Run, Actor, Run

Towards Cross-Actor Language Benchmarking

Sebastian Blessing
Imperial College London
United Kingdom
sebastian.blessing12@imperial.ac.uk

Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

Sophia Drossopoulou
Imperial College London
United Kingdom
s.drossopoulou@imperial.ac.uk

Tobias Wrigstad
Uppsala University
Sweden
tobias.wrigstad@it.uu.se

Abstract

The actor paradigm supports the natural expression of concurrency. It has inspired the development of several actor-based languages, whose adoption depends, to a large extent, on the *runtime characteristics* (*i.e.,* the performance and scaling behaviour) of programs written in these languages.

This paper investigates the relative runtime characteristics of Akka, CAF and Pony, based on the Savina benchmarks. We observe that the scaling of many of the Savina benchmarks does not reflect their categorization (into essentially sequential, concurrent and parallel), that many programs have similar runtime characteristics, and that their runtime behaviour may drastically change nature (*e.g.,* go from essentially sequential to parallel) by tweaking some parameters.

These observations lead to our proposal of a single benchmark program which we designed so that through tweaking of some knobs (we hope) we can simulate most of the programs of the Savina suite.

CCS Concepts • **Computing methodologies** → *Distributed programming languages*; • **General and reference** → *Evaluation*; *Performance*.

Keywords actor programming; benchmarks

ACM Reference Format:
Sebastian Blessing, Kiko Fernandez-Reyes, Albert Mingkun Yang, Sophia Drossopoulou, and Tobias Wrigstad. 2019. Run, Actor, Run: Towards Cross-Actor Language Benchmarking. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358499.3361224

ACM ISBN 978-1-4503-6982-4/19/10…$15.00
https://doi.org/10.1145/3358499.3361224

1 Introduction

Modern computers are all multicore or manycore machines that need software written with concurrency and parallelism in mind to utilise their full power. This has sparked a renewed interest in the actor paradigm, which aims to express concurrency in a natural way, and provide a high-level abstraction that, at least in theory, exploits the inherent concurrency of a program's design. This has inspired the development of several actor languages [6, 11–13, 16, 19, 24, 32, 36, 38].

For wide adoption of actor languages, the *runtime characteristics*—performance and scaling—are important considerations. The benchmark suite Savina [30], which aims to provide a collection of programs which are "diverse, realistic and compute intensive (rather than I/O intensive)", is the current de facto benchmark for actor languages. Benchmarking is difficult; language and runtime features may interact in ways which, if not carefully taken into account, may lead to a misinterpretation of the results.

To deepen the understanding of Savina and corroborate that a program's scaling depends on the *inherent concurrency* in its design, we compare the performance of the Savina programs across three actor languages on two architectures, scaling from 1 to 112 cores.

The contributions of the paper are as follows:

- Description of the aims and (common) pitfalls of cross-language benchmarking (§ 2).
- Reproduction study of the Savina paper with explanations of runtime characteristics (§ 4 and § 5).
- An observation that many Savina benchmarks have similar runtime characteristics, and that tuning parameters in these programs drastically affects their characteristics so that they become indistinguishable from those of other programs in the suite (§ 5).

– A proposal for a single benchmark program that (we believe) can simulate most of the programs of the Savina benchmark suite (§6).

§3 gives an overview of the three languages in our study, and §4 introduces the Savina benchmarks. §7 concludes.

2 Cross-Language Benchmarking: Pitfalls

A central question for the adoption of a concurrent programming language is its performance, and how well it scales when we increase, or decrease, the number of cores.

This section briefly touches on some of the problems when comparing performance across multiple languages that introduce differences in the benchmark programs.

2.1 Pitfalls of Cross-Language Benchmarking

Cross-language benchmarking is difficult. Considering simple programs (aka micro benchmarking), that typically stress a single or a few aspects, like message passing overhead or backpressure handling, isolating aspects is a delicate matter as many seemingly innocuous operations performed *e.g.,* to simulate "actual work" can easily become dominating performance factors. For example, many actor micro benchmark programs generate random numbers: even with actor-local generators initialised with constant seeds, generators in different languages may favour one implementation or suppress a pain point in another. Furthermore, random number generators may perform differently which may obscure the actual benchmark result and the performance of the operations it is used in combination with.

In the past, we have seen results in micro benchmarks skewed (or obscured) by *e.g.,*: random number generation (such as *different numbers* generated across platforms; *varying performance* of generators across platforms—leading to accidentally comparing performance of random number generators rather than the actual metric of interest); string libraries (such as the same frequent operation is $O(1)$ on one platform and $O(n)$ on another); accidentally measuring performance of "irrelevant" libraries (such as regular expressions); creation of garbage objects during warm-up that penalises subsequent iterations due to *e.g.,* worse locality or additional GC cycles.

These problems largely go away in *macro* benchmarking—comparing the performance of entire applications without synthesized behaviour. But macro benchmarks make it harder to isolate the effects of single aspects.

Measuring scalability is delicate regardless of micro/macro. When using only a fraction of a large machine to simulate performance on a smaller machine, one typically wants cores on the same NUMA node to avoid slowdowns due to remote memory access, or suffer from unfairness when a lock is suddenly closer to one core than another, etc. This is relatively straightforward in compiled systems without a runtime component, but for full-blown VMs with background compilation threads, GC threads, and many other runtime management entities—what is the *right thing to do* is a lot less clear. Pin all, pin no, or pin only non-VM threads? Ultimately, what is a "fair" comparison is not obvious.

There are multiple other reasons why fairness is difficult. For example, Akka allows users to supply custom scheduling behaviours for actors which ultimately hurts its scheduling performance because it limits the extent to which things can be optimised as fewer facts are set in stone. This (at least in theory) makes it easier to "beat" the default Akka scheduling behaviour which, unless one goes through the trouble of defining benchmarks where custom scheduling makes sense, paints Akka in less favourable light. A similar argument can be made about Erlang favouring fairness over performance.

2.2 The Objective and The Rules of the Game

The above discussion brings us to the point of what we call "the rules of the game". For a micro benchmark to be useful, it needs a *clearly* stated *objective*: what it intends to show (*e.g.,* how a program that is under-saturated with respect to parallel units of behaviour is penalised by actor scheduling) as well as the *rules for meeting the objective*—what "tricks" are allowed in implementations. For example, in the context of actor programs, a rule could be not breaking actor isolation. As §5 will discuss further, some of the initial implementations of Savina programs break isolation to avoid message passing overhead for coordination. Is the *ability* to do so a strength, or is a language's ability to enforce actor isolation a superior quality?

In this example, the question may be formulated as "when is a program an actor program?" As there is no universally accepted truth, we believe each benchmark should come with its own set of rules, *and changing the rules by definition means the creation of a different benchmark.*

3 Actor Languages

Actors are concurrent objects interacting via asynchronous point-to-point messages [2, 26]. In the initial model, every actor maintains a private heap inaccessible to other actors [26]. Hence, the state of an actor can only be manipulated through messages. Messages are sent by adding them to the mailbox of the receiving actor (ideally the only point of synchronisation in a program) after which the sender can continue without waiting for an explicit reply. Actors process incoming messages in the order they arrive or out of order (called *selective receive*). Actors are thus sequential, but actor *programs* are inherently concurrent. Consequently, actor-based languages are concurrent-by-default [2, 26].

3.1 Akka

Akka [24, 38] is a library-based implementation of the actor model on the JVM with widespread adoption. Akka supports

Table 1. Overview of Akka, CAF, and Pony.

Features	Akka	CAF	Pony
Model	Actor	Actor	Actor
Concurrent	Yes	Yes	Yes
Distributed	Yes	Yes	No
Data Races	Possible	Possible	Impossible *(Type system)*
Fault Tolerance	Supervisor	Supervisor	None
GC	Any JVM collector	Automatic reference counting	Orca [17]
Runtime	JVM	Compiled	Compiled

distributed programming through a configuration deployment mechanism that decouples the program logic from the network (deployment) architecture. Unlike Pony, Akka does not enforce actor isolation, meaning data-races are possible if a programmer is not careful. Fault tolerance is achieved through a supervisor hierarchy, *i.e.,* the common *let-it-crash* semantics of actor-based languages.

3.2 C++ Actor Framework (CAF)

The C++ Actor Framework (CAF) is a C++ library for building concurrent and distributed, actor-based programs [14]. CAF has dynamic and statically typed actors in a high-performant implementation that is data-race free and fault tolerant. Dynamic actors accept any message and dispatch dynamically while interaction with statically typed actors is type-checked. CAF does not enforce actor isolation, but allows some middle-ground through *copy-on-write* semantics: immutable data can be shared by reference between actors and copied locally when an actor creates a *mutable* reference to it. Altough this appears to be an optimization compared to full isolation, copying potentially immutable data is important for GC, see *e.g.,* Erlang [6]. As Akka, CAF decouples the actor logic from its deployment; fault tolerance is achieved by using a supervisor hierarchy.

3.3 Pony

Pony is an actor language for building concurrent, parallel, type-safe, and high-performant systems [15–17, 22]. Pony enforces actor isolation through a capability-based type system, which also provides guarantees to the runtime that are used to implement a zero-copying policy of shared data.

3.4 Similarities and Differences

In this section we highlight the main similarities and differences between Akka, CAF, and Pony (summary in Table 1). The three languages are concurrent and parallel by default. Akka and CAF support distributed programming, with a supervisor hierarchy to provide fault tolerance. Support for distributed programming in Pony is under development [8].

Akka and CAF leave actor isolation to the programmer. Pony guarantees isolation statically through its type system.

Akka runs on the JVM and thus may use whatever garbage collector the JVM supports (starting from JDK 9, the default is G1 [20]); which typically introduces some stop-the-world pauses and does not leverage actor isolation. CAF implements its own garbage collection using reference counting [1]; Pony has its own runtime and uses its own concurrent and parallel garbage collector [17] including actor collection [15].

3.5 Cross-Language Benchmarking—The Rules of the Game revisited

Despite Akka, Pony and CAF being all actor languages, they are implemented quite differently. This stresses the importance of the rules to guide benchmark implementation in any of these (or future) languages. Using only constructs that are present in all languages compared does not scale well to future languages, may not be possible, and also hurts the relevance of the benchmarks. Instead we propose a set of rules for what *not to do* that we believe any actor-based language can easily adhere to (even if not always idiomatic).

R0: No Breaking of The Actor Model Actor isolation is an intrinsic property of the actor paradigm. That is, state changes must be caused by sending messages. Sharing mutable data or using thread-safe data structures such as atomic integers or maps and containers is forbidden.

R1: No Poison Pills In our experience, actor programs tend to be long-running with potentially no intention to ever terminate. Thus, including termination in measurements only serves to add jitter. Consequently, we consider a benchmark "done" when the desired final state is reached. This means that cleaning up (say final GC'ing or poisoning live actors to terminate quickly) is outside of measurements.

R2: No Selective Receive Programs should not require the causal nature of actor systems to be broken. Available mechanisms behave differently with varying impact on actor responsiveness (*e.g.,*, Akka's *stash* vs. *receive*), leading to noise when reporting benchmark results. Moreover, not all languages and frameworks allow for pattern matching on message queues, which leaves too much room for implementing a custom (language-level based) stash and unstash mechanism posing an additional emphasis on object allocation.

R3: No Randomness Relying on randomness hampers reproducibility, even with constant seeds, and introduces reliance of random number generators with associated performance noise. In the spirit of [18], the input should be externally supplied and lead to deterministic program behaviour.

R4: No Focus on Highly Optimized Libraries The purpose of cross-language benchmarking is comparing *languages, frameworks and runtime systems*, not libraries such as regular expressions (*e.g., pcre2*) or numerical computations (*e.g., libgmp*). A set of primitive data types and standard collections (array, list, maps and sets) as well as networking (TCP/UDP) should be enough and are most likely available in any standard library.

R5: No Focus on Embarrassingly Parallel Algorithms
In our experience, embarassingly parallel data processing on large shared mutable state is rare in the context of actors, and do not stress the asynchronous behaviour for which actor languages were developed.

4 Benchmarking Savina

Savina consists of 30 programs, grouped into three categories: *Micro*, *Concurrency* and *Parallelism*. We took the benchmarks as already implemented for Akka and CAF from [31, 37], and implemented their counterpart in Pony. We left the Akka and CAF implementation unchanged, and developed the Pony versions from scratch, in accordance with Rules R0–R5. We omitted *Successive Over-Relaxation*, as the Akka implementation appeared to be infinitely running. Moreover, a few Pony implementations are missing (*apsp,astar*, *bitonicsort*, *piprecision*, *facloc*, *NQueenk* and *Uct*) due to the lack of standard library support (such as a comparable *big decimal* implementation required for *piprecision*). The code for all the programs in all three programming languages can be found at [9]. It shall be noted at this point, that most of the savina programs violate rule *R3* by making heavy and inconsistent use of random number generators [31]. Some use *java.util.Random* while others use a locally defined congruential generator.

We ran the benchmarks on a medium-sized machine (an Intel Xeon(R) CPU E5-2690 v3 2.6 GHz, with 2 sockets, and 24 cores with hyperthreading and 256 GB main memory), as well as a larger machine (an Intel Xeon Platinum 8180M 2.5 GHz, with 4 sockets, and 112 physical cores with hyperthreading and 3TB main memory). We ran on SuSE Linux Enterprise 15 HPC with Linux Kernel 4.12.14-150.14.

We measure time from a start event until the specific program event which signals the finished computation. We run 12 warm-up iterations on the JVM, and rule out significant outliers before reporting statistics data. We implement a generic benchmark runner to automate the benchmarking process, including plotting the statistics. The runner detects the hardware resources including NUMA arrangement, physical cores and hyperthreads, and disables/enables cores (using *numactl* or *CPU offlining*) until all languages and iterations have been executed on all available core counts. CPU disabling is NUMA-aware to mitigate the risk of stalling a thread due to far-distance memory accesses.

Runtimes are the median taken from all iterations per core, in milliseconds. We report the results for the larger machine; those for the smaller machine can be found at [10].

4.1 Benchmarking Results

The Savina benchmarks are grouped into three categories: *Essentially Sequential* (called *Micro* by [30]), *Concurrency* and *Parallelism*. In our opinion, all 30 Savina benchmarks are *micro benchmarks*, as we used the term in section 2. The Savina authors use the term *Micro* in a different sense: they call *Micro* those benchmarks which exercise a single implementation aspect, and which do not benefit from concurrency. From now on, we stick to *their* definition.

We now describe the three Savina categories, their programs, and the results we obtained. We show the plots in Figures 1 to 3, where the x-axis is the number of cores (1–112, no hyperthreading), and the y-axis is time in milliseconds.

4.1.1 Micro Benchmarks

The *Micro* benchmarks are small, simple programs whose objective is to test specific components of an actor runtime.*big* (from [7]) measures the performance of many-to-many message passing as well as mailbox contention. *count* measures message passing overhead by sending a large number of messages from one actor to another. *chameneos* is similar to *big*, but all message exchanges go through a single broker and thus stresses mailbox contention on the disproportionally popular broker actor. *fib* measures creation and destruction of a large number of actors of varying lifetime by recursive fibonacci number calculation, spawning actors in-place of recursive calls. *fjcreate* and *fjthrput* measure actor creation and destruction, and messaging throughput respectively using parallel fork–join calculations. *Ping Pong* and *Thread Ring* measure the scheduling behaviour, context switching overhead, and message passing overhead of sequential programs implementing a logical thread of control in form of a message being passed around a ring topology.

Measurements and Observations Figure 1 shows the results of the eight microbenchmarks. None of the benchmarks show a considerable speed-up with increasing number of cores. This is not surprising, as they are *essentially sequential*: In *Ping Pong* and *Thread Ring*, no more than one actor can be active at a time. In *count*, at most two actors are active at a time. In *chameneos* the broker is the bottleneck. Even though in *fib* an exponentially growing number of actors is active at a time, each actor's work is so small that the parallelism is dwarved by actor-creation and communication overhead.

We notice a considerable speed-up between 1 and 2 cores in CAF, even in benchmarks which exhibit no concurrency (or parallelism), *e.g.,* Thread Ring or Ping Pong. We believe this is because CAF oversubscribes to scheduler threads on small core numbers. In some cases, we notice a small slowdown in Pony and Akka going from 1 to 2 cores; we believe this is due to mailbox contention.

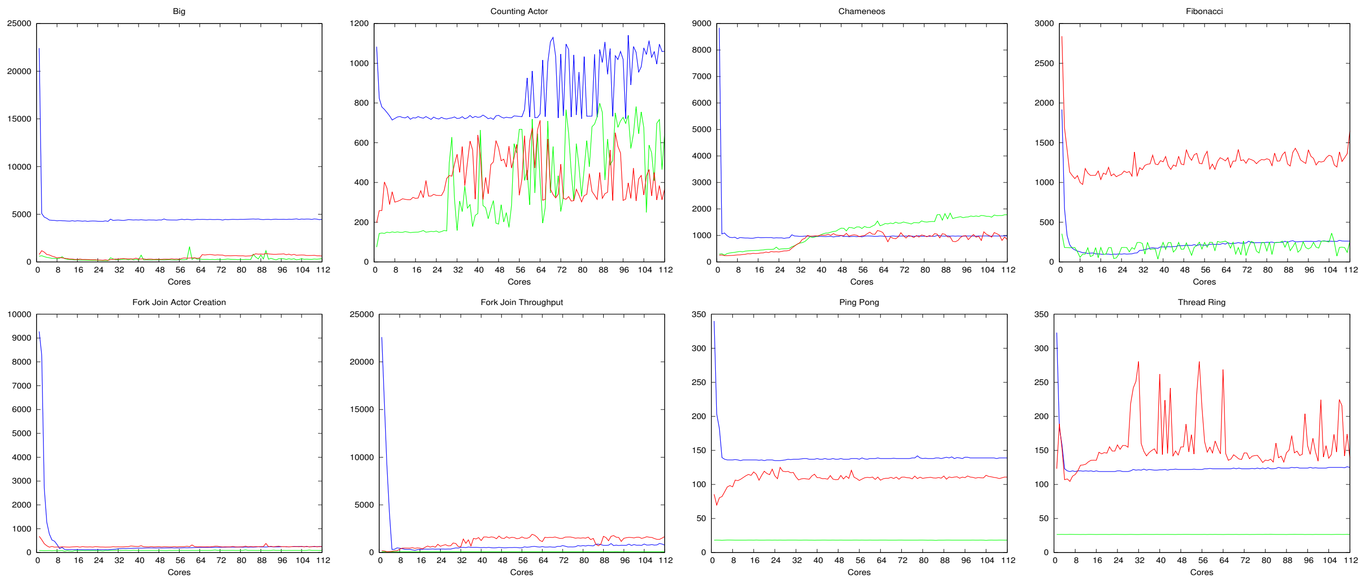


Figure 1. Microbenchmark performance and scaling plots. Legend: Akka 2.3.2: 🟥, CAF 0.16.3: 🟦, Pony 0.28.0: 🟩.

4.1.2 Concurrency Benchmarks

The objective of the concurrency benchmarks is to measure contention and coordination of nondeterministic communication between multiple actors. *Banking* and *Logmap* both measure cost of synchronising request–response patterns with interfering and non-interfering transactions respectively. Both violate rule *R2*, by stashing and unstashing messages to defer the receipt of subsequent requests until a response for previous request has been received. *philosophers* (from [28]) measure contention on shared resources governed by a central arbitrator actor. The implementation provided at [31] violates *R0* by using shared atomics for reporting results. *Bounded Buffer* is an instance of a multi-process synchronisation problem [27, 28]. It measures the overhead of (effectively) starting and stopping actors depending on increasing supply and demand. *concdict* and *concsll* both measure the performance of multiple readers–single writer concurrency (which the actor model does not support). The purpose of *cigsmok* (from [34]) is unclear; its actual challenge is to avoid deadlocks. *barber* (from [21]) measures contention on a single bottleneck, while avoiding starvation. Just as for *philosopher*, rule *R0* is violated by the implementation provided at [31].

Measurements and Observations Figure 2 shows the results of the eight concurrency benchmarks. With the exception of *bndbuffer*, the concurrency benchmarks do not speedup with increasing number of cores. This might be initially surprising, as in these benchmarks there is (theoretically) a large number of actors which can be active concurrently. However, on closer inspection, we observe that these concurrent actors tend to do very little work when active. For example, in the *philosophers*, when a philosopher is given the forks she relinquishes them immediately, and notifies the arbitrator that she is hungry. Thus the arbitrator immediately becomes a bottleneck, and philosophers mostly wait.

Increasing the amount of work each philosopher does upon receiving two forks also increases the *possible parallelism*, as shown by the pragmatically placed Figure 3 (a). The same "trick" can be applied successfully to several other benchmarks, for example, *chameneos*, *fib* and *barber*.

Combining the results of all concurrency benchmarks, we obtain the plot in Figure 3 (b).

4.1.3 Parallelism Benchmarks

The parallelism benchmarks aim to take full advantage of parallel hardware architectures by task decomposition. The challenge is to translate this decomposition efficiently into actor-style computations. There are different kinds of parallelism, such as pipelines, phased computations, divide-and-conquer, master–worker schemes as well as graph and tree navigation scenarios.

apsp is a phased computation where actors join a barrier (implemented via messages) at the end of each iteration, while mutating a matrix. Each actor owns a block of the matrix and is responsible for updating the state of its block and inform its neighbor about state changes.

astar (from [25]) is a graph traversal benchmark searching for the path with the lowest expected total cost/distance. The implementation as defined in [31] breaks actor isolation (and thus rule *R0*) by sharing and mutating a central graph data structure [31]. Implementing this benchmark whilst preserving parallelism faithfully with actor isolation is nontrivial. Our Akka implementation from [30] inherits artefacts

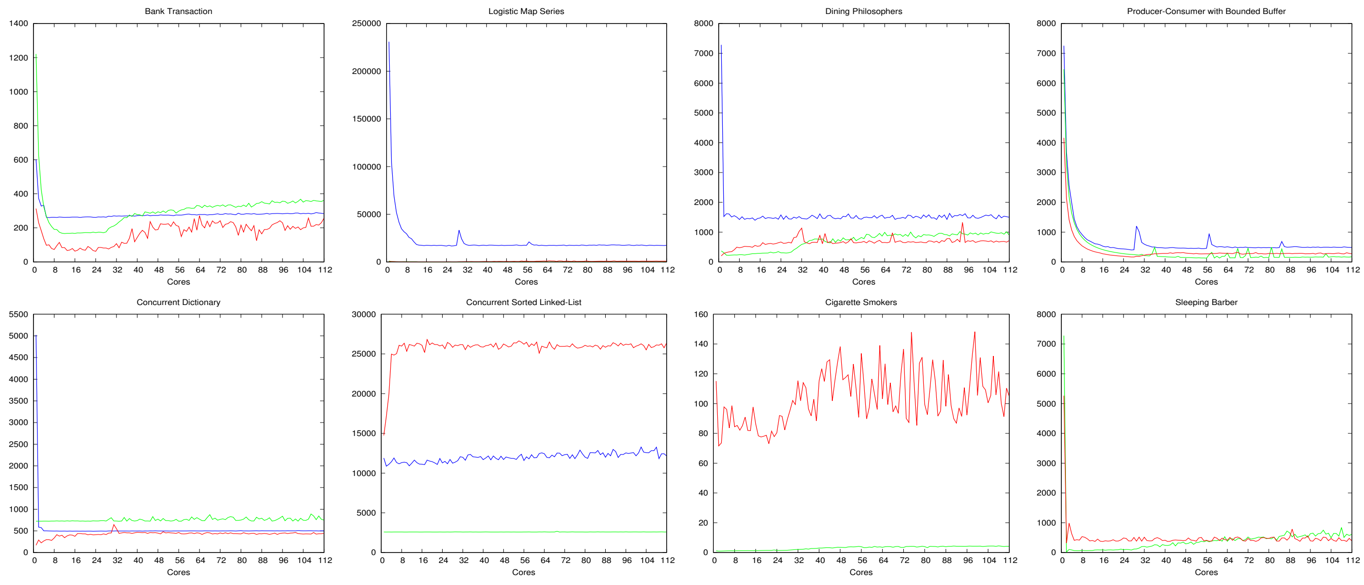


Figure 2. Concurrency performance and scaling plots. Legend: Akka 2.3.2: 🟥, CAF 0.16.3: 🟦, Pony 0.28.0: 🟩.

due to the use of a priority queue in the original implementation. Prioritising inter-actor messages is not part of the actor model, which makes achieving good performance difficult.

filterbank is a stream-pipelining implementation with join semantics. The benchmark stresses the speed of message delivery and the scheduling of actors onto worker threads.

NQueens measures the overhead of speculative parallelism with a single master and several worker actors each attempting to find a valid solution. Prioritising solutions that place more queens on a board makes this benchmark difficult to implement in the actor model.

piprecision exhibits a master–worker pattern with dynamic load-balancing computing the value of π in a way that requires an implementation of a Big Decimal datatype. Similar to *philosopher* and *barber*, this benchmark violates rule *R0*.

trapezoid is master-worker style implementation to approximate the integral over a given function.

recmatmul uses static load-balancing performing an embarassingly parallel recursive matrix multiplication. The nature of this benchmark is violating rule *R5* and, by sharing the central mutable result matrix, does not adhere to rule *R0*. In Pony, an additional is required to collect the values for the result matrix. *sieve* (from [29]) implements a dynamic pipeline. Each time a pipeline overflows, a new actor is created and attached to the pipeline meaning messages have a varying "hop length". *uct* exhibits non-uniform load in a tree-traversal benchmark stressing dynamic load-balancing of the scheduler.

facloc (from [5]) uses speculative parallelism on a dynamically generated tree.

Finally, *quicksort*, *bitonicsort* and *radixsort* are divide-and-conquer based sorting algorithms.

Measurements and Observations Figure 3 shows the results of the parallel benchmarks. (The last three subfigures were placed in the same figure to save space.) Several of these benchmarks show a considerable speed-up with increasing number of cores, but this improvement tends to peter off at between 8 and 32 cores.

The matrix multiplication program is interesting—the typical way to implement this in a threaded system is by sharing read-only input matrices and a write-only result matrix across all workers. The latter is safe because workers will write to disjoint locations in the matrix. As Pony's type system enforces actor isolation, this strategy is forbidden (although see [3] for an extension to a similar capability system that handles precisely this scenario). Both the Akka and CAF programs pragmatically break actor isolation and share the result matrix.

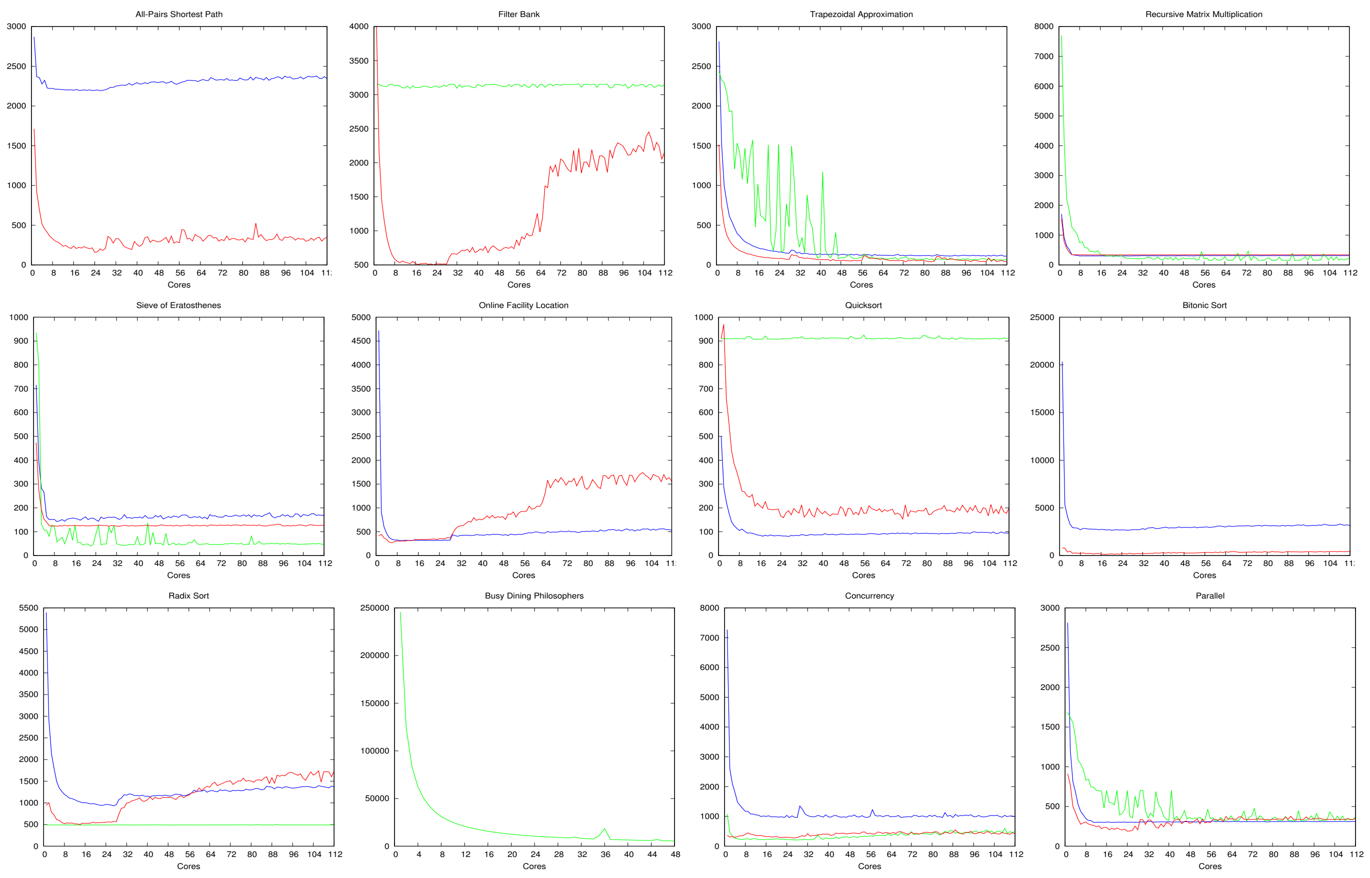
We combine the results of all the parallel benchmarks per language, and show the results in Figure 3 (c).

5 Performance Analysis

In the previous section, we described the key characteristics of each benchark, and pointed to the benchmark results. In this section, we perform a more thorough analysis and argue for a recategorisation of the Savina benchmarks.

5.1 Recategorisation

The fact that the plot for *philosophers* could change drastically after increasing the amount of work done by actors suggests the original categorisation does not capture the inherent concurrency of the benchmark. Instead, we propose the following recategorisation based on potential parallelism if actors do a non-trivial amount of work.



(a) Busy Philosophers **(b)** All Concurrency **(c)** All Parallelism

Figure 3. Parallel performance and scaling plots. Legend: Akka 2.3.2: 🟥, CAF 0.16.3: 🟦, Pony 0.28.0: 🟩. Subfigure (a) shows *philosophers* with extra work. Subfigures (b) and (c) coalesces concurrency and parallelism results respectively.

Sequential: only one actor is runnable at any instant of time; this category contains *Thread Ring* and *Ping Pong*. Therefore, both benchmarks are inherently sequential; we expect to see a flat line in the execution time plot. The plot for Pony is indeed a flat line; Akka shows some jitter, but mostly a flat line as well. CAF shows a flat line once the number of cores goes above 4, because CAF spawns multiple threads according to the number of cores in the machine, but never less than 4. The large execution time when the core count is less than 4 is the cost of over-provisioning the system and potentially disrespecting the NUMA affinity map given by *numactl*.

Constant parallelism: only a constant number of actors are runnable and doing non-trivial work; this category contains *count*, *chameneos*, *big*, *concdict*, *Banking*, *facloc*, *concsll*, *philosophers*, *barber*, *sieve*, and *uct*. These benchmarks show visible speedup when the number of cores are small, but flatten afterwards due to limited parallelism. For example, *barber* flattens at core 3.

Master & Slaves: a master pushes work to several slaves which can run in parallel; this category contains: *bndbuffer*, *cigsmok*, *Logmap*, *astar*, *NQueens*, *trapezoid*, *filterbank*, and *piprecision*. The trend in the plot should be similar to the previous category except that parallelism is tunable.

Recursive Divide-and-Conquer: work is distributed by recursively decomposing the problem into smaller subproblems; this category contains: *fjcreate*, *fjthrput*, *fib*, *apsp*, *recmatmul*, *quicksort*, *bitonicsort* and *radixsort*. It differs from the previous category only in how parallelism is created. Depending on the cost of actor creation relative to other workloads, that difference may not manifest itself in the plots. Indeed, in the Savina benchmarks it does not manifest itself.

5.2 Actor Benchmarks?

The Savina benchmarks are a collection of "found benchmarks," many of which were taken from a threaded domain and ported to an actor setting in a way where an actor is made to operate in a thread-like fashion. Porting sometimes breaks with the actor model: using shared counters manipulated by atomic operations; the sharing of the result matrix; message priorities. Furthermore, task-based parallelism (*e.g.,* parallel quicksort) typically relies on a notion of a task graph that captures dependencies between tasks (which *e.g.,* influences work-stealing)—actors are unstructured and cannot express such dependencies, making these algorithms unsuited for actor programs. Using futures to block execution (and possibly selective receive) can be used to encode similar dependencies, but is arguably not part of the actor model. Finally, the actor paradigm was developed for *concurrent* programs, not *parallel.* Few of the Savina benchmarks deal with responding to truly asynchronous behaviour.

A survey of how programmers tend to use actors in Scala can be found at [35]. In our experience, actor-based applications tend to consist of a large number of actors of varying lifetime, which receive messages frequently, and whose behaviours are relatively short-lived. We have *not* observed this kind of design pattern with the Savina benchmark suite.

5.3 Memory Management

A note on memory management is in place. We use the default G1 collector for our benchmarks. G1 has been claimed to scale as the number of cores increases [23] in a study which did not take into account NUMA-awareness when allocating objects [33]. Carpen-Amarie et al. measure performance of various collectors (including G1) in terms of responsiveness, throughput and GC pause times. The authors conclude that G1 performs badly in applications with a small memory footprint but reduce pause times for applications with a large memory footprint [4]. Ultimately, we are faced with a choice between GC behaviour which is ill-chosen for benchmark programs, or measurements on a GC which is unrepresentative for real programs.

The memory footprint of the Savina benchmark suite is something to be taken into account and appears to be difficult to tune using the available input parameters to the individual programs. Using the default configurations, during our measurements, the memory consumption across all languages did not grow beyond 11 GB. Scheduling and memory management are seperate, but equally difficult tasks to perform well in. Consequently, memory-boundedness as well as computation-boundedness should be tuning knobs of a cross-language benchmark suite.

6 One Benchmark to Rule them All

To mitigate many of the problems and pitfalls discussed in §2.1, we propose a malleable synthesised benchmark modelled after typical actor uses in industry: *ChatApp*. The idea of a malleable benchmark application is to provide several tuning knobs (see §6.2) for the application that can be used to put pressure on particular design points of an actor language, all within a single benchmark program without the need to create many specialised and highly different programs for various tasks. We believe that using the same program for multiple investigations will lead to intimate understanding of its characteristics in a way that can avoid or at least mitigate the obscuring of results in macro benchmarks when trying to study specific language aspects. Depending on the configuration, the program can be memory-bound, computation-bound, message-bound or mixed. Having such tuning knobs allows benchmarking to be more *experimental* and *focussed* at the same time, ultimately helping in understanding and comparing cross-language behaviors and draw conclusions for real-world application scenarios. Section §6.2 suggests some tuning knob settings to serve different objectives.

In the remainder of this section, we briefly sketch the ChatApp design and architecture, tuning knobs, and state the rules for its implementation.

6.1 ChatApp – A Malleable Benchmark

ChatApp models a chat application in the spirit of *e.g.,* Facebook chats. It consists of three types of actors: *clients*, *directories* and *chats.* A client is a server-side proxy for an imagined client on the client-side. A directory is a load-balancing mechanism that maps a number of client ids to their corresponding actor handles. A chat is a representation of a conversation between multiple client actors that keeps a history of a conversation, and a list of clients involved in the chat to which it forwards all incoming chat messages. Ideally, for running the benchmark, there should be an "external world" actor to supply non-random, repeatable input to the system. Figure 4 shows an example topology with two directories, three clients and three chats.

Input that spawns parallel activity arrives in *turns* and causes each actor to make one move (and possibly respond to many more). When an actor makes a move it rolls a die, and acts depending on the outcome: it may do nothing, start a new chat, leave an existing chat in which it is a member, or post a message in an existing chat in which it is a member. When a new chat is started, we iterate over the list of friends, roll a die for each one, and with some low probability invite them to the chat.

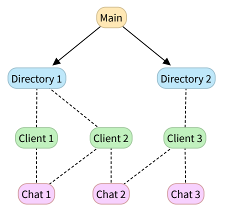


Figure 4. ChatApp example topology.

In addition, a client actor may respond to external stimuli. It may respond to a befriend request from another Client actor by adding the other actor to its friends; respond to an invite to join a chat by joining the chat, send an acknowledgement when receiving a message from one of its chats, or respond to a logout request from the system after the desired number of turns is completed, by setting its status in its containing directory to offline. When an actor is added to the system during start up, we iterate over the global list of actors and roll a die for each actor, and with some low probability send it a friend request. As the collection of actors grow during start up, an older actor is likely to be more well-connected than a younger actor, which models typical social network topologies.

Directories respond to requests for actor handles. When a chat receives a message, it forwards it to all of its members. It also maintains a list of clients and responds to join and leave requests.

6.2 Tuning Knobs

In addition to the number of turns (amount of work) and the number of actors (possible span), the ChatApp benchmark supports (at least) the following tuning knobs:

Work Per Client Move In addition to the behaviour described above, each client move may perform a variable amount of extra "busy work" to simulate more involved computation. This work can be compute-bound or memory-bound. It should be possible to assign different busy work to different actors, both with respect to type and amount. This tuning knob can be used to expose hidden parallelism, or sequential execution due to actors doing only trivial work, as shown in *philosopher*.

Client Behaviour The probabilities with which clients act in the various ways described above should be tuneable. For example, one can have a system with millions of actors with a 99% chance of doing nothing per turn, or a smaller number of less idle actors. As another example, decreasing the possibility of leaving chats to zero will cause the system's memory to get increasingly bloated as only more and more connections and objects will be created.

Clients Per Directory Whenever a client wants to obtain a handle to another client, it must go via a directory lookup. A single centralised directory will immediately become a bottleneck (possible objective) of the system as it will be disproportionally flooded by requests. Increase the number of directories to make sure clients are runnable without being blocked on a single shared resource.

Size of Messages Similar to the above, the size of messages sent should also be tuneable to be able to investigate message sending implementations. Some actor systems (*e.g.,* Erlang [6]) traverse and clone the transitive closure of argument objects in a message, possibly optimised through copy-on-write (*e.g.,* CAF). Others support message passing by pointer swizzle (*e.g.,* Akka, Pony and Encore [11]) but may require scanning outgoing and incoming messages for GC purposes (*e.g.,* Pony and Encore). Small message payloads (*e.g.,* primitive values and single-object structures) downplay these differences whereas larger message payloads highlight them. For example, a message to a chat can be a graph of objects of size 0 or more.

Degree of Connectedness The probability by which an actor will add another actor as a friend or add a friend to a newly created chat controls the degree of connectedness in the system which affects how many parallel units of work are created by each actor move.

6.3 Game Rules for ChatApp

Following our own recommendation, we define a clear set of rules for implementation of every aspect of the ChatApp's design considerations (which already satisfies R4 and R5) and tuning knobs (take this to be in addition to §3.5). Most importantly, directories need to be actors, not shared read-only collections of some sort (R0). Consequently, a message round-trip and actor-local hash map look-up is required to retrieve an actor handle from its identifier. The amount of actors a directory can handle is implicitly decided by the number of client actors in the system, which are distributed uniformly between all available directories.

Each turn, each client is sent one message to move, and executes exactly one move as a result (which may amount to "do nothing"), without coordinating its decision with any other clients. Moreover, following our initial proposition, an actor picks its behavior to execute based on a factory class (local to each client) using a congruential (but deterministic) random number generator based on 32-bit integers (breaking R3!), which should be implementable on almost any platform and any language or framework as demonstrated in [9].

Ideally, we would like to eliminate any degree of number generation in future versions of the ChatApp (R3), and control the systems behavior based on some pre-defined repeatable input. This could be an external benchmark driver communicating against TCP sockets (to which the directories listen), allowing us to control the behavior of clients, chats as well as the size of messages posted to any of the actors (even complex object graphs). The specification of such a driver is subject to future work.

7 Conclusion

We compared the runtime characteristics of Akka, CAF and Pony based on the Savina benchmark suite. We observed similarities between many of the benchmarks, and drastic effects of small tweaks to the benchmarks. As a result, we proposed one, highly tunable benchmark, which we plan to implement and study in a range of actor-based languages.

References

[1] 2019. *CAF, C++ Actor Framework version 0.17.* http://www.actor-framework.org/pdf/manual.pdf

[2] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[3] Beatrice Åkerblom, Elias Castegren, and Tobias Wrigstad. 2020. Reference Capabilities for Safe Parallel Array Programming. *The Art, Science, and Engineering of Programming* (2020). To appear.

[4] Maria Carpen Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. 2015. A performance study of Java garbage collectors on multicore

architectures. 20–29. https://doi.org/10.1145/2712386.2712404

[5] Aris Anagnostopoulos, Russell Bent, Eli Upfal, and Pascal Van Hentenryck. 2004. A Simple and Deterministic Competitive Algorithm for Online Facility Location. *Inf. Comput.* 194, 2 (Nov. 2004), 175–202. https://doi.org/10.1016/j.ic.2004.06.002

[6] Joe Armstrong. 2007. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 6–1–6–26. https://doi.org/10.1145/1238844.1238850

[7] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. 2012. A Scalability Benchmark Suite for Erlang/OTP. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop (Erlang '12)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/2364489.2364495

[8] Sebastian Blessing. 2013. *A String of Ponies. Transparent Distributed Programming with Actors.* Master's thesis. Imperial College of Science. https://www.doc.ic.ac.uk/~scb12/publications/s.blessing.pdf

[9] Sebastian Blessing. 2019. GitHub - Pony Savina. https://github.com/sblessing/pony-savina. (2019). Accessed: 2019-08-13.

[10] Sebastian Blessing. 2019. Savina Results - Medium Machine. https://www.doc.ic.ac.uk/~scb12/pony.html. (2019). Accessed: 2019-08-13.

[11] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming - 15th SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*. 1–56. https://doi.org/10.1007/978-3-319-18941-3_1

[12] Denis Caromel, Christian Delbe, Alexandre Di Costanzo, and Mario Leyton. 2006. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology* 12 (2006), issue 1. https://hal.archives-ouvertes.fr/hal-00125034

[13] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2014. CAF - the C++ Actor Framework for Scalable and Resource-Efficient Applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*. 15–28. https://doi.org/10.1145/2687357.2687363

[14] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2016. Revisiting actor programming in C++. *Computer Languages, Systems & Structures* 45 (2016), 105–131. https://doi.org/10.1016/j.cl.2016.01.002

[15] Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully concurrent garbage collection of actors on many-core machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 553–570. https://doi.org/10.1145/2509136.2509557

[16] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*. 1–12. https://doi.org/10.1145/2824815.2824816

[17] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and type system co-design for actor languages. *PACMPL* 1, OOPSLA (2017), 72:1–72:28. https://doi.org/10.1145/3133896

[18] The TPC consortium. 2019. TPC-C - An On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/default.asp. (2019). Accessed: 2019-08-13.

[19] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*. 230–254. https://doi.org/10.1007/11785477_16

[20] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*. ACM, 37–48.

[21] Edsger W. Dijkstra. 2002. The Origin of Concurrent Programming. Springer-Verlag New York, Inc., New York, NY, USA, Chapter Cooperating Sequential Processes, 65–138. http://dl.acm.org/citation.cfm?id=762971.762974

[22] Juliana Franco, Sylvan Clebsch, Sophia Drossopoulou, Jan Vitek, and Tobias Wrigstad. 2018. Correctness of a Concurrent Object Collector for Actor Languages. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 885–911. https://doi.org/10.1007/978-3-319-89884-1_31

[23] Lokesh Gidra, Gael Thomas, Julien Sopena, and Mark Shapiro. 2012. Assessing the Scalability of Garbage Collectors on Many Cores. *SIGOPS Oper. Syst. Rev.* 45, 3 (Jan. 2012), 15–19.

[24] P. Haller and M. Odersky. 2009. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410, 2-3 (2009), 202–220.

[25] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2) (1968), 100–107.

[26] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*. 235–245. http://ijcai.org/Proceedings/73/Papers/027B.pdf

[27] C. A. R. Hoare. 1974. Monitors: An Operating System Structuring Concept. *Commun. ACM* 17, 10 (Oct. 1974), 549–557. https://doi.org/10.1145/355620.361161

[28] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. https://doi.org/10.1145/359576.359585

[29] Samuel Horsley. 1772. The Sieve of Eratosthenes. Being an Account of His Method of Finding All the Prime Numbers, by the Rev. Samuel Horsley, F. R. S. *Philosophical Transactions (1683-1775)* 62 (1772), 327–347.

[30] 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 &#38; Decentralized Control (AGERE! '14)*. ACM, New York, NY, USA, 67–80. https://doi.org/10.1145/2687357.2687368

[31] Shams M. Imam and Vivek Sarkar. 2019. GitHub - Savina. https://github.com/shamsimam/savina. (2019). Accessed: 2019-08-13.

[32] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2010. ABS: A Core Language for Abstract Behavioral Specification. 142–164. https://doi.org/10.1007/978-3-642-25271-6_8

[33] Tomas Kalibera, Matthew Mole, Richard E. Jones, and Jan Vitek. 2012. A black-box approach to understanding concurrency in DaCapo. In *Proceedings of the 27th Annual ACM SIGPLAN OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. 335–354. https://doi.org/10.1145/2384616.2384641

[34] Suhas S. Patil. 1971. Limitations and capabilities of dijkstra's semaphore primitives for coordination among processes.

[35] Tasharofi Samira, Dinges Peter, and Johnson Ralph. 2012. Why Do Scala Developers Mix the Actor Modelwith Other Concurrency Models?. In *ECOOP*.

[36] Dave Thomas. 2018. *Programming Elixir ≥ 1.6: Functional |> Concurrent |> Pragmatic |> Fun.* Pragmatic Bookshelf.

[37] Sebastian Woelke. 2019. GitHub - CAF Fork Savina. https://github.com/woelke/savina. (2019). Accessed: 2019-08-13.

[38] Derek Wyatt. 2013. *Akka Concurrency.* Artima.