABS the problem of arranging N queens on a $N \times N$ chessboard. It provides a master-slave model that illustrates very well the advantage of using actors together with coroutines.

The benchmark divides the task of finding all the valid solutions to the N queens problem into subtasks sent to a fixed number of workers. The `board` is defined as a list of integers where the index of each element represents the line (equivalent to the `depth` of the board) and the number represents the column. Each subtask sent to a worker (line 21 in Listing8.2) requires finding all possible valid solutions of placing the next queen on a `board` filled up to the current `depth`. Once an intermediary solution is found the worker sends an asynchronous call to the `master` (line 12in Listing8.3) to create a new subtask for the new board and the incremented `depth`. The master aggregates the results using a coroutine model (line 24) to await all the solutions starting at depth 0.

We ran the benchmark with a board size varying from 7 to 14 with a fixed number of 4 workers. The results compare the implementations of the NQueens problem and are shown in Figure 8.2. The first two implementations are direct translations in Java from ABS source code with the two co-routine approaches (thread-abstraction and JAAC(spawning)). It is important to observe that as the board size increases, the number of solutions grows from 40 to 14200. The results show that using thread abstraction (where each method invocation generates a corresponding thread) the time taken grows exponentially and cannot complete once the board size reaches 11 while the approach that uses tasks remains unaffected.

The next result (the blue plot) shows the improvement brought by using Java data structures. These results are at a comparable level with the Savina implementation using Akka actors(orange plot). ABS has limited support for data structures (offering only lists, sets and associative lists that can be used as

Listing 8.2: NQueens Master Class Snippet

```
1  class Master (Int numWorkers, Int threshold, Int boardSize, ...) implements IMaster {
2
3   List<IWorker> workers = Nil;
4
5   //... constructor and initializations
6   {
7    Int i = 0;
8    while (i <= numWorkers) {
9    IWorker w = new Worker(this,threshold,size);
10   workers = Cons(w,workers);
11   i = i+1;
12  }
13
14
15  this!sendWork(Nil, 0, ...); // triggers computation
16  }
17
18  //method for receiving solutions
19  Unit sendWork(List<Int> board, Int depth, ...){
20   Fut<Unit> f = nth(workers,messageCounter)!
21   nqueensKernelPar(board,depth,priorities);
22   messageCounter = (messageCounter + 1) % numWorkers;
23   if(depth==0){
24    await f? ;
25    //handling program completion
26   }
27  }
28 }
```

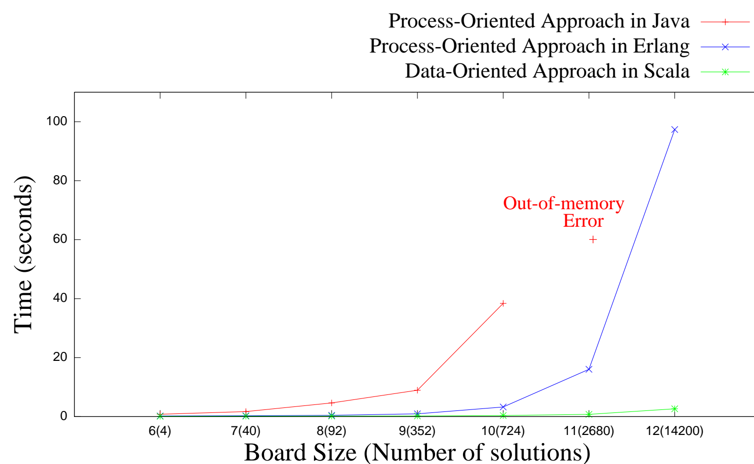

Figure 8.2: Results for the N-Queens problem using two different coroutine approaches

Listing 8.3: NQueens Worker Class Snippet

```
1  class Worker(IMaster master, Int threshold, Int size) implements IWorker {
2
3   Unit nqueensKernelPar(List<Int> board, Int depth, ...) {
4    Int i = 0;
5    if (size != depth) {
6     if (depth >= threshold) {
7      //handle the rest of the solution sequentially and send it to the master
8     } else {
9      while (i < size) {
10      List<Int> newboard = appendright(board,i);
11      if (boardValid(0, newboard,depth+1)) {
12       master!sendWork(newboard,depth+1, ...);
13      }
14      i = i+1;
15     }
16    }
17   }
18   else { //send a solution to the master
19   }
20  }
21 }
```

maps) and by changing the `board` from an ABS list to a Java Array we obtain a significant improvement. This enforces the need of a foreign language interface for ABS to be used a full-fledged programming language.

8.2 Benchmarking the ASCOOP Library

In this section, we evaluate the ASCOOP library in terms of the overhead of creating actors, asynchronous message passing between them, and suspended sub-tasks in actors. To this end, we use some of the examples in the Savina benchmark for programming with actors [IS14], each focusing on one aspect. We compare the execution time of our ASCOOP implementation with those available in the Savina Suite for standard Scala actors and Akka actors. All benchmarks are run on a core i5 machine with 2 CPU that support hyperthreading and 8GB memory.

We use the Ping Pong benchmark for exchanging a large number of messages between two actors. This tests how well each of the libraries handles message passing to the correct actor and how quickly an actor receives a message and processes it. Then we use the Fibonacci benchmark to see how well the library behaves with an exponentially growing number of actors. Finally, we take two typical benchmarks for CPU and memory operations: the Sieve of Eratosthenes and the NQueens problem.

For each of the benchmarks we offer two implementations using ASCOOP Actors. The first one is a typical actor implementation similar to the other libraries with a fire-and-forget approach. This approach, additionally, requires some sort of termination protocol and a prior known condition of termination to ensure correct completion of the application. In the second approach we use the cooperative scheduling feature of ASCOOP actors. This reduces the need for extra message passing to explicitly return the result of a computation; instead, the sender awaits the availability of a future and gets the result from there. Similarly, the termination of the program can be determined by completion of the future corresponding

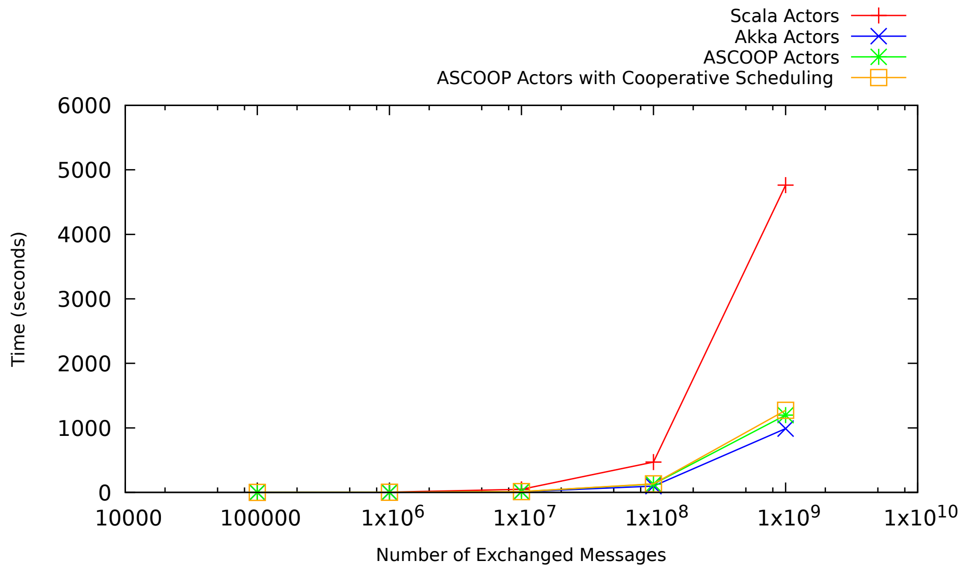

Figure 8.3: Performance figures Ping Pong Benchmark

to the task that starts the whole benchmark, but this can be tweaked to a different task or actor state depending on the benchmark.

8.2.1 Message Passing Overhead

The first benchmark we look at is the Ping Pong benchmark where two actors exchange a fixed number of empty messages. These messages do not perform any computations so this benchmark strictly tests only the overhead of communication between actors. The results are presented in Figure 8.3.

Listing 8.4: Pong Actor code in ASCOOP

```
1  class PongActor extends TypedActor {
2
3    var pongCount = 0
4
5    def ping(sender: PingActor): Future[Void] = messageHandler {
6      sender.pong
7      pongCount += 1
8      done
9    }
10
11   def stop: Future[Void] = messageHandler {
12     println("Pong: pongs = " + pongCount)
13     done
14   }
15 }
```

From the results we see that up to 10^9 messages the overhead of both ASCOOP implementations is comparable to that of Akka and significantly better than Scala Actors. What we want to stress is that

even for a simple program like this, the size of the code is reduced from 150 to 110 lines of code (loc). For example we compare the implementation of the Pong Actor with Akka and ASCOOP Actors in Listings 8.4 and 8.5. We can observe that explicit message types and pattern matching (on line 7 of Listing 8.5) are not longer required, as they are replaced by method definitions for each type of message. This also gives the code more readability and modularity as an actor no longer has just a single method for processing all messages. Also in Akka each type of message needs to be defined separately in PingPongConfig and this is a requirement that is specific to each benchmark.

Listing 8.5: Pong Actor code in Akka

```
1  class PongActor extends AkkaActor[Message] {
2
3    private var pongCount: Int = 0
4
5    override def process(msg: PingPongConfig.Message) {
6      msg match {
7        case message: PingPongConfig.SendPingMessage =>
8          val sender = message.sender.asInstanceOf[ActorRef]
9          sender ! new PingPongConfig.SendPongMessage(self)
10         pongCount = pongCount + 1
11       case _: PingPongConfig.StopMessage =>
12         exit( )
13       case message =>
14         val ex = new IllegalArgumentException("Unsupported message: " + message)
15         ex.printStackTrace(System.err)
16 } } }
```

The implementation with cooperative scheduling also uses the code sequence presented previously in Listing 6.11 to ensure correct termination of the program. In Listings 8.6 and 8.7 we compare the termination of the benchmark in ASCOOP and Akka. This is part of the Ping Actor implementation.

Listing 8.6: Ping Actor code in ASCOOP

```
1  class PingActor(pongActor: PongActor) extends PingInterface {
2
3    var pingsLeft = 0
4    var t1 = 0L
5
6    override def start(iterations: Int) = messageHandler {
7      t1 = System.currentTimeMillis
8      pongActor.ping(this)
9      pingsLeft = iterations − 1
10
11     on (pingsLeft == 0) execute {
12       val t2 = System.currentTimeMillis
13       pongActor.stop
14       println(s"Finished in" +{t2−t1}+ " milliseconds")
15       ActorSystem.shutdown( )
16       done
17     }
18     done
19 } }
```

Listing 8.7: Ping Actor code in Akka

```
1  private class PingActor(count: Int, pong: ActorRef) extends AkkaActor[Message] {
2
3    private var pingsLeft: Int = count
4
5    override def process(msg: PingPongConfig.Message) {
6      msg match {
7        //...other case branches for prcessing messages
8        case _: PingPongConfig.SendPongMessage =>
9          if (pingsLeft > 0) {
10           self ! PingMessage.ONLY
11         } else {
12           pong ! StopMessage.ONLY
13           exit( )
14         }
15     }
16 }
```

The termination that uses ASCOOP Actors delegates the verification of the number of pings left to issue to the underlying scheduler (see Section 4.2.2) instead of verifying this state upon every execution of a ping as Akka Actors implementation shows(line 9 in Listing 8.7). As such the termination in ASCOOP is based on a particular actor state that the user can specify(line 11 of Listing 8.6) and needs not to be checked upon every ping.

8.2.2 Actors Overhead

The second benchmark calculates the nth number of the Fibonacci sequence. The sequence is defined with two starting numbers with the value of 1 and each number that follows is the sum of the two preceding ones. The implementation in the Savina benchmark calculates number n by creating two actors to calculate numbers $n-1$ and $n-2$ and then adding them together. These two actors in turn will each create two more actors (for calculating the sums of $n-2$ *and* $n-3$ and $n-3$ *and* $n-4$ respectively) and so on. This sequence is repeated until $n=1$ or $n=2$ which both have value 1. This benchmark is very suitable for testing the performance of a program with a large number of actors as it will create 2^n actors. We observe the results in Figure 8.4, which show the overhead for up to $2^3$3 actors.

The implementation with cooperative scheduling for this benchmark is shown in Listing 8.8. The completion of the Fibonacci number (starting on line 10) is scheduled to run only when the two previous numbers have been computed by futures ff1 and ff2. This example best illustrates how much code can be reduced using cooperative scheduling as a normal fire-and-forget implementation would have required two separate response messages before computing the number.

8.2.3 CPU and Memory benchmarks

The next two benchmarks are typical benchmarks for programming with actors that perform a lot of CPU and memory operations. The Sieve of Eratosthenes [O'N09] calculates prime numbers from the first n candidates [Tip13, Bok87, pri]. The implementation first creates two actors:one that generates candidates starting from 3 and skipping even numbers. The candidates are passed to a second actor that determines if they are prime based on the prime numbers it has already identified by checking if they are divisors of the current candidate. If a number has no divisors it is stored in an array inside the actor. To increase parallelism, each actor stores a fixed number of primes, which once exceeded will create a new actor

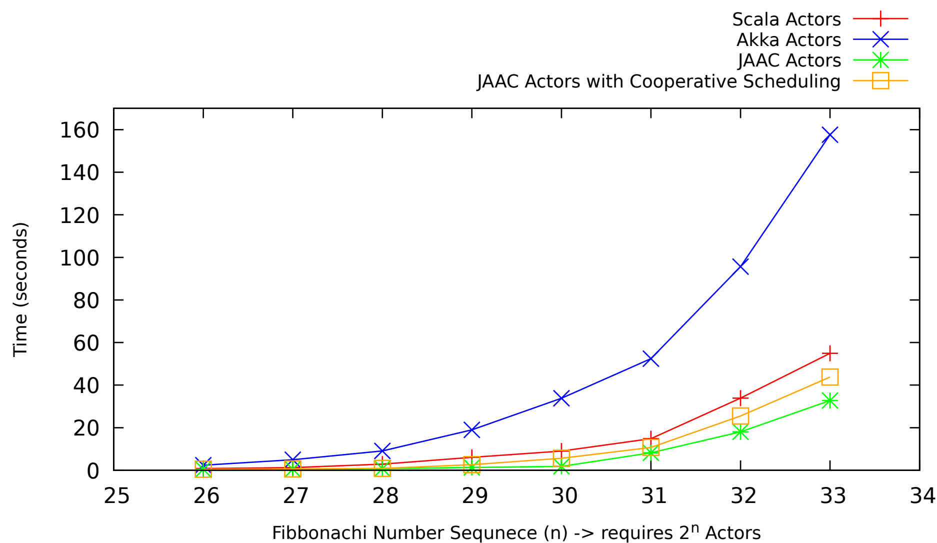

Figure 8.4: Performance figures Fibonacci Benchmark

Listing 8.8: Fibonacci Actor code in ASCOOP with Cooperative Scheduling

```
1   class FibActorAwaiting extends TypedActor{
2     def request(n: Int): Future[Int] = messageHandler {
3       if (n <= 2) {
4         Future.done(1)
5       }
6       else {
7         val ff1 = (new FibActorAwaiting).request(n − 1)
8         val ff2 = (new FibActorAwaiting).request(n − 2)
9         List(ff1, ff2) onSuccessAll {
10          ns =>
11          Future.done(ns.head + ns.tail.head)
12        }
13      }
14    }
15  }
```

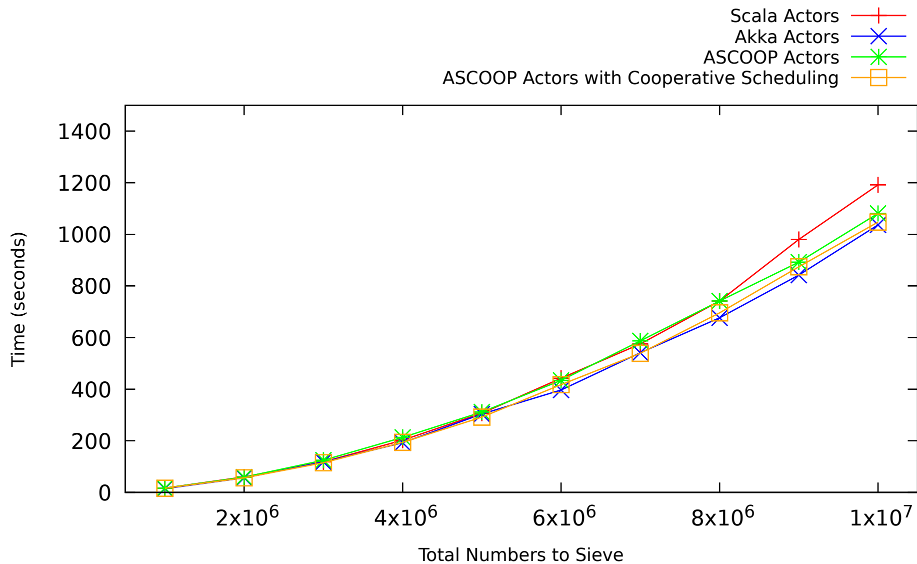

Figure 8.5: Performance figures Eratosthenes Sieve Benchmark

with the same behaviour. Once all candidates up to n have been processed, the total number of primes is aggregated from each array. Am important observation is that this is an actor-based implementation and comparison of the case study and it is not meant to be the fastest implementation of the sieve, as there exist many algorithms of segmenting the data to obtain better parallelism and scaling [yS05].

The results of this benchmark are shown in Figure 8.5, with each actor storing a fixed number of 2000 primes before an new actor is created. In this benchmark the number of messages and actors is significantly lower and we can see that there is no particular implementation that is constantly better in terms of performance. However when it comes to writing the actual application, once again we observe that the size of the code compared to the Savina benchmarks in Scala and Akka has been reduced from 160 loc to 110 loc.

The NQueens benchmark is reused because the master-slave model relies heavily on the cooperative scheduling properties. 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, a threshold is imposed where the worker has to sequentially 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 11 to 15 with a fixed number of 4 workers. The results are shown in Figure 8.6. It is important to observe that as the board size increases, the number of solutions grows from 2680 - 2,279,184.

These results also have very small differences in performance and it is very important to know that the ASCOOP model for this benchmark is 150 lines of code in total, while the benchmark written directly using the Scala or Akka libraries is around 400 lines of code. Furthermore the ASCOOP implementation with cooperative scheduling does not require that the number of solutions to be found is known a priori, as the master actor can simply wait on the future that generated all of the searches (line 9 of Listing 8.9) and generate a sub-task to terminate the application once it completes (line 10).

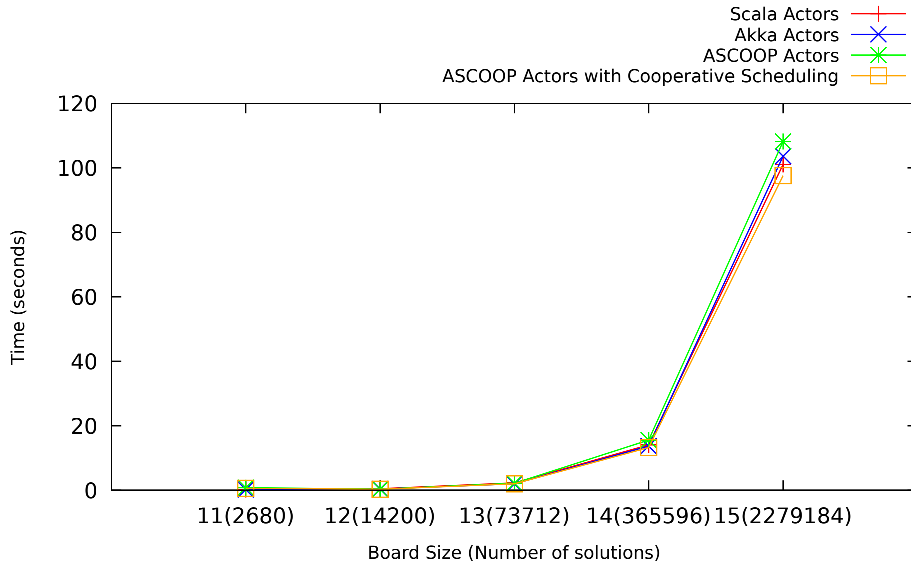

Figure 8.6: Performance figures Nqueens Benchmark

Listing 8.9: NQueens Master Methods for Initialization and Completion

```
1   def sendWork(list: Array[Int], depth: Int, priorities: Int):
2     Future[Iterable[Array[Int]]] = messageHandler {
3    val worker = new Worker(threshold, size)
4    worker.nqueensKernelPar(list, depth, priorities)
5   }
6
7   def init : Future[Void] = messageHandler {
8    val inArray: Array[Int] = new Array[Int](0)
9    val f = this.sendWork(inArray, 0, priorities)
10   f onSuccess(result => {
11     println(s"Found␣{result.size}␣solutions")
12    }
13  }
```

8.3 Map Reduce

The solution provided in this thesis is tailored towards the efficient integration of actors, futures and coroutines on a single-machine. As we have seen in Chapter 5, there are significant challenges to maintaining this efficiency in a distributed setting. This section covers a comparative analysis between the research and solution proposed in this thesis and the research conducted in [Roc16] where ABS is extended with distributed support.

Through the use of a representative HPC application: pattern matching of a DNA sequence programmed in the MapReduce model [DG08] we compare the performance of the two models deployed on an increasing number of nodes. The case study is computation-intensive and yet does no have a lot of active object communication. On one hand, ProActive relies on physical threads for active objects and

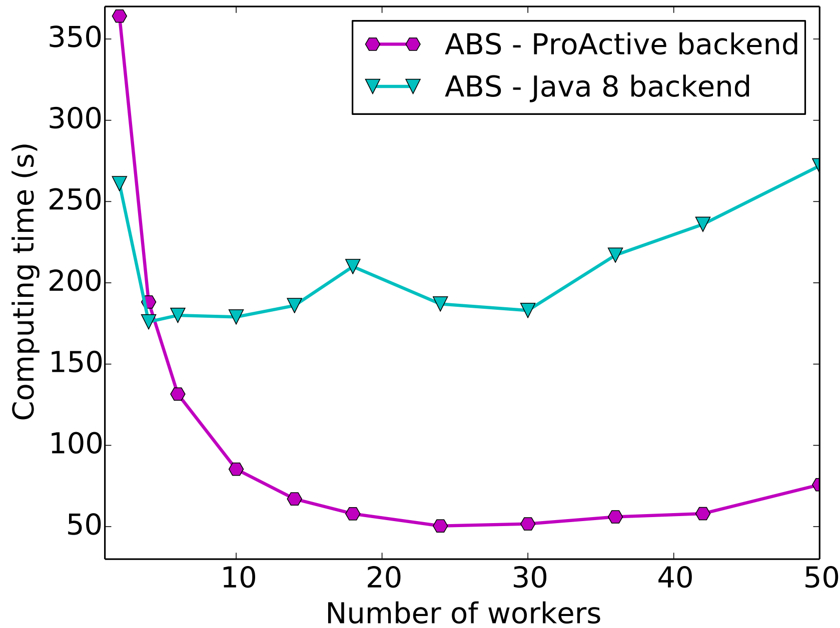

Figure 8.7: Execution time of DNA-matching ABS application

data copying occurs in each communication, thus ProActive is better suited for computation-intensive scenarios than for communication-intensive ones. On the other hand, the JAAC backend is perfectly suited when the number of nodes does not exceed the machine's cores as the premise of the `LocalActor` implementation is to communicate with other actors at no additional cost. We compared performance of the code generated with the ProActive backend for ABS run in distributed mode, the code generated with the Java 8 backend of ABS run on a single machine, and native code written manually in ProActive for the same algorithm.

The application creates `Map` instances (workers) each in their own COG. The search pattern is of 250 bytes inside a database of 5 MB. Each worker queries and processes the maximum matching sequence of a chunk while a reducer aggregates and prints out the global maximum matching sequence. When deploying with ProActive for the best mapping to cores, each machine is instantiated with two workers (as it has two dual-core CPUs). One of the biggest drawbacks of ABS was its standard library data structures, so in both generated programs, we manually replaced the translation of functional ABS types (integers, booleans, lists, maps) with standard Java types to avoid search spaces of very high complexities (i.e a search in an ABS functional hash map is $O(n)$).

The hardware setup for this case study was with machines that have 2 dual core CPUs at 2.6GHz, and 8 GB of RAM[1].

Fig. 8.3 shows the execution times of both ABS backends for the application ranging from 2 to 50 workers, and 1 to 25 physical machines with ProActive. The execution times of the ProActive backend are sharply decreasing for the first few added machines and then decrease at a slower rate. The first instance added in the JAAC backend also improves significantly the performance of the program and the JAAC backend performs better with one or two workers. Thanks to the efficient thread management of the Java

[1] We use a cluster of Grid5000 platform [BCC+06]: `http://grid5000.fr`

8 backend, the performance stays stable until 30 workers. With a high number of instances, the degree of parallelism becomes harmful. In contrast, increasing the degree of parallelism for the ProActive backend results in a linear speedup, because it balances the load between machines and benefits from distribution.

8.4 Cache coherency

This section covers a case study where ABS is used to simulate the memory model of a multi-core architecture. It is a model of the entire multi-core system that includes computing cores, a scheduler, the bus and the different memory levels that include cache levels and the main memory. The simulation models how data is loaded and written into main memory and how it passes through the caches when different operations are executed. For the research conducted in this thesis the operations abstract from the actual data by working only with address requests like loading an address for reading or writing purposes. Operations are grouped together in sets to represent tasks that are to be executed on a core. It is important to understand that the entire system is a model that is independent from the actual system it runs on. All the operations corresponding to loading addresses, looking them up in the cache levels, evicting addresses to create space in a cache or flushing addresses to main memory for maintaining persistent data are all programmed in the ABS source code. As such at the end of executing the model, the user can observe the state of the system such as the data stored in the cache or the number of hits and misses when an address is required.

This case study illustrates a powerful usage of the resource modeling extension of ABS. Each core is modeled as an ABS object that can executed its assigned tasks in parallel to other cores. The simulated core scheduler assigns tasks to the available cores in the system in a round-robin manner. Using the resource provisioning model, core objects can each be assigned a Deployment Component with a finite amount of resources (Listing 8.10), thus ensuring a load-balancing factor between the cores. On lines 6- 10 the model determines whether a new Deployment Component (referenced by `dc`) is instantiated with an infinite (0) or finite amount of resources. Then upon setup of the system's cores (lines 14 or 18), these are assigned the newly created component with finite resources if this was the configured setup.

Listing 8.10: Assigning finite resources to a new core object

```
1  class Config(Maybe<Rat> resources) implements IConfig {
2
3   Unit runConfig(Int nCores_, ...){
4    / * other configuration setups * /
5    while (nCores > 0) {
6     Rat rc = case resources {
7      Nothing => 0;
8      Just(x) => x;
9     };
10    DeploymentComponent dc = new DeploymentComponent(name, map[Pair(Speed,rc)]);
11    / * ... * /
12    case resources {
13     Nothing => {
14      ICore c = new Core(name,s,l1);
15      //core setup Deployment Component with infinite resources
16     }
17     _ => {
18      [DC: dc] ICore c = new Core(name,s,l1);
19      //core setup on a Deployment Component with finite resources
20 } } } } }
```

Even more powerful is the fact the by giving the Deployment Components different speeds (as in line 10 of the previous listing) or the tasks different costs by using annotations (line 4 Listing 8.11), the model can simulate cores that execute faster or tasks than take longer to run (thus the reason for abstracting from actual data as far as execution speed is concerned). A simple algorithm of assigning equal resources to all Deployment Components on which `Core` objects are instantiated and giving all tasks the same Cost will simulate complete parallelism in the system: a core will execute a second task only after all cores have executed the first task. Together with a round-robin scheduler, the model also simulates perfect load-balancing between all the cores.

Listing 8.11: Assigning taks execution a particular cost

```
1  class Core( ... ) implements ICore {
2
3    Unit run( ) {
4     [Cost:1] skip;
5     case currentTask {
6       //execution of current task assigned to the core
7     }
8    }
9  }
```

The Scala backend was compared with the Erlang backend for ABS in this case study. This backend is the most stable and well-maintained backend of ABS and it offers full support for all of ABS features including error handling and custom-defined schedulers, however it does use the thread-abstraction approach for asynchronous communication and cooperative scheduling. In Listing 8.8 the performance times for simulating a system with 2 cores and three levels of cache are presented. In this first simulation (the red and green plots) there is no usage of resource models assigned to ABS `Core` objects and as such these can work at non-deterministic speeds resulting in an uneven number of tasks executed between the cores. In the second simulation (the orange and blue plots) the ABS model uses Deployment Components as explained in Listing 8.10 to obtain a load balance of the tasks executed on the two `core` objects. There are two important observations to withdraw from these results:

1. The tasks spawning approach allows the ABS model to run much faster as the number of tasks simulated in the system grows
2. The simulation of the load balancing and resources creates a small overhead in order to provide a much more realistic version of the memory system.

8.5 German Railway Example

The final case study involves testing and validating support for the time model of ABS in the runtime library. The case study simulates physical behaviour of trains and events occur happen on the railway track and its components such as signals, stations and switches. For a detailed formal description of the model we refer to [HM16, KH16, KH17]. The main focus of using this example in the research is its usage of the time model of ABS. Throughout the source model several events occur after a fixed amount of time units passes and in turn trains have to react to those events in correctly establish capacity and safety properties.

The railway is modeled as an oriented graph where nodes represent fixed points of information flow (PIF) and are connected by edges that represent the tracks between them. In short PIFs are structural elements where trains can receive or send information such as a signal or a track clearance detection device or a gradient change in the track. Both tracks and nodes are represented as ABS objects as in

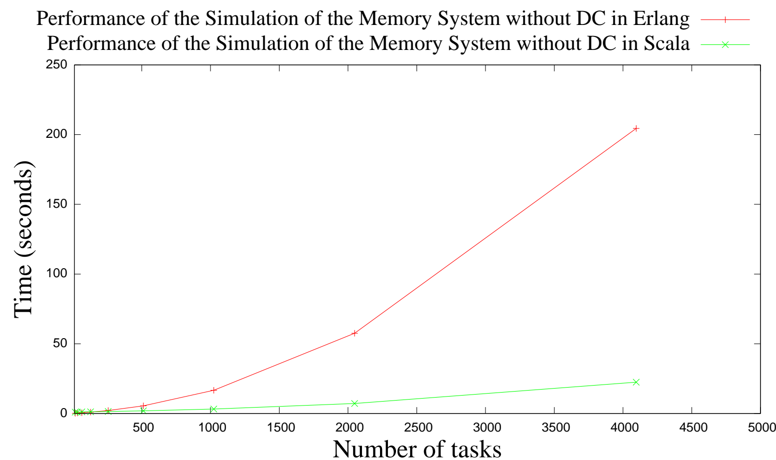

Figure 8.8: Results for the simulation of the memory system

Listing 8.12. A `NodeImpl` object contains a list of connecting **Edges** that make up the graph and represent the tracks on the railway. The `EdgeImpl` class represents tracks with a particular length `l` and the two nodes it connects(`frNode` and `toNode`). Multiple lines between the same two nodes may be represented by edges.

Listing 8.12: Graph and Edge Classes

```
1  class NodeImpl(Int x, Int y, String name) implements Node {
2
3    List<Edge> edges = Nil;
4
5  }
6  class EdgeImpl(..., Node frNode, Node toNode, Int l, String name) implements Edge {
7
8  }
```

Trains are also ABS objects are represented by the their front and back position relative to the most recent node they passed. A train has a series of attributes to describe its behavior such as speed, accelerations state and length as well as fixed attributes such as maximum acceleration and brake retardation. An outline of the class is presented in Listing 8.13. Trains are modeled to drive on simulation events. At every PIF where the train is active, it computes its next event and the time until this event must be processed. This is where the time-model of ABS is used by calling the `advance` method whenever an event is computed to simulate the amount of time that passes.

Listing 8.13: Train Class

```
1  class TrainImpl(App app, String name, Int length) implements Train {
2    //attributes that define a train's state and behaviour
3
4    Unit advance(Rat r){
5      await duration(round(r), round(r));
6    }
7  }
```

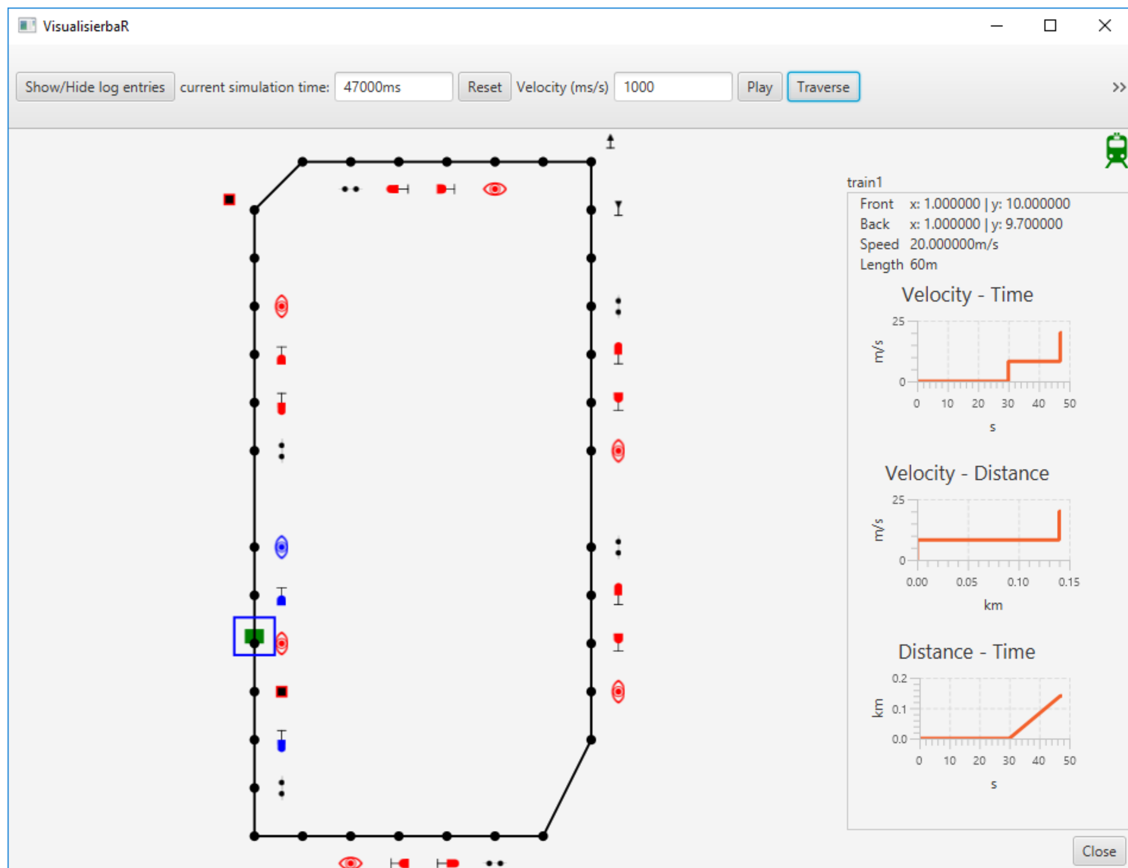

Figure 8.9: The railway model through the visualization tool

The model was run with the ABS Scala backend with time support and a very important result that was obtained is that the output of the model is deterministic and the simulation always yields that same result given the same input. The output can then be processed using a visualization tool developed in [KS19]. Figure 8.9 presents a snapshot of the model where the user can observe and monitor the behaviour of trains by advancing time in the simulation model.